# Integrity and access control in untrusted content distribution networks

by

Kevin E. Fu

B.S. Computer Science and Engineering, MIT, 1998
M.Eng. Electrical Engineering and Computer Science, MIT, 1999

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 2005





Author .....
Department of Electrical Engineering and Computer Science
September 6, 2005

Certified by .....
M. Frans Kaashoek
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Ronald L. Rivest
Andrew and Erna Viterbi Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

THIS PAGE INTENTIONALLY LEFT BLANK

# Integrity and access control in untrusted content distribution networks

by

Kevin E. Fu

Submitted to the Department of Electrical Engineering and Computer Science
on September 6, 2005, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

## Abstract

A content distribution network (CDN) makes a publisher's content highly available to readers through replication on remote computers. Content stored on *untrusted servers* is susceptible to attack, but a reader should have confidence that content originated from the publisher and that the content is unmodified. This thesis presents the SFS read-only file system (SFSRO) and *key regression* in the Chefs file system for secure, efficient content distribution using untrusted servers for public and private content respectively.

SFSRO ensures integrity, authenticity, and freshness of single-writer, many-reader content. A publisher creates a digitally-signed database representing the contents of a source file system. Untrusted servers replicate the database for high availability. Chefs extends SFSRO with key regression to support *decentralized access control* of private content protected by encryption. Key regression allows a client to derive past versions of a key, reducing the number of keys a client must fetch from the publisher. Thus, key regression reduces the bandwidth requirements of publisher to make keys available to many clients.

Contributions of this thesis include the design and implementation of SFSRO and Chefs; a concrete definition of security, provably-secure constructions, and an implementation of key regression; and a performance evaluation of SFSRO and Chefs confirming that latency for individual clients remains low, and a single server can support many simultaneous clients.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Ronald L. Rivest
Title: Andrew and Erna Viterbi Professor of Electrical Engineering and Computer Science

# Acknowledgments

A USENIX Scholars Fellowship, an Intel PhD Fellowship, and Project Oxygen provided partial support for this research. Part of this research was performed while visiting the Johns Hopkins University.

I especially thank my thesis advisors (Frans Kaashoek and Ron Rivest) and my thesis committee readers (Mahesh Kallahalla, David Mazières, Avi Rubin, and Ram Swaminathan) for their service.

Ram Swaminathan and Mahesh Kallahalla provided the early design of the key regression protocols. I thank Ron Rivest and David Mazières for suggestions on formalizing definitions of security, and Frans Kaashoek for his persistence, guidance, and unending support.

Giuseppe Ateniese, Breno de Medeiros, Susan Hohenberger, Seny Kamara, and Yoshi Kohno deserve many thanks for their suggestions on theoretical definitions of security for key regression. Seny and Yoshi in particular deserve the most credit in forming the definitions and proofs of security for key regression.

I thank Emmett Witchel for producing the first implementation of the SFS read-only file system. I thank Anjali Prakash, an undergraduate from Johns Hopkins, for implementing the initial version of hash-based key regression and finding statistics about group dynamics.

Portions of the thesis appeared in the following publications:

"Fast and secure distributed read-only file system" by Kevin Fu, M. Frans Kaashoek, David Mazières. In ACM Transactions on Computer Systems 20(1), February 2002, pages 1-24. (Originally published in the Proceedings of the 4th Symposium on Operating System Design & Implementation (OSDI). San Diego, California, October, 2000.)

"Plutus: Scalable secure file sharing on untrusted storage" by Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, Kevin Fu. In Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST), 2003.

"Improved proxy re-encryption schemes with applications to secure distributed storage" by Giuseppe Ateniese, Kevin Fu, Matthew Green, Susan Hohenberger. In Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS). San Diego, CA, February 2005.

"Key regression: Enabling efficient key distribution for secure distributed storage" by Kevin Fu, Seny Kamara, Tadayoshi Kohno. Manuscript, August 2005.

## Academic acknowledgments

Over the years many people have answered my relentless questions. I thank the many members of CSAIL, PDOS, NMS, and my now defunct MIT Applied Security Reading Group for their feedback on paper submissions and general mayhem: Daniel Aguayo (superman!), David Andersen (always scavenging for free food), Sameer Ajmani (always willing to have discussions),

Magdalena Balazinska ("milk pool" makes research happen), John Bicket (roof-scaling officemate), Sanjit Biswas (another roofnet hacker), Chuck Blake (dealing with my computer hardware woes and wearing tank tops), Josh Cates (the quiet officemate), Benjie Chen (golfing with alligators), Russ Cox (bottomless source of knowledge and boy scout mishaps), Frank Dabek (golfing at Shoreline, hacking SFS code, even Latin scholarship), Doug De Couto (sailing sailing...), Nick Feamster (working on the cookie paper and reminiscing about Latin/JCL), Bryan Ford (I'll never forget Orgazmo), Simson Garfinkel (likes to argue, but that's what science requires), Thomer Gil (a Dutchman who can bake), Michel Goraczko (for computing support and even automobile support), John Jannotti (easy going but always able to precisely shoot down stupid ideas), Jaeyeon Jung (security and sushi along with Stuart Schechter), Michael Kaminsky (SFS forever!), Eddie Kohler (Metapost!), Max Krohn (your SCI paper is better organized than my paper), Chris T. Lesniewski-Laas (SIPB can be smart and have style), Jingyang Li (my virtual officemate), Anna Lysyanskaya (theory of cryptography), David Mazières (always the docile, well-behaved child), Allen Miu (HP Labs marches on), Robert Morris (graduate if you should be so lucky), Athicha Muthitacharoen (fun in Palo Alto at HP Labs too), Emil Sit (our work on cookies never seems to die), Alex Snoeren (for camaraderie and hosting me at UCSD), Jacob Strauss (for letting me steal office space), Jeremy Stribling (for voluntarily disabling SCIgen [114] so that I could benchmarks Chefs), Godfrey Tan (Intel fellowship), Michael Walfish (I hesitate to use improper grammar in front of you), Alex Yip (former student who could teach me a thing or two).

Thank you to Abhi Shelat and Thien-Loc Nguyen for making 6.857 (Network and Computer Security) a great course to TA.

Jerry Saltzer was a welcome source of computing history. Upon my presenting an idea, he quickly recalls related work from decades ago.

Thanks to Sam Hartman, my undergraduate roommate, for perhaps unintentionally steering me into the field of computer security.

Be "Super Be" Blackburn and Neena Lyall deserve an ACM award for all their help assisting with administration. Thanks go to Paula Michevich and Maria Sensale for their help locating hard-to-find research papers available only in the LCS-now-CSAIL reading room.

Thanks to the folks at the Johns Hopkins University and the SPAR Lab in the Information Security Institute for making my two year visit enjoyable: Giuseppe Ateniese, Kenneth Berends, Lucas Ballard, Steve Bono, Randal Burns, Reza Curtmola, Darren Davis, Breno de Medeiros, Sujata Doshi, Matt Green, Seny Kamara, Mike LaBarre, Gerald Masson, Fabian Monrose, Zachary Peterson, Jackie Richardson, Avi Rubin, Robin Shaw, Christopher Soghoian, Adam Stubblefield, Sophie Qiu, Jing Wang, and Charles Wright.

Prof. Herb Dershem at Hope College in Holland, Michigan deserves thanks for letting me take computer science courses while still enrolled in high school. Alan Parker from Chapel Hill, NC first got me interested in computing on the IBM PC Jr., and the late Kenneth Glupker taught my first computer programming courses at West Ottawa High School. Pat Seif (English and broadcast journalism) and the late Mary Jo LaPlante (Latin) did their best to guide my creative energies in my formative years. Whoever says we do not need more English majors is wrong.

## Industrial acknowledgments

I thank Avi Rubin and Dan Boneh for taking the risk of mentoring a mere college sophomore at Bell Communications Research. Avi Rubin unintentionally piqued my interests in cryptographic file systems. In 1996, Avi made me propose several summer research topics, with cryptographic file systems my top choice. He then left Bellcore, leaving me to my own devices. But I got my revenge. He's stuck on my committee now. Thanks go out to my Bellcore officemate (and housemate) Giovanni Di Crescenzo for his guidance in cryptography and Manhattan jazz.

Many companies and computing groups provided me with invaluable experience in computer security: HP Labs (Ram and Mahesh), Bellcore (Bill Aiello, Dan Boneh, Rich Graveman, Neil Haller, Raj Rajagopalan, Avi Rubin), Sighpath/Cisco, Holland Community Hospital (Bruce Eckert), and MIT Information Systems (Bob Mahoney, Jeff Schiller, Oliver Thomas and many others).

## Family and friends

Thank you Teresa, my little piece of toast, for letting me spend eleven years in college. It's your turn now. Thanks go to my parents, Mary Claire and Wallace, for their tireless efforts and co-signing my student loans. Thanks to my in-laws, Richard and Maureen, for bringing Teresa into this world. Thanks to my sister Laura for contributing to the breaking of my right arm on March 10, 1987 (admit it, a treehouse should not have planks lying on the forest floor), forcing me to learn how to type on a computer. Thanks to Sarah for housing me in Boston, and thanks to Eric for countering my type-A personality.

Thanks go to my undergraduate students at English House (Conner 2) from 1999-2003 for giving me a reason to prepare group dinners and cook fast, scalable meals.

Finally, thanks go to the "fine" accommodations of MIT Housing for giving me many solid and liquid incentives to graduate.

# Contents

# List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

# List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 1

# Introduction

*Nullius boni sine socio iucunda possessio est.*
*(No good thing is pleasing to possess without a friend to share it.)*
– Lucius Annaeus Seneca,
Epistulae Morales Liber I §VI (4)

A content distribution network (CDN) reduces the bandwidth requirements of a publisher to make single-writer content available to many readers. For instance, one could pay a service provider or enlist volunteers to replicate and serve content. While such a model achieves the goal of making content widely available, it does not guarantee secure distribution of content. Volunteers could damage integrity by maliciously modifying content. A compromised service provider could leak confidential, access-controlled content to unauthorized parties. Thus, secure content distribution can fail when the trusted computing base no longer includes the servers replicating content.

Responding to these security shortfalls, this thesis explores how to ensure integrity protection of public content and access control of private content using *untrusted servers* as depicted by Figure 1-1.

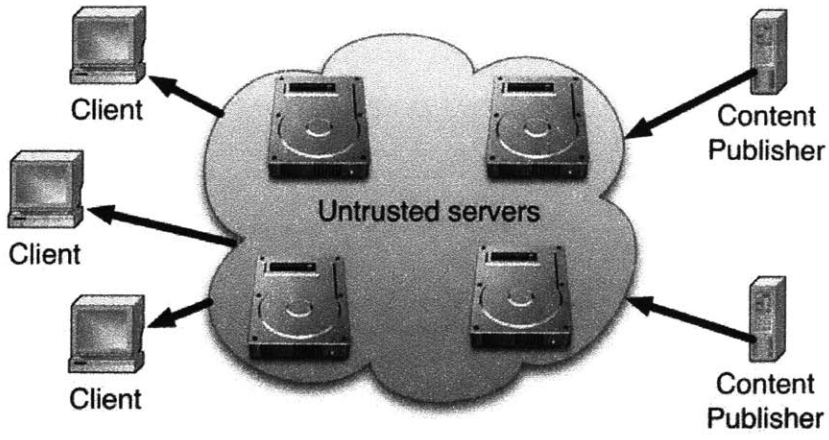


Figure 1-1: Overview of content distribution using untrusted storage. Content publishers produce content, which is replicated on untrusted servers. Users on client machines then query untrusted servers for content.

## 1.1 Challenges

Two challenges motivated the research of this thesis in secure content distribution. First, clients must be able to verify integrity of content. The content must appear as the content publisher intended. Second, content publishers should be able to control access to content replicated on untrusted servers.

### 1.1.1 How to guarantee integrity in untrusted storage?

Users and content publishers desire two basic properties for secure content distribution. First, users want to ensure that *collections* of file content appear as the content publisher intended. For instance, signed software packages guarantee integrity of individual packages, but do not guarantee the integrity of an operating system composed of several thousand packages. Second, content publishers do not want to sacrifice performance for security. If a publisher were to sign an entire collection of files in the same way as signing individual packages, then a single change to a package would require the resigning and redistribution of the entire collection.

System designers have introduced a number of *ad hoc* mechanisms for dealing with the security

of public content, but these mechanisms often prove incomplete and of limited utility to other applications. For instance, binary distributions of Linux software packages in RPM [100] format can contain PGP signatures. However, few people actually check these signatures, and packages cannot be revoked. In addition, when packages depend on other packages being installed first, the dependencies are not secure (e.g., one package cannot explicitly require another package to be signed by the same author).

The approach of the SFS read-only file system (SFSRO)[1] maps a Merkle hash tree [76] over the directory structure of a file system. By preserving the integrity of an entire file system rather than of individual files, a user can be sure that not only does a file have integrity, but also that a file appears in the context the content publisher intended. A user can detect the misrepresentation or absence of a file.

An administrator creates a database of a file system's contents and digitally signs it offline using the file system's private key. The administrator then replicates the database on untrusted machines. There a simple and efficient program serves the contents of the database to clients, without needing access to the file system's private key. DNS round-robin scheduling or more advanced techniques [59] can be used to distribute the load among multiple servers. A trusted program on the client machine checks the authenticity of content before returning it to the user.

SFSRO avoids performing any cryptographic operations on servers and keeps the overhead of cryptography low on clients. SFSRO achieves this performance improvement by using two simple techniques. First, file system structures are named by *handles*, which are collision-resistant cryptographic hashes of their contents. Second, groups of handles are hashed recursively, producing a tree of hashes [76]. Using the root handle of a file system, a client can verify the contents of any directory subtree by recursively checking hashes.

The SFSRO protocol allows the content producer to replicate a database at many servers, supports frequent updates to the database, and enables clients to check the freshness of the content retrieved from a server. Since the protocol does not need access to the private key that signed the

[1]The SFS read-only file system (SFSRO) was named as such because it uses SFS's naming scheme and fits into the SFS framework. However, the system does support updates. More accurately, the file system provides single-writer, many-reader access. Despite the misnomer, we continue with the shorthand "SFSRO" to name our system.

database, the database can be replicated at many servers without compromising security. To allow for efficient updates, servers incrementally update their copies. Clients check the freshness of the content using time stamps.

### 1.1.2 How to control access in untrusted storage?

Already a large quantity of access controlled content exists on the Internet. For instance, the Wall Street Journal online edition boasts of having 701,000 paid subscribers [117]. Publishers cannot use untrusted servers to enforce access control in content distribution. Traditional access control uses a reference monitor for complete mediation of access to a protected resource [104]. When a CDN is untrusted and the content publisher cannot rely on the network to enforce proper access control, the content publisher can achieve *decentralized access control* by encrypting the content and distributing the cryptographic keys to legitimate users [46, 54, 58, 79, 90]. Using *lazy revocation* [43, 58] after evicting a member, the content publisher encrypts new content with a new cryptographic key distributed to remaining members. The content publisher does not re-encrypt existing content since the evicted member could have already cached that content.

Lazy revocation delays the re-encryption of files until absolutely necessary. This approach, however, can result in the proliferation of many keys. As evictions and updates occur, subsets of files in a single database become encrypted with different group keys. A straightforward approach makes users query the publisher for every key. This quickly becomes infeasible as the number of keys and number of users can easily overwhelm the bandwidth of a publisher's network connection. This thesis proposes *key regression* to reduce the bandwidth requirements of such key distribution. Key regression coalesces many different versions of a key into a compact form.

To demonstrate the practical benefits of key regression, this thesis presents the Chefs file system for decentralized access control of private content served by untrusted servers. Chefs extends SFSRO by additionally encrypting content for confidentiality. Clients with the proper decryption key can then access the content. Chefs uses key regression to enable a publisher on a low-bandwidth connection to efficiently distribute keys to clients.

## 1.2 Applications

Practical applications motivated the design and implementation of SFSRO and Chefs. This chapter highlights a few of the specific applications this thesis envisions. Some of these applications are used to demonstrate the performance benefits of SFSRO and Chefs.

### 1.2.1 Integrity-protected public content

Certificate authorities and software distribution are two applications that require integrity protection of content, and distribution that scales well with the number of clients.

**Certificate authorities.** Certificate Authorities (CAs) [61] for the Internet are servers that publish signed certificates binding hostnames to public keys. Three goals guide the design of CAs: high availability, integrity, and freshness of certificates. High availability ensures that a client can easily access a certificate. Integrity ensures that a certificate is valid and properly signed by a CA. Freshness ensures that a client will be made aware of revoked certificates.

Using SFSRO is one way to build a CA for high availability, integrity, and freshness. A CA could replicate an SFSRO database containing unsigned certificates, which could be implemented as unsigned X.509 certificates [39] or symbolic links to self-certifying paths [73]. Unlike traditional certificate authorities, SFS certificate authorities get queried interactively. This design simplifies certificate revocation, since revoking a key amounts to removing a file or symbolic link.

Instead of signing each certificate individually, SFSRO signs a database containing a collection of unsigned certificates [83]. The database signature provides for the authenticity and integrity of all the certificates. SFSRO improves performance by making the amount of cryptographic computation proportional to the file system's size and rate of change, rather than to the number of clients connecting. SFSRO also improves integrity by freeing SFS certificate authorities from the need to keep any online copies of their private keys. Finally, SFS read-only improves availability because it can be replicated on untrusted machines.

**Software distribution.** Sites distributing popular software have high availability, integrity, and performance needs. Open source software is often replicated at several mirrors to support a high number of concurrent downloads. If users download a distribution with anonymous FTP, they have low content integrity: a user cannot tell whether it is downloading a trojan-horse version instead of the correct one. If users connect through the Secure Shell (SSH) [127] or secure HTTP, then the server's throughput is low because of cryptographic operations it must perform. Furthermore, that solution does not protect against attacks where the server is compromised and the attacker replaces a program on the server's disk with a trojan horse.

By distributing software through SFS read-only servers, one can provide integrity, performance, and high availability. Users with an SFSRO client can even browse the distribution as a regular file system and compile the software directly from the sources stored on the SFS file system. The SFSRO client will transparently check the authenticity of content, and raise an error if it detects unauthorized modification of content. To distribute new versions of the software, the administrator simply updates the database. Users with only a browser could get all the benefits by connecting through a Web-to-SFS proxy to the SFS file system.

### 1.2.2 Access-controlled private content

The concept of using untrusted storage to replicate private content is somewhat counterintuitive. After all, what server would want to replicate content that the server itself cannot read? Nevertheless, there are signs that such a sharing infrastructure may become a reality, as demonstrated by the example applications in this section.

**Subscriptions.** We target key regression at publishers of popular content who have limited bandwidth to their trusted servers, or who may not always be online, but who can use an untrusted CDN to distribute encrypted content at high throughput. Our experimental results show that a publisher using key regression on a low-bandwidth connection can serve more clients than the strawman approach of having the publisher distribute all keys directly to members.

Such a publisher might be an individual, startup, or cooperative with popular content but with

few network resources. The possibilities for such content ranges from blogs, photo galleries, and amateur press to mature content and operating systems. To elaborate on one such form of content: Mandriva Linux [71] currently uses the BitTorrent [30] CDN to distribute its latest Linux distributions to its paid members. Mandriva controls access to these distributions by only releasing the `.torrent` files to its members. Using key regression, Mandriva could exercise finer-grained access control over its distributions, allowing members through time period $i$ to access all versions of the operating system including patches, minor revisions and new applications added through time period $i$, but no additions to the operating system after time period $i$. To grant universal access to security-critical patches, Mandriva could encrypt security-critical patches with the key for the time period to which the patch is first applicable.

**Time-delayed release.** In the financial world, making sure every party can receive public financial statements at the same time ensures fairness. Otherwise, a party who opens the statement earlier would have an advantage. Perhaps that party would sell its shares of stock ahead of the tide. A content distribution network can preposition content so that content is readily available to all users. However, the prepositioning may finish on servers at different times. Therefore, users of early servers will have an advantage.

Chefs can enable time-delayed release of content by publishing the group key at the release time. The distribution of sealed content can happen asynchronously. Once all the prepositioning finishes, the content owner can distribute the group key from a well-known location. Because the group key is usually much smaller than the content (16 bytes versus hundreds or thousands of gigabytes), Chefs enables fairness of time-delayed release of content.

This application is particularly well-suited to the semantics provided by Chefs because no evictions ever happen in time-delayed release. Instead, content begins private, but ends up completely public.

**Versioning file systems** The semantics of Chefs work best when a system never revokes a member's access to old content, such as in the case of versioning file systems that never forget [89, 102]. In a versioning file system, saving modifications to a file results in a new version of a file. Yet the

old file remains accessible through filenames based on time. The old data never disappears.

## 1.3 Contributions

This thesis makes three primary contributions:

- **Efficient integrity protection of public content distributed by untrusted servers**

  SFSRO system makes the security of published content independent from that of the distribution infrastructure. In a secure area, a publisher creates a digitally-signed database out of a file system's contents. The private key of the file system does not have to be online, allowing the database to be replicated on many untrusted machines. The publisher then replicates the database on untrusted content distribution servers, allowing for high availability.

  SFSRO is one of the first systems to demonstrate the full potential of the Merkle hash tree [76] in secure storage. First, the Merkle tree provides efficient integrity protection in a block-store-based file system. The network round-trip time overshadows the cost of verifying a hash, making the Merkle tree verification a zero cost operation. Second, the Merkle tree serves as a block naming mechanism. The naming scheme allows for efficient incremental updates because an untrusted server replicating content can immediately prune out unmodified subtrees of the file system. Third, the Merkle tree allows SFSRO to use a simple client-server protocol consisting of only two remote procedure calls: one for mounting a file system and one for fetching blocks of content.

- **Key regression for efficient key distribution**

  The Plutus file system [58] introduced the notion of *key rotation* as a means to derive a sequence of temporally-related keys from the most recent key. This thesis shows that, despite natural intuition to the contrary, key rotation schemes cannot generically be used to key other cryptographic objects; in fact, keying an encryption scheme with the output of a key rotation scheme can yield a composite system that is insecure.

To address the shortcomings of key rotation, a new cryptographic object called a *key regression* scheme is introduced. Proofs demonstrate that key regression based on SHA1, AES, and RSA are secure in the reduction-based provable security approach pioneered by Goldwasser and Micali [50] and lifted to the concrete setting by Bellare, Kilian, and Rogaway [12].

- **Implementation and evaluation of SFSRO and Chefs**

  The read-only file system avoids performing any cryptographic operations on servers and keeps the overhead of cryptography low on clients, allowing servers to scale to a large number of clients. Measurements of an implementation show that an individual server running on a 550 MHz Pentium III with FreeBSD can support 1,012 connections per second and 300 concurrent clients compiling a large software package.

  Chefs extends SFSRO by encrypting content for confidentiality using key regression. Measurements of users downloading keys to search encrypted content show that key regression can significantly reduce the bandwidth requirements of a publisher distributing keys to members. On a 3.06 GHz Intel Xeon CPU with a simulated cable modem, a publisher using key regression can make 1,000 keys available to 181 clients per second whereas without key regression the cable modem limits the publisher to 20 clients per second. The significant gain in throughput conservation comes at no measurable cost to client latency, even though key regression requires more client-side computation. Furthermore, key regression reduces client latency in cases of highly dynamic group membership.

## 1.4 Roadmap

The rest of this thesis is organized into seven additional chapters.

Chapter 2 presents the design of SFSRO, a scalable content distribution system for clients to securely access public content replicated on untrusted servers. The chapter enumerates the goals, components, protocol, data structures, and incremental update mechanism of SFSRO.

Chapter 3 explains decentralized access control and the design of the Chefs file system. Chefs extends SFSRO to ensure that only authorized members may access the protected content. Even

the servers replicating the content do not have access to the protected content.

Lazy revocation in encrypted storage causes a proliferation of group keys. Different files in the collection may be encrypted with different versions of the same group key. Chapter 4 explains how key regression allows group members with a current group key to easily derive old versions of the group key. Yet only the group owner should be able to generate the future versions of the group key, preventing evicted members from discovering future keys. The chapter also explains the cryptographic theory of key regression.

Chapter 5 presents the implementations of SFSRO and Chefs so that Chapter 6 may evaluate the performance of SFSRO and Chefs under selected workloads.

Chapter 7 compares SFSRO and Chefs to related work in secure file systems, content distribution networks, and cryptography.

Finally, Chapter 8 closes this thesis with conclusions and suggestions for future work.

# Chapter 2

# Integrity in the SFS read-only file system

Доверяй, но проверяй

*(Trust, but verify)*

– Russian proverb

This chapter explains the design of the SFS read-only file system, beginning with a discussion of the goals that guided the design of SFSRO. Section 2.2 presents an overview of the file system's architecture and the cryptographic primitives it employs. Section 2.3 describes the read-only file system protocol. Section 2.4 describes the data structures that comprise the file system. Section 2.5 describes the process of updating the file system.

SFSRO consists of three programs: a database generator, a server, and a client. In a trusted area the database generator creates a signed database from a file system's contents. The database is replicated on untrusted machines. In response to requests from clients, the server looks up content from its copy of the database and returns it to the client. The client verifies the retrieved content for authenticity and recentness, and interprets it as a file system.

Each read-only file system has a public key associated with it. SFSRO uses the naming scheme of SFS [73], in which file names contain public keys. Thus, users can employ any of SFS's various key management techniques to obtain the public keys of file systems.

The protocol between the client and server consists of only two remote procedure calls: one

to fetch the signed handle for the root inode of a file system, and one to fetch the data (inode or file content) for a given handle. Since the server does not have to understand what it is serving, its implementation is both trivial and highly efficient: it simply looks up handles in the database and sends them back to the client.

## 2.1 Goals

Single-writer, many-reader content can have high performance, availability, and security needs. Some examples include executable binaries, software distribution, bindings from hostnames to addresses or public keys, and popular Web pages. In many cases, people widely replicate and cache such content to improve performance and availability—for instance, volunteers often set up mirrors of popular operating system distributions. Unfortunately, replication generally comes at the cost of security. Each server adds a new opportunity for attackers and server operators to tamper with content. Therefore, SFSRO aims to satisfy a number of performance and security goals.

**Scalability to many clients.** A scalable infrastructure is desired for distribution of public content. Publishers should be able to reach a wide audience without needing a high-bandwidth connection. Furthermore, each server replicating content should provide high throughput to support as many clients as possible. A server should not have to perform any complex operations that would decrease throughput. The use of untrusted servers leads to the next goal.

**Integrity and freshness of content.** The SFS read-only file system assumes that an attacker may compromise and assume control of any read-only server machine. It therefore cannot prevent denial-of-service from an attacker penetrating and shutting down every server for a given file system. However, the client does ensure that any data retrieved from a server is authentic, no older than a file system-configurable consistency period, and also no older than any previously retrieved data from the same file system.

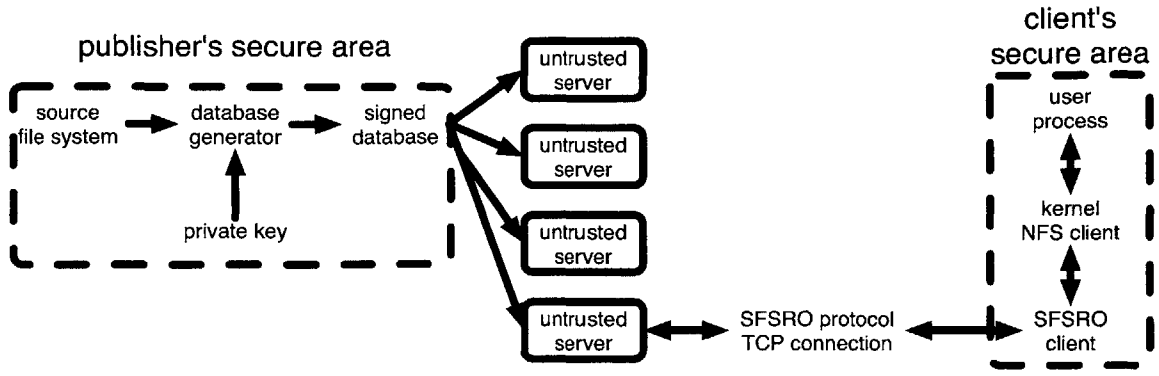


Figure 2-1: The SFS read-only file system. Dashed boxes indicate the trusted computing base.

**Denial of service.** A large number of servers may help to withstand the impact of denial of service. An attacker would have to disable all servers to completely shutdown content distribution. An attacker could delay a targeted client from accessing content, but integrity verification makes sure that the client will not accept unauthentic content from an attacker.

**No confidentiality.** SFSRO provides no confidentiality of content. All content distributed through SFSRO is public. Confusion in the past resulted in a number of papers describing SFSRO as confidential content distribution. This is not true. Rather, this thesis addresses the problem of providing confidentiality in the Chefs file system, described in Chapter 3.

## 2.2 Overview

Figure 2-1 shows the overall architecture of the SFS read-only file system. In a secure area, a content publisher runs the SFS database generator, passing as arguments a directory of files to export and a file containing a private key. The directory represents the origin file system. The SFSRO database will incorporate the entire contents of the given directory tree. The publisher replicates this database on a number of untrusted servers, each of which runs the SFS read-only server daemon.

The server is a simple program that looks up and returns blocks of data from the database at the request of clients. The following two sections describe the database generator and client daemon.

### 2.2.1 Database generation

To export a file system, a system administrator produces a signed database from a source directory in an existing file system. The database generator reads content from the directory tree, inserting the content into a database. The database contains the file data blocks and inodes indexed by their hash values as shown in Figure 2-3. In essence, it is analogous to a file system in which inode and block numbers have been replaced by cryptographic hashes.

The database generator utility traverses the given file system depth-first to build the database. The leaves of the file system tree are files or symbolic links. For each regular file in a directory, the database generator creates a read-only inode structure and fills in the metadata. Then, it reads the blocks of the file. For each block, the database generator hashes the data in that block to compute its handle, and then inserts the block into the database under the handle (i.e., a lookup on the handle will return the block). The hash value is also stored in an inode. When all file blocks of a file are inserted into the database, the filled-out inode is inserted into the database under its hash value.

Inodes for symbolic links are slightly different from the one depicted in Figure 2-3. Instead of containing handles of blocks, the inode directly contains the destination path for the symbolic link.

After the whole directory tree has been inserted into the database, the generator utility fills out an `FSINFO` structure and signs it with the private key of the file system. For simplicity, the database generator stores the signed `FSINFO` structure in the database under a well-known, reserved handle.

The database of file system structures stored under their content hashes serves two purposes. First, the content hash provides a naming mechanism to locate content. Second, the resulting Merkle hash tree provides integrity protection [76]. The next section explains how a client traversing the directory tree can efficiently verify integrity.

We chose the Rabin public key cryptosystem [121] for database authentication because of its fast signature verification time. SFS also uses the SHA1 [84] cryptographic hash function. SHA1

is a collision-resistant hash function that produces a 20-byte output from an arbitrary-length input. Finding any two inputs of SHA1 that produce the same output is believed to be computationally intractable. Modern machines can typically compute SHA1 at a rate greater than the local area network bandwidth. Thus, a client can reasonably verify integrity by hashing all the content resulting from RPC calls of an SFSRO server.

### 2.2.2 Client daemon

The most challenging part of the file system is implemented by the client. The client handles all file system requests for read-only file systems. It requests blocks of data from an appropriate server and interprets the data as file system structures (e.g., inodes and directories). The client is responsible for parsing pathnames, searching directories, looking up blocks of files, and so forth. To locate a server for a given file system, the client can use DNS round-robin scheduling or more advanced techniques such as consistent hashing [59]. Since the server is untrusted, the client must verify that any data received from the server was indeed signed by the database generator using the appropriate private key.

SFSRO provides integrity by way of a hash tree and digital signature. To verify that a particular block at a leaf node of the tree has integrity, the client hashes the block's content and verifies that the resulting hash matches the handle of the block (itself a hash). This process is recursive. The client hashes content up the tree, eventually reaching the root that has a digital signature.

The digital signature provides a link from the integrity of the file system to the authenticity of the publisher. The collision resistance property of the hash function ensures that no adversary can feasibly substitute false content. The hash tree also allows efficient operation. To verify integrity of a block, a client must only fetch a set of nodes on the directory path to block. The client does not have to download the entire database of blocks.

Our design can also in principle be used to provide non-repudiation of file system contents. An administrator of a server could commit to keeping every file system ever signed. Then, clients could just record the signed root handle. The server would be required to prove what the file system contained on any previous day. In this way, an administrator could never falsely deny that a file

```
struct FSINFO {
  sfs_time start;
  unsigned duration;
  opaque iv[16];
  sfs_hash rootfh;
  sfs_hash fhdb;
};
```

Figure 2-2: Contents of the digitally signed root of an SFS read-only file system.

previously existed.

## 2.3 SFS read-only protocol

The read-only protocol consists of one new RPC: *getdata*. The read-only server also implements two RPCs specified by the general SFS framework: *connect* and *getfsinfo*. *connect* names a particular service by way of a self-certifying pathname. This RPC allows a single server to multiplex several read-only file systems. *getfsinfo* takes no arguments and returns a digitally signed `FSINFO` structure, depicted in Figure 2-2. The SFS client verifies the signature using the public key embedded in the server's name. As long as the user received the key from a trusted source, the signature guarantees the integrity of the structure.

The *getdata* RPC takes a 20-byte argument and returns a data block whose collision-resistant cryptographic hash should match the 20-byte argument. The client uses *getdata* to retrieve parts of the file system requested by the user. It hashes every response to check it against the requested hash. The collision-resistant property of the hash function ensures that an attacker cannot construct a different data block with the same hash as a chunk of file system data. Thus, as long as the requested hash itself is authentic, the response will be, too.

Because read-only file systems reside on untrusted servers, the protocol relies on time to enforce consistency loosely but securely. The `start` field of `FSINFO` indicates the time (in seconds since 1970) at which a file system was signed. Clients cache the highest value they have seen to prevent an attacker from rolling back the file system to a previous state. The `duration` field

signifies the length of time for which the data structure should be considered valid. It represents a commitment on the part of a file system's owner to issue a signature within a certain period of time. Clients reject an `FSINFO` structure when the current time exceeds `start` + `duration`. An attacker who skews a client's clock may delay the time it takes for the client to see an update beyond the old version's expiration time. However, once a client has seen a particular version of the file system, it will never accept an older version with a lower `start` time.

The read-only file system references arbitrary-length blocks of data using fixed-size cryptographic hashes known as handles. The handle for a data item $x$ is computed using SHA1: $H(x) = \text{SHA1}(\texttt{iv}, x)$. `iv`, the initialization vector, is chosen randomly by the database generator the first time it publishes a file system. Currently the value chosen is just a hash of the file system's name and public key. The initialization vector ensures that simply knowing one particular collision of SHA1 will not immediately give attackers collisions of functions in use by SFS read-only file systems.

`rootfh` is the handle of the file system's root directory. It is a hash of the root directory's *inode* structure, which through recursive use of $H$ specifies the contents of the entire file system, as described below. `fhdb` is the hash of the root of a tree that contains every handle reachable from the root directory. `fhdb` lets clients securely verify that a particular handle does not exist, so that they can return stale file handle errors when file systems change. The latest version of SFSRO deprecates `fhdb` in favor of an internal file handle naming scheme described in Section 2.5.

## 2.4 SFS read-only data structures

Each data block a client retrieves from a server contains a file system data structure. The primary read-only data structure clients interpret is the read-only inode structure, which specifies the entire contents of a file. However, data blocks can also contain file or directory data, or index structures for large files.

The database stores these file system data structures in XDR marshaled form [112]. Using XDR has three advantages. First, it simplifies the client implementation, as the client can use the SFS RPC and crypto libraries to parse file system data. Second, the XDR representation clearly defines

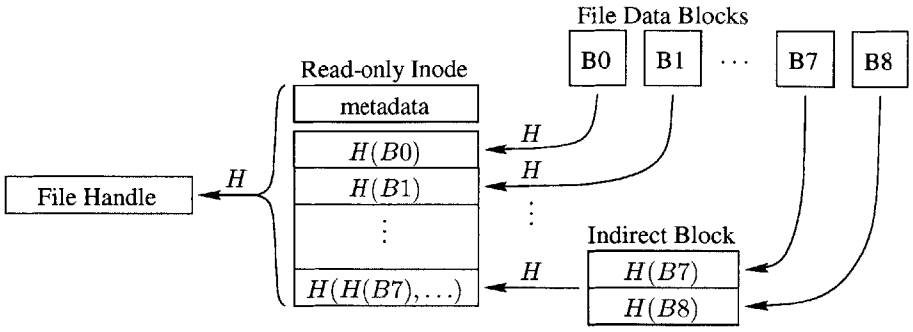


Figure 2-3: Format of a read-only file system inode.

what the database contains, which simplifies writing programs that process the database (e.g., a debugging program). Finally, it improves performance of the server by saving it from doing any marshaling—anything retrieved from the database can be directly transmitted to a client.

### 2.4.1 Read-only inode

Figure 2-3 shows the format of an inode in the read-only file system. The inode begins with some metadata, including the file's type (regular file, executable file, directory, opaque directory, or symbolic link), size, and modification time. Permissions are not included because they can be synthesized on the client. The inode then contains handles of successive 8 Kbyte blocks of file data. If the file contains more than eight blocks, the inode contains the handle of an *indirect block*, which in turn contains handles of file blocks. Similarly, for larger files, an inode can also contain the handles of double- and triple-indirect blocks. In this way, the blocks of small files can be verified directly from the inode, while inodes can also indirectly verify large files—an approach similar to the on-disk data structures of the Unix File System [93].

Inodes for symbolic links differ slightly from the depiction in Figure 2-3. Instead of containing handles of blocks, the inode directly contains the destination path of the symbolic link.

### 2.4.2 Directories

An SFS read-only directory is simply an inode of type directory or opaque directory. The inode specifies data blocks and possibly indirect blocks, just as for a regular file. However, the data blocks

of a directory have a fixed format known to the client. They consist of lists of ⟨name, handle⟩ pairs binding file names to the hashes of those files' inodes. Thus, the directory inode lets clients verify directory data blocks, and directory data blocks in turn let clients verify the inodes of files or subdirectories.

Directory entries are sorted lexicographically by name. Thus, clients can avoid traversing the entire directory by performing a binary search when looking up files in large directories. This property also allows clients to inexpensively verify whether a file name exists or not, without having to read the whole directory.

To avoid inconveniencing users with large directories, server administrators can set the type field in an inode to "opaque directory." When users list an opaque directory, they see only entries they have already referenced—somewhat like Unix "automounter" directories [23]. Opaque directories are well-suited to giant directories containing, for instance, all names in the `.com` domain or all name-to-key bindings issued by a particular certificate authority. If one used non-opaque directories for these applications, users could inadvertently download hundreds of megabytes of directory data by typing `ls` or using file name completion in the wrong directory.

## 2.5 File system updates

The SFS read-only file system allows a publisher to update the contents of a file system. When a file system changes, the administrator generates a new database and pushes it out to the untrusted servers replicating content. Files that persist across file system versions will keep the same handles. However, when a file is removed or modified, clients can end up requesting handles no longer in the database. In this case, the read-only server replies with an error.

Unfortunately, since read-only servers (and the network) are not trusted, clients cannot necessarily believe "handle not found" errors they receive. Though a compromised server can hang a client by refusing to answer RPCs, it must not be able to make programs spuriously abort with stale file handle errors. Otherwise, for instance, an application looking up a key revocation certificate in a read-only file system might falsely believe that the certificate did not exist.

We have two schemes to let clients securely determine whether a given file handle exists. The

old scheme uses the `fhdb` field of the `FSINFO` structure to verify that a handle no longer exists. The new scheme is based on the pathnames of files.

### 2.5.1 Using `fhdb` in updates

`fhdb` is the root of a hash tree, the leaf nodes of which contain a sorted list of every handle in the file system. Thus, clients can easily walk the hash tree (using *getdata*) to see whether the database contains a given file handle.

The `fhdb` scheme has advantages. It allows files to persist in the database even after they have been deleted, as not every handle in the database needs to be reachable from the root directory. Thus, by keeping handles of deleted files in a few subsequent revisions of a database, a system administrator can support the traditional Unix semantics that one can continue to access an open file even after it has been deleted.

Unfortunately, `fhdb` has several drawbacks. Even small changes to the file system cause most of the hash tree under `fhdb` to change—making incremental database updates unnecessarily expensive. Furthermore, there is no distinction between modifying a file and deleting then recreating it because handles are based on file contents in the read-only file system. In some situations, one does not want to have to close and reopen a file to see changes. (This is always the case for directories, which therefore need a different mechanism anyway.) Finally, under the `fhdb` scheme, a server cannot change its `iv` without causing all open files to become stale on all clients.

### 2.5.2 Pathname-based approach to updates

To avoid the problems associated with `fhdb`, the latest version of the software introduces a level of indirection between NFS and SFSRO file handles. In the new scheme, the client tracks the pathnames of all files accessed in read-only file systems. It chooses NFS file handles that are bound to the pathnames of files, rather than to hashes of the read-only inodes. When a server `FSINFO` structure is updated, the client walks the file namespace to find the new inode corresponding to the name of each open file.

Those who really want an open file never to change can still emulate the old semantics (albeit

somewhat inelegantly) by using a symbolic link to switch between the old and new version of a file while allowing both to exist simultaneously.

### 2.5.3 Incremental update and transfer

Untrusted servers can be updated without transferring the entire contents of the file system over the network. We built a simple utility program, `pulldb`, that incrementally transfers a newer version of a database from a primary server to a server. The program fetches `FSINFO` from the source server, and checks if the local copy of the database is out of date. If so, the program recursively traverses the entire file system, starting from the new root file handle, building on the side a list of all active handles. For each handle encountered, if the handle does not already exist in the local database, `pulldb` fetches the corresponding data with a *getdata* RPC and stores it in database. After the traversal, `pulldb` swaps the `FSINFO` structure in the database and then deletes all handles no longer in the file system. If a failure appears before the transfer is completed, the program can just be restarted, since the whole operation is idempotent.

The read-only inode structure contains the modification and "inode change" times of a file. Thus, `sfsrodb` could potentially update the database incrementally after changes are made to the file system, recomputing only the hashes from changed files up to the root handle and the signature on the `FSINFO` structure. Our current implementation of `sfsrodb` creates a completely new database for each version of the file system, but we plan to support incremental updates in a future release.

## 2.6 Discussion

**File system abstraction.** One of the benefits of the file system abstraction is the ease with which one can refer to the file namespace in almost any context—from shell scripts to C code to a Web browser's location field. Moreover, because of its security and scalability, SFSRO can support a wide range of applications, such as certificate authorities, that one could not ordinarily implement using a network file system.

**Optimizations.** The design of SFSRO opts for simplicity over performance. All data structures are stored in separate blocks in the database, for instance. Grouping inodes into a single block as does the UNIX Fast File System (FFS) [75], embedding inodes in directories [44], or using a B-tree instead of indirect data pointers would likely increase the throughput capabilities of servers.

**Hash collisions.** SFSRO relies on the security of a collision-resistant hash function. Information theory tells us that a hash function will not withstand an adversary with infinite computing power. However, realistic adversaries are computationally limited. Modern cryptography uses this notion to reduce the security of one system to the difficulty of a well-known, hard problem. In the case of SFSRO, we reduce the correctness and security to the difficulty of finding particular collisions in SHA1.

The research in this thesis was performed under the assumption that SHA1 was effectively "ideal." In particular, it was presumed that finding a collision for SHA1 would take time approximately $2^{80}$. Because of the recent work of Wang et al. [119, 120], SHA1 has become a disappointment; SHA1 collisions can be found in time $2^{69}$ (and believed to be only $2^{63}$ according to a CRYPTO 2005 rump session announcement of Wang et al.).

Nonetheless, the problems with SHA1 are likely repairable with reasonable cost. This thesis therefore leaves the references to SHA1 in this thesis untouched, and asks for the reader's indulgence to interpret a reference to SHA1 as a reference to a generic SHA1-like hash function with difficulty $2^{80}$ of finding collisions. Of course, the implementation timings are based on the use of the "standard" SHA1.

# Chapter 3

# Decentralized access control in Chefs

*Faite simple*

*(Make it simple)*

– Georges-Auguste Escoffier, legendary French chef

Chefs[1] is a read-only file system that adds access control to the SFSRO design. Because Chefs caters to private content, we use the term *member* to describe a client who is part of the access group.

As in SFSRO, the content publisher is the writer while the members are strictly readers. In addition to managing a database of content, the content publisher manages key distribution for access control. Chefs provides access control at the granularity of a database. Each database is associated with one group key, shared with all the group members. Possession of the group key enables a member to access content. Because the group of members is dynamic (members come and go), the group key changes after various membership events. Chefs uses key regression to provide efficient access control in such a dynamic environment.

The Chefs file system addresses three problems in secure content distribution.

[1]Hold a finger to your lips and breath through your teeth to properly pronounce the name of this file system. We leave the secret expansion of the acronym to the imagination of reader.

**Confidentiality.** Chefs must ensure that only those authorized members possessing the appropriate group key may access the protected content. Even the servers replicating the content may not have access to the protected content.

**Scalability to many members.** Chefs must preserve the scalability of SFSRO content distribution and also support scalable access control for large groups. Content publishers should be able to reach a large membership without needing a high-bandwidth network connection. The amount of direct communication between the content publisher and members should be independent of the amount of content and rate of membership turnover. Furthermore, each untrusted server should provide high throughput to support as many members as possible. An untrusted server should not have to perform any complex operations that would decrease throughput.

**Decentralized access control.** A content publisher must manage group membership, but may not directly mediate access to content. If the content publisher were involved in mediation, Chefs could not satisfy the scalability requirement. Thus, the process of mediating access to content must be decentralized and not involve online communication with the content publisher. For instance, group members should be able to access content even if the content publisher is offline or positioned behind a low-bandwidth network connection.

## 3.1 Security model

To protect a resource, traditional access control often uses a physical barrier, modeled as a guard. For instance, a guard may verify a principal's capability before granting access to an object. In Chefs, no physical barrier separates content from principals (here, called members). Instead, Chefs adopts a model of decentralized access control to protect content.

### 3.1.1 Decentralized access control.

One of the benefits of SFSRO is that content distribution scales independent of the publisher's resources. The publisher takes advantage of untrusted servers to make content highly available.

Clients may continue to fetch content from servers, even if the publisher is offline. For Chefs to provide the same scalable distribution of private content, it must also have scalable access control.

One approach for access control is for group members to contact the content publisher every time a member references content. This approach limits availability to the publisher's direct resources and violates our goal of scalability. A member would lose access to content if the content publisher is offline, for instance.

To achieve scalable access control, Chefs separates content publishing from mediation. Chefs could enlist the help of servers to mediate access. Alas, servers are untrusted and cannot properly mediate access to private content. A malicious server could simply ignore an access control list. Consequently, Chefs encrypts all content for confidentiality before distributing content to servers. Only those members with the proper key can decrypt the content. This form of storage protection is called *decentralized access control.*

Mediation in decentralized access control takes place in a client's local environment rather than on a server. If a client possesses the proper key, the client can decrypt and access the content. Decentralized access control has the same advantages and disadvantages as capabilities. The key can be delegated to others and revocation is difficult.

A number of secure storage systems have used semantics similar to decentralized access control. In particular, Gifford [46] calls the model of decentralized access control by using encryption "passive protection." Chapter 7 discusses such related work in more detail.

### 3.1.2 Lazy revocation

A relaxation of security semantics, *lazy revocation* posits that immediately protecting old content from an evicted member is not worth the computational trouble [43, 58]. Lazy revocation is an explicit way of noting that an eviction does not result in immediate revocation of access to content. A member is evicted, preventing access to future content. Yet the evicted member may continue to have access to already cached content. Lazy revocation weakens the security semantics of content protection in return for having efficient, incremental re-encryption of content.

Two events may cause a content publisher to re-encrypt content. A compromise of a key

would necessitate immediate re-encryption of all content. More often, re-encryption results from an eviction of a group member. Chefs was designed for this common case [67], while still allowing a content publisher to re-encrypt content as a result of events other than eviction.

Eviction prevents a former member from accessing *future* content. Eviction makes sense in a model where a content publisher has no control over a resource once granted. Revocation prevents a member from accessing both *future* and *past* content. Revocation results in the reversal of a privilege. Revocation makes sense in a model where a member must demonstrate authorization for every reference to a resource.

With decentralized access control, the content publisher cannot recall content already granted to a member. The member could easily cache a copy prior to revocation, and thus revocation is not a design option.

## 3.2 Keys

Two types of keys guard the confidentiality of content in Chefs:

**Group key.** After a membership event (e.g., an eviction), the content publisher distributes a new *group key*. The remaining group members request this new group key on-demand through a secure, out-of-band channel.

A group key protects content at the granularity of a whole database. Chefs does not support access to selected portions of a database. We imagine that a database will contain a collection of related documents, such as a directory tree of Web pages. Separate databases are required for content having different sets of members.

**Content key.** A group key does not directly encrypt content. Instead, Chefs uses a *content key* to encrypt content. A member obtains a content key by opening a lockbox that is encrypted with the group key. The separation of content keys from group keys helps to isolate the long-term effects of leaking secrets. The more a single key is used to encrypt content, the more secret information leaks.

```
struct sfsro_sealed {
  unsigned gk_vers;
  sfsro_lockboxtype lt;
  opaque lockbox<SFSRO_MAX_PRIVATE_KEYSIZE>;
  opaque pkcs7<>; /* Encryption of PKCS-7 encoded plaintext */
};
```

Figure 3-1: A data structure in Chefs to hold encrypted content. The PKCS#7-encoded plaintext contains a marshaled SFSRO structure.

Figure 3-1 shows the structure in Chefs for encrypted content. After marshaling an SFSRO structure, Chefs formats the content with PKCS#7 [57]. PKCS#7 provides secure padding of plaintext to ensure that the decryption process later returns an unambiguous plaintext. Chefs encrypts the PKCS#7-formatted content with a content key and stores the resulting ciphertext in the `pkcs7` field. The `lockbox` field serves as a level of indirection. The `lockbox` contains the content key encrypted with the group key. `gk_vers` indicates which version of the group key to use for encrypting and decrypting the lockbox. `lt` enables future expansion for alternative cryptosystems.

### 3.2.1 Key regression

Chefs uses key regression, described in Chapter 4, to reduce the amount of out-of-band communication necessary for group key distribution. Key regression conserves the publisher's network resources necessary to give group members continued access to content. After an eviction and modification of content, the content is re-encrypted with a new content key. With key regression, the new content key is related to the old content key. Given the new content key, it is easy to derive the old content key. Yet given the old content key, it is infeasible to derive the new content key.

In Chefs, a content publisher issues a new group key for encrypting new content after evicting a group member. Old, encrypted content remains unmodified. An evicted member may continue to have access to the unmodified content; immediately re-encrypting all data would have limited benefit, because the evicted member may have cached the content locally. A content publisher may choose to re-encrypt old content lazily, when convenient.

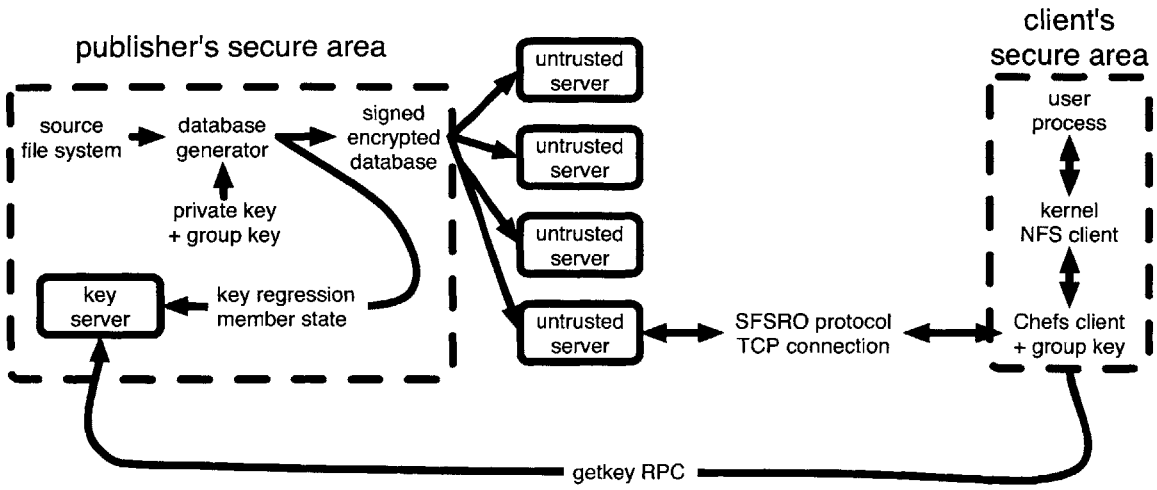


Figure 3-2: The Chefs file system. Dashed boxes indicate the trusted computing base.

## 3.3 Components of Chefs

Chefs consists of two major components. The SFS read-only file system provides the file system environment. Key regression generates new group keys as group members come and go. Figure 3-2 illustrates how components in the file system interact.

Chefs requires few changes to the base SFSRO system. In fact, the untrusted server is completely unaware that it is serving encrypted content. Only the database generator and client daemon require significant modification. Both programs use a shared, symmetric group key to enable confidentiality. The server requires one new remote procedure call, *getkey*, to enable a default key distribution mechanism.

### 3.3.1 Database generator

Chefs uses the same database generator program as SFSRO, but with a few additional features. First, the database generator takes two extra arguments: a self-certifying path to the publisher's key distriubtion server, and the version of the key to protect new content. For a new database, the database generator stores a new key regression publisher state in an auxiliary file.

When traversing the origin file system, the Chefs database generator encrypts content with a symmetric cipher, 128-bit AES in CBC mode. To store a block of content, the database generator first generates a random, 128-bit AES content key. After encrypting the content with this key, Chefs stores the content key in a *lockbox* that is itself encrypted with AES in CBC mode using the group key. The database generator now creates a structure consisting of three components: the encrypted lockbox, the encrypted content, and the version of the group key that encrypted the lockbox. The entire structure is hashed with SHA1 and an initialization vector to obtain the familiar SFSRO handle. Finally, the structure is stored in the database under the newly computed handle.

### 3.3.2 Client daemon

The Chefs client daemon works exactly as the SFSRO client daemon, except that it decrypts content after verifying integrity of the ciphertext.

To read a file, Chefs first mounts a remote Chefs file system by connecting to a server using a self-certifying path [73]. The server responds with a digitally signed `sfsro_private` structure as shown in Figure 3-3. Chefs verifies the digital signature on the `sfsro_sealed` structure. The `ct` field contains an encrypted, marshaled `FSINFO` structure as shown earlier by Figure 3-1. Chefs looks up the group key by the name given in the `gk_id` field of the `sfsro_sealed` structure. If the appropriate version of the group key is not available via key regression, Chefs contacts the publisher to securely download the latest key with the *getkey* RPC.

### 3.3.3 Server daemon

The content publisher controls the distribution of group keys that provide the decentralized access control. To evict a group member, the publisher winds the group key to the next version using key regression. Remaining group members contact the publisher on-demand for the latest group key using the *getkey* RPC. The publisher first verifies that the user is authorized to receive the key. If so, the publisher responds to a *getkey* RPC by encrypting the latest key regression member state with the requesting user's public RSA key. A user with the corresponding private RSA key may

```
struct sfsro_private {
  filename3 keymgr_sname;
  sfs_hash keymgr_hash;
  unsigned gk_id;
  sfsro_sealed ct;
};
```

Figure 3-3: An FSINFO data structure in Chefs to help clients mount a file system. `keymgr_sname` and `keymgr_hash` represent a self-certifying path to a publisher's key distribution service. `gk_id` is the version of the group key necessary to decrypt `ct` which itself contains an SFSRO-style FSINFO structure.

decrypt the response.

## 3.4 Updates

Chefs starts with the same basic algorithm as SFSRO to update a database. The update program traverses the origin file system and generates a new database. In addition, Chefs includes extra routines necessary to maintain lazy revocation and incremental distribution of updates.

To understand why Chefs has a more complicated task for updates than SFSRO, let us first recall how SFSRO updates a public database. The content publisher creates a new database of the updated origin file system. This process is not incremental. Rather, SFSRO servers provide incremental distribution of updates by downloading only the changes between the old and new databases. If a server encounters an unchanged handle, it halts the traversal. In this way, an update to a database requires a server to only download the structures on the path from the update to the root, resulting in a logarithmic number of content fetches with respect to the depth of the directory tree.

The database generator in SFSRO is deterministic. Regenerating a database of a particular origin file system results in the exact same SFSRO database. This is not the case in Chefs because the encrypting introduces non-determinism. Each content key is randomly generated. If a database were regenerated, the entire contents would require redistribution. Even if no changes occur, the old and new databases would be different.

The Chefs database generator includes extra logic to maintain the semantics of lazy revocation, thus ensuring incremental distribution of updates. When generating a database, Chefs traverses both the origin file system and the original, encrypted database. Instead of encrypting old content with new content keys, Chefs reuses the old content key for *unchanged* content. A new content key will encrypt new or changed content. The resulting database contains ciphertext that matches the previous database, except for changed content and the path from the content to the root.

Lazy revocation prevents evicted group members from accessing new content. However, an evicted member is not prevented from continued access to old, unchanged data. This relaxation in security semantics allows for efficient distribution of updates.

Content publishers must decide if the performance benefit of lazy revocation is worth the risk of less resistance to traffic analysis. Because of lazy revocation, an attacker can determine which content remains unchanged. A content publisher who wants better security against traffic analysis should disable lazy revocation.

## 3.5 Discussion

### 3.5.1 Intialization vectors

The Chefs prototype uses an initialization vector of zero with AES in CBC mode. Fortunately, each block has its own unique, random content key that contributes entropy to the encryption. The content key is never reused for encrypting new blocks. Alternatively, we could use the same content key for encrypting all blocks within a database and incorporate a random initialization vector per block. Another alternative would generate a random content key for the `FSINFO` but then generate all other content keys by using a pseudorandom function seeded with the content key of the `FSINFO`. Chefs uses a random content key for each block because of its simplicity of implementation.

### 3.5.2 Sliding window

Key regression ensures that the size of the encrypted content is independent of the number of evictions. Only one lockbox is necessary to allow members with various versions of the group key to access the content. To implement lazy revocation *without* key regression, encrypted content could include a list of all lockboxes. Each lockbox would contain the same content key, but each lockbox would be encrypted with a different version of the group key—one for each version of the group key since the last modification to the content.

The long list of lockboxes is a result of the content publisher not wanting to re-encrypt the content with a new content key. Instead of an indefinitely long list of lockboxes, one could use a sliding window of lockboxes [115]. The assumption is that re-encryption, while slow, can usually finish within a reasonable amount of time, and thus only a constant number of lockboxes will exist.

One side effect of this design is that a legitimate group member with a group key version older than the sliding window may unnecessarily communicate with the content publisher. Recall that one goal of Chefs is to reduce the amount of direct communication between the content publisher and members. In the sliding window method, group members may have to contact the content publisher for an updated group key even if the plaintext content remains unchanged, which is why Chefs does not use sliding windows.

# Chapter 4

# Key regression

*Definitions, when they are first proposed, will often encompass poor decisions or errors. For a good definition, these don't greatly diminish the value of the contribution.*

*Definitions can be worthwhile even in the absence of theorems and proofs.*

– Phillip Rogaway [97]

Key regression[1] enables a content publisher to efficiently share a contiguous sequence of group keys with a dynamic set of members. A publisher produces new group keys after a membership event (e.g., an eviction). Group members can independently compute old versions of the group key by unwinding the current group key.

To prevent key distribution from becoming a bottleneck, the Plutus file system [58] introduced a new cryptographic object called a *key rotation scheme*. Plutus uses the symmetric key $K_i$ to encrypt stored content during the $i$-th time period, e.g., before the $i$-th eviction. If a user becomes a member during the $i$-th time period, then Plutus gives that member the $i$-th key $K_i$.

[1] This chapter comes from the paper "Key regression: Enabling efficient key distribution for secure distributed storage" by Kevin Fu, Seny Kamara, Tadayoshi Kohno. Manuscript, August 2005.

From the Plutus paper [58], the desired properties of a key rotation scheme are that:

1. given the $i$-th key $K_i$ it is easy to compute the keys $K_j$ for all previous time periods $j < i$,

2. but for any time period $l > i$ after $i$, it should be computationally infeasible to compute the keys $K_l$ for time period $l$ given only $K_i$.

Property (1) enables the content publisher to transfer only a single small key $K_i$ to new members wishing to access all current and past content, rather than the potentially large set of keys $\{K_1, K_2, \ldots, K_i\}$; this property reduces the bandwidth requirements on the content publisher. Property (2) is intended to prevent a member evicted during the $i$-th time period from accessing (learning the contents of) content encrypted during the $l$-th time period, $l > i$.

We begin by presenting a design flaw with the definition of key rotation: for any realistic key rotation scheme, even though a member evicted during the $i$-th time period *cannot predict* subsequent keys $K_l$, $l > i$, the evicted member *can distinguish* subsequent keys $K_l$ from random. The lack of pseudorandomness follows from the fact that if an evicted member is given the real key $K_l$, then by definition (i.e., by property (2)) the evicted member can recover the real key $K_i$; but given a random key instead of $K_l$, the evicted member will with high probability recover a key $K_i' \neq K_i$.

The difference between unpredictability and lack of pseudorandomness can have severe consequences in practice. To illustrate the seriousness of this design flaw, we describe a key rotation scheme and a symmetric encryption scheme that individually meet their desired security properties (property (2) for key rotation and IND-CPA privacy for symmetric encryption [9]), but when combined (e.g, when a content publisher uses the keys from the key rotation scheme to key the symmetric encryption scheme) result in a system that fails to provide even a weak form of privacy.[2]

While the above counter example does not imply that all systems employing key rotation will fail just as drastically, it does motivate finding a key rotation-like object that still achieves the spirit

[2]We stress that the novelty here is in identifying the design flaw with key rotation, not in presenting a specific counter example. Indeed, the counter example follows naturally from our observation that a key rotation scheme does not produce pseudorandom keys.

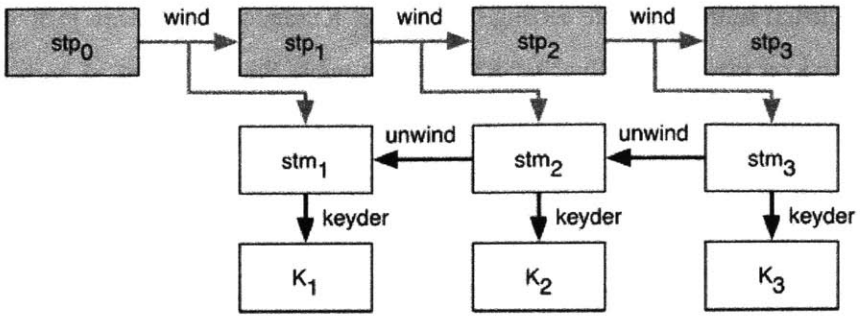


Figure 4-1: Key regression overview; $\mathsf{stp}_i$ and $\mathsf{stm}_i$ respectively represent the $i$-th publisher and member states.

---

of property (1) but (a new property 2′) produces future keys that are pseudorandom to evicted members — as opposed to just unpredictable. Assuming the new object achieves pseudorandomness, one could use it as a black box to key other cryptographic constructs without worrying about the resulting system failing as drastically as the one described above. A *key regression* scheme is such a key rotation-like object.

In key regression, rather than give a new member the $i$-th key $K_i$ directly, the content publisher gives the member a *member state* $\mathsf{stm}_i$. From the member state, the member could derive the encryption key $K_i$ for the $i$-th time period, as well as for all previous member states $\mathsf{stm}_j$, $j < i$. By transitivity, a member given the $i$-th member state could also derive all previous keys $K_j$. By separating the member states from the keys, we can build key regression schemes where the keys $K_l$, $l > i$, are pseudorandom to evicted members possessing only the $i$-th member state $\mathsf{stm}_i$. Intuitively, the trick used in the constructions to make the keys $K_l$ pseudorandom is to ensure that given both $K_l$ and $\mathsf{stm}_i$, it is still computationally infeasible for the evicted member to compute the $l$-th member state $\mathsf{stm}_l$. Viewed another way, there is no path from $K_l$ to $\mathsf{stm}_i$ in Figure 4-1 and vice-versa.

The newly constructed key regression schemes are referred to as KR-SHA1, KR-AES, and KR-RSA. Rather than rely solely on potentially error-prone heuristic methods for analyzing the security of our constructions, we prove that all three are secure key regression schemes. Our security proofs use the reduction-based provable security approach pioneered by Goldwasser and

Micali [50] and lifted to the concrete setting by Bellare, Kilian, and Rogaway [12]. For KR-RSA, our proof is based on the assumption that RSA is one-way. For the proof of both KR-RSA and KR-SHA1, we assume that SHA1 is a random oracle [10].

Section 4.1 explains the notation to describe key regression. Section 4.2 discusses problems with the Plutus-style [58] key rotation. Section 4.3 presents a definition of security for key regression, followed by several provably-secure constructions in Section 4.4.

## 4.1 Notation

If $x$ and $y$ are strings, then $|x|$ denotes the length of $x$ in bits and $x\|y$ denotes their concatenation. If $x$ and $y$ are two variables, we use $x \leftarrow y$ to denote the assignment of the value of $y$ to $x$. If $Y$ is a set, we denote the selection of a random element in $Y$ and its assignment to $x$ as $x \xleftarrow{R} Y$. If $f$ is a deterministic (respectively, randomized) function, then $x \leftarrow f(y)$ (respectively, $x \xleftarrow{R} f(y)$) denotes the process of running $f$ on input $y$ and assigning the result to $x$. We use the special symbol $\perp$ to denote an error.

We use $\text{AES}_K(M)$ to denote the process of running the AES block cipher with key $K$ and input block $M$. An RSA [95] key generator for some security parameter $k$ is a randomized algorithm $\mathcal{K}_{\text{rsa}}$ that returns a triple $(N, e, d)$. The modulus $N$ is the product of two distinct odd primes $p, q$ such that $2^{k-1} \leq N < 2^k$; the encryption exponent $e \in \mathbb{Z}^*_{\varphi(N)}$ and the decryption exponent $d \in \mathbb{Z}^*_{\varphi(N)}$ are such that $ed \equiv 1 \bmod \varphi(N)$, where $\varphi(N) = (p-1)(q-1)$. Section 4.4 describes what it means for an RSA key generator to be one-way.

## 4.2 Problems with key rotation

A key rotation scheme [58] consists of three algorithms: setup, windkey, and unwindkey. Figure 4-2 shows the original (RSA-based) Plutus key rotation scheme. Following Plutus, one familiar with hash chains [66] and S/KEY [52] might design the key rotation scheme in Figure 4-3, which is more efficient than the scheme in Figure 4-2, but which is limited because it can only produce MaxWind ("max wind") keys, where MaxWind is a parameter chosen by the implementor. A

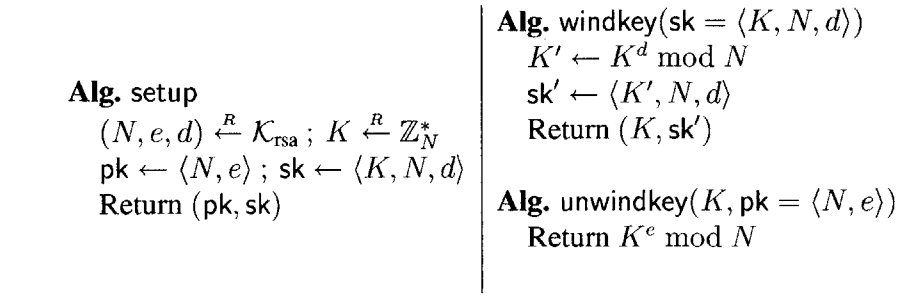


Figure 4-2: The Plutus key rotation scheme; $\mathcal{K}_{\text{rsa}}$ is an RSA key generator.

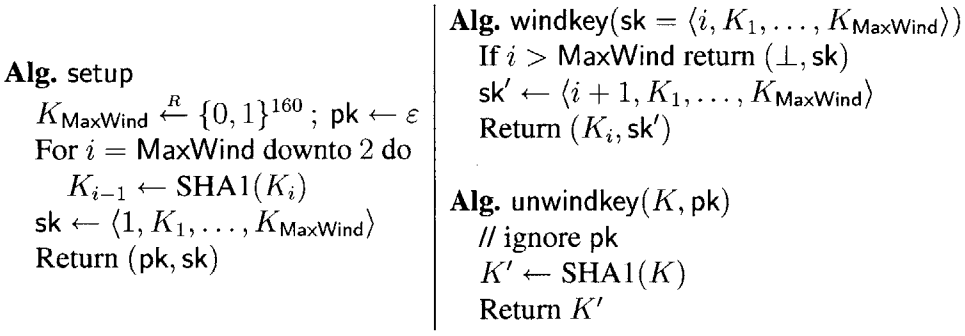


Figure 4-3: A hash chain-based key rotation scheme.

content publisher runs the setup algorithm to initialize a key rotation scheme; the result is public information pk for all users and a secret $\mathsf{sk}_1$ for the content publisher. The content publisher invokes windkey($\mathsf{sk}_i$) to obtain the key $K_i$ and a new secret $\mathsf{sk}_{i+1}$. Any user in possession of $K_i$, $i > 1$, and pk can invoke unwindkey($K_i$, pk) to obtain $K_{i-1}$. Informally, the desired security property of a key rotation scheme is that, given only $K_i$ and pk, it should be computationally infeasible for an evicted member (the adversary) to compute $K_l$, for any $l > i$. The Plutus construction in Figure 4-2 has this property under the RSA one-wayness assumption (defined in Section 4.4), and the construction in Figure 4-3 has this property assuming that SHA1 is one-way.

The scheme in Figure 4-3 is reasonable in practice, even though it has a limit on the number of windings. A publisher could precompute a chain long enough for most applications. For instance, our client in Table 6.4 can perform 687,720 unwindings per second. A user willing to spend a week (604,800 seconds) precomputing keys could produce a chain of length 415,933,056,000. We imagine most groups will not ever have to worry about exceeding 415,933,056,000 membership events.

In Section 1.1.2 we observed that the $l$-th key output by a key rotation scheme cannot be pseudorandom, i.e., will be distinguishable from a random string, to an evicted member in possession of the key $K_i$ for some previous time period $i < l$.[3] We consider the following example to emphasize how this lack of pseudorandomness might impact the security of a real system that combines a key rotation scheme and a symmetric encryption scheme as a black boxes.

For our example, we first present a key rotation scheme $\overline{\mathcal{WS}}$ and an encryption scheme $\overline{\mathcal{SE}}$ that individually both satisfy their respective security goals (unpredictability for the key rotation scheme and IND-CPA privacy [9] for the symmetric encryption scheme). To build $\overline{\mathcal{WS}}$, we start with a secure key rotation scheme $\mathcal{WS}$; $\overline{\mathcal{WS}}$ outputs keys twice as long as $\mathcal{WS}$. The $\overline{\mathcal{WS}}$ winding algorithm $\overline{\mathsf{windkey}}$ invokes $\mathcal{WS}$'s winding algorithm to obtain a key $K$; $\overline{\mathsf{windkey}}$ then returns $K\|K$ as its key. On input a key $K\|K$, $\overline{\mathsf{unwindkey}}$ invokes $\mathcal{WS}$'s unwinding algorithm with input $K$ to

[3]Technically, there may be pathological examples where the $l$-th key is pseudorandom to a member given the $i$-th key, but these examples seem to have other problems of their own. For example, consider a key rotation scheme like the one in Figure 4-3, but where SHA1 is replaced with a function mapping all inputs to some constant string $C$, e.g., the all 0 key. Now set MaxWind $= 2$, $i = 1$, and $l = 2$. In this pathological example $K_2$ is random to the evicted member, meaning (better than) pseudorandom. But this construction still lacks our desired pseudorandomness property: the key $K_1$ is always the constant string $C$.

obtain a key $K'$; $\overline{\mathsf{unwindkey}}$ then returns $K'\|K'$ as its key. If the keys output by windkey are unpredictable to evicted members, then so must the keys output by $\overline{\mathsf{windkey}}$. To build $\overline{\mathcal{SE}}$, we start with a secure symmetric encryption scheme $\mathcal{SE}$; $\overline{\mathcal{SE}}$ uses keys that are twice as long as $\mathcal{SE}$. The $\overline{\mathcal{SE}}$ encryption and decryption algorithms take the key $K$, split it into two halves $K = L_1\|L_2$, and run $\mathcal{SE}$ with key $L_1 \oplus L_2$. If the key $K$ is random, then the key $L_1 \oplus L_2$ is random and $\overline{\mathcal{SE}}$ runs the $\mathcal{SE}$ encryption algorithm with a uniformly selected random key. This means that $\overline{\mathcal{SE}}$ satisfies the standard IND-CPA security goal if $\mathcal{SE}$ does.

Despite the individual security of both $\overline{\mathcal{WS}}$ and $\overline{\mathcal{SE}}$, when the keys output by $\overline{\mathcal{WS}}$ are used to key $\overline{\mathcal{SE}}$, $\overline{\mathcal{SE}}$ will always run $\mathcal{SE}$ with the all-zero key; i.e., the content publisher will encrypt all content under the same constant key. An adversary can thus trivially compromise the privacy of all encrypted data, including data encrypted during time periods $l > i$ after being evicted. Although the construction of $\overline{\mathcal{WS}}$ and $\overline{\mathcal{SE}}$ may seem somewhat contrived, this example shows that combining a key rotation scheme and an encryption scheme may have undesirable consequences and, therefore, that it is not wise to use (even a secure) key rotation scheme as a black box to directly key other cryptographic objects.

Figure 4-1 might suggest an alternative approach for fixing the problems with key rotation. Instead of using the keys $K_i$ from a key rotation scheme to directly key other cryptographic objects, use a function of $K_i$, like $\mathrm{SHA1}(K_i)$, instead. If one models SHA1 as a random oracle and if the key rotation scheme produces unpredictable future keys $K_l$, then it might seem reasonable to conclude that an evicted member given $K_i$ should not be able to distinguish future values $\mathrm{SHA1}(K_l)$, $l > i$, from random. While this reasoning may be sound for some specific key rotation schemes (this reasoning actually serves as the basis for our derivative of the construction in Figure 4-2, KR-RSA in Construction 4.4.12) we dislike this approach for several reasons. First, we believe that it is unreasonable to assume that every engineer will know to or remember to use the hash function. Further, even if the engineer knew to hash the keys, the engineer might not realize that simply computing $\mathrm{SHA1}(K_l)$ may not work with all key rotation schemes, which means that the engineer cannot use a key rotation scheme as a black box. For example, while $\mathrm{SHA1}(K_l)$ would work for the scheme in Figure 4-2, it would cause problems for the scheme in Figure 4-3. We

choose to consider a new cryptographic object, key regression, because we desire a cryptographic object that is not as prone to accidental misuse.

## 4.3 Key regression

The negative result in Section 4.2 motivates our quest to find a new cryptographic object, similar to key rotation, but for which the keys generated at time periods $l > i$ are pseudorandom to any adversary evicted at time $i$. Here we formalize such an object: a key regression scheme. Following the reduction-based practice-oriented provable security approach [12, 50], our formalisms involve carefully defining the syntax, correctness requirements, and security goal of a key regression scheme. These formalisms enable us to, in Section 4.4, prove that our preferred constructions are secure under reasonable assumptions. We desire provable security over solely *ad hoc* analyses since, under *ad hoc* methods alone, one can never be completely convinced that a cryptographic construction is secure even if one assumes that the underlying components (e.g., block ciphers, hash functions, RSA) are secure.

Figure 4-1 gives an abstract overview of a key regression scheme. The content publisher has content publisher states $\mathsf{stp}_i$ from which it derives future publisher and member states. When using a key regression scheme, instead of giving a new member the $i$-th key $K_i$, the content publisher would give the member the $i$-th member state $\mathsf{stm}_i$. As the arrows in Figure 4-1 suggest, given $\mathsf{stm}_i$, a member can efficiently compute all previous member states and the keys $K_1, \ldots, K_i$. Although it would be possible for an evicted member to distinguish future member states $\mathsf{stm}_l$, $l > i$, from random (the evicted member would extend our observation on the lack of pseudorandomness in key rotation schemes), because there is no efficient path between the future keys $K_l$ and the evicted member's last member state $\mathsf{stm}_i$, it is possible for a key regression scheme to produce future keys $K_l$ that are pseudorandom (indistinguishable from random). We present some such constructions in Section 4.4.

### 4.3.1 Syntax and correctness requirements

**Syntax.** Here we formally define the syntax of a key regression scheme $\mathcal{KR} = (\mathsf{setup}, \mathsf{wind}, \mathsf{unwind}, \mathsf{keyder})$. Let $H$ be a random oracle; all four algorithms are given access to the random oracle, though they may not use the random oracle in their computations. Via $\mathsf{stp} \xleftarrow{R} \mathsf{setup}^H$, the randomized setup algorithm returns a publisher state. Via $(\mathsf{stp}', \mathsf{stm}) \xleftarrow{R} \mathsf{wind}^H(\mathsf{stp})$, the randomized winding algorithm takes a publisher state stp and returns a pair of publisher and member states or the error code $(\perp, \perp)$. Via $\mathsf{stm}' \leftarrow \mathsf{unwind}^H(\mathsf{stm})$ the deterministic unwinding algorithm takes a member state stm and returns a member state or the error code $\perp$. Via $K \xleftarrow{R} \mathsf{keyder}^H(\mathsf{stm})$ the deterministic key derivation algorithm takes a member state stm and returns a key $K \in \mathsf{DerKeys}_{\mathcal{KR}}$, where $\mathsf{DerKeys}_{\mathcal{KR}}$ is the *derived key space* for $\mathcal{KR}$. Let $\mathsf{MaxWind} \in \{1, 2, \ldots\} \cup \{\infty\}$ denote the maximum number of derived keys that $\mathcal{KR}$ is designed to produce. We do not define the behavior of the algorithms when input the error code $\perp$. A construction may use multiple random oracles, but since one can always obtain multiple random oracles from a single random oracle [10], our definitions assume just one.

**Correctness.** Our first correctness criterion for a key regression scheme is that the first MaxWind times that wind is invoked, it always outputs valid member states, i.e., the outputs are never $\perp$. Our second correctness requirement ensures that if $\mathsf{stm}_i$ is the $i$-th member state output by wind, and if $i > 1$, then from $\mathsf{stm}_i$, one can derive all previous member states $\mathsf{stm}_j$, $0 < j < i$. Formally, let $\mathsf{stp}_0 \xleftarrow{R} \mathsf{setup}$ and, for $i = 1, 2, \ldots$, let $(\mathsf{stp}_i, \mathsf{stm}_i) \xleftarrow{R} \mathsf{wind}^H(\mathsf{stp}_{i-1})$. We require that for each $i \in \{1, 2, \ldots, \mathsf{MaxWind}\}$, that $\mathsf{stm}_i \neq \perp$ and that, for $i \geq 2$, $\mathsf{unwind}^H(\mathsf{stm}_i) = \mathsf{stm}_{i-1}$.

**Remarks on syntax.** Although we allow wind to be randomized, the wind algorithms in all of our constructions are deterministic. We allow wind to return $(\perp, \perp)$ since we only require that wind return non-error states for its first MaxWind invocations. We use the pair $(\perp, \perp)$, rather than simply $\perp$, to denote an error from wind since doing so makes our pseudocode cleaner. We allow unwind to return $\perp$ since the behavior of unwind may be undefined when input the first member state.

### 4.3.2 Security goal

For security, we desire that if a member (adversary) is evicted during the $i$-th time period, then the adversary will not be able to distinguish the keys derived from any subsequent member state $\mathsf{stm}_l$, $l > i$, from randomly selected keys. Definition 4.3.1 captures this goal as follows. We allow the adversary to obtain as many member states as it wishes (via a WindO oracle). Note that the WindO oracle returns only a member state rather than both a member and publisher state. Once the adversary is evicted, its goal is to break the pseudorandomness of subsequently derived keys. To model this, we allow the adversary to query a key derivation oracle KeyderO. The key derivation oracle will either return real derived keys (via internal calls to wind and keyder) or random keys. The adversary's goal is to guess whether the KeyderO oracle's responses are real derived keys or random keys. Since the publisher is in charge of winding and will not invoke the winding algorithm more than the prescribed maximum number of times, MaxWind, the WindO and KeyderO oracles in our security definition will respond only to the first MaxWind queries from the adversary.

**Definition 4.3.1 [Security for key regression schemes.]** Let $\mathcal{KR} = (\mathsf{setup}, \mathsf{wind}, \mathsf{unwind}, \mathsf{keyder})$ be a key regression scheme. Let $\mathcal{A}$ be an adversary. Consider the experiments $\mathbf{Exp}^{\text{kr-}b}_{\mathcal{KR},\mathcal{A}}$, $b \in \{0,1\}$, and the oracles WindO and KeyderO in Figure 4-4. The oracles WindO and KeyderO and the experiment $\mathbf{Exp}^{\text{kr-}b}_{\mathcal{KR},\mathcal{A}}$ share global variables $i$ (an integer) and stp (publisher state). The adversary runs in two stages, member and non-member, and returns a bit. The *KR-advantage* of $\mathcal{A}$ in breaking the security of $\mathcal{KR}$ is defined as

$$\mathbf{Adv}^{\text{kr}}_{\mathcal{KR},\mathcal{A}} = \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1\right] - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1\right].$$

Note that $\mathbf{Adv}^{\text{kr}}_{\mathcal{KR},\mathcal{A}}$ could be negative. But for every adversary with a negative advantage, there is one adversary (obtained by switching outputs) that has the corresponding positive advantage.

Under the concrete security approach [12], we say that the scheme $\mathcal{KR}$ is *KR-secure* if for any adversary $\mathcal{A}$ attacking $\mathcal{KR}$ with resources (running time, size of code, number of oracle queries) limited to "practical" amounts, the KR-advantage of $\mathcal{A}$ is "small." Formal results are stated with concrete bounds. ▮

**Experiment** $\mathbf{Exp}^{\text{kr-}b}_{\mathcal{KR},\mathcal{A}}$
  Pick random oracle $H$
  $i \leftarrow 0$
  $\mathsf{stp} \xleftarrow{R} \mathsf{setup}^H$
  $\mathsf{st} \xleftarrow{R} \mathcal{A}^{\mathsf{WindO},H}(\mathsf{member})$
  $b' \xleftarrow{R} \mathcal{A}^{\mathsf{KeyderO}_b,H}(\mathsf{non\text{-}member}, \mathsf{st})$
  Return $b'$

**Oracle** WindO
  $i \leftarrow i + 1$
  If $i >$ MaxWind then
    return $\perp$
  $(\mathsf{stp}, \mathsf{stm}) \xleftarrow{R} \mathsf{wind}^H(\mathsf{stp})$
  Return stm

**Oracle** $\mathsf{KeyderO}_b$
  $i \leftarrow i + 1$
  If $i >$ MaxWind then return $\perp$
  $(\mathsf{stp}, \mathsf{stm}) \xleftarrow{R} \mathsf{wind}^H(\mathsf{stp})$
  If $b = 1$ then
    $K \leftarrow \mathsf{keyder}^H(\mathsf{stm})$
  If $b = 0$ then
    $K \xleftarrow{R} \mathsf{DerKeys}_{\mathcal{KR}}$
  Return $K$

Figure 4-4: The experiment and oracles used in the definition of security for key regression.

| | KR-SHA1 | KR-AES | KR-RSA |
|---|---|---|---|
| MaxWind $= \infty$ | No | No | Yes |
| setup cost | MaxWind SHA1 ops | MaxWind AES ops | 1 RSA key generation |
| wind cost | no crypto | no crypto | 1 RSA decryption |
| unwind cost | 1 SHA1 op | 1 AES op | 1 RSA encryption |
| keyder cost | 1 SHA1 op | 1 AES op | 1 SHA1 op |

Table 4.1: Our preferred constructions. There are ways of implementing these constructions with different wind costs.

## 4.4 Constructions

We are now in a position to describe our three preferred key regression schemes, KR-SHA1, KR-AES, and KR-RSA. Table 4.1 summarizes some of their main properties. KR-SHA1 is a derivative of the key rotation scheme in Figure 4-3 and KR-RSA is a derivative of the Plutus key rotation scheme in Figure 4-2. The primary differences between the new key regression schemes and the original key rotation schemes are the addition of the new, SHA1-based keyder algorithms, and the adjusting of terminology (e.g., member states in these key regression schemes correspond to keys in the original key rotation schemes).

We begin by defining KR-SHA1. In the construction of KR-SHA1, we prepend the string $0^8$ to the input to SHA1 in keyder to ensure that the inputs to SHA1 never collide between the keyder and unwind algorithms.

### 4.4.1 Hash-based key regression

**Construction 4.4.1** [KR-SHA1] The key regression scheme KR-SHA1 = (setup, wind, unwind, keyder) is defined in Figure 4-5. MaxWind is a positive integer and a parameter of the construction. There are no shared global variables in KR-SHA1. The derived key space for KR-SHA1 is $\mathsf{DerKeys}_{\mathcal{KR}} = \{0,1\}^{160}$. ∎

The following theorem states that KR-SHA1 is secure in the random oracle model for adversaries that make a reasonable number of queries to their random oracles. Here we view the application of $\mathrm{SHA1}(\cdot)$ in unwind as one random oracle and the application of $\mathrm{SHA1}(0^8\|\cdot)$ in keyder as another random oracle. The proof of Theorem 4.4.2 is in the random oracle model [10].

**Theorem 4.4.2** *Let $\mathcal{KR}$ be a generalization of* KR-SHA1 *(Construction 4.4.1) in which $SHA1(\cdot)$ in* unwind *is replaced by a random oracle $H_1$: $\{0,1\}^{160} \to \{0,1\}^{160}$ and in which $SHA1(0^8\|\cdot)$ in* keyder *is replaced by another random oracle $H_2$: $\{0,1\}^{160} \to \{0,1\}^{160}$. Then $\mathcal{KR}$ is KR-secure in the random oracle model. Concretely, for any adversary $\mathcal{A}$ we have*

$$\mathbf{Adv}^{\mathrm{kr}}_{\mathcal{KR},\mathcal{A}} \leq \frac{(\mathsf{MaxWind})^2}{2^{k+1}} + \frac{q \cdot \mathsf{MaxWind}}{2^k - \mathsf{MaxWind} - q},$$

**Alg.** setup
  $\mathsf{stm}_{\mathsf{MaxWind}} \xleftarrow{R} \{0,1\}^{160}$
  For $i = \mathsf{MaxWind}$ downto 2 do
    $\mathsf{stm}_{i-1} \leftarrow \mathsf{unwind}(\mathsf{stm}_i)$
  $\mathsf{stp} \leftarrow \langle 1, \mathsf{stm}_1, \ldots, \mathsf{stm}_{\mathsf{MaxWind}} \rangle$
  Return stp

**Alg.** unwind(stm)
  $\mathsf{stm}' \leftarrow \mathrm{SHA1}(\mathsf{stm})$
  Return stm'

**Alg.** wind(stp)
  If $\mathsf{stp} = \perp$ then return $(\perp, \perp)$
  Parse stp as $\langle i, \mathsf{stm}_1, \ldots, \mathsf{stm}_{\mathsf{MaxWind}} \rangle$
  If $i > \mathsf{MaxWind}$ return $(\perp, \perp)$
  $\mathsf{stp}' \leftarrow \langle i+1, \mathsf{stm}_1, \ldots, \mathsf{stm}_{\mathsf{MaxWind}} \rangle$
  Return $(\mathsf{stp}', \mathsf{stm}_i)$

**Alg.** keyder(stm)
  $\mathsf{out} \leftarrow \mathrm{SHA1}(0^8 \| \mathsf{stm})$
  Return out

Figure 4-5: The algorithms of KR-SHA1.

*where $q$ is the maximum number of queries that $\mathcal{A}$ makes to its random oracles.* ∎

**Proof of Theorem 4.4.2:** Theorem 4.4.2 follows immediately from Theorem 4.4.4 in the next section since the latter makes a more general statement. ∎

### 4.4.2 Generalization of KR-SHA1 and proof of Theorem 4.4.2.

Construction 4.4.3 below shows a generalization of KR-SHA1 in which $\mathrm{SHA1}(\cdot)$ and $\mathrm{SHA1}(0^8\|\cdot)$ are respectively replaced by two random oracles, $H_1$ and $H_2$. For Construction 4.4.3, in order for setup and wind to be "efficient," we assume that MaxWind has some "reasonable" value like $2^{20}$; in the asymptotic setting we would require that MaxWind be polynomial in some security parameter. Besides KR-SHA1, one can envision a number of other natural instantiations of Construction 4.4.3.

**Construction 4.4.3** Let $H_1$: $\{0,1\}^k \rightarrow \{0,1\}^k$ and $H_2$: $\{0,1\}^k \rightarrow \{0,1\}^l$ be random oracles. Figure 4-6 shows how to construct a key regression scheme $\mathcal{KR} = (\mathsf{setup}, \mathsf{wind}, \mathsf{unwind}, \mathsf{keyder})$ from $H_1$ and $H_2$; MaxWind is a positive integer and a parameter of the construction. The derived key space for $\mathcal{KR}$ is $\mathsf{DerKeys}_{\mathcal{KR}} = \{0,1\}^l$. ∎

The following theorem states that Construction 4.4.3 is secure in the random oracle model for adversaries that make a reasonable number of queries to their random oracles.

**Alg.** $\mathsf{setup}^{H_1,H_2}$
$\mathsf{stm}_{\mathsf{MaxWind}} \xleftarrow{R} \{0,1\}^k$
For $i = \mathsf{MaxWind}$ downto 2 do
$\mathsf{stm}_{i-1} \leftarrow \mathsf{unwind}^{H_1,H_2}(\mathsf{stm}_i)$
$\mathsf{stp} \leftarrow \langle 1, \mathsf{stm}_1, \ldots, \mathsf{stm}_{\mathsf{MaxWind}} \rangle$
Return stp

**Alg.** $\mathsf{unwind}^{H_1,H_2}(\mathsf{stm})$
$\mathsf{stm}' \leftarrow H_1(\mathsf{stm})$
Return stm′

**Alg.** $\mathsf{wind}^{H_1,H_2}(\mathsf{stp})$
If $\mathsf{stp} = \perp$ then return $(\perp, \perp)$
Parse stp as
$\langle i, \mathsf{stm}_1, \ldots, \mathsf{stm}_{\mathsf{MaxWind}} \rangle$
If $i > \mathsf{MaxWind}$ return $(\perp, \perp)$
$\mathsf{stp}' \leftarrow \langle i+1, \mathsf{stm}_1, \ldots, \mathsf{stm}_{\mathsf{MaxWind}} \rangle$
Return $(\mathsf{stp}', \mathsf{stm}_i)$

**Alg.** $\mathsf{keyder}^{H_1,H_2}(\mathsf{stm})$
$\mathsf{out} \leftarrow H_2(\mathsf{stm})$
Return out

Figure 4-6: Hash chains-based algorithms for Construction 4.4.3. $H_1$ and $H_2$ are random oracles. The setup algorithm uses the unwind algorithm defined in the second column.

---

**Theorem 4.4.4** *The key regression scheme in Construction 4.4.3 is secure in the random oracle model. Formally, let* $H_1$: $\{0,1\}^k \to \{0,1\}^k$ *and* $H_2$: $\{0,1\}^k \to \{0,1\}^l$ *be random oracles and let* $\mathcal{KR}$ *be the key regression scheme built from* $H_1, H_2$ *via Construction 4.4.3. Then for any adversary* $\mathcal{A}$ *we have that*

$$\mathbf{Adv}^{\mathrm{kr}}_{\mathcal{KR},\mathcal{A}} \leq \frac{(\mathsf{MaxWind})^2}{2^{k+1}} + \frac{q \cdot \mathsf{MaxWind}}{2^k - \mathsf{MaxWind} - q},$$

*where* $q$ *is the maximum number of queries total that adversary* $\mathcal{A}$ *makes to its* $H_1$ *and* $H_2$ *random oracles.* ∎

**Proof of Theorem 4.4.4:** Consider the experiments $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ and $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}}$. Let $\mathsf{stm}_1, \mathsf{stm}_2, \ldots, \mathsf{stm}_{\mathsf{MaxWind}}$ denote the member states as computed by setup, and let $w'$ denote the variable number of WindO oracle queries that $\mathcal{A}$ made in its member stage. Let $\mathcal{E}_1$ be the event in $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ that $w' \leq \mathsf{MaxWind} - 1$ and that $\mathcal{A}$ queries either its $H_1$ or $H_2$ random oracles with some string $x \in \{\mathsf{stm}_{w'+1}, \ldots, \mathsf{stm}_{\mathsf{MaxWind}}\}$. Let $\mathcal{E}_0$ be the event in $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}}$ that $w' \leq \mathsf{MaxWind} - 1$ and that $\mathcal{A}$ queries either its $H_1$ or $H_2$ random oracles with some string $x \in \{\mathsf{stm}_{w'+1}, \ldots, \mathsf{stm}_{\mathsf{MaxWind}}\}$. Let $\mathcal{F}_1$ be the event in $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ that there exist two distinct indices $i, j \in \{1, \ldots, \mathsf{MaxWind}\}$ such that $\mathsf{stm}_i = \mathsf{stm}_j$ and let $\mathcal{F}_0$ be the event in $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}}$ that there exist two distinct indices $i, j \in \{1, \ldots, \mathsf{MaxWind}\}$ such that $\mathsf{stm}_i = \mathsf{stm}_j$.

We claim that

$$\mathbf{Adv}^{\mathrm{kr}}_{\mathcal{KR},\mathcal{A}} \leq \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{F}_1\right] + \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_1 \wedge \overline{\mathcal{F}_1}\right], \tag{4.1}$$

that

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{F}_1\right] \leq \frac{(\mathsf{MaxWind})^2}{2^{k+1}}, \tag{4.2}$$

and that

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_1 \wedge \overline{\mathcal{F}_1}\right] \leq \frac{q \cdot \mathsf{MaxWind}}{2^k - \mathsf{MaxWind} - q}, \tag{4.3}$$

from which the inequality in the theorem statement follows.

To justify Equation (4.1), let $\Pr_1[\cdot]$ and $\Pr_0[\cdot]$ denote the probabilities over $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ and $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}}$, respectively. From Definition 4.3.1, we have

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{kr}}_{\mathcal{KR},\mathcal{A}} &= \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1\right] - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1\right] \\
&= \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{F}_1\right] + \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_1 \wedge \overline{\mathcal{F}_1}\right] \\
&\quad + \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_1} \wedge \overline{\mathcal{F}_1}\right] - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{F}_0\right] \\
&\quad - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_0 \wedge \overline{\mathcal{F}_0}\right] - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_0} \wedge \overline{\mathcal{F}_0}\right] \\
&\leq \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{F}_1\right] + \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_1 \wedge \overline{\mathcal{F}_1}\right] \\
&\quad + \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_1} \wedge \overline{\mathcal{F}_1}\right] - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_0} \wedge \overline{\mathcal{F}_0}\right]. \qquad (4.4)
\end{aligned}$$

By conditioning,

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_1} \wedge \overline{\mathcal{F}_1}\right] = \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \mid \overline{\mathcal{E}_1} \wedge \overline{\mathcal{F}_1}\right] \cdot \Pr_1\left[\overline{\mathcal{E}_1} \wedge \overline{\mathcal{F}_1}\right]$$

and

$$\Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_0} \wedge \overline{\mathcal{F}_0}\right] = \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \mid \overline{\mathcal{E}_0} \wedge \overline{\mathcal{F}_0}\right] \cdot \Pr_0\left[\overline{\mathcal{E}_0} \wedge \overline{\mathcal{F}_0}\right].$$

Prior to the adversary causing the events $\mathcal{E}_1 \vee \mathcal{F}_1$ and $\mathcal{E}_0 \vee \mathcal{F}_0$ to occur in their respective experiments, $\mathcal{A}$'s view is identical in both experiments, meaning that

$$\Pr_1\left[\overline{\mathcal{E}_1} \wedge \overline{\mathcal{F}_1}\right] = \Pr_0\left[\overline{\mathcal{E}_0} \wedge \overline{\mathcal{F}_0}\right].$$

Similarly, if the events do not occur, then the outcome of $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ and $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}}$ will be the same since the output of a random oracle is random if the input is unknown; i.e., the response to $\mathcal{A}$'s key derivation oracle query in the non-member stage will be random in both cases and therefore

$$\Pr\left[\,\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \mid \overline{\mathcal{E}_1} \wedge \overline{\mathcal{F}_1}\,\right] \;=\; \Pr\left[\,\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \mid \overline{\mathcal{E}_0} \wedge \overline{\mathcal{F}_0}\,\right] .$$

Consequently,

$$\Pr\left[\,\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_1} \wedge \overline{\mathcal{F}_1}\,\right] \;=\; \Pr\left[\,\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_0} \wedge \overline{\mathcal{F}_0}\,\right] .$$

Combining the above equation with Equation (4.4) gives Equation (4.1).

Returning to Equation (4.2), we first note that

$$\Pr\left[\,\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{F}_1\,\right] \;\leq\; \Pr_1\left[\,\mathcal{F}_1\,\right] .$$

If we consider the event $\mathcal{F}_1$, we note that the setup algorithm selects the points $\mathsf{stm}_{\mathsf{MaxWind}}$, $\mathsf{stm}_{\mathsf{MaxWind}-1}$, $\mathsf{stm}_{\mathsf{MaxWind}-2}$, and so on, uniformly at random from $\{0,1\}^k$ until a collision occurs. Since this is exactly the standard birthday paradox [12], we can upper bound $\Pr_1\left[\,\mathcal{F}_1\,\right]$ as

$$\Pr_1\left[\,\mathcal{F}_1\,\right] \;\leq\; \frac{(\mathsf{MaxWind})^2}{2^{k+1}} .$$

Equation (4.2) follows.

To justify Equation (4.3), we begin by noting that

$$\Pr\left[\,\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_1 \wedge \overline{\mathcal{F}_1}\,\right] \;\leq\; \Pr_1\left[\,\mathcal{E}_1 \mid \overline{\mathcal{F}_1}\,\right] \cdot \Pr_1\left[\,\overline{\mathcal{F}_1}\,\right] \;\leq\; \Pr_1\left[\,\mathcal{E}_1 \mid \overline{\mathcal{F}_1}\,\right] .$$

Consider the adversary $\mathcal{A}$ in $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ and assume that $\mathcal{F}_1$ does not occur. Consider any snapshot of the entire state of $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ before $\mathcal{A}$ causes $\mathcal{E}_1$ to occur, and let $q'$ denote the number of $H_1$ and $H_2$ oracle queries that $\mathcal{A}$ has made prior to the snapshot being taken. Then the member states $\mathsf{stm}_{w'+1}, \ldots, \mathsf{stm}_{\mathsf{MaxWind}}$ are restricted only in that they are distinct strings from $\{0,1\}^k$ and that none of the strings are from $\{\mathsf{stm}_1, \ldots, \mathsf{stm}_{w'}\}$ or the set of $\mathcal{A}$'s $q'$ queries to its random oracles; i.e., the member states that $\mathcal{A}$ obtained in its member stage and the responses from the KeyderO oracle do not reveal additional information to the adversary. This means that if the adversary's next oracle query after this snapshot is to one of its random oracles, and if that input for that oracle query is some string $x$, then the probability that $x \in \{\mathsf{stm}_{w'+1}, \ldots, \mathsf{stm}_{\mathsf{MaxWind}}\}$, i.e., the probability

**Alg.** setup
$\mathsf{stm}_{\mathsf{MaxWind}} \xleftarrow{R} \{0,1\}^{128}$
For $i = \mathsf{MaxWind}$ downto 2 do
$\mathsf{stm}_{i-1} \leftarrow \mathsf{unwind}(\mathsf{stm}_i)$
$\mathsf{stp} \leftarrow \langle 1, \mathsf{stm}_1, \ldots, \mathsf{stm}_{\mathsf{MaxWind}} \rangle$
Return stp

**Alg.** unwind(stm)
$\mathsf{stm}' \leftarrow \mathrm{AES}_{\mathsf{stm}}(0^{128})$
Return stm'

**Alg.** wind(stp)
If $\mathsf{stp} = \perp$ then return $(\perp, \perp)$
Parse stp as $\langle i, \mathsf{stm}_1, \ldots, \mathsf{stm}_{\mathsf{MaxWind}} \rangle$
If $i > \mathsf{MaxWind}$ return $(\perp, \perp)$
$\mathsf{stp}' \leftarrow \langle i+1, \mathsf{stm}_1, \ldots, \mathsf{stm}_{\mathsf{MaxWind}} \rangle$
Return $(\mathsf{stp}', \mathsf{stm}_i)$

**Alg.** keyder(stm)
$\mathsf{out} \leftarrow \mathrm{AES}_{\mathsf{stm}}(1^{128})$
Return out

Figure 4-7: AES-based algorithms for key regression.

that $\mathcal{A}$'s oracle query would cause $\mathcal{E}_1$ to occur, is at most $(\mathsf{MaxWind} - w')/(2^k - (w' + q')) \leq \mathsf{MaxWind}/(2^k - \mathsf{MaxWind} - q')$. Summing over all of $\mathcal{A}$'s $q$ random oracle queries and taking an upper bound, we have

$$\Pr_1 \left[ \mathcal{E}_1 \mid \overline{\mathcal{F}_1} \right] \leq \frac{q \cdot \mathsf{MaxWind}}{2^k - \mathsf{MaxWind} - q},$$

which completes the proof. ∎

### 4.4.3 AES-based key regression

Our next hash-based construction, KR-AES, uses AES in a construction similar to KR-SHA1. One can view KR-AES as using two hash functions:

$$H_1(x) = \mathrm{AES}_x(0^{128})$$
$$H_2(x) = \mathrm{AES}_x(1^{128})$$

Using the hash function input as a block cipher key produces a one-way function for fixed plaintext [33, 37]. The following construction is four times faster at unwinding than KR-SHA1 (see Table 6.4).

| **Algorithm** unwind(stm) | **Algorithm** keyder(stm) |
|---|---|
| $x \leftarrow G(\mathsf{stm})$ | $x \leftarrow G(\mathsf{stm})$ |
| $\mathsf{stm}' \leftarrow$ first $k$ bits of $x$ | out $\leftarrow$ last $l$ bits of $x$ |
| Return $\mathsf{stm}'$ | Return out |

Figure 4-8: The unwind and keyder algorithms for Construction 4.4.6. $G$: $\{0,1\}^k \rightarrow \{0,1\}^{k+l}$ is a function. The setup and wind algorithms are as in Figure 4-6 except that setup and wind do not receive access to any random oracles.

---

**Construction 4.4.5** [KR-AES] The key regression scheme KR-AES = (setup, wind, unwind, keyder) is defined in Figure 4-7. MaxWind is a positive integer and a parameter of the construction. There are no shared global variables in KR-AES. The derived key space for KR-AES is $\mathsf{DerKeys}_{\mathcal{KR}} = \{0,1\}^{128}$. ∎

### 4.4.4 Key regression using forward-secure pseudorandom bit generators

Construction 4.4.6 below generalizes KR-AES, and is essentially one of Bellare and Yee's [17] forward secure PRGs in reverse. Construction 4.4.6 uses a pseudorandom bit generator, which is a function $G$: $\{0,1\}^k \rightarrow \{0,1\}^{k+l}$ that takes as input a $k$-bit seed and returns a string that is longer than the seed by $l$ bits, $k, l \geq 1$. Pseudorandom bit generators were defined first in [21] and lifted to the concrete setting in [32]. As with Construction 4.4.3, in order for setup and wind to be "efficient," we assume that MaxWind has some "reasonable" value like $2^{20}$; in the asymptotic setting we would require that MaxWind be polynomial in some security parameter. To instantiate KR-AES from Construction 4.4.6, we set $k = l = 128$ and, for any $X \in \{0,1\}^{128}$, we define $G$ as $G(X) = \mathrm{AES}_X(0^{128}) \| \mathrm{AES}_X(1^{128})$. Numerous other instantiations exist. The security proof for Construction 4.4.6 is in the standard, as opposed to the random oracle, model.

**Construction 4.4.6** Let $G$: $\{0,1\}^k \rightarrow \{0,1\}^{k+l}$ be a pseudorandom bit generator. Figure 4-8 shows how to construct a key regression scheme $\mathcal{KR}$ = (setup, wind, unwind, keyder) from $G$; MaxWind is a positive integer and a parameter of the construction. The derived key space for the scheme $\mathcal{KR}$ is $\mathsf{DerKeys}_{\mathcal{KR}} = \{0,1\}^l$. ∎

Toward proving the security of Construction 4.4.6, we begin by defining our security assumptions on the base PRG [17, 21, 126]. If $\mathcal{A}$ is an adversary, we let

$$\mathbf{Adv}^{\mathrm{prg}}_{F,\mathcal{A}} \;=\; \Pr\left[\, K \xleftarrow{R} \{0,1\}^k \,;\, x \leftarrow G(K) \;:\; \mathcal{A}(x) = 1 \,\right] - \Pr\left[\, x \xleftarrow{R} \{0,1\}^{k+l} \;:\; \mathcal{A}(x) = 1 \,\right]$$

denote the *prg-advantage* of $\mathcal{A}$ in attacking $G$. Under the concrete security approach [12], there is no formal definition of what it means for $G$ to be a "secure PRG," but in discussions this phrase should be taken to mean that, for any $\mathcal{A}$ attacking $G$ with resources (running time, size of code) limited to "practical" amounts, the prg-advantage of $\mathcal{A}$ is "small." Formal results are stated with concrete bounds.

**Theorem 4.4.7** *If* $G\colon\ \{0,1\}^k \to \{0,1\}^{k+l}$ *is a secure PRG, then the key regression scheme* $\mathcal{KR}$ *built from* $G$ *via Construction 4.4.6 is KR-secure. Concretely, given an adversary* $\mathcal{A}$ *attacking* $\mathcal{KR}$*, we can construct an adversary* $\mathcal{B}$ *attacking* $G$ *such that*

$$\mathbf{Adv}^{\mathrm{kr}}_{\mathcal{KR},\mathcal{A}} \;\leq\; 2 \cdot (q+1)^2 \cdot \mathbf{Adv}^{\mathrm{prg}}_{G,\mathcal{B}}$$

*where* $q$ *is the minimum of* MaxWind *and the maximum number of queries* $\mathcal{A}$ *makes to its* WindO *and* KeyderO *oracles. Adversary* $\mathcal{B}$ *uses within a small constant factor of the resources of* $\mathcal{A}$*, plus the time to compute* setup *and* $G$ MaxWind *times.* ∎

For our proof of Theorem 4.4.7, we remark that the internal structure of the member states and derived keys in Construction 4.4.6 is very similar to the internal structure of the states and output bits in a forward-secure pseudorandom bit generator, as defined in [17] and recalled below. Our proof therefore proceeds first by showing how to build a secure key regression scheme from any forward-secure pseudorandom bit generator, essentially by running the forward-secure pseudorandom bit generator in reverse during the key regression scheme's setup algorithm (Construction 4.4.8). This intermediate result suggests that future work in forward-secure pseudorandom bit generators could have useful applications to key regression schemes. To prove Theorem 4.4.7, we then combine this intermediate result with a lemma in [17] that shows how to create a forward-secure pseudorandom bit generator from a conventional pseudorandom bit generator.

| **Algorithm** seed | **Algorithm** next($\mathsf{stg}_i$) |
|---|---|
| $\mathsf{stg}_0 \xleftarrow{R} \{0,1\}^k$ | $r \xleftarrow{R} G(\mathsf{stg}_i)$ |
| return $\mathsf{stg}_0$ | $\mathsf{stg}_{i+1} \leftarrow$ first $k$ bits of $r$ |
| | out $\leftarrow$ last $l$ bits of $r$ |
| | return $(\mathsf{stg}_{i+1}, \mathsf{out})$ |

Figure 4-9: Algorithms for Construction 4.4.10.

---

**Forward-secure pseudorandom generators.** In [17], Bellare and Yee define stateful pseudorandom bit generators and describe what it means for a stateful pseudorandom bit generator to be forward-secure. Intuitively a stateful PRG is forward-secure if even adversaries that are given the generator's current state cannot distinguish previous outputs from random.

SYNTAX. A stateful PRG consists of two algorithms: $\mathcal{SBG} = (\mathsf{seed}, \mathsf{next})$ as shown in Figure 4-9. The randomized setup algorithm returns an initial state; we write this as $\mathsf{stg} \xleftarrow{R} \mathsf{seed}$. The deterministic next step algorithm takes a state as input and returns a new state and an output from $\mathsf{OutSp}_{\mathcal{SBG}}$, or the pair $(\perp, \perp)$; we write this as $(\mathsf{stg}', \mathsf{out}) \leftarrow \mathsf{next}(\mathsf{stg})$. We require that the set $\mathsf{OutSp}_{\mathcal{SBG}}$ is efficiently samplable. $\mathsf{MaxLen}_{\mathcal{SBG}} \in \{1, 2, \ldots\} \cup \{\infty\}$ denotes the maximum number of output blocks that $\mathcal{SBG}$ is designed to produce.

CORRECTNESS. The correctness requirement for stateful PRGs is as follows: let $\mathsf{stg}_0 \xleftarrow{R} \mathsf{seed}$ and, for $i = 1, 2, \ldots$, let $(\mathsf{stg}_i, \mathsf{out}_i) \xleftarrow{R} \mathsf{next}(\mathsf{stg}_{i-1})$. We require that for $i \leq \mathsf{MaxLen}_{\mathcal{SBG}}$, $(\mathsf{stg}_i, \mathsf{out}_i) \neq (\perp, \perp)$.

SECURITY. Let $\mathcal{SBG} = (\mathsf{seed}, \mathsf{next})$ be a stateful bit generator. Let $\mathcal{A}$ be an adversary. Consider the experiments $\mathbf{Exp}^{\text{fsprg-}b}_{\mathcal{SBG},\mathcal{A}}$, $b \in \{0,1\}$, and the oracle NextO below with shared global variable $stg$. The adversary runs in two stages: find and guess.

| **Experiment** $\mathbf{Exp}^{\text{fsprg-}b}_{\mathcal{SBG},\mathcal{A}}$ | **Oracle** $\mathsf{NextO}_b$ |
|---|---|
| $\mathsf{stg} \xleftarrow{R} \mathsf{seed}$ | $(\mathsf{stg}, \mathsf{out}) \leftarrow \mathsf{next}(\mathsf{stg})$ |
| $\mathsf{st} \xleftarrow{R} \mathcal{A}^{\mathsf{NextO}_b}(\mathsf{find})$ | If $b = 0$ then $\mathsf{out} \xleftarrow{R} \mathsf{OutSp}_{\mathcal{SBG}}$ |
| $b' \xleftarrow{R} \mathcal{A}(\mathsf{guess}, \mathsf{stg}, \mathsf{st})$ | Return out |
| Return $b'$ | |

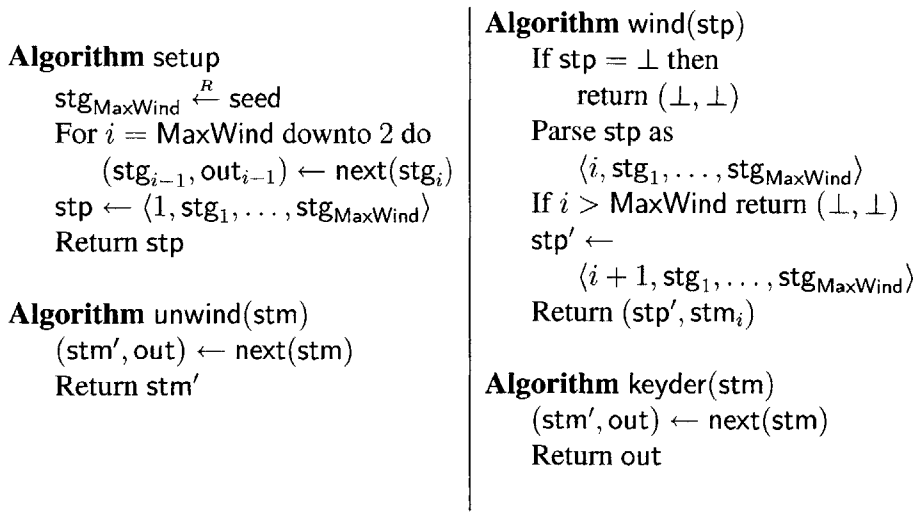


Figure 4-10: Algorithms for KR-SBG (Construction 4.4.8). Construction KR-SBG demonstrates how to build a KR-secure key regression scheme from any FSPRG-secure stateful bit generator $\mathcal{SBG} = (\mathsf{seed}, \mathsf{next})$. There are no shared global variables in KR-SBG.

---

The *FSPRG-advantage* of $\mathcal{A}$ in breaking the security of $\mathcal{SBG}$ is defined as

$$\mathbf{Adv}^{\text{fsprg}}_{\mathcal{SBG},\mathcal{A}} = \Pr\left[\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{A}} = 1\right] - \Pr\left[\mathbf{Exp}^{\text{fsprg-0}}_{\mathcal{SBG},\mathcal{A}} = 1\right].$$

Note that $\mathbf{Adv}^{\text{fsprg}}_{\mathcal{SBG},\mathcal{A}}$ could be negative. But for every adversary with a negative advantage, there is one adversary (obtained by switching outputs) that has the corresponding positive advantage.

Under the concrete security approach, the scheme $\mathcal{SBG}$ is said to be *FSPRG-secure* if the FSPRG-advantage of all adversaries $\mathcal{A}$ using reasonable resources is "small."

**Key regression from forward-secure pseudorandom bit generators.**

**Construction 4.4.8** [KR-SBG] Given a stateful generator $\mathcal{SBG} = (\mathsf{seed}, \mathsf{next})$, we can construct a key regression scheme $\mathsf{KR\text{-}SBG} = (\mathsf{setup}, \mathsf{wind}, \mathsf{unwind}, \mathsf{keyder})$ as follows. For the construction, we set MaxWind to a positive integer at most $\mathsf{MaxLen}_{\mathcal{SBG}}$; MaxWind is a parameter of our construction. The derived key space for KR-SBG is $\mathsf{DerKeys}_{\mathcal{KR}} = \mathsf{OutSp}_{\mathcal{SBG}}$. The algorithms for KR-SBG are shown in Figure 4-10. ∎

**Lemma 4.4.9** If $\mathcal{SBG}$ is FSPRG-secure, then $\mathcal{KR}$ built from $\mathcal{SBG}$ via Construction KR-SBG is KR-secure. Concretely, given an adversary $\mathcal{A}$ attacking $\mathcal{KR}$, we can construct an adversary $\mathcal{B}$ attacking $\mathcal{SBG}$ such that

$$\mathbf{Adv}^{\text{kr}}_{\mathcal{KR},\mathcal{A}} \leq (q+1) \cdot \mathbf{Adv}^{\text{fsprg}}_{\mathcal{SBG},\mathcal{B}}$$

where $q$ is the minimum of MaxWind and the maximum number of wind and key derivation oracle queries that $\mathcal{A}$ makes. Adversary $\mathcal{B}$ makes up to MaxWind queries to its oracle and uses within a small constant factor of the other resources of $\mathcal{A}$ plus the time to run the setup algorithm. ∎

**Proof of Lemma 4.4.9:** The adversary $\mathcal{B}$ is shown in Figure 4-11. The main idea is that if $\mathcal{B}$ correctly guesses the number of WindO queries that $\mathcal{A}$ will make, then $\mathcal{B}$'s simulation is perfect for either choice of bit $b$. If $\mathcal{B}$ does not correctly guess the bit the number of WindO oracle queries, then it always returns 0, regardless of the value of the bit $b$. We restrict $q$ to the minimum of MaxWind and the maximum number of wind and key derivation oracle queries that $\mathcal{A}$ makes since wind is defined to return $(\perp, \perp)$ after MaxWind invocations.

Formally, we claim that

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1\right] = (q+1) \cdot \Pr\left[\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}} = 1\right] \tag{4.5}$$

$$\Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1\right] = (q+1) \cdot \Pr\left[\mathbf{Exp}^{\text{fsprg-0}}_{\mathcal{SBG},\mathcal{A}} = 1\right], \tag{4.6}$$

from which it follows that

$$\begin{aligned}
\mathbf{Adv}^{\text{kr}}_{\mathcal{KR},\mathcal{A}} &= \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1\right] - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1\right] \\
&= (q+1) \cdot \left(\Pr\left[\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}} = 1\right] - \Pr\left[\mathbf{Exp}^{\text{fsprg-0}}_{\mathcal{SBG},\mathcal{A}} = 1\right]\right) \\
&\leq (q+1) \cdot \mathbf{Adv}^{\text{fsprg}}_{\mathcal{SBG},\mathcal{B}}
\end{aligned}$$

as desired.

It remains to justify Equation (4.5), Equation (4.6), and the resources of $\mathcal{B}$. Let $\mathcal{E}_1$ and $\mathcal{E}_0$ respectively denote the events that $\mathcal{B}$ sets bad to true in the experiments $\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}}$ and $\mathbf{Exp}^{\text{fsprg-0}}_{\mathcal{SBG},\mathcal{A}}$, i.e.,

**Adversary** $\mathcal{B}^{\mathsf{NextO}_b}(\mathsf{find})$
$q' \xleftarrow{R} \{0, 1, \ldots, q\}$
For $i = \mathsf{MaxWind}$ downto $q' + 1$ do
$\mathsf{out}_{i-1} \leftarrow \mathsf{NextO}_b$
Return $\langle q', \mathsf{out}_{q'}, \ldots, \mathsf{out}_{\mathsf{MaxWind}-1} \rangle$

**Adversary** $\mathcal{B}(\mathsf{guess}, \mathsf{stg}, \mathsf{st})$
Parse st as $\langle q', \mathsf{out}_{q'}, \ldots, \mathsf{out}_{\mathsf{MaxWind}-1} \rangle$
$\mathsf{stg}_{q'} \leftarrow \mathsf{stg}$
For $i = q'$ downto 2 do
$(\mathsf{stg}_{i-1}, \mathsf{out}_{i-1}) \leftarrow \mathsf{next}(\mathsf{stg}_i)$
$i \leftarrow 0$
$\mathsf{bad} \leftarrow \mathsf{false}$
$\mathsf{st}_{\mathcal{A}} \xleftarrow{R} \mathcal{A}^{\mathsf{SimWindO}}(\mathsf{member})$
If $i \neq q'$ then $\mathsf{bad} \leftarrow \mathsf{true}$
If $\mathsf{bad} = \mathsf{true}$ then return 0
$b' \xleftarrow{R} \mathcal{A}^{\mathsf{SimKeyderO}}(\mathsf{non\text{-}member}, \mathsf{st}_{\mathcal{A}})$
Return $b'$

**Oracle** SimWindO
If $i \geq q'$ then $\mathsf{bad} \leftarrow \mathsf{true}$
If $i \geq \mathsf{MaxWind}$ or $\mathsf{bad} = \mathsf{true}$
then return $\perp$
Else $i \leftarrow i + 1$
return $\mathsf{stg}_i$

**Oracle** SimKeyderO
If $i \geq \mathsf{MaxWind}$ then return $\perp$
$i \leftarrow i + 1$
Return $\mathsf{out}_{i-1}$

Figure 4-11: The adversary $\mathcal{B}$ in the proof of Theorem 4.4.9. Note that bad is a shared boolean global variable.

when $\mathcal{B}$ fails to correctly guess the number of wind oracle queries that $\mathcal{A}$ makes. Let $\Pr_1[\cdot]$ and $\Pr_0[\cdot]$ respectively denote probabilities over $\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}}$ and $\mathbf{Exp}^{\text{fsprg-0}}_{\mathcal{SBG},\mathcal{A}}$. We now claim that

$$\begin{aligned}
&\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1\right] && (4.7)\\
&\quad = \Pr\left[\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}} = 1 \mid \overline{\mathcal{E}_1}\right] && (4.8)\\
&\quad = \Pr\left[\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}} = 1 \wedge \overline{\mathcal{E}_1}\right] \cdot \frac{1}{\Pr_1\left[\overline{\mathcal{E}_1}\right]} && (4.9)\\
&\quad = (q+1) \cdot \Pr\left[\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}} = 1 \wedge \overline{\mathcal{E}_1}\right] && (4.10)\\
&\quad = (q+1) \cdot \left(\Pr\left[\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}} = 1 \wedge \overline{\mathcal{E}_1}\right] + \Pr\left[\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}} = 1 \wedge \mathcal{E}_1\right]\right) && (4.11)\\
&\quad = (q+1) \cdot \Pr\left[\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}} = 1\right].
\end{aligned}$$

Equation (4.8) is true because when the event $\mathcal{E}_1$ does not occur, i.e., when $\mathcal{B}$ correctly guesses the number of wind oracle queries that $\mathcal{A}$ will make, then $\mathcal{B}$ in $\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}}$ runs $\mathcal{A}$ exactly as $\mathcal{A}$ would be run in $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$. Equation (4.9) follows from conditioning off $\Pr_1\left[\overline{\mathcal{E}_1}\right]$ and Equation (4.10) is true because $\mathcal{B}$ chooses $q'$ from $q+1$ possible values and therefore $\Pr_1\left[\overline{\mathcal{E}_1}\right] = 1/(q+1)$. To justify Equation (4.11), note that $\Pr\left[\mathbf{Exp}^{\text{fsprg-1}}_{\mathcal{SBG},\mathcal{B}} = 1 \wedge \mathcal{E}_1\right] = 0$ since $\mathcal{B}$ always returns 0 whenever it fails to correctly guess the number of wind oracle queries that $\mathcal{A}$ will make. This justifies Equation (4.5).

To justify Equation (4.6), note that

$$\Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1\right] = \Pr\left[\mathbf{Exp}^{\text{fsprg-0}}_{\mathcal{SBG},\mathcal{B}} = 1 \mid \overline{\mathcal{E}_0}\right]$$

since when the event $\mathcal{E}_0$ does not occur, $\mathcal{B}$ in $\mathbf{Exp}^{\text{fsprg-0}}_{\mathcal{SBG},\mathcal{B}}$ runs $\mathcal{A}$ exactly as $\mathcal{A}$ would be run in $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}}$. The remaining justification for Equation (4.6) is analogous to our justification of Equation (4.5) above.

The resources for $\mathcal{B}$ is within a small constant factor of the resources for $\mathcal{A}$ except that $\mathcal{B}$ must execute the setup algorithm itself, which involves querying its oracle up to MaxWind times. ∎

**Forward-secure pseudorandom bit generators from standard PRGs.**

**Construction 4.4.10 [Construction 2.2 of [17].]** Given a PRG $G : \{0,1\}^k \to \{0,1\}^{k+l}$ we can construct a FSPRG $\mathcal{SBG} = (\text{seed}, \text{next})$ as described in Figure 4-9. The output space of $\mathcal{SBG}$ is $\mathsf{OutSp}_{\mathcal{SBG}} = \{0,1\}^l$ and $\mathsf{MaxLen}_{\mathcal{SBG}} = \infty$. ∎

The following theorem comes from Bellare and Yee [17] except that we treat $q$ as a parameter of the adversary and we allow the trivial case that $q = 0$.

**Lemma 4.4.11 [Theorem 2.3 of [17].]** Let $G : \{0,1\}^k \to \{0,1\}^{k+l}$ be a PRG, and let $\mathcal{SBG}$ be the FSPRG built using $G$ according to Construction 4.4.10. Given an adversary $\mathcal{A}$ attacking $\mathcal{SBG}$ that makes at most $q$ queries to its oracle, we can construct an adversary $\mathcal{B}$ such that

$$\mathbf{Adv}^{\text{fsprg}}_{\mathcal{SBG},A} \leq 2q \cdot \mathbf{Adv}^{\text{prg}}_{G,B}$$

where $\mathcal{B}$ uses within a small constant factor of the resources of adversary $\mathcal{A}$ and computes $G$ up to $q$ times. ∎

**Proof of Theorem 4.4.7. Proof:** Construction 4.4.6 is exactly Construction 4.4.8 built from the forward secure pseudorandom bit generator defined by Construction 4.4.10. The theorem statement therefore follows from Lemma 4.4.9 and Lemma 4.4.11. ∎

### 4.4.5 RSA-based key regression

Our final construction, KR-RSA derives from the key rotation scheme in Figure 4-2; KR-RSA differs from KR-SHA1 and KR-AES in that MaxWind $= \infty$, meaning that there is no specific limit to the number of times the content publisher can invoke the KR-RSA winding algorithm. Given the finite number of outputs, the member states produced by winding will eventually cycle. Fortunately, the difficulty of finding a cycle is usually as hard as factoring the RSA modulus for most choices of RSA parameters [42]. Since factoring is considered hard, the number of times one can safely run the KR-RSA winding algorithm is proportional to the strength of the underlying RSA construction — practically infinite for a reasonably sized modulus (today, at least 1,024 bits).

**Alg.** $\mathsf{setup}^H(k)$
$(N, e, d) \xleftarrow{R} \mathcal{K}_{\text{rsa}}(k)$
$S \xleftarrow{R} \mathbb{Z}_N^*$
$\mathsf{stp} \leftarrow \langle N, e, d, S \rangle$
Return stp

**Alg.** $\mathsf{unwind}^H(\mathsf{stm})$
Parse stm as $\langle N, e, S \rangle$
$S' \leftarrow S^e \bmod N$
$\mathsf{stm}' \leftarrow \langle N, e, S' \rangle$
Return $\mathsf{stm}'$

**Alg.** $\mathsf{wind}^H(\mathsf{stp})$
Parse stp as $\langle N, e, d, S \rangle$
$S' \leftarrow S^d \bmod N$
$\mathsf{stp}' \leftarrow \langle N, e, d, S' \rangle$
$\mathsf{stm} \leftarrow \langle N, e, S \rangle$
Return $(\mathsf{stp}', \mathsf{stm})$

**Alg.** $\mathsf{keyder}^H(\mathsf{stm})$
Parse stm as $\langle N, e, S \rangle$
$\mathsf{out} \leftarrow H(S)$
Return out

Figure 4-12: Algorithms for Construction 4.4.12. $H$ is a random oracle, but we substitute SHA1 for $H$ in our implementation.

---

**Construction 4.4.12** [KR-RSA] The key regression scheme KR-RSA $=$ (setup, wind, unwind, keyder) is defined as follows. Given an RSA key generator $\mathcal{K}_{\text{rsa}}$ for some security parameter $k$, and a random oracle $H$: $\mathbb{Z}_{2^k} \to \{0,1\}^l$, Figure 4-12 shows how to construct a key regression scheme $\mathcal{KR} =$ (setup, wind, unwind, keyder). Let $m$: $\mathbb{Z}_{2^k} \to \{0,1\}^k$ denote the standard big-endian encoding of the integers in $\mathbb{Z}_{2^k}$ to $k$-bit strings. There are no shared global variables in KR-RSA. The derived key space for KR-RSA is $\mathsf{DerKeys}_{\mathcal{KR}} = \{0,1\}^l$. In our experiments, we set $l = 160$, $k =$1,024, and $\mathcal{K}_{\text{rsa}}(k)$ returns $e = 3$ as the RSA public exponent. ∎

Before presenting Theorem 4.4.13, we first recall the standard notion of one-wayness for RSA. Let $\mathcal{K}_{\text{rsa}}$ be an RSA key generator with security parameter $k$. If $\mathcal{A}$ is an adversary, we let

$$\mathbf{Adv}^{\text{rsa-ow}}_{\mathcal{K}_{\text{rsa}},\mathcal{A}} = \Pr\left[\, (N,e,d) \xleftarrow{R} \mathcal{K}_{\text{rsa}}(k)\ ;\ x \xleftarrow{R} \mathbb{Z}_N^*\ ;\ y \leftarrow x^e \bmod N \ :\ \mathcal{A}(y,e,N) = x \,\right]$$

denote the RSA one-way advantage of $\mathcal{A}$ in inverting RSA with the key generator $\mathcal{K}_{\text{rsa}}$. Under the concrete security approach [12], there is no formal definition of what it means for $\mathcal{K}_{\text{rsa}}$ to be "one-way." In discussions this phrase should be taken to mean that for any $\mathcal{A}$ attacking $\mathcal{K}_{\text{rsa}}$ with resources (running time, size of code, number of oracle queries) limited to "practical" amounts, the RSA one-way advantage of $\mathcal{A}$ is "small." The formal result below is stated with concrete bounds. The proof is in the random oracle model.

**Theorem 4.4.13** *If $\mathcal{K}_{rsa}$ is an RSA key generator with security parameter $k$, then* KR-RSA *is KR-secure under the RSA one-wayness assumption. Concretely, given an adversary $\mathcal{A}$ attacking* KR-RSA, *we can construct an adversary $\mathcal{B}$ attacking $\mathcal{K}_{rsa}$ such that*

$$\mathbf{Adv}^{\mathrm{kr}}_{\mathcal{KR},\mathcal{A}} \leq 2q^2 \cdot \mathbf{Adv}^{\text{rsa-ow}}_{\mathcal{K}_{rsa},\mathcal{B}} ,$$

*where $q$ is the maximum number of winding and key derivation oracle queries that $\mathcal{A}$ makes. Adversary $\mathcal{B}$ uses resources within a constant factor of $\mathcal{A}$'s resources plus the time to perform $q$ RSA encryption operations.* ∎

The proof of Theorem 4.4.13 uses the two following Lemmas.

**Lemma 4.4.14** If a key regression scheme is secure when an adversary is limited to one KeyderO oracle query, then the key regression scheme is secure when an adversary is allowed multiple KeyderO oracle queries. Concretely, let $\mathcal{KR}$ be a key regression scheme. Given an adversary $\mathcal{A}$ attacking $\mathcal{KR}$ that makes at most $q_1$ queries to WindO and $q_2$ queries to KeyderO, we can construct an adversary $\mathcal{B}$ attacking $\mathcal{KR}$ such that

$$\mathbf{Adv}^{\mathrm{kr}}_{\mathcal{KR},\mathcal{A}} \leq q_2 \cdot \mathbf{Adv}^{\mathrm{kr}}_{\mathcal{KR},\mathcal{B}} , \tag{4.12}$$

$\mathcal{B}$ makes at most $q_1 + q_2 - 1$ queries to WindO (or 0 queries if $q_1 + q_2 = 0$), $\mathcal{B}$ makes at most one query to KeyderO, and $\mathcal{B}$ has other resource requirements within a small constant factor of the resource requirements of $\mathcal{A}$. ∎

**Lemma 4.4.15** If $\mathcal{K}_{\mathrm{rsa}}$ is an RSA key generator with security parameter $k$, then the key regression scheme $\mathcal{KR}$ built from $\mathcal{K}_{\mathrm{rsa}}$ via Construction 4.4.12 is KR-secure assuming that $\mathcal{K}_{\mathrm{rsa}}$ is one-way. Concretely, given an adversary $\mathcal{A}$ attacking $\mathcal{KR}$ that makes at most one key derivation oracle query, we can construct an adversary $\mathcal{B}$ attacking $\mathcal{K}_{\mathrm{rsa}}$ such that

$$\mathbf{Adv}^{\mathrm{kr}}_{\mathcal{KR},\mathcal{A}} \leq (q+1) \cdot \mathbf{Adv}^{\text{rsa-ow}}_{\mathcal{K}_{\mathrm{rsa}},\mathcal{B}} , \tag{4.13}$$

where $q$ is the maximum number of winding oracle queries that $\mathcal{A}$ makes. Adversary $\mathcal{B}$ uses within a small constant factor of the resources as $\mathcal{A}$ plus performs up to $q$ RSA encryption operations. ∎

**Proof of Theorem 4.4.13:** The proof of Theorem 4.4.13 follows from Lemma 4.4.14 and Lemma 4.4.15. Note that for the application of Lemma 4.4.14 we set $q_1 = q$ and $q_2 = q$, meaning the adversary $\mathcal{B}$

from Lemma 4.4.14 may make up to $2q - 1$ queries to its WindO oracle, or $2q$ if $q = 0$. ∎

**Proof of Lemma 4.4.14.** **Proof:** We consider the case where $q_2 = 0$ separately. If $q_2 = 0$ then

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1\right] = \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1\right]$$

since the adversary $\mathcal{A}$'s view in the experiments $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ and $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}}$ is identical. Therefore, when $q_2 = 0$,

$$\begin{aligned}
\mathbf{Adv}^{\text{kr}}_{\mathcal{KR},\mathcal{A}} &= \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1\right] - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1\right] \\
&= 0 \\
&= q_2 \cdot \mathbf{Adv}^{\text{kr}}_{\mathcal{KR},\mathcal{B}}
\end{aligned}$$

for all adversaries $\mathcal{B}$.

We now restrict our analysis to the case where $q_2 \geq 1$. Consider the experiments $\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},i}$ in Figure 4-13, $i \in \{0, \ldots, q_2\}$. When $i = q_2$, $\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},i}$ uses keyder to reply to all of $\mathcal{A}$'s HKeyderO oracle queries, which means that

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1\right] = \Pr\left[\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},q_2} = 1\right].$$

On the other hand, when $i = 0$, $\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},i}$ replies to all of $\mathcal{A}$'s HKeyderO oracle queries with random values from $\mathsf{DerKeys}_{\mathcal{KR}}$, which means that

$$\Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1\right] = \Pr\left[\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},0} = 1\right].$$

From these two equations we conclude that

$$\mathbf{Adv}^{\text{kr}}_{\mathcal{KR},\mathcal{A}} = \Pr\left[\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},q_2} = 1\right] - \Pr\left[\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},0} = 1\right]. \tag{4.14}$$

Note that $\mathbf{Adv}^{\text{kr}}_{\mathcal{KR},\mathcal{A}}$ could be negative. But for every adversary with a negative advantage, there is one adversary (obtained by switching outputs) that has the corresponding positive advantage.

**Experiment** $\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},i}$
Pick random oracle $H$
$l \leftarrow 0$
$\mathsf{stp} \xleftarrow{R} \mathsf{setup}^H$
$\mathsf{st} \xleftarrow{R} \mathcal{A}^{\mathsf{HWindO},H}(\mathsf{member})$
$j \leftarrow 0$
$b' \xleftarrow{R}$
$\mathcal{A}^{\mathsf{HKeyderO}_i,H}(\mathsf{non\text{-}member}, \mathsf{st})$
Return $b'$

**Oracle** HWindO
$l \leftarrow l + 1$
If $l >$ MaxWind then
return $\perp$
$(\mathsf{stp}, \mathsf{stm}) \xleftarrow{R} \mathsf{wind}^H(\mathsf{stp})$
Return stm

**Oracle** $\mathsf{HKeyderO}_i$
$l \leftarrow l + 1$
If $l >$ MaxWind then
return $\perp$
$(\mathsf{stp}, \mathsf{stm}) \xleftarrow{R} \mathsf{wind}^H(\mathsf{stp})$
If $j < i$ then
$K \leftarrow \mathsf{keyder}^H(\mathsf{stm})$
Else
$K \xleftarrow{R} \mathsf{DerKeys}_{\mathcal{KR}}$
$j \leftarrow j + 1$
Return $K$

Figure 4-13: Hybrid experiments for the proof of Lemma 4.4.14. The algorithms share the global variables $l$ and $j$.

**Adversary** $\mathcal{B}^{\mathsf{WindO},H}(\mathsf{member})$
$i \xleftarrow{R} \{0,\ldots,q_2-1\}$
$l \leftarrow 0$
Run $\mathcal{A}^{\mathsf{WindO}',H}(\mathsf{member})$,
replying to $\mathcal{A}$'s oracle queries as follows:
For each query to WindO′ do
$\mathsf{stm} \xleftarrow{R} \mathsf{WindO}$
$l \leftarrow l+1$
If $l > \mathsf{MaxWind}$ then $\mathsf{stm} \leftarrow \perp$
Return stm to $\mathcal{A}$
Until $\mathcal{A}$ halts outputting a state st′
For $j = 0$ to $i-1$ do
$\mathsf{stm} \xleftarrow{R} \mathsf{WindO}$
$K_j \leftarrow \mathsf{keyder}^H(\mathsf{stm})$
$\mathsf{st} \leftarrow (\mathsf{st}', i, l, K_0, \ldots, K_{i-1})$
Return st

**Adversary** $\mathcal{B}^{\mathsf{KeyderO}_b,H}(\mathsf{non\text{-}member},\mathsf{st})$
Parse st as $(\mathsf{st}', i, l, K_0, \ldots, K_{i-1})$
$j \leftarrow 0$
Run $\mathcal{A}^{\mathsf{KeyderO}',H}(\mathsf{non\text{-}member},\mathsf{st}')$,
replying to $\mathcal{A}$'s oracle queries as follows:
For each query to KeyderO′ do
If $j < i$ then $K \leftarrow K_j$
Else if $j = i$ then $K \leftarrow \mathsf{KeyderO}_b$
Else $K \xleftarrow{R} \mathsf{DerKeys}_{\mathcal{KR}}$
$j \leftarrow j+1$ ; $l \leftarrow l+1$
If $l > \mathsf{MaxWind}$ then $K \xleftarrow{R} \perp$
Return $K$ to $\mathcal{A}$
Until $\mathcal{A}$ halts outputting a bit $b$
Return $b$

Figure 4-14: Adversary $\mathcal{B}$ for the proof of Lemma 4.4.14. We describe in the body an alternate description with reduced resource requirements.

---

Consider now the adversary $\mathcal{B}$ in Figure 4-14. We claim that

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{B}} = 1\right] = \frac{1}{q_2} \cdot \sum_{i=0}^{q_2-1} \Pr\left[\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},i+1} = 1\right] \tag{4.15}$$

and

$$\Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{B}} = 1\right] = \frac{1}{q_2} \cdot \sum_{i=0}^{q_2-1} \Pr\left[\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},i} = 1\right] . \tag{4.16}$$

Subtracting Equation (4.16) from Equation (4.15) and using Definition 4.3.1, we get

$$\begin{aligned} \mathbf{Adv}^{\text{kr}}_{\mathcal{KR},\mathcal{B}} &= \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{B}} = 1\right] - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{B}} = 1\right] \\ &= \frac{1}{q_2} \cdot \left(\Pr\left[\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},q_2} = 1\right] - \Pr\left[\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},0} = 1\right]\right) . \end{aligned} \tag{4.17}$$

Equation (4.12) follows from combining Equation (4.14) with Equation (4.17).

It remains to justify Equation (4.15), Equation (4.16), and the resources of $\mathcal{B}$. To justify Equation (4.15), note that in the experiment $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{B}}$, when $\mathcal{B}$ picks some value for $i$, the view of $\mathcal{A}$ becomes equiv-

alent to $\mathcal{A}$'s view in $\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},i+1}$; namely, $\mathcal{A}$'s first $i+1$ queries to its KeyderO oracle will be computed using keyder, and the remaining KeyderO oracle queries will return random values from $\mathsf{DerKeys}_{\mathcal{KR}}$. More formally, if $I$ denotes the random variable for the $\mathcal{B}$'s selection for the variable $i \in \{0, \ldots, q_2 - 1\}$, then

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{B}} = 1 \mid I = i\right] = \Pr\left[\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},i+1} = 1\right]$$

for each $i \in \{0, \ldots, q_2 - 1\}$. Letting $\Pr_1[\cdot]$ denote the probability over $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{B}}$, we then derive Equation (4.15) by conditioning off the choice of $i$:

$$\begin{aligned}
\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{B}} = 1\right] &= \sum_{i=0}^{q_2-1} \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{B}} = 1 \mid I = i\right] \cdot \Pr_1[I = i] \\
&= \frac{1}{q_2} \cdot \sum_{i=0}^{q_2-1} \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{B}} = 1 \mid I = i\right] \\
&= \frac{1}{q_2} \cdot \sum_{i=0}^{q_2-1} \Pr\left[\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},i+1} = 1\right]
\end{aligned}$$

The justification for Equation (4.16) is similar. When $\mathcal{B}$ picks some value for $i$ in $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{B}}$, the view of $\mathcal{A}$ in $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{B}}$ becomes equivalent to $\mathcal{A}$'s view in $\mathbf{ExpH}_{\mathcal{KR},\mathcal{A},i}$ since in both cases the responses to $\mathcal{A}$'s first $i$ (not $i+1$ this time) queries to its KeyderO oracle will be computed using keyder, and the remaining KeyderO oracle queries will return random values from $\mathsf{DerKeys}_{\mathcal{KR}}$.

We now turn to the resource requirements of $\mathcal{B}$. The pseudocode for $\mathcal{B}$ in Figure 4-14 suggests that $\mathcal{B}$ might invoke WindO and keyder up to $q_2$ times more than $\mathcal{A}$ (since the last for loop of $\mathcal{B}$'s member stage runs for up to $q_2$ interactions even though $\mathcal{A}$ may not make that many KeyderO oracle queries). We describe $\mathcal{B}$ this way since we feel that Figure 4-14 better captures the main idea behind our proof and what $\mathcal{B}$ does. Equivalently, $\mathcal{B}$ could split $\mathcal{A}$'s non-member stage between its ($\mathcal{B}$'s) own member and non-member stages and invoke WindO and keyder only the number of times that it needs to simulate $i$ of $\mathcal{A}$'s $\mathsf{KeyderO}_i$ oracle queries. When viewed this way, $\mathcal{B}$ uses resources equivalent, within a constant factor, to the resources of $\mathcal{A}$. ∎

**Adversary** $\mathcal{B}(y, e, N)$
bad $\leftarrow$ false
$\alpha \leftarrow \perp$
$j \leftarrow 0$
$w \xleftarrow{R} \{0, 1, 2, \ldots, q\}$
$\mathsf{stm}_w \leftarrow y$
For $i = w - 1$ downto 1 do
  $\mathsf{stm}_i \leftarrow (\mathsf{stm}_{i+1})^e \mod N$
st $\xleftarrow{R} \mathcal{A}^{\mathsf{SimWindO},\mathsf{SimH}}(\mathsf{member})$
If $j \neq w$ then
  bad $\leftarrow$ true
  Return $\perp$
$b \xleftarrow{R} \mathcal{A}^{\mathsf{SimKeyderO},\mathsf{SimH}}(\text{non-member}, \mathsf{st})$
Return $\alpha$

**Oracle** SimWindO
$j \leftarrow j + 1$
If $j \leq w$ then return $\mathsf{stm}_j$
Else return $\perp$

**Oracle** SimKeyderO
$K \xleftarrow{R} \mathsf{DerKeys}_{\mathcal{KR}}$
Return $K$

**Oracle** SimH$(x)$
If $x^e = y \mod n$ then $\alpha \leftarrow x$
If $H[x]$ undefined then
  $H[x] \xleftarrow{R} \mathsf{DerKeys}_{\mathcal{KR}}$
Return $H[x]$

Figure 4-15: The adversary $\mathcal{B}$ in the proof of Lemma 4.4.15.

---

**Proof of Lemma 4.4.15.** **Proof:** Consider the experiments $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ and $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}}$; let $(N, e, S_1)$, $(N, e, S_2), \ldots, (N, e, S_{w'})$ denote the responses to $\mathcal{A}$'s wind oracle queries when $\mathcal{A}$ is in the member stage, $w' \in \{0, 1, \ldots, q\}$. Let $\mathcal{E}_1$ and $\mathcal{E}_0$ respectively be the events in $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ and $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}}$ that $\mathcal{A}$ queries its wind oracle with a value $S$ such that $S^e \equiv S_{w'} \mod N$. We claim that

$$\mathbf{Adv}^{\text{kr}}_{\mathcal{KR},\mathcal{A}} = \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_1\right] - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_0\right]. \tag{4.18}$$

Consider now the adversary $\mathcal{B}$ in Figure 4-15. We additionally claim that

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_1\right] \leq (q+1) \cdot \Pr\left[\mathbf{Exp}^{\text{rsa-ow}}_{\mathcal{K}_{\text{rsa}},\mathcal{B}} = 1\right]. \tag{4.19}$$

Combining these two equations and the definition of security for $\mathcal{K}_{\text{rsa}}$ gives Equation (4.13).

It remains to justify Equation (4.18), Equation (4.19), and the resource requirements for $\mathcal{B}$. We first justify Equation (4.18). Let $\Pr_1[\cdot]$ and $\Pr_0[\cdot]$ denote the probabilities over $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ and

$\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}}$, respectively. From Definition 4.3.1, we have

$$\begin{aligned}\mathbf{Adv}^{\text{kr}}_{\mathcal{KR},\mathcal{A}} &= \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1\right] - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1\right] \\ &= \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_1}\right] + \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_1\right] \\ &\quad -\Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_0}\right] - \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_0\right]. \qquad (4.20)\end{aligned}$$

By conditioning,

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_1}\right] = \Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \mid \overline{\mathcal{E}_1}\right] \cdot \Pr_1\left[\overline{\mathcal{E}_1}\right]$$

and

$$\Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_0}\right] = \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \mid \overline{\mathcal{E}_0}\right] \cdot \Pr_0\left[\overline{\mathcal{E}_0}\right].$$

Prior to $\mathcal{E}_1$ and $\mathcal{E}_0$, $\mathcal{A}$'s view is identical in both experiments, meaning that

$$\Pr_1\left[\overline{\mathcal{E}_1}\right] = \Pr_0\left[\overline{\mathcal{E}_0}\right].$$

Further, if the events do not occur, then the outcome of $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ and $\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}}$ will be the same since the output of a random oracle is random if the input is unknown; i.e., the response to $\mathcal{A}$'s key derivation oracle query in the non-member stage will be random in both cases and therefore

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \mid \overline{\mathcal{E}_1}\right] = \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \mid \overline{\mathcal{E}_0}\right].$$

Consequently,

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_1}\right] = \Pr\left[\mathbf{Exp}^{\text{kr-0}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \overline{\mathcal{E}_0}\right].$$

Combining the above equation with Equation (4.20) gives Equation (4.18).

We now turn to Equation (4.19). Note that $\mathcal{B}$ runs $\mathcal{A}$ exactly as in $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ assuming that $\mathcal{B}$ correctly guesses the number of wind oracle queries that $\mathcal{A}$ will make in its member stage; i.e., if $\mathcal{B}$ does not set bad to true. Here we use the fact that RSA encryption and decryption is a permutation and therefore $\mathcal{B}$ is justified in unwinding a starting state from its input $(y, e, N)$. Also observe that if $\mathcal{E}_1$ in $\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}}$ occurs and if $\mathcal{B}$ does not set bad to true, then $\mathcal{B}$ will succeed in

inverting RSA. Letting BAD denote the event that $\mathcal{B}$ sets bad to true, it follows that

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_1\right] \leq \Pr\left[\mathbf{Exp}^{\text{rsa-ow}}_{\mathcal{K}_{\text{rsa}},\mathcal{B}} = 1 \mid \overline{\mathsf{BAD}}\right]$$

and, by conditioning, that

$$\Pr\left[\mathbf{Exp}^{\text{kr-1}}_{\mathcal{KR},\mathcal{A}} = 1 \wedge \mathcal{E}_1\right] \leq \Pr\left[\mathbf{Exp}^{\text{rsa-ow}}_{\mathcal{K}_{\text{rsa}},\mathcal{B}} = 1 \wedge \overline{\mathsf{BAD}}\right] \cdot \frac{1}{\Pr_2\left[\overline{\mathsf{BAD}}\right]}$$

where $\Pr_2[\cdot]$ denotes the probability over $\mathbf{Exp}^{\text{rsa-ow}}_{\mathcal{K}_{\text{rsa}},\mathcal{B}}$. Equation (4.19) follows from the above equation and the fact that $\Pr_2\left[\overline{\mathsf{BAD}}\right] = 1/(q+1)$.

Turning to the resource requirements of $\mathcal{B}$, note that the for loop in $\mathcal{B}$ is not present $\mathcal{A}$ (nor in the algorithm setup nor the experiment $\mathbf{Exp}^{\text{kr-}b}_{\mathcal{KR},\mathcal{A}}$). This means that $\mathcal{B}$ may perform $q$ more RSA encryption operations than in the $\mathbf{Exp}^{\text{kr-}b}_{\mathcal{KR},\mathcal{A}}$ experiment running $\mathcal{A}$; $\mathcal{B}$ does not, however, invoke any RSA decryption operations. ∎

# Chapter 5

# Implementation

*Plan to throw one away.*

– Frederick P. Brooks, Jr.

*cd build*

*rm -rf sfs1*

*cd /disk/ex0/sfscvs*

↑↑←⏎

– Kevin Fu

SFSRO and Chefs are two file systems that demonstrate the practical applications of integrity protection and access control for content distribution using untrusted servers. We describe the implementation of both prototypes.

## 5.1 SFSRO

As illustrated in Figure 5-1, the read-only file system is implemented as two daemons (`sfsrocd` and `sfsrosd`) in the SFS system [73]. `sfsrodb` is a stand-alone program.

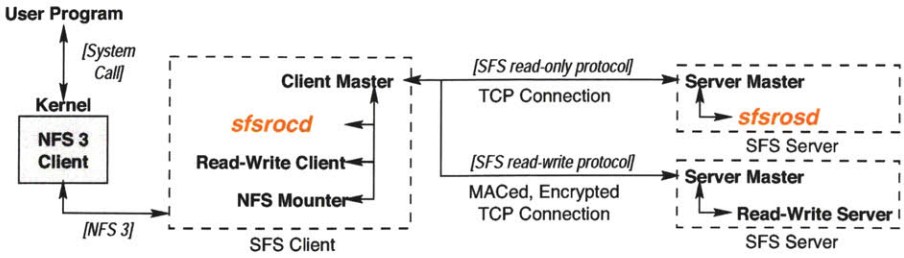


Figure 5-1: Implementation overview of the read-only file system in the SFS framework.

`sfsrocd` and `sfsrosd` communicate with Sun RPC over a TCP connection. (The exact message formats are described in the XDR protocol description language [112].) We also use XDR to define cryptographic operations. Any content that the read-only file system hashes or signs is defined as an XDR data structure; SFSRO computes the hash or signature on the raw, marshaled bytes.

`sfsrocd`, `sfsrosd`, and `sfsrodb` are written in C++. To handle many connections simultaneously, the client and server use SFS's asynchronous RPC library. Both programs are single-threaded, but the RPC library allows the client to have many outstanding RPCs.

Because of SFS's support for developing new servers and the read-only server's simplicity, the implementation of `sfsrosd` measured by Chapter 6 is trivial—only 400 lines of C++. It gets requests for content blocks by file handle, looks up pre-formatted responses in a B-tree, and responds to the client. The current implementation uses the synchronous version of Sleepycat database's B-tree [108].

The implementations of the other two programs (`sfsrodb` and `sfsrocd`) are more interesting; we discuss them in more detail.

### 5.1.1 Database generator

The database generator is a 1,500-line, stand-alone C++ program. To publish a file system, a system administrator runs `sfsrodb` to produce a signed database from a private key and a directory in an existing file system. The database generator computes every content block, indirect block, inode, and directory block required for the file system, and stores these structures in the database,

indexed by hash value.

The database generator utility traverses the given file system depth-first to build the database. The leaves of the file system tree are files or symbolic links. For each regular file in a directory, the database generator creates a read-only inode structure and fills in the metadata. Then, it reads the blocks of the file. For each block, `sfsrodb` hashes the content in that block to compute its handle, and then inserts the block into the database under the handle (i.e., a lookup on the handle will return the block). The hash value is also stored in an inode. When all file blocks of a file are inserted into the database, the filled-out inode is inserted into the database under its hash value.

When all files in a given directory have been inserted into the database, the generator utility inserts a file corresponding to the directory itself—it hashes blocks of ⟨name, handle⟩ pairs into an inode data structure. After the root directory tree has been inserted into the database, the generator utility fills out an `FSINFO` structure and signs it with the private key of the file system. For simplicity, `sfsrodb` stores the signed `FSINFO` structure in the database under a well-known, reserved key.

As motivated in Section 2.4, the database contains data structures in XDR marshaled form. One disadvantage of this is that physical representation of the content is slightly larger than the actual content. For instance, an 8 Kbyte file block is slightly larger than 8 Kbyte.

A benefit of storing blocks under their hash is that blocks from different files that have the same hash will only be stored once in the database. If a file system contains blocks with identical content among multiple files, then `sfsrodb` stores just one block under the hash. In the RedHat 6.2 distribution, 5,253 out of 80,508 file content blocks share their hash with another block. The overlap is much greater if one makes the same content available in two different formats (for instance, the contents of the RedHat 6.2 distribution, and the image of a CD-ROM containing that distribution).

### 5.1.2 Client daemon

The client constitutes the bulk of the code in the read-only file system (1,500 lines of C++). The read-only client behaves like an NFS3 [24] server, allowing it to communicate with the operating

system through ordinary networking system calls. The read-only client resolves pathnames for file name lookups and handles reads of files, directories, and symbolic links. It relies on the server only for serving blocks of content, not for interpreting or verifying those blocks. The client checks the validity of all blocks it receives against the hashes by which it requested them.

The client implements four caches with LRU replacement policies to improve performance by avoiding RPCs to `sfsrosd`. It maintains an inode cache, an indirect-block cache, a small file-block cache, and a cache for directory entries. The client also maintains a file handle translation table, mapping NFS file handles to SFSRO handles as described by Section 2.5.2.

`sfsrocd`'s small file-block cache primarily optimizes the case of the same block appearing in multiple files. In general, `sfsrocd` relies on the local operating system's buffer cache to cache the file content. Thus, any additional caching of file content will tend to waste memory unless the content in appears multiple files. The small block cache optimizes common cases—such as a file containing many blocks of all zeros—without dedicating too much memory to redundant caching.

Indirect blocks are cached so that `sfsrocd` can quickly fetch and verify multiple blocks from a large file without refetching the indirect blocks. `sfsrocd` does not prefetch because most operating systems already implement prefetching locally.

Whenever the client must return an NFS file handle to the kernel, it computes a hash of the file's pathname and adds a mapping from the new NFS file handle to the corresponding SFSRO handle. For instance, SFSRO directory entries map filenames to SFSRO handles. When responding to a directory lookup, the client must translate each SFSRO handle into an NFS file handle.

When receiving a kernel's request containing an NFS file handle, the client first translates the NFS file handle to an SFSRO handle. Because NFSv3 does not keep track of which files are actively opened, the SFSRO client will not know when it can safely remove a table entry. As a result, the table will continue to grow as more files are accessed. To keep the table to a reasonable size in the client's working memory, we added an NFS_CLOSE RPC such that a modified kernel can simulate the closing of a file.

### 5.1.3 Example

We demonstrate how the client works by example. Consider a user reading the file `/sfs/@sfs.fs.net,uzwadtctbjb3dg596waiyru8cx5kb4an/README`, where `uzwadtctbjb3dg596waiyru8cx5kb4an` is the representation of the public key of the server storing the file `README`. (In practice, symbolic links save users from ever having to see or type pathnames like this.)

The local operating system's NFS client will call into the protocol-independent SFS client software, asking for the directory `/sfs/@sfs.fs.net,uzwadtctbjb3dg596waiyru8cx5kb4an/`. The client will contact `sfs.fs.net`, which will respond that it implements the read-only file system protocol. At that point, the protocol-independent SFS client daemon will pass the connection off to the read-only client, which will subsequently be asked by the kernel to interpret the file named `README`.

The client makes a *getfsinfo* RPC to the server to get the file system's signed `FSINFO` structure. It verifies the signature on the structure, ensures that the `start` field is no older than its previous value if the client has seen this file system before, and ensures that `start` + `duration` is in the future. Checking the `start` and `duration` fields ensures that content is fresh.

The client then obtains the root directory's inode by doing a *getdata* RPC on the `rootfh` field of `FSINFO`. Given that inode, it looks up the file `README` by doing a binary search among the blocks of the directory, which it retrieves through *getdata* calls on the block handles in the directory's inode (and possibly indirect blocks). When the client has the directory entry $\langle$`README`, handle$\rangle$, it calls *getdata* on handle to obtain `README`'s inode. Finally, the client can retrieve the contents of `README` by calling *getdata* on the block handles in its inode.

## 5.2 Chefs

Chefs is a confidentiality-enabled version of SFSRO. Chefs consists of three programs: a database generator, a client, and a server. During creation of a new database, the publisher may specify a group key for access control of the entire database. Our level of granularity is the database.

Each file block is encrypted with a randomly generated 128-bit AES key. A lockbox in the inode contains this random AES key, itself encrypted with the group key. To read a file, a client contacts a content publisher (or a designated proxy) for the appropriate group key. The client decrypts the lockbox to obtain the file key, which provides access to the plaintext.

Chefs implements the SHA1-based version of key regression. Each user of a Chefs client daemon has an RSA key pair. To obtain new key regression member state, the client makes a *getkey* RPC of the publisher's key distribution service. The publisher then responds with the latest key regression member state encrypted in the requesting user's RSA public key. The client decrypts this message, then uses key regression to compute the version of the group key that protects the database. The SFSRO client daemon includes a single convenient location to decrypt all content. Chefs adds code to the callback that unmarshals responses from *getdata*.

The demands of Chefs revealed many latent bugs in the SFSRO client daemon. In particular, a race condition existed because several asynchronous callbacks shared a common cache of content blocks. The encryption in Chefs affected the timings of callbacks, making the race condition apparent. In the worst case, the race would cause the Chefs client daemon to fetch each block twice. The race happens when one callback fetches a content block that a second callback has started to fetch. This occurs when the NFS client in the kernel fetches content that crosses block boundaries. Chefs and SFSRO now use a locking protocol to prevent this performance-dehancing race condition.

### 5.2.1 Example

To illustrate how to use Chefs, we give an example scenario. Alice is a content publisher. Bob is a member of the group that has access to the content. Alice runs the database generator and an untrusted server. Bob runs a Chefs client.

**Generating a group key and database.** Alice uses `sfsrodb` to create a database and group key. If `sfsrodb` does not find an existing group key, it creates a new one with the maxwind parameter set by Alice. The database is encrypted block by block.

**Reading a file.** Bob wishes to read a file from Alice's database. Bob attempts to mount the remote file system. This causes Bob's client daemon to automatically call *getkey* to the Alice's server. Alice's key management server responds with the latest key regression member state, encrypted in Bob's well-known RSA public key. Once Bob's client daemon has decrypted the key regression member state, the client daemon can unwind and derive group keys suitable to decrypt a lockbox. A decrypted lockbox produces a content key that finally decrypts the sought-after content. This entire process is transparent to the users.

**Evicting a group member.** Alice evicts Bob from the group by disallowing him access to new keys. Alice winds the key regression publisher state forward and makes new member state available to remaining members via *getkey*.

**Updating a database.** Alice modifies a file after evicting Bob. This event requires that Alice re-run the database generator to update and re-encrypt changes in the database. Bob will continue to have access to unchanged content, but will not have access to newly encrypted content.

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 6

# Performance

*Bread, like so many of life's formulas, relies on the 80/20 principle, in this instance, 80 percent technique and 20 percent equipment.*

– Peter Reinhart

This chapter presents microbenchmarks and application-level benchmarks to demonstrate that (1) SFSRO keeps the overhead of cryptography low on clients, allowing servers to scale to a large number of clients, and that (2) key regression in Chefs can significantly reduce the bandwidth requirements of a publisher distributing keys to clients.

A performance evaluation of SFSRO on a 550 MHz Pentium III with 256 Mbyte of memory running FreeBSD shows that the server can support 1,012 short-lived connections per second, which is 26 times better than a standard read-write SFS file server and 92 times better than a secure Web server. The performance of the read-only server is limited mostly by the number of TCP connections per second, not by the overhead of cryptography, which is offloaded to clients. For applications such as sustained downloads that require longer-lived connections, the server can support 300 concurrent sessions while still saturating a fast Ethernet.

The measurements of an encrypted search workload show that key regression can significantly reduce the bandwidth requirements of a publisher distributing keys to clients. On a simulated cable modem, a publisher using key regression can distribute 1,000 keys to 181 clients per second whereas without key regression the cable modem limits the publisher to 20 clients per second.

The significant gain in throughput conservation comes at no cost to client latency, even though key regression requires more client-side computation. The measurements show that key regression actually reduces client latency in cases of highly dynamic group membership. Even though hash functions normally have higher throughput than block ciphers, the efficient KR-AES construction performs more than four times faster than KR-SHA1.

## 6.1 Measurements of SFS read-only

This section presents the results of measurements to support the claims that (1) SFS read-only provides acceptable application performance for individual clients and (2) SFS read-only scales well with the number of clients.

To support the first claim, this thesis presents microbenchmarks and performance measurements of a large software compilation. The performance of SFS read-only is compared with the performance of the local file system, insecure NFS, and the secure SFS read-write file system (SFSRW).

To support the second claim, the maximum number of connections per server and the throughput of software downloads with an increasing number of clients are measured.

The main factors expected to affect SFS read-only performance are the user-level implementation of the client, hash verification in the client, and database lookups on the server.

### 6.1.1 Experimental setup and methodology

The historical results in this section were measured several years before the results in Section 6.2. Although the experimental environment is no longer modern, the conclusions are still justified by the measurements.

Measurements were conducted on 550 MHz Pentium IIIs running FreeBSD 3.3. The client and server were connected by 100 Mbit, full-duplex, switched Ethernet. Each machine had a 100 Mbit Tulip Ethernet card, 256 Mbyte of memory, and an IBM 18ES 9 Gigabyte SCSI disk. The client maintains inode, indirect-block, and directory entry caches that each have a maximum of 512

| Operation | Cost ($\mu$sec) |
|---|---|
| Sign 68 byte fsinfo | 24,400 |
| Verify 68 byte fsinfo | 82 |
| SHA1 256 byte iv+inode | 17 |
| SHA1 8,208 byte iv+block | 406 |

Table 6.1: Performance of base primitives on a 550 MHz Pentium III. Signing and verification use 1,024-bit Rabin-Williams keys.

entries, while the file-block cache has maximum of 64 entries. It suffices for the client to use small caches because the in-kernel NFS client already caches file and directory content. Maximum TCP throughput between client and server, as measured by `ttcp` [116], was 11.31 Mbyte/sec.

Because the certificate authority benchmark in Section 6.1.4 requires many CPU cycles on the client, two 700 MHz Athlons running OpenBSD 2.7 were added as clients to produce a workload saturating the server. Each Athlon had a 100 Mbit Tulip Ethernet card and 128 Mbyte of memory. Maximum TCP throughput between an Athlon and the FreeBSD server, as measured by `ttcp`, was 11.04 Mbyte/sec. The Athlon machines generated the client SSL and SFSRW requests; this thesis reports the sum of the performance measured on the two machines.

For some workloads, traces of the RPC traffic between a client and a server were collected. A simple client program replays these traces to evaluate the performance of the server itself and to determine how many clients a single server can support.

For all experiments this thesis reports the average of five runs. To demonstrate the consistency of measurements, this thesis reports the percentage at which the minimum and maximum samples are within the average.

### 6.1.2 Microbenchmarks

Small and large file microbenchmarks help to evaluate the performance of the SFS read-only system. Table 6.1 lists the cost of cryptographic primitives used in the read-only file system. The implementation is secure against chosen-message attacks, using the redundancy function proposed by Bellare and Rogaway [16]. Table 6.1 shows that computing digital signatures is somewhat expensive, but verifying takes only 82 $\mu$sec—far cheaper than a typical network round-trip time.

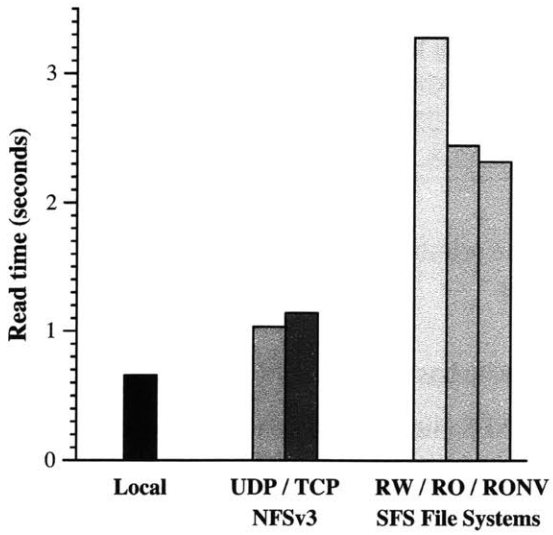


Figure 6-1: Time to sequentially read 1,000 files each 1 Kbyte. Local is FreeBSD's local FFS file system on the server. The local file system was tested with a cold cache. The network tests were applied to warm server caches, but cold client caches. RW, RO, and RONV denote respectively the read-write protocol, the read-only protocol, and the read-only protocol with integrity verification disabled.

**Small file benchmark.** The read phases of the LFS benchmarks [99] provide a basic understanding of single-client, single-server performance. Figure 6-1 shows the latency of sequentially reading 1,000 files each 1 Kbyte on the different file systems. The files contain random content and are distributed evenly across ten directories. For the read-only and NFS experiments, all samples were within 0.4% of the average. For the read-write experiment, all samples were within 2.7% of the average. For the local file system, all samples were within 6.9% of the average.

| Breakdown | Cost (sec) | Percent |
|---|---|---|
| NFS loopback | 0.661 | 26% |
| Computation in client | 1.386 | 54% |
| Communication with server | 0.507 | 20% |
| Total | 2.55 | 100% |

Table 6.2: Breakdown of SFS read-only performance as reported in Figure 6-1. The actual measured latency is 2.43 sec, whereas the estimated total is 2.55 sec. This overestimate is attributed to a small amount of double counting of cycles between the NFS lookback measurement and the computation in the client.

As expected, the SFS read-only server performs better than the SFS read-write server (2.43 vs. 3.27 seconds). The read-only file server performs worse than NFSv3 over TCP (2.43 vs. 1.14 seconds). Table 6.2 and the following paragraphs analyze in more detail the 2.43 seconds spent in the read-only client.

To determine the cost of the user-level implementation, the time spent in the NFS loopback is measured. The `fchown` operation used against a file in a read-only file system effectively measures the time spent in the user-level NFS loopback file system. This operation generates NFS RPCs from the kernel to the read-only client, but no traffic between the client and the server. The average over 1,000 `fchown` operations is 167 $\mu$sec. By contrast, the average for attempting an `fchown` of a local file with permission denied is 2.4 $\mu$sec. The small file benchmark generates 4,015 NFS loopback RPCs. Hence, the overhead of the client's user-level implementation is at least (167 $\mu$sec - 2.4 $\mu$sec) $*$ 4,015 = 0.661 seconds.

The CPU time spent during the small file benchmark in the read-only client is 1.386 seconds. With integrity verification disabled, this drops to 1.300 seconds, indicating that for this workload, file handle verification consumes little CPU time.

To measure the time spent communicating with the read-only server, a trace of the 2,101 *getdata* RPCs of the benchmark were played back to the read-only server. This took 0.507 seconds.

These three measurements total to 2.55 seconds. With an error margin of 5%, this accounts for the 2.43 seconds to run the benchmark. This error is attributed to a small amount of double counting of cycles between the NFS loopback measurement and the computation in the client.

In summary, the cryptography accounts for little of the time in the SFS read-only file system. The CPU time spent on verification is only 0.086 seconds. Moreover, end-to-end measurements show that content verification has little impact on performance. RONV performs slightly better than RO (2.31 vs. 2.43 seconds). Any optimization will have to focus on the non-cryptographic portions of the system.

**Large file benchmark** Figure 6-2 shows the performance of sequentially and randomly reading a 40 Mbyte file containing random content. Blocks are read in 8-Kbyte chunks. In the network experiments, the file is in the server's cache, but not in the client's cache. Thus, the experiment

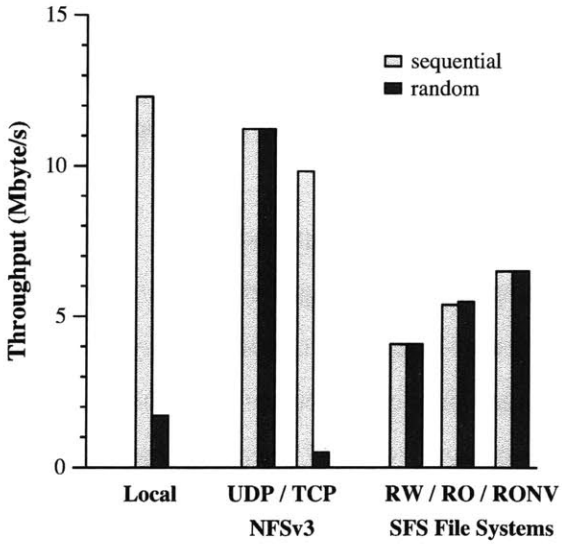


Figure 6-2: Throughput of sequential and random reads of a 40 Mbyte file. The experimental conditions are the same as in Figure 6-1 where the server has a warm cache and the client has a cold cache.

does not measure the server's disk. This experiment isolates the software overhead of cryptography from SFS's user-level design.

For the local file system, all samples were within 1.4% of the average. For NFSv3 over UDP and the read-write experiments, all samples were within 1% of the average. For NFSv3 over TCP and the read-only experiments, all samples were within 4.3% of the average. This variability and the poor NFSv3 over TCP random read performance appears to be due to a pathology of FreeBSD.

The SFS read-only server performs better than the read-write server because the read-only server performs no online cryptographic operations. On the sequential workload, verification costs 1.4 Mbyte/s in throughput. NFSv3 over TCP performs substantially better for sequential reads (9.8 vs. 6.5 Mbyte/s) than the read-only file system without verification, even though both run over TCP and do similar amounts of work; the main difference is that NFS is implemented in the kernel.

In SFSRO, the random read has a slight 0.1 Mbyte/s advantage over the sequential read. The random read is faster than the sequential read because of interactions between caches and cryptography. The random read test consists of reading 5,120 8 Kbyte blocks at random locations from the 40 Mbyte file. Thus, it is expected to re-read some blocks and to not read other blocks at all.

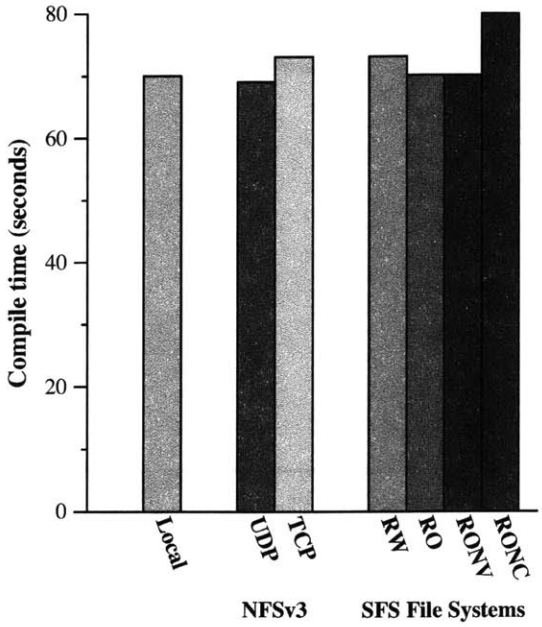


Figure 6-3: Compiling the Emacs 20.6 source. Local is FreeBSD's local FFS file system on the server. The local file system was tested with a cold cache. The network tests were applied to warm server caches, but cold client caches. RW, RO, RONV, and RONC denote respectively the read-write protocol, the read-only protocol, the read-only protocol with integrity verification disabled, and the read-only protocol with caching disabled.

Because no cryptography re-verification is necessary on an already cached block, it makes sense that random reads in SFSRO would slightly outperform sequential reads.

If the large file contains only blocks of zeros, SFS read-only obtains a throughput of 17 Mbyte/s since all blocks hash to the same handle. In this case, the measurement is dominated by the throughput of loopback NFSv3 over UDP within the client machine.

### 6.1.3 Software distribution

To evaluate how well the read-only file system performs on a larger application benchmark, Emacs 20.6 was compiled (with optimization and debugging disabled) with a local build directory and a remote source directory. The results are shown in Figure 6-3. The RO experiment performs 1% worse (1 second) than NFSv3 over UDP and 4% better (3 seconds) than NFSv3 over TCP. Dis-

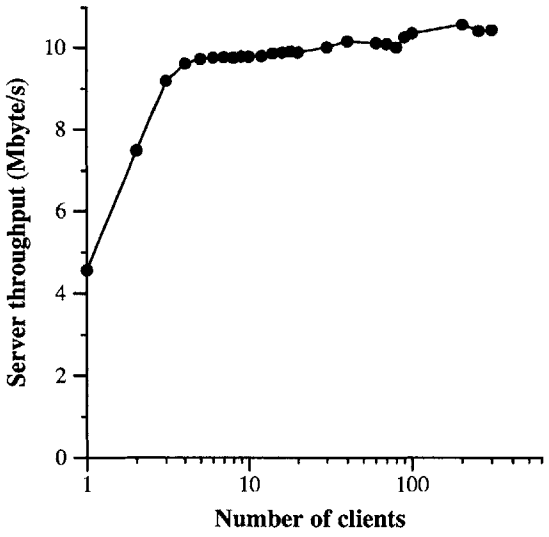


Figure 6-4: The aggregate throughput delivered by the read-only server for an increasing number of clients simultaneously compiling the Emacs 20.6 source. The number of clients is plotted on a log scale.

abling integrity checks in the read-only file system (RONV) does not speed up the compile because the caches absorb the cost of hash verification. However, disabling caching does decrease performance (RONC). During a single Emacs compilation, the read-only server consumes less than 1% of its CPU while the read-only client consumes less than 2% of its CPU. This benchmark demonstrates that the read-only protocol introduces negligible performance degradation in an application benchmark.

To evaluate how well the server scales, a trace of a single client compiling the Emacs 20.6 source tree was repeatedly played back to the server from an increasing number of simulated, concurrent clients. Figure 6-4 plots the aggregate throughput delivered by the server. Each sample represents the throughput of playing traces for 100 seconds. Each trace consists of 1,428 *getdata* RPCs. With 300 simultaneous clients, the server consumes 96% of the CPU.

With more than 300 clients, FreeBSD reboots because of a bug in its TCP implementation. In an experiment replacing FreeBSD with OpenBSD, the server maintains a rate of 10 Mbyte/s of file system content for up to 600 simultaneous clients.

### 6.1.4 Certificate authority

To evaluate whether the read-only file system performs well enough to function as an online certificate authority, the number of connections a single read-only file server can sustain is compared with the number of connections to the SFS read-write server, the number of SSL connections to an Apache Web server, and the number of HTTP connections to an Apache server.

The SFS servers use 1,024-bit keys. The SFS read-write server performs one Rabin-Williams decryption per connection while the SFS read-only server performs no online cryptographic operations. The Web server was Apache 1.3.12 with OpenSSL 0.9.5a and ModSSL 2.6.3-1.3.12. The SSL ServerID certificate and Verisign certificate use 1,024-bit RSA keys. All the SSL connections use the TLSv1 cipher suite consisting of Ephemeral Diffie-Hellman key exchange, DES-CBC3 for confidentiality, and SHA1 HMAC for integrity.

To generate enough load to saturate the servers, a simple client program sets up connections, reads a small file containing a self-certifying path, and terminates the connection as fast as it can. This client program runs simultaneously on the two OpenBSD machines. In all experiments, the certificate is in the main memory of the server, limiting the experiment by software performance, not by disk performance. This scenario is realistic since this thesis envisions that important online certificate authorities would have large enough memories to avoid frequent disk accesses.

The SFS read-only protocol performs client-side name resolution, unlike the Web server which performs server-side name resolution. Both single-component and multi-component lookups are measured. (For instance, `http://host/a.html` causes a single-component lookup while `http://host/a/b/c/d.html` causes a multi-component lookup.) The read-only client makes a linear number of *getdata* RPCs with respect to the number of components in a lookup. On the other hand, the HTTP client makes only one HTTP request regardless of the number of components in the URL path.

The HTTP and SSL single- and multi-component tests consist of a GET `/symlink.txt` and GET `/one/two/three/symlink.txt` respectively, where `symlink.txt` contains the string `/sfs/new-york.lcs.mit.edu:bzcc5hder7cuc86kf6qswyx6yuemnw69/`. The SFS read-only and read-write tests consist of comparable operations. A trace is played back

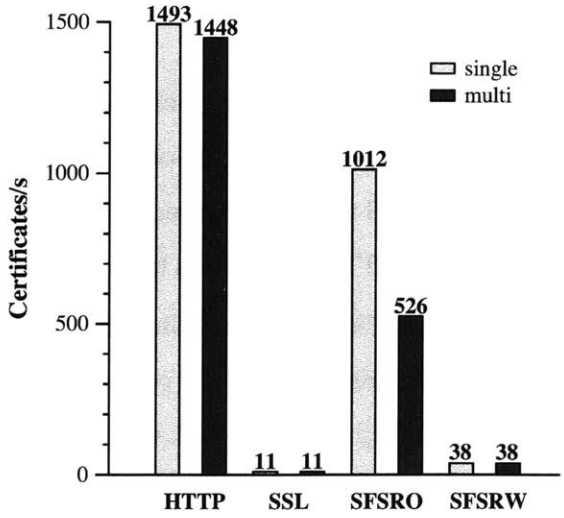


Figure 6-5: Maximum sustained certificate downloads per second. HTTP is an insecure Web server, SSL is a secure Web server, SFSRW is the secure SFS read-write file system, and SFSRO is the secure read-only file system. Light bars represent single-component lookups while dark bars represent multi-component lookups.

of reading a symlink that points to the above self-certifying path. The single-component trace of the read-only file system consists of 5 *getdata* RPCs to read a symlink in the top-level directory. The multi-component trace consists of 11 *getdata* RPCs to read a symlink in a directory three levels deep. The single-component SFSRW trace consists of 6 RPCs while the multi-component trace consists of 12 RPCs. Measuring the performance of an actual client would not measure the server throughput; the client overhead would distort the measurement of the server. Thus, the trace is necessary to measure the aggregate throughput of many simulated clients accessing a single server.

The benchmark uses a trace rather than an actual client in order to isolate the measurement of server performance.

Figure 6-5 shows that the read-only server scales well. For single-component lookups, the SFS read-only server can process 26 times more certificate downloads than the SFS read-write server because the read-only server performs no online cryptographic operations. The read-write server is bottlenecked by public key decryptions, which each take 24 msec. Hence, the read-write server

can at best achieve 38 (= 1,000/24) connections per second.

By comparing the read-only server with an insecure Apache server, this thesis concludes that the read-only server is a good platform for serving read-only content to many clients; the number of connections per second is only 32% lower than that of the *insecure* Apache server. In fact, the performance of SFS read-only is within an order of magnitude of the performance of a DNS root server, which according to Network Solutions can sustain about 4,000 lookups per second (DNS uses UDP instead of TCP). Since the DNS root servers can support online name resolution for the Internet, this comparison suggests that it is reasonable to build a distributed online certificate authority using SFS read-only servers.

A multi-component lookup is faster with HTTP than with the SFS read-only file system. The SFS read-only client must make two *getdata* RPCs per component. Hence, there is a slowdown for deep directories. In practice, the impact on performance will depend on whether clients do multi-component lookups once, and then never look at the same directory again, or rather, amortize the cost of walking the file system over multiple lookups. In any situation in which a single read-only client does multiple lookups in the same directory, the client should have performance similar to the single-component case because it will cache the components along the path.

In the case of the certificate authority benchmark, it is realistic to expect all files to reside in the root directory. Thus, this usage scenario minimizes people's true multi-component needs. On the other hand, if the root directory is huge, then SFS read-only will require a logarithmic number of round-trips for a lookup. However, SFS read-only will still outperform HTTP on a typical file system because Unix normally performs directory lookups in time linear in the number of directory entries; SFS read-only performs a lookup in logarithmic time in the number of directory entries.

## 6.2 Measurements of Chefs

Performance measurements validate that (1) key regression allows a publisher to serve many keys per second to clients effectively independent of the publisher's network throughput and the rate of membership turnover, and (2) key regression does not degrade client latency. To test these hypotheses, this thesis compares the performance of Chefs to Sous-Chefs, a version of Chefs without

key regression.

Microbenchmarks of Chefs in a static environment ensure confidence that the basic file system performs in a reasonable manner. The microbenchmarks help to explain the consumption of time Chefs. Application-level measurements show that a publisher can serve many keys per second to clients when using key regression — even on a low-bandwidth connection.

### 6.2.1 Experimental setup and methodology

The results in this section were measured several years after the results in Section 6.1. Over time the test equipment and software evolved significantly. Thus, the reader should not to draw comparisons between measurements in this section with measurements from Section 6.1. This thesis therefore provides benchmarks of SFSRO on the new equipment to make a fair comparison to Chefs.

Three machines were used to benchmark Chefs:

1. a *client* accessing the file system and performing key regression;
2. a *server* to distribute encrypted content; and
3. a *publisher* to serve keys to clients.

The client and server contained the same hardware: a 2.8 GHz Intel Pentium 4 with 512 MB RAM. Each machine used a 100 Mbit/sec full-duplex Intel PRO/1000 Ethernet card and a Maxtor 250 GB, Serial ATA 7200 RPM hard drive with an 8 MB buffer size, 150 MB/sec transfer rate, and less than 9.0 msec average seek time. The publisher was a 3.06 GHz Intel Xeon with 2 GB RAM, a Broadcom BCM5704C Dual Gigabit Ethernet card, and a Hitachi 320 GB SCSI-3 hard drive with a 320 MB/sec transfer rate.

The machines were connected on a 100 Mbit/sec local area network and all used FreeBSD 4.9. NetPipe [109] measured the round-trip latency between the pairs of machines at 249 $\mu$sec, and the maximum sustained TCP throughput of the connection at 88 Mbit/sec when writing data in 4 MB chunks and using TCP send and receive buffers of size 69,632 KB. When writing in 8 KB chunks (the block size in Chefs), the peak TCP throughput was 66 Mbit/sec.

| | Small file | Large file | Emacs compilation |
|---|---|---|---|
| SFSRO | 1.38 sec | 8.21 Mbyte/sec | 50.78 sec |
| Chefs | 1.52 sec | 5.13 Mbyte/sec | 51.43 sec |

Table 6.3: Small-file, large-file, and emacs-compilation microbenchmarks of Chefs versus SFSRO. In all tests the server has a warm cache, and the client has a cold cache.

The dummynet [96] driver in FreeBSD was used to simulate cable modem and analog modem network conditions. For the cable modem on the publisher machine, the round-trip delay was set to 20 msec and the download and upload bandwidth to 4 Mbit/sec and 384 Kbit/sec respectively. For the analog modem, the round-trip delay was set to 200 msec and the upload and download bandwidth each to 56 Kbit/sec.

In the Chefs measurements, the inode cache has 16,384 entries, a directory block cache has 512 entries, an indirect block cache has 512 entries, and a file block cache has 64 entries. A large file block cache is unnecessary because the NFS loopback server performs most of the file data caching.

For each measurement, the median result of five samples are reported. Error bars indicate minimum and maximum samples.

### 6.2.2 Microbenchmarks

Table 6.3 presents the same small-file, large-file, and emacs-compilation benchmarks described in Section 6.1 on the new experimental environment. SFSRO is compared to Chefs to evaluate the cost of adding confidentiality. Measurements of key regression algorithms show the relative performance of KR-SHA1, KR-AES, and KR-RSA.

The SFSRO and Chefs small-file benchmarks each generate 2,022 RPCs to fetch and decrypt content from a server (1,000 files, 10 directories, and one root directory — each generating two RPCs: one for the inode, one for the content). The measurements below show that there is a performance cost to adding confidentiality in Chefs for I/O-intensive workloads, but that the cost is not noticeable in application-level benchmarks.

The SFSRO and Chefs small file benchmarks finish in 1.38 seconds and 1.52 seconds respec-

| Key regression protocol | Winds/sec | Unwinds/sec |
|---|---|---|
| KR-SHA1 | Not applicable | 687,720 |
| KR-AES | Not applicable | 3,303,900 |
| KR-RSA | 158 | 35,236 |

Table 6.4: Microbenchmarks of KR-SHA1, KR-AES, KR-RSA key regression.

tively. The small overhead in Chefs comes as a result of decrypting content with 128-bit AES in CBC mode, downloading a 20-byte key regression member state from the publisher, and decrypting the member state with 1,024-bit RSA. In this local area network, the network latency accounts for nearly 30% of the overall latency; 2,022 RPCs with a 249 $\mu$sec round-trip time yields 503 msec. Content distribution networks are more commonly found in wide-area networks, where longer round-trip times would absorb the cost of the cryptography in Chefs.

The large-file benchmark generates 5,124 RPCs to fetch 40 Mbytes of content from the server (two RPCs for the root directory, two for the file, and 5,120 for the file content). The cost of cryptography in Chefs comes at a cost of 3.08 MByte/sec in throughput. The Chefs client takes approximately 32% of the client CPU, whereas the SFSRO client takes only 14% of the CPU.

The software distribution benchmark consists of an Emacs version 21.3 compilation as described by Section 6.1.3. The source code is stored in the file system, while the resulting binaries are written to a local disk. The experiment mounts the remote file system, runs `configure`, then compiles with `make`. This CPU-intensive workload requires access to approximately 300 files. The cost of cryptography is no longer noticeable. The Chefs client program consumes less than 1% of the CPU while the compiler takes nearly 90% of the CPU.

Table 6.4 displays the performance of basic key regression operations. The internal block size of the hash function matters significantly for the throughput of KR-SHA1 key regression. Because SHA1 uses an internal 512-bit block size, hashing values smaller than 512 bits results in poorer throughput than one would expect from SHA1 hashing longer inputs. Contrary to conventional wisdom, KR-AES can perform more than four times as many unwinds/sec than KR-SHA1.

### 6.2.3 Secure content distribution on untrusted storage

The large-file, small-file, and emacs compilation microbenchmarks evaluate Chefs in a static environment. Because files and membership do not change, the cost of unwinding in key regression does not appear. A standard benchmark is not available for measuring the effects of group membership dynamics. Therefore, this thesis evaluates Chefs based on how a client might search for content in a subscription-based newspaper.

**Searching encrypted content.** The benchmarks were inspired by the membership dynamics reported at Salon.com, a subscription-based online journal[1]. Salon announced that in the year 2003, they added 31,000 paid subscribers (for a total of 73,000) and maintained a 71% renewal rate. Thus, a 29% eviction rate would generate an expected 21,170 evictions in one year. This suggests that the total number of membership events would reach 52,170.

To represent a workload of searching newspaper content, the experiment tests a file system containing 10,000 8 KB encrypted files and the associated content keys. The experiment consists of mounting the file system and reading all the files. This causes the client machine to fetch all the content keys.

While there is promising research in how to search encrypted content [111], the untrusted server cannot perform the search because a client could not believe in the response. For instance, an untrusted server could respond, "No results found." Moreover, the server is not able to selectively return ciphertexts that would match the search. The server would still have to prove to the client that no other matching ciphertexts exist. Because Chefs extends the SFS read-only file system, it inherits the property that the client can verify when it has received all intended content (i.e., the whole truth) from the server. Therefore, the Chefs client downloads all the encrypted content and keys to perform the search itself.

**Sous-Chefs.** To determine the cost of key regression, Chefs is compared to a version of Chefs with key regression disabled. This strawman file system is called Sous-Chefs[2]. Chefs and Sous-

[1] `http://www.salon.com/press/release/`

[2] Pronounced "sioux chefs," a sous-chef is an assistant to the chef.

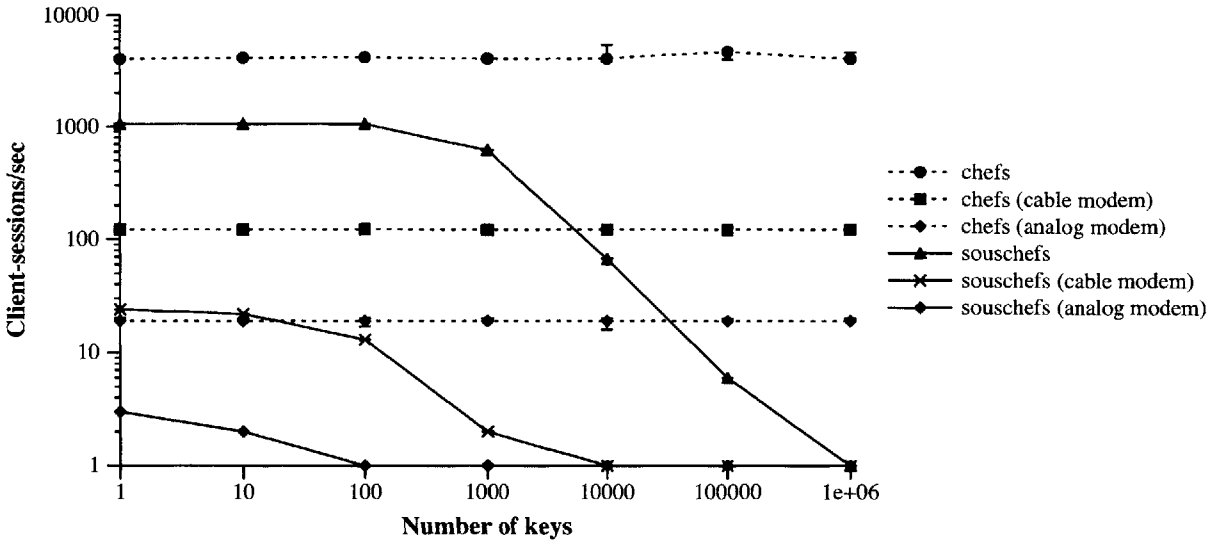


Figure 6-6: Aggregate publisher throughput for key distribution plotted on a log-log graph. A client-session consists of fetching key material sufficient to generate all the keys to decrypt the published content. Key regression enables a publisher to support many client-sessions per second. Chefs always performs better than Sous-Chefs because key regression performance is effectively independent of the rate of membership turnover.

Chefs differ only in how they fetch group keys from the publisher. When using KR-SHA1 for key regression, Chefs fetches a 20-byte member state, encrypted in the client's public 1,024-bit RSA key with low exponent $e = 3$. Chefs then uses key regression to unwind and derive all past versions of the group key. Sous-Chefs fetches all the derived group keys at once (each 16 bytes). The group keys themselves are encrypted with 128-bit AES in CBC mode. The AES key is encrypted with the client's RSA public key. A Sous-Chefs client is allowed to request a single bulk transfer of every version of a group key to fairly amortize the cost of the transfer.

**Reduced throughput requirements.** The log-log graph in Figure 6-6 shows that a publisher can serve many more clients in Chefs than Sous-Chefs in low-bandwidth, high-latency conditions. The CPU utilization for Chefs under no bandwidth limitation is negligible, indicating that the cost of RSA encryptions on the publisher is not the bottleneck.

The benchmark measured in Figure 6-6 effectively simulates the effect of 20 clients applying the same key distribution workload to the server. After all traces have completed, the effective

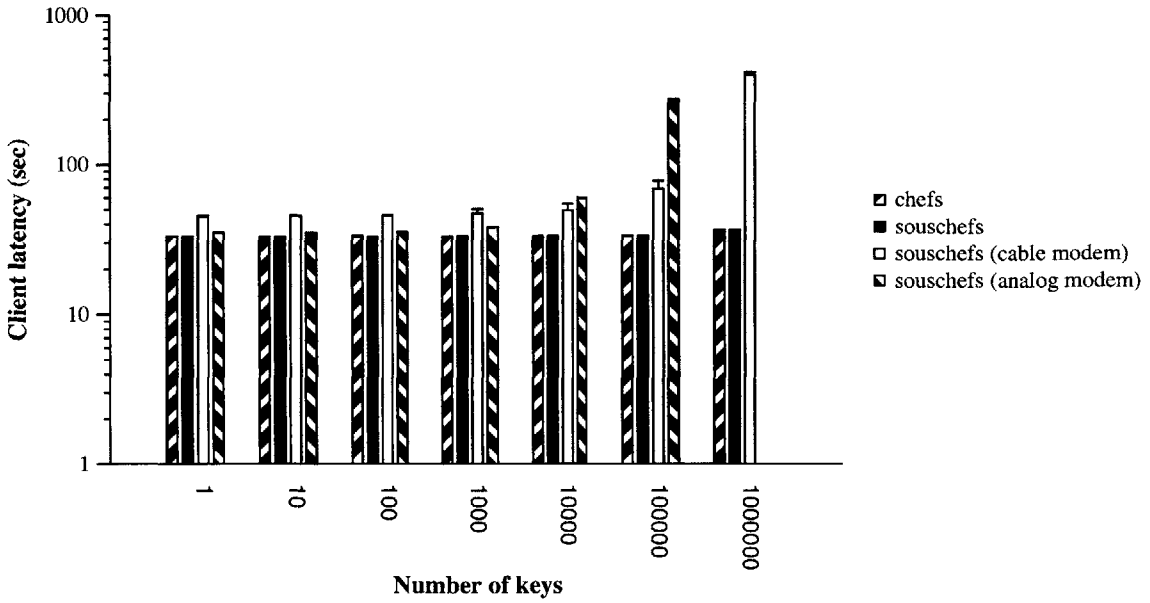


Figure 6-7: A log-log chart of single client latency to read 10,000 8 KB encrypted files and the associated content keys. Key regression maintains a constant client latency regardless of the number of keys. Under low-bandwidth, high-latency conditions, Sous-Chefs latency is dominated my the transfer time of keys after reaching 10,000 keys. Key regression enables much better latency in Chefs.

number of trace playbacks per second is recorded. Each test runs for 1–2 seconds, asynchronously playing back 20 traces of a single client fetching the keys for the search workload.

A Chefs trace consists of a TCP connection setup, followed by a *getkey* RPC. Chefs always generates a single *getkey* remote procedure call, regardless of the number of key versions.

A Sous-Chefs trace consists of a TCP connection setup, followed by a read of an encrypted file containing a set of keys. The file read is further composed of an *sfsconnect* RPC, a *getfsinfo* RPC, a *getkey* RPC, and a number of *getdata* RPCs sufficient to download the file of keys. The Sous-Chefs traces of fetching 1, 10, $10^2$, $10^3$, $10^4$, $10^5$, and $10^6$ keys generate 4, 4, 4, 5, 24, 200, and 1,966 asynchronous RPCs respectively.

Over fast network connections, the cost of transferring the 10,000 8 KB files dominates the client latency. A new trend appears after 100 transferred keys in the measurements of Sous-Chefs in Figure 6-6. The network bandwidth and latency of the publisher begin to dominate the client

latency. For instance, Sous-Chefs running on the simulated cable modem with 100 keys results in a publisher having 13 client-sessions/sec. This measurement meets the expectations. With a 384 Kbit/sec upload bandwidth, 20 ms round-trip delay, and the transfer of 100 keys each of size 16 bytes using 4 RPCs, one would expect a single client to take at least 50 msec simply to download the keys. This translates to at most 20 client-sessions/sec under perfectly asynchronous RPC conditions—confirming the measurements as reasonable.

**Improved client latency.** The client latency experiment measures the time for a single client to execute the search workload. The untrusted server and publisher have warm caches while the client has a cold cache.

The log-log chart in Figure 6-7 shows that Chefs outperforms Sous-Chefs for the search workload under several network conditions. In Sous-Chefs, the network transfer time dominates client latency because of the sheer volume of keys transferred from the publisher to the client. There is no measurement for Sous-Chefs downloading 1,000,000 keys because the kernel assumes that the mount failed after waiting 1,000 seconds. On a 56 Kbit/sec network, Sous-Chefs is expected to take over 2,232 seconds to download 1,000,000 keys each 16 bytes. Key regression itself is a small component of the Chefs benchmark. With $10^6$ keys, key regression on the client takes less than 1.5 sec with CPU utilization never exceeding of 42%.

# Chapter 7

# Related work

*What does that have to do with operating systems?*

– Andrew Tanenbaum, during Q/A at the ACM Symposium on Operating Systems Principles (SOSP), St. Malo, France, 1997

SFSRO and Chefs extend research in both computer systems and cryptography. Section 7.1 begins with a discussion of related work in secure file systems. SFSRO and Chefs support sharing of single-writer, many-reader content. This style of sharing lends itself conveniently to the application of content distribution. Section 7.2 examines the properties of already deployed content distribution systems, and draws comparisons with SFSRO and Chefs. Finally, Section 7.3 explains the cryptographic fundamentals that SFSRO and key regression rely upon, and other secure systems that use these fundamentals in a similar manner.

## 7.1 Secure file systems

Several file systems provide secure sharing, using various notions of untrusted storage. The following sections discuss file systems that use encrypted storage to protect local storage or networked storage.

### 7.1.1 Local cryptographic storage

Cryptographic storage can provide confidentiality for personal storage. For instance, a user may encrypt an entire file system so that an intruder who gains access to the hard drive cannot read any content. This section reviews file systems that provide confidentiality of local storage by encrypting.

**Cryptographic File System (CFS).** Blaze's CFS implements cryptography inside a file system [19]. As a user-level process, CFS encrypts content at the file system level, rather than at the level of individual files. A user sets a password to protect each directory's content. A user would worry less if a hard drive is compromised, because an adversary without the password is unable to make sense of the encrypted hard drive. CFS provides protection of personal storage in a single-writer, single-reader model, but does not address the problem of how to securely share files over a network. Chefs combines cryptographic storage with key regression and lazy revocation to provide efficient sharing of single-writer, many-reader content.

**Transparent cryptographic file system (TCFS).** TCFS extends CFS for transparent group sharing within cryptographic storage [28]. If a threshold number of users are logged into a given machine, then those users can access encrypted files. Chefs adopts the traditional notion of group sharing where a group member can access any file designated to the group—independent of whether other group members are logged in.

**NCryptfs.** NCryptfs [124] provides confidentiality by layering an encrypted file system on top of any existing file system. The in-kernel implementation of NCryptfs allows for better client performance than user mode file systems such as SFSRO and Chefs.

Users of NCryptfs inherit the sharing semantics of the underlying file system. Layered above a local file system, NCryptfs lets local users share files encrypted on disk. Layered above the client side of a network file system, NCryptfs lets users share files across the network. Key management is not part of the NCryptfs design.

### 7.1.2 Networked cryptographic storage

SFSRO and Chefs are network file systems that provide security for single-writer, many-reader content. Several other network file systems provide various notions of security for many-reader and either single-writer or many-writer content. To support single-writer, many-reader content, SFSRO and Chefs use separate mechanisms for reading and writing. Clients read content from a replicated file system, but publishers write content by using a command-line database generator.

**General purpose file systems for trusted storage.** Some file systems provide high security (e.g., the SFS read-write file system [73] or Echo [18]), but compared to the SFS read-only file system these servers do not scale well with the number of clients because their servers perform expensive cryptographic operations in the critical path. For instance, the SFS read-write server performs one private-key operation per client connection, which takes about 24 msec on a 550 MHz Pentium III.

**Secure Untrusted Data Repository (SUNDR).** SUNDR [68, 72, 74] enables integrity protection of shared read-write content even when malicious parties control the network storage. SUNDR and Chefs have several architectural similarities because the initial implementation of Chefs[1] was built on SUNDR instead of SFSRO. The Chefs implementation switched to using SFSRO because single-writer content is sufficient to demonstrate the benefits of key regression. Both systems implement a client daemon and block store (called an untrusted server in Chefs). The block store provides an extremely simple, disk-like interface for the client to store and fetch data blocks. The client presents a file system interface for users to store and retrieve files.

SUNDR does not provide confidentiality or read-access control. SUNDR's objectives are to protect the integrity and freshness of content. In particular, SUNDR provides the notion of fork consistency, whereby clients can detect any integrity or consistency failures as long as they see each other's file modifications. SUNDR includes a consistency server that maintains versioning information to detect forking attacks and damage to integrity. SFSRO and Chefs do not consider fork consistency because they are single-writer file systems.

[1] At the time called SUNDRIED.

**Plutus.** Plutus also introduced the notion of key rotation, the precursor to key regression. The key regression protocols in Chapter 4 evolved from the protocol originally proposed by Plutus. While the design of Plutus explains key rotation, the Plutus implementation itself did not use key rotation. Chefs provides an implementation of cryptographic storage that uses key regression.

Like Chefs, Plutus uses lazy revocation [43] to reduce the cost of file re-encryption following a member eviction. We chose SFSRO instead of Plutus to implement key regression because SFSRO offered a code base free of intellectual property restrictions.

**Decentralized access control.** Several systems already use encryption to provide access control. For instance, one could consider PGP-encrypted email as form of decentralized access control. Only those in possession of the decryption key can determine the plaintext message. The encrypted message travels through untrusted mail servers, similar to the untrusted servers in SFSRO. The same technique works for sending secure messages over radio and USENET newsgroups. However, these are all examples of secure *communication* over untrusted networks. This thesis focuses on secure *storage* on untrusted servers.

The Swallow [90] read-write object store suggests the use of encryption to provide what this thesis calls decentralized access control. Svobodova and Reed envisioned an access control system based on encrypting files. Only parties with the proper decryption key could access the objects. Swallow uses encryption to both protect against unauthorized modification and unauthorized access to data. In our file systems, we have split these goals into SFSRO and Chefs respectively. Unfortunately, Swallow was never completed because of funding constraints.

Gifford [46] calls the model of decentralized access control by encryption "passive protection" and discusses the problem of revoking access to encrypted content — the same problem that motivated lazy revocation and thus key regression. Miklau and Suciu [79] propose a similar mechanism to protect access to XML documents using encryption.

Harrington and Jensen revive the goals of Swallow and explain how Cryptographic NFS (CNFS) uses "cryptographic access control" to protect access to shared read-write content [54]. Like the SFSRO server, a CNFS server provides an immutable block store. Rather than use reference monitors to mediate access, CNFS encrypts content for confidentiality. In Chefs, we call this type of

mediation decentralized access control.

**Block-level security.** SFS read-only utilizes a block store and protects content both at the block level and as a file system unit. Several other file systems add security at the block-level. Oprea et al. [85] present methods to add block-level integrity without changing the block size. Aguilera et al. [4] propose a scheme for adding security to network-attached disks without changing disk layout. SFS read-only changes the disk layout, but uses an NFS loopback server to localize the changes.

**Versioning file systems.** The ext3cow file system extends the Linux ext3 file system to support individual file versioning [87]. A modification to ext3cow stores all content encrypted on disk to support access control and secure deletion [88]. The ext3cow file system efficiently deletes the entire plaintext of a file by removing only a small "stub" of the ciphertext[2]. Without the ciphertext stub, an adversary given the remaining ciphertext is unable to retrieve the plaintext. The stub approach is closely related to authenticated encryption [15] and all-or-nothing transforms [22] where decryption cannot begin until after receiving all ciphertext.

A stub-enabled version of Chefs could provide revocation rather than eviction of members. A content publisher would distribute both group keys and ciphertext stubs. The untrusted servers would replicate all ciphertext except the stubs. To revoke a member, the content publisher would replace the stub with a new stub—effectively making the ciphertext on the untrusted servers useless to revoked members.

Venti [89] provides an immutable block store similar in spirit to the SFSRO server, but Venti focuses on archival storage and not security. Versioning file systems could use Venti as a building block. The Cedar file system [47] provides a system of immutable files. Like the SFS read-only file system, Cedar communicates block requests over the network and has the client synthesize file system structures from these blocks.

[2]The "stub" was privately revealed as a tongue-and-cheek gesture to Adam Stubblefield.

**Cepheus.** Cepheus [43] is a read-write file system that provides group sharing of files by encrypting file keys in lockboxes. A lockbox contains a public-key encryption of the file key. A file will have a lockbox for each member. This method does not scale well for large groups. Moreover, the owner must compute one public-key encryption for each remaining member following an eviction. Chefs uses both lockboxes and lazy revocation, originally proposed by Cepheus under the deprecated term "delayed re-encryption."

**SiRiUS.** The SiRiUS file system [49] enables read-write group sharing of encrypted files on untrusted storage. Like Cepheus, each file in SiRiUS includes a list of key lockboxes—one for each group member. To cope with large groups, the SiRiUS-NNL dialect uses the NNL [81] group key distribution protocol. The dialect allows SiRiUS to keep the number of lockboxes independent of the group size, but the number of lockboxes will grow linearly with respect to the cumulative number of evictions. Chefs desires to keep lockboxes constant in size, independent of both the group size and the rate of membership turnover. As a result, Chefs uses key regression instead of NNL.

In addition to confidentiality, SiRiUS provides integrity protection and freshness of data by using a Merkle hash tree and time stamps respectively. SFSRO uses the same techniques to provide integrity and freshness.

**OceanStore.** Pond, the OceanStore prototype, provides read-write sharing of collections of immutable objects replicated on semi-trusted servers [63, 92]. To provide confidentiality, the OceanStore design encrypts content before replication. Users with the decryption key can read the content. To provide integrity, Pond uses proactive threshold signatures. As long as the majority of a set of semi-trusted servers remain honest, a threshold signature provides for authenticity and integrity of content. SFSRO and Chefs use similar methods to provide authenticity, integrity, and confidentiality, but OceanStore does not use key regression to manage group keys as members come and go.

**Andrew File System (AFS) read-only volumes.** Many sites use a separate file system to replicate and export read-only binaries, providing high availability and high performance. AFS supports read-only volumes to achieve replication [103]. However, in all these cases replicated content is stored on trusted servers. SFSRO and Chefs are designed to operate on untrusted storage.

**Secure replication.** Reiter and Birman [91] describe methods to securely replicate services even if several servers and clients are compromised. Similarly, Herlihy and Tygar [55] discuss ways to make replicated data secure — including a method of re-encryption. SFS read-only takes a different approach in that a compromise will not damage security because it already assumes untrusted servers replicate content. Clients will never accept unauthentic content and will at worst be denied service if too many servers behave maliciously. By encrypting content with convergent encryption, Farsite [3] functions as a centralized file server even though it is replicated on untrusted hosts.

## 7.2 Content distribution networks

SFSRO and Chefs are content distribution networks because they support replication of single-writer, many-reader content. The following paragraphs describe how other content distribution systems relate to SFSRO and Chefs.

**Web content distribution.** Content distribution networks deployed by companies such as Akamai [5] and research initiatives such as Coral [40] are an efficient and highly-available way of distributing static Web content. Content stored on these networks are dynamically replicated on trusted servers scattered around the Internet. Web browsers then connect to a cache that provides high performance. The approach of SFSRO allows for secure replication of Web content on *untrusted* servers. Users without an SFSRO client could configure a proxy Web server on an SFSRO client to serve the `/sfs` directory, trivially creating a Web-to-SFSRO gateway for any Web clients that trust the proxy.

**Software distribution.** Signed software distributions are common in the open-source community. In the Linux community, for example, a creator or distributor of a software package can sign RPM [100] files with PGP or GNU GPG. RPM also supports MD5 hashes. A person downloading the software can optionally check the signature or hash. RedHat Software, for example, publishes their PGP public key on their Web site and signs all of their software distributions with the corresponding private key. This arrangement provides some integrity and authenticity guarantees to the person who checks the signature on the RPM file and who makes sure that the public key indeed belongs to RedHat. However, RPMs do not provide an expiration time or revocation support. If RPMs were distributed on SFSRO servers, users were running an SFSRO client could transparently verify the RPMs for integrity, authenticity, and freshness.

**Distributing popular content with BitTorrent.** A peer-to-peer file sharing system, BitTorrent is a decentralized content distribution system for individual, public files [30]. Clients and servers are not distinguishable. All clients are servers, and all servers are clients. In SFSRO, a client can access content without having to also serve the content. Moreover, a server can replicate content without running a client.

Both SFSRO and BitTorrent implement scalable replication of public content on untrusted servers. Whereas BitTorrent concentrates on replication of popular content, SFSRO and Chefs focus on secure distribution of files. SFSRO addresses problems of storage, file system integrity, incremental updates of dynamic content, and distribution of large collections of files. BitTorrent addresses the problems of distributed caching of content, file integrity, on-demand replication of static content, and distribution of large individual files.

**Secure Web servers.** Secure Web servers are another example of servers that provide access to mostly single-writer, many-reader data. These servers are difficult to replicate on untrusted machines, however, since their private keys have to be online to prove their identity to clients. Furthermore, private-key operations are expensive and are in the critical path: every SSL connection requires the server to compute modular exponentiations as part of the public-key cryptography [41]. As a result, software-only secure Web servers achieve low throughput. For instance, our

Apache SSL server in Chapter 6 accepted only 11 new connections per second.

**SSL splitting.** SSL splitting [65] is a technique to provide integrity and freshness of public Web content distributed by untrusted servers. The technique allows a publisher on a low-bandwidth connection to reach a large audience without compromising integrity or freshness. Clients downloading content are limited by the aggregate bandwidth made available by servers, rather than by the bandwidth of the publisher's server.

Key regression in Chefs is motivated by the same goal as SSL splitting. Namely, key regression allows a content publisher on a low-bandwidth connection to make content available to many clients. Unlike SSL splitting, Chefs provides decentralized access control of private content rather than public content.

**Secure DNS.** Secure DNS [36] is an example of a read-only data service that provides security, high availability, and high performance. In secure DNS, each individual resource record is signed. This approach would not work for distributed file systems. If each inode and 8 Kbyte-block of a moderate file system—for instance, the 635 Mbyte RedHat 6.2 i386 distribution—were signed individually with a 1,024-bit Rabin-Williams key, the signing alone would take about 36 minutes ($90,000 \times 24$ msec) on a 550 MHz Pentium III. A number of read-only data services, such as FTP archives, are 100 times larger, making individual block signing impractical—particularly since we want to allow frequent updates of the database and rapid expiration of old signatures.

## 7.3 Cryptography

SFSRO and key regression rely on cryptography for integrity protection, confidentiality, and the security of key regression. The first section reviews fundamental notions of security from theory of cryptography—central to the design of SFSRO and key regression. The following section discusses related applications of cryptography that share in the goals of SFSRO and key regression.

### 7.3.1 Fundamental cryptographic notions

**Related-key attacks.** Extending the notion of secure symmetric encryption, Bellare and Kohno investigate Related-Key Attacks (RKAs) against block ciphers [13]. In an RKA, the adversary has access to a related-key oracle that takes not only a plaintext message, but also a related-key-deriving function. The adversary can encrypt messages not with chosen keys, but with chosen functions of a hidden key.

Because key regression produces a set of related encryption keys, the notion of security for RKAs may help to craft a better definition of security for key regression. Key regression could be viewed as a particular related-key-deriving function.

**Re-keying.** Re-keyed symmetric encryption allows for two parties to derive a sequence of keys from one shared master key, effectively extending the lifetime of the master key. Examples of such systems (discussed in this chapter) include Micali's key trees for key escrow, Snoeren's session keys, and several multicast key agreement protocols. The definition of security for key regression is modeled after the notion of security for re-keyed symmetric encryption [1] in that key regression substitutes for a key generation algorithm. Whereas Abdalla and Bellare propose a tree-based scheme for deriving keys from a master key [1], key regression could be thought of as producing a chain of keys, each derived from other elements in the chain.

**On-line ciphers.** An on-line cipher takes as input an arbitrary length plaintext, and will output the $i$th block of ciphertext after having only processed the first $i$ blocks of plaintext [11]. On-line ciphers share with key regression the security notion of "behaving as randomly as possible."

The designers of on-line ciphers found that no on-line cipher could satisfy the existing security definition based on pseudorandomness. Namely, an adversary can trivially distinguish between the output of an on-line cipher and a pseudorandom function. For this reason, on-line ciphers use a weaker notion of security.

Key rotation [58] suffers from the same problem. The correctness requirement (that unwinding undoes winding) makes any key rotation protocol insecure in the pseudorandom sense. Consider the task of distinguishing between a random sequence of keys and a sequence generated by key

rotation. Given a challenge sequence, an adversary simply unwinds a key from the sequence. If the previous key matches the unwound key, then key regression generated the sequence. Thus, key regression communicates member states rather than keys. Clients run a key derivation algorithm on a member state to compute the actual key.

In a similar spirit, Shamir [105] explains how to securely generate a sequence of pseudorandom values by using RSA. The security goal is to prevent an adversary given some sequence of keys from deriving other keys in the sequence. In key regression, we require a stronger notion. Namely, it should be infeasible for an adversary given a new value to guess whether that value is a real key or random bit string with probability greater than 50%.

### 7.3.2 Cryptographic applications

Many systems use cryptography to accomplish the similar goals shared by SFSRO and key regression.

**One-time passwords and hash chains.** Key regression borrows from the techniques used in one-time passwords [66]. Namely, the publisher precomputes a hash chain of group keys. Just as with one-time passwords, the hash function limits the number of times a publisher can wind the key forward. Recent efforts have shown how to more efficiently traverse such hash chains [31]. These techniques may improve the efficiency of hash-based key regression for both owners and members.

One-way functions can generate a sequence of keys in a hierarchy [6, 69], but do not necessarily provide for computational indistinguishability. Gudes [51] uses one-way functions to protect content in a file system by repeatedly encrypting data with keys derived from a hash chain.

**Key derivation in key escrow.** Micali introduces a key derivation construction for time-bounded key escrow with fair public key cryptosystems [78]. Similar to key regression, time-bounded escrow uses a single key to effectively grant access to a sequence of keys. An escrow agent can give authorities a small piece of information to allow derivation (and therefore eavesdropping) of a set of keys in use during a given time period. With a secure hash function $h : \{0,1\}^k \rightarrow \{0,1\}^{2k}$,

one can create a $d$-level binary hash tree rooted with a $k$-bit master key. Each of the $2^d$ leaves represents a key. A parent node can derive its children keys. This technique of handing out one key to cover a set of subkeys is similar in spirit to the NNL protocol [81]. By giving out only $d$ nodes in the tree (logarithmic in size with respect to the number of keys, $2^d$), a publisher can give access to any contiguous sequence of keys in the leaves of the tree.

Chefs does not use Micali's scheme because the publisher must be able to function with a low-bandwidth connection. Key regression makes the bandwidth consumption of key distribution effectively independent of the number of evictions, at the cost of increasing the computational demands on a member's machine.

**Forward security.** Anderson describes the notion of forward security [7] for signatures [2, 14, 62] and encryption [25]. An extension of perfect forward secrecy [34], forward security, ensures that exposure of a key will not cause exposure of messages encrypted with past keys. This property limits the damage resulting from key compromise. For instance, the compromise of a session key should not expose the contents of past communication.

Key regression provides effectively the opposite property of forward-secure encryption. A single key allows derivation of all past keys, but not future keys. In fact, Anderson describes a simple hash chain for forward-secure encryption [7]. To compute a future key, one hashes the current key. A version of hash-based key rotation uses the same construction, but hashing backwards instead of forwards. Old keys are hashes of new keys.

**Merkle hash tree.** SFSRO makes extensive use of Merkle hash trees, which have appeared in numerous other systems. Merkle used a hierarchy of hashes for an efficient digital signature scheme [77]. In the context of file systems, the Byzantine-fault-tolerant file system uses hierarchical hashes for efficient state transfers between clients and servers [26, 27]. Cepheus [43] uses cryptographic hashes in a similar fashion to SFSRO, except that the Cepheus guarantees integrity for individual files rather than a complete file system. Duchamp uses hierarchical hashes to efficiently compare two file systems in a toolkit for partially-connected operation [35]. TDB [70] uses hash trees combined with a small amount of trusted storage to construct a trusted database system

on untrusted storage. A version of Network-Attached Secure Disks (NASD) uses an incremental "Hash and MAC" scheme to reduce the cost of protecting the integrity of read traffic in storage devices that are unable to generate a MAC at full data transfer rates [48].

A number of proposals have been developed to make digital signatures cheaper to compute [45, 98], some involving hash trees [122]. These proposals enable signing hundreds of packets per second in applications such as multicast streams. However, if applied to file systems, these techniques would introduce complications such as increased signature size. Moreover, because SFSRO was designed to avoid trusting servers, read-only servers must function without access to a file system's private key. This prevents any use of dynamically computed digital signatures, regardless of the computational cost.

**Multicast security.** There are several ways to improve the performance of access control using untrusted servers when key distribution is on the critical path. For instance, multicast security and broadcast encryption often make the key distribution process itself efficient. The approach in key regression is simply to reduce the number of keys that need to be distributed in the first place.

Group key distribution is often discussed in the context of multicast security [106, 107]. These protocols wish to efficiently distribute a new group key to remaining group members after an eviction — as in tree-based group key agreement [60]. The NNL revocation protocol [81] uses a hash tree for key updates. Each group member has a leaf in the tree. The parent nodes are computed by hashing children. Initially, the group key is the root node. Messages are encrypted with this group key. After an eviction, the evicted member must not know the new group key. NNL splits the tree in two pieces, such that remaining group members can compute at least of the two roots. Each message requires two encryptions: one for each root. Consequently, a key update message is linear in the number of evicted members. However, the NNL protocol becomes impractical as members leave. The key tree becomes fragmented by evictions, and the size of the ciphertext grows linearly with the number of evictions. The NNL protocol works best in large groups expecting few evictions relative to the number of users. One of the goals of key regression is to keep all operations constant in space and time. The NNL protocol does not satisfy this goal.

**Broadcast encryption.** In broadcast encryption [38], a group owner can efficiently evict a subset of the group members without explicitly distributing a new set of keys to individual, remaining group members.

Similar to broadcast encryption, a non-interactive group key distribution protocol does not require a group owner to communicate with each group member individually after an eviction. Instead, the owner distributes a new group key by either contacting a small subset of the members or by broadcasting a message. ELK [86], based on LKH [53, 118, 123], is one protocol that provides non-interactive group key distribution.

The author of this thesis developed the first (yet unpublished) key rotation construction in Java together with non-interactive group key distribution by adapting the Naor-Pinkas traitor tracing protocol [82]. Because the non-interactivity added significant complexity to the protocol and proofs, the author decided to separate the key distribution mechanism from key rotation.

**Self-healing key distribution.** Self-healing key distribution with revocation [80, 113] protocols are resilient even when broadcasts are lost on the network. A user who misses a small number of key update messages can reconstruct the lost key. In this manner, one can view key regression as having the self-healing property. The self-healing protocols are resistant to collusion. Because all group members in key regression have the same secret information, there is no reason to collude. Self-healing protocols run in time linear in the number of colluding users. Key regression requires that basic operations run in constant time and space.

A second feature of self-healing is that group members can recover missing keys within a given window of time. Kurnio et al. [64] and Yang et al. [125] propose mechanisms for similarly coping with key update messages over a lossy network. Key regression protocols can effectively self-heal in perpetuity.

**Evolving session keys.** Snoeren uses SHA1 hashing to compute a sequence of session keys that protect migrated TCP connections [110, p. 111]. Two parties securely negotiate an initial symmetric key. To generate a session key for a new connection, each party computes a one-way function of the symmetric key and two public sequence numbers that change for each new connection. Sno-

eren argues that the one-wayness property makes it difficult for an adversary who sees the public sequence numbers (but not the symmetric key) to compute session keys.

**Master keys.** In key regression, we impose a linear order on group keys with respect to time for a single service. That is, access to a service at one time implies access to the same service at an earlier time, but not necessarily future times.

Master key systems and several group key establishment protocols impose a partial order on group keys with respect to subordinate services or group members. That is, some members are deemed to have more authority than other members. A member at the top of a tree of authority can derive the key of any member within the tree. For instance, Chick and Tavares show how to create a tree-based master key scheme to control access to a hierarchy of services [29]. A node in the tree represents a one-way function of its children. Thus, the root key allows the possessor to derive any key in the tree. Inner nodes give access only to keys beneath the node in the tree.

**Key distribution.** Blaze [20] and Saltzer [101] describe the challenges specific to storage security that often do not appear in communications security. We keep the support of key management in Chefs as simple as possible by using key regression and simple access control policies (e.g., access to all the files or none of the files).

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 8

# Conclusion

*It's Hard To Finish Papers.*
– Ira Haverson & Tiffany Fulton-Pearson

This thesis presented the design, implementation, and measurements of the SFS read-only file system and provably-secure key regression in the Chefs file system.

The SFS read-only file system is a distributed file system that makes extensive use of Merkle hash trees [76] to allow a high number of clients to securely access public, read-only content. The content of the file system is stored in an integrity-protected database, which is signed offline with the private key of the file system. The private key of the file system does not have to be online, allowing the database to be replicated on many untrusted machines. To allow for frequent updates, the database can be replicated incrementally. The read-only file system pushes the cost of cryptographic operations from the server to the clients, allowing read-only servers to be simple and to support many clients. Measurements of an implementation confirm that the read-only file system can support a large number of clients downloading integrity-protected content, while providing individual clients with acceptable application performance.

Chefs extends the SFS read-only file system with key regression to enable decentralized access control of private content replicated on untrusted servers. Key regression allows a client to derive a sequence of past keys from the most recent key. Only the publisher can compute future keys, which

are distributed to clients on-demand. This thesis presented provably-secure constructions for key regression using SHA1, AES, and RSA. Measurements of client latency and server throughput in Chefs demonstrate that key regression enables efficient key distribution for publishers on low-bandwidth, high-latency connections. Using key regression, a publisher can efficiently control access to content effectively independent of group membership dynamics and without needing a fast network connection.

## 8.1 Future work

This thesis leaves a number of topics for future work:

**Windowing in key regression.** It may be desirable to limit how far back in time a member can unwind a key. For instance, a group member may only be able to derive keys as far back in time as when the member first joined. A publisher may want to prevent a new member from reading deleted files left on tape backups or from reading not-yet declassified files. A key regression protocol could achieve this goal by establishing a different window for each group member. One straightforward approach uses two hash chains of member state flowing in opposite directions. One chain is easy to compute backwards (traditional member state in key regression), and one chain is easy to compute forward (proposed windowing). A member could receive an element from the forward windowing chain at join time. Only where the chains overlap could the member derive content keys.

Windowing itself presents additional challenges. For instance, collusion now becomes a problem. Two members with disjoint access to the set of key versions could combine their hash chains to derive keys to which neither should have access. Moreover, it is not clear what semantics are desirable when a member joins a group, gets evicted, and later rejoins the group. Should the member have access to key versions that protect content created during the period of exile?

**Improvements to key regression.** The performance of the key regression algorithms in Chapter 4 could be improved with a batch key unwinding function. For instance, the unwinding algorithms are tailored to sequential access patterns. When a member wants to derive a single key in

the past, data structures other than a simple hash chain may be more desirable [31, 56].

**Proxy re-encryption.** After a publisher evicts a member, files encrypted with the old group key are still available to the evicted member. Chefs uses lazy revocation such that the evicted member cannot see future updates. One could re-encrypt files in a new key to expedite the eviction. With proxy re-encryption [8], the work of re-encrypting can be offloaded from the publisher to untrusted proxy machines; the proxy translates ciphertexts into new ciphertexts without seeing the underlying plaintext or knowing any private keys.

**Workloads.** Many surveys characterize file system workloads in terms of the number, size, and distribution of files. There is little work on how such workloads interact with access control list dynamics. Gathering data on group dynamics would help to determine the most appropriate workloads for key regression.

We had great difficultly obtaining statistics on group dynamics from Web site operators. We contacted online newspapers and a content network distribution company, but were unable to obtain any meaningful statistics without first signing a non-disclosure agreement. We tried analyzing SEC financial statements, records from the Audit Bureau of Circulations, and public relations newswires to estimate basic workload characteristics. At best we could determine the growth of subscribership. Companies deem group dynamics as competitive information.

We recently received nearly a terabyte of logs containing four years worth of downloads from an adult entertainment Web site. We hope to convert the logs into a meaningful benchmark. If a few more companies donate anonymous Web logs of access-controlled content, we could develop a much better picture of how group dynamics would affect the performance of key regression and lazy revocation.

**Economics of decentralized access control.** One of the barriers to practical deployment of Chefs is an incentive system to convince volunteers to replicate encrypted content. After all, what server would want to replicate content that the server itself cannot read?

One way to construct an incentive is to bind popular content to unpopular content, forcing

volunteers to replicate the unpopular content before access is granted to the popular content. The all-or-nothing (AON) transform [94] is one method to help implement such semantics. The AON transform guarantees in an information theoretic sense that someone decrypting a message cannot obtain a single bit of the plaintext without first possessing every bit of the ciphertext. This effectively disables random access. Therefore, one could apply the AON transform to popular and unpopular content before encrypting for confidentiality. For the volunteer to access the popular content (e.g., music), it would first have to download the ciphertext of the unpopular content (e.g., encrypted photo galleries).

## 8.2 Code availability

Chefs and SFSRO are available from anonymous CVS on `http://www.fs.net/`. Feedback and improvements to the open-source software are welcomed.

# Bibliography


[1] Michel Abdalla and Mihir Bellare. Increasing the lifetime of a key: a comparative analysis of the security of re-keying techniques. In Tatsuaki Okamoto, editor, *Proceedings of Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[2] Michel Abdalla and Leonid Reyzin. A new forward-secure digital signature scheme. In Tatsuaki Okamoto, editor, *Proceedings of Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[3] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

[4] Marcos Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertliand, Dave Andersen, Mike Burrows, Timothy Mann, and Chandramohan Thekkath. Block-level security for network-attached disks. In *2nd USENIX Conference on File and Storage Technologies*, 2003.

[5] *Akamai Technologies.* `http://www.akamai.com/ (Last viewed August 28, 2005).`

[6] Selim G. Akl and Peter D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *ACM Transactions on Computer Systems*, 1(3):239–248, 1983.

[7] Ross Anderson. ACM CCS invited lecture: Two remarks on public key cryptology, April 1997. `http://www.cl.cam.ac.uk/ftp/users/rja14/forwardsecure.pdf` (Last viewed August 28, 2005).

[8] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. In *Proceedings of the Twelfth Network and Distributed System Security (NDSS) Symposium*. Internet Society (ISOC), February 2005.

[9] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, pages 394–403. IEEE Computer Society, 1997.

[10] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93: 1st Conference on Computer and Communications Security*, Lecture Notes in Computer Science, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press.

[11] Mihir Bellare, Alexandra Boldyreva, Lars Knudsen, and Chanathip Namprempre. On-line ciphers and the hash-CBC construction. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO '01*, volume 2139 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2001.

[12] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In Yvo G. Desmedt, editor, *Advances in Cryptology – CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 341–358. Springer-Verlag, 1994.

[13] Mihir Bellare and Tadayoshi Kohno. A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications. In E. Biham, editor, *Advances in Cryptology – EUROCRYPT '03*, volume 2656 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[14] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448. Springer-Verlag, 1999.

[15] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: relations about notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Proceedings of Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[16] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures—how to sign with RSA and Rabin. In U. Maurer, editor, *Advances in Cryptology—Eurocrypt 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer-Verlag, 1996.

[17] Mihir Bellare and Bennet Yee. Forward security in private key cryptography. In Marc Joye, editor, *Topics in Cryptology – CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 1–18, San Francisco, CA, USA, April 13–17, 2003. Springer-Verlag, Berlin, Germany.

[18] Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Technical Report 111, Digital Systems Research Center, Palo Alto, CA, September 1993.

[19] Matt Blaze. A cryptographic file system for UNIX. In *Proceedings of 1st ACM Conference on Communications and Computing Security*, 1993.

[20] Matt Blaze. Key management in an encrypting file system. In *Summer USENIX*, June 1994.

[21] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science (FOCS '82)*, 1982.

[22] Victor Boyko. On the security properties of OAEP as an all-or-nothing transform. In Michael Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[23] Brent Callaghan and Tom Lyon. The automounter. In *Proceedings of the Winter 1989 USENIX*, pages 43–51. USENIX, 1989.

[24] Brent Callaghan, Brian Pawlowski, and Peter Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[25] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. Cryptology ePrint Archive, Report 2003/083, 2003. `http://eprint.iacr.org/2003/083` (Last viewed August 28, 2005).

[26] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, LA, February 1999.

[27] Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, October 2000.

[28] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Giuseppe Persiano. The design and implementation of a transparent cryptographic file system for UNIX. In *FREENIX Track: USENIX Annual Technical Conference*, June 2001.

[29] Gerald C. Chick and Stafford E. Tavares. Flexible access control with master keys. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 316–322. Springer-Verlag, 1989.

[30] Bram Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the First Workshop on the Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[31] Don Coppersmith and Markus Jakobsson. Almost optimal hash sequence traversal. In *Proceedings of Financial Crypto 2002*, Southampton, Bermuda, March 2003.

[32] A. Desai, A. Hevia, and Y. Yin. A practice-oriented treatment of pseudorandom number generators. In *Advances in Cryptology - EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 368–383. Springer-Verlag, 2002.

[33] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT-22:644–654, November 1976.

[34] Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes, and Cryptography*, 2(2):107–125, June 1992.

[35] Dan Duchamp. A toolkit approach to partially disconnected operation. In *Proceedings of the 1997 USENIX*, pages 305–318, January 1997.

[36] Donald Eastlake and Charles Kaufman. Domain name system security extensions. RFC 2065, Network Working Group, January 1997.

[37] A. Evans, W. Kantrowitz, and E. Weiss. A user authentication scheme not requiring secrecy in the computer. *Communications of the ACM*, 17:437–442, August 1974.

[38] Amos Fiat and Moni Naor. Broadcast encryption. In *Advances in Cryptology – CRYPTO '93*, volume 773 of *Lecture Notes in Computer Science*, pages 480–491. Springer-Verlag, 1994.

[39] Warwick Ford and Michael S. Baum. *Secure electronic commerce*. Prentice Hall, 1997.

[40] Michael Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[41] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol version 3.0. Internet draft (draft-freier-ssl-version3-02.txt), Network Working Group, November 1996. Work in progress.

[42] John Friedlander, Carl Pomerance, and Igor Shparlinski. Period of the power generator and small values of carmichael's function. *Mathematics of Computation*, 70(236):1591–1605, 2001.

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

[44] Greg Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–17, January 1997.

[45] Rosario Gennaro and Pankaj Rohatgi. How to sign digital streams. In *Advances in Cryptology—CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 180–197. Springer-Verlag, 1997.

[46] David K. Gifford. Cryptographic sealing for information secrecy and authentication. *Communications of the ACM*, 25(4):274–286, 1982.

[47] David K. Gifford, Roger M. Needham, and Michael D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, 1988.

[48] Howard Gobioff, David Nagle, and Garth Gibson. Embedded security for network-attached storage. Technical Report CMU-CS-99-154, CMU, June 1999.

[49] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*. Internet Society (ISOC), Feburary 2003.

[50] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences (JCSS)*, 28(2):270–299, April 1984.

[51] E. Gudes. The design of a cryptography based secure file system. *IEEE Transactions on Software Engineering*, SE-6(5):411–420, Sept 1980. Secondary source.

[52] Neil M. Haller. The S/KEY one-time password system. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, February 1994.

[53] Hugh Harney and Eric Harder. Logical key hierarchy protocol, March 1999. `http://www.securemulticast.org/draft-harney-sparta-lkhp-sec-00.txt` (Last viewed August 28, 2005).

[54] Anthony Harrington and Christian Jensen. Cryptographic access control in a distributed file system. In *Proceedings of 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, Villa Gallia, Como, Italy, June 2003.

[55] Maurice P. Herlihy and J.D. Tygar. How to make replicated data secure. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO '87*, pages 379–391. Springer-Verlag, 1988. Lecture Notes in Computer Science No. 293.

[56] Yih-Chun Hu, Markus Jakobsson, and Adrian Perrig. Efficient constructions for one-way hash chains. In *Proceedings of Applied Cryptography and Network Security (ACNS '05)*, volume 3531, page 423. Springer-Verlag, 2005.

[57] Burt Kaliski. PKCS #7 cryptographic message syntax version 1.5, March 1998. `http://www.ietf.org/rfc/rfc2315.txt` (Last viewed August 28, 2005).

[58] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *2nd USENIX Conference on File and Storage Technologies*, 2003.

[59] David Karger, Tom Leighton, Danny Lewin, and Alex Sherman. Web caching with consistent hashing. In *The Eighth Word Wide Web Conference*, Toronto, Canada, May 1999.

[60] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Tree-based group key agreement. *ACM Transactions on Information and System Security*, 7(1):60–96, 2004.

[61] Loren M. Kohnfelder. Towards a practical public-key cryptosystem, May 1978. Bachelor's thesis supervised by L. Adleman. Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA USA.

[62] Hugo Krawczyk. Simple forward-secure signatures from any signature scheme. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS)*, Athens, Greece, November 2000.

[63] John Kubiatowicz, David Bindel, Yan Chen, Steve Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Westley Weimer, Christopher Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, December 2000.

[64] Hartono Kurnio, Reihaneh Safavi-Naini, and Huaxiong Wang. A secure re-keying scheme with key recovery property. In *7th Australasian Conference on Information Security and Privacy (ACISP)*, pages 40–55, 2002.

[65] Chris Laas and M. Frans Kaashoek. SSL splitting: securely serving data from untrusted caches. In *Proceedings of 12th USENIX Security Symposium*, August 2003.

[66] Leslie Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–771, November 1981.

[67] Butler Lampson. Hints for computer system design. *ACM Operating Systems Review*, 15(5):33–48, October 1983.

[68] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[69] Stephen MacKinnon and Selim G. Akl. New key generation algorithms for multilevel security. In *SP '83: Proceedings of the 1983 IEEE Symposium on Security and Privacy*, page 72, Washington, DC, USA, 1983. IEEE Computer Society.

[70] Umesh Maheshwari and Radek Vingralek. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, October 2000.

[71] *Mandriva.* `http://club.mandriva.com/xwiki/bin/view/Downloads/TorrentAccess (Last viewed August 28, 2005).`

[72] D. Mazières and D. Shasha. Don't trust your file server. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 113–118, May 2001.

[73] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawah Island, SC, 1999.

[74] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 1–15, July 2002.

[75] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[76] Ralph Merkle. Protocols for public key cryptosystems. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[77] Ralph Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer-Verlag, 1987.

[78] Silvio Micali. Fair public-key cryptosystems. In Ernest F. Brickell, editor, *Advances in Cryptology – CRYPTO '92*, volume 740 of *Lecture Notes in Computer Science*, pages 113–138. Springer-Verlag, 1992.

[79] Gerome Miklau and Dan Suciu. Controlling access to published data using cryptography. In *International Conference on Very Large Data Bases*, pages 898–909, September 2003.

[80] Sara Miner More, Michael Malkin, Jessica Staddon, and Dirk Balfanz. Sliding-window self-healing key distribution. In *2003 ACM Workshop on Survivable and Self-Regenerative Systems*, 2003.

[81] Dalit Naor, Moni Naor, and Jeff Lotspiech. Revocation and tracing schemes for stateless receivers. In Joe Killian, editor, *Advances in Cryptology – CRYPTO '01*, volume 2139 of *Lecture Notes in Computer Science*, pages 41–62. Springer-Verlag, 2001.

[82] Moni Naor and Benny Pinkas. Efficient trace and revoke schemes. In *Proceedings of Financial Crypto 2000*, Anguilla, British West Indies, February 2000.

[83] Ivan Nestlerode. Implementing EFECT. Master's thesis, MIT, June 2001. `http://theses.mit.edu/` (Last viewed August 28, 2005).

[84] National Bureau of Standards. Secure hash standard. Technical Report FIPS Publication 180-2, National Bureau of Standards, August 2002.

[85] Alina Oprea, Michael Reiter, and Ke Yang. Space-efficient block storage integrity. In *Proceedings of the Twelfth Network and Distributed System Security (NDSS) Symposium.* Internet Society (ISOC), Feburary 2005.

[86] Adrian Perrig, Dawn Song, and J. D. Tygar. ELK, a new protocol for efficient large-group key distribution. In *Proceedings of IEEE Symposium on Security and Privacy*, 2001.

[87] Zachary N. J. Peterson and Randal C. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, May 2005.

[88] Zachary N. J. Peterson, Randal C. Burns, and Adam Stubblefield. Limiting liability in a federally compliant file system. In *Proceedings of the PORTIA workshop on sensitive data in medical, financial, and content-distribution systems*, 2004.

[89] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *FAST*, pages 89–102, Monterey, CA, January 2002.

[90] David Reed and Liba Svobodova. Swallow: A distributed data storage system for a local network. In A. West and P. Janson, editors, *Local Networks for Computer Communications*, pages 355–373. North-Holland Publ., Amsterdam, 1981.

[91] Michael Reiter and Kenneth Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

[92] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, March 2003.

[93] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[94] Ronald L. Rivest. All-or-nothing encryption and the package transform. In *Proceedings of the 1997 Fast Software Encryption Conference*, volume 1267 of *Lecture Notes in Computer Science*, pages 210–218. Springer-Verlag, 1997.

[95] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[96] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

[97] Phillip Rogaway. On the role definitions in and beyond cryptography. In *ASIAN '04 The Nineth Asian Computing Science Conference*, volume 3321 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[98] Pankaj Rohatgi. A compact and fast hybrid signature scheme for multicast packet authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security (CCS)*, Singapore, November 1999.

[99] Mendel Rosenblum and John Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, CA, October 1991.

[100] *RPM software packaging tool.* `www.rpm.org` (Last viewed August 28, 2005).

[101] J. H. Saltzer. Hazards of File Encryption. MIT Laboratory for Computer Science Request for Comments 208, MIT, May 1981. `http://mit.edu/Saltzer/www/publications/csrrfc208.html` (Last viewed August 28, 2005).

[102] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *17th SOSP*, pages 110–123, December 1999.

[103] Mahadev Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, pages 9–21, May 1990.

[104] Roger Schell, Peter Downey, and Gerald Popek. Preliminary notes on the design of secure military computer systems. Technical Report MCI-73-1, The MITRE Corporation, Bedford, MA, January 1973. `http://csrc.nist.gov/publications/history/sche73.pdf` (Last viewed August 28, 2005).

[105] Adi Shamir. On the generation of cryptographically strong pseudo-random sequences. *ACM Transactions on Computer Systems*, 1(1):38–44, February 1983.

[106] Alan Sherman. A proof of security for the LKH and OFC centralized group keying algorithms. Manuscript, November 2002.

[107] Alan Sherman and David McGrew. Key establishment in large dynamic groups using one-way function trees. *IEEE Transactions on Software Engineering*, 29(5):444–458, May 2003.

[108] *Sleepycat Berkeley Database (version 3.0.55).* `www.sleepycat.com` (Last viewed August 28, 2005).

[109] Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A network protocol independent performace evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[110] Alex Snoeren. *A session-based architecture for Internet mobility*. PhD thesis, Massachusetts Institute of Technology, February 2003.

[111] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[112] Raj Srinivasan. XDR: External data representation standard. RFC 1832, Network Working Group, August 1995.

[113] Jessica Staddon, Sara Miner, Matt Franklin, Dirk Balfanz, Michael Malkin, and Drew Dean. Self-healing key distribution with revocation. In *Proceedings of IEEE Symposium on Security and Privacy*, 2002.

[114] Jeremy Stribling, Daniel Aguayo, and Maxwell Krohn. Rooter: A methodology for the typical unification of access points and redundancy. 2005. Accepted in the 9th World Multi-Conference on Systemics, Cybernetics and Informatics (2005).

[115] Adam Stubblefield. Personal communication, 2004.

[116] TTCP. `ftp://ftp.sgi.com/sgi/src/ttcp/` (Last viewed August 28, 2005).

[117] *The Wall Street Journal Online.* `http://www.dowjones.com/FactSheets/WSJcomFactSheet.htm` (Last viewed November 9, 2004).

[118] Debby Wallner, Eric Harder, and Ryan Agee. Key management for multicast: Issues and architectures. RFC 2627, Network Working Group, June 1999.

[119] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. `http://eprint.iacr.org/2004/199` (Last viewed August 28, 2005).

[120] Xiaoyun Wang, Yiqun Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO '05*, volume 3621 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[121] Hugh C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Transactions on Information Theory*, IT-26(6):726–729, November 1980.

[122] Chung Wong and Simon Lam. Digital signatures for flows and multicasts. In *IEEE International Conference on Network Protocols (ICNP'98)*, Austin, TX, October 1998.

[123] Chung Kei Wong, Mohamed Gouda, and Simon Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, February 2000.

[124] Charles Wright, Michael Martino, and Erez Zadok. NCryptfs: A secure and conventient cryptographic file system. In *Proceedings of USENIX Annual Technical Conference*, June 2003.

[125] Yang Richard Yang, X. Steve Li, X. Brian Zhang, and Simon S. Lam. Reliable group rekeying: a performance analysis. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–38, New York, NY, USA, 2001.

[126] A Yao. Theory and applications of trapdoor functions. In *Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science (FOCS '82)*, 1982.

[127] Tatu Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, San Jose, CA, July 1996.

# Biographical note

Kevin Edward Fu graduated from West Ottawa High School in Holland, Michigan in 1994, then attended MIT for undergraduate and graduate school for the next eleven years. Kevin had a dual appointment as a visiting scholar at the Johns Hopkins University Information Security Institute from 2003–2005.

Kevin served four semesters as a teaching assistant in 6.857 Network and Computer Security and 6.033 Computer Systems Engineering. Kevin served as a Graduate Resident Tutor for English House (Conner 2) at Burton-Conner for one ant nest shy of four years.

During his graduate tenure at MIT, Kevin has been active with the USENIX Association, the ACM, and the National Senior Classical League. Kevin is married to Teresa K. Lai '98. When not researching, Kevin enjoys artisanal bread making and the culinary arts.