Attribute Based Encryption Made Practical

by

Long Zhang

B.S., Peking University, Beijing, China, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

April 2012

© Long Zhang, 2012
Ciphertext-Policy Attribute Based Encryption (CP-ABE) is a promising method for end-to-end, fine grained access control. However, based on our knowledge, there is no massive deployment of CP-ABE based systems. Expensive and insecure key revocation should be one of the major reasons. In this thesis, we hypothesize that key revocation can be performed client side by combining existing trust computing technologies and validate this hypothesis with a prototype file system called ABFS. ABFS uses CP-ABE to do client side access control, at the same time, provide strong assurance on key revocation. Enterprises equipped with ABFS can reliably relocate their data from centralized storage to unused space on untrusted client machines and thus decentralize most aspects of their storage, mitigate data backup cost, improve storage durability and remove the threat of single point of failure. ABFS combines existing TPM and attribute-based encryption technologies to perform access control checks on otherwise untrusted clients and ensure confidentiality of data.
Preface

All the work described in this thesis was performed under the supervision of Professor William Aiello with regular consultation from Dutch Meyer, a current Ph.D candidate, Professor Andrew Warfield and Wenhao Xu, a former master student from NSS lab.

The idea of ABFS project was first developed between Wenhao and me in a Starbucks cafe close to East Mall and Agronomy Rd, after reading the Persona paper accepted by Sigcomm 2009. We then ran into Bill’s office immediately when we got the first draft, and persuaded him to jump to the boat. After Wenhao’s graduation, we were very lucky to have Dutch and Andy in, and they indeed, brought lots of fresh ideas into the project, such as using TPM. The whole project is on decentralizing most aspects of centralized filer to the clients and employing client side desktops to provide secure, durable storage. Fixing key revocation problem for CP-ABE is a necessary part of the whole project, and this work is independently done by the author.
# Table of Contents

Abstract ......................................................... ii  
Preface ............................................................. iii  
Table of Contents ................................................ iv  
List of Tables ..................................................... vii  
List of Figures ..................................................... viii  
Acknowledgments ................................................ ix  

1 Introduction ..................................................... 1  

1.1 Background ................................................... 3  
  1.1.1 Attribute Based Encryption ............................... 4  
  1.1.2 TPM Based Attestation and Sealed Storage .......... 4  
  1.1.3 Xen and Virtualization .................................. 6  
1.2 Roadmap ..................................................... 6  

2 System Overview ............................................. 8  

2.1 System Overview ............................................. 8  
2.2 Assumptions ................................................ 9  
2.3 Deployment Model ......................................... 10  

3 File Level Access Control ................................. 13  

3.1 Terms .................................................... 13  

iv


3.2 File Format ........................................... 14
   3.2.1 File Metadata .................................. 14
   3.2.2 Block Metadata ................................ 15
3.3 Read, Write, Share and Verification ..................... 15
3.4 Attribute Revocation .................................. 17
   3.4.1 Lazy Revocation ................................ 17
   3.4.2 Server-verified Writes ......................... 21
   3.4.3 Owner’s Identity and File’s Ownership .......... 22
3.5 Future Work ........................................ 23
   3.5.1 Attribute Based Signature ....................... 23
   3.5.2 Merkle Hash Tree ............................... 24
3.6 Weakness ............................................ 24

4 TPM based Attestation and Key Revocation .................. 25
   4.1 Design Criteria ................................... 26
   4.2 Xen Boot Sequence ................................ 26
   4.3 Boot Time Integrity ............................... 27
   4.4 System Provisioning and ASK Revocation ........... 28
   4.5 Runtime Integrity ................................. 31
   4.6 Secure Channel ................................... 31
   4.7 Usability ......................................... 31
   4.8 Discussions ...................................... 32

5 Implementation ......................................... 33
   5.1 File System Frontend .............................. 33
   5.2 TPM Based Attestation and Sealed Storage .......... 34
      5.2.1 Extend PCRs .................................. 34
      5.2.2 Seal Indexed NVRAM Area .................... 35

6 Performance Evaluation .................................. 37
   6.1 Boot Time Overhead ............................... 38
   6.2 Runtime Overhead ................................ 38
      6.2.1 Sequential I/O performance ................. 39
      6.2.2 Postmark Benchmark .......................... 40
6.3 Storage Overhead ................................................. 42
6.4 Runtime Speedup ............................................... 43

7 Related Work ...................................................... 45
  7.1 Access Control and Secure Storage ......................... 45
  7.2 TPM Based Attestation ....................................... 46
  7.3 Attribute Based Encryption (ABE) ......................... 47

8 Conclusions and Future Work ................................. 48
  8.1 Conclusions .................................................. 48
  8.2 Future Work ................................................ 48

Bibliography ...................................................... 50
List of Tables

Table 6.1  TPM centric overhead ........................................... 38
Table 6.2  Postmark configuration ........................................... 42
Table 6.3  Storage overhead for ABFS file abstraction ................. 42
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>System architecture</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Deployment model</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>File format</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Updated file format</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Event timeline</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Current file state</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Server-verified writes</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Final version of file metadata</td>
<td>23</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Code for extending indexed PCR’s value</td>
<td>34</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Code for sealing indexed NVRAM space</td>
<td>36</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Sequential I/O on local machine</td>
<td>39</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Sequential I/O on network filer</td>
<td>40</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Postmark benchmark on local machine</td>
<td>41</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Postmark benchmark on network filer</td>
<td>41</td>
</tr>
</tbody>
</table>
Acknowledgments

I would like to thank my supervisor Prof. William Aiello for all kinds of things he has done for me, such as, but not limited to, guiding my research and developing my skills on solving challenging problems with encouragement and patience, teaching me how to land airplanes (he said it’s easy...) and introducing me great western food, places of interest and movies. I am also very grateful to have Prof. Andrew Warfield as the second reader of this thesis. Andy provides valuable guidance and feedback on ABFS project from very beginning, and is actually my “unofficial” supervisor. I would like to thank Mr. Dutch Meyer for his effort on developing the idea of ABFS, giving a first read to this thesis, and providing helpful suggestions for the writing of it. I would like to thank my friend Mr. Wenhao Xu for helping me understand complex operating systems, getting me involved in several promising projects when I first came here and with barely no knowledge on this field. I would also thank Mr. Jean-Sébastien Lègarè for introducing me Javascript, DOM model and several other browser side technologies and bringing me into another super fantastic project, Pando. Apart from the above, there are too many great others to be mentioned here. In general, NSS lab is a great place to meet great people, and it was a wonderful experience to accomplish master degree here.

Last but not the least, I would also thank NESRC ISSNET for their great effort and generous funding on organizing annual summer school and workshop on network and systems security (the watermelon party and Calgary Stampede were both impressive), and supporting me to attend USENIX Security Symposium.
To my parents,
with love.
Chapter 1

Introduction

In several scenarios, encryption is not enough for exchanging sensitive data. It is always imperative to convince the message sender that his message can only be revealed to recipients whose identities strictly confirm to the specific access control policy defined by himself. For example, when FBI wants to report a highly confidential clue of a potential terrorism attack to extremely limited members inside Congress for security consideration, or Apple CTO wants to deliver Iphone 4S prototype to a limited number of senior stuffs for improvements, it is always necessary to ensure that no one else can snoop the secret. Currently, the dominating method to enforce such access control policies in file sharing is through a trusted server/cluster to store the data and mediate access control. This solution inevitably introduce third parties (such as system administrators) stand “in the middle” of every message transmission and have sufficient privilege to disturb the transmission and snoop the secret. Thus, it is always nice to have a “perfect” end-to-end access control solution to directly share confidential data. By “perfect” here, we mean an access control method that is both reliable (with high assurance for data access, strong revocation, simple management logic, etc), scalable and with minimal number of third parties to be trusted with.

Enterprises will also benefit from “perfect” end-to-end access control. Enterprise file servers provide centralized access to storage that is assumed to be ideally durable and secure. However, when we take a second thought, the centralized design of these storage system places significant limitations on both of those ideals.
Durability is limited by the number of physically redundant components that can be placed in a single server, including but not limited to hard drives. Adoption of security is limited by it’s complexity on management logic and impact on performance, which is similarly bottlenecked at a single source. Since most organizations already have ample resources available on each end-host, with the help of “ideally perfect” end-to-end access control, enterprise can use some of its excess disk capacity for efficient, effective data mirroring and backup. Adoption of end-to-end access control will both enhance the security and durability of enterprise file system, at the same time, mitigate the cost of maintaining centralized primary/secondary storage infrastructure and deploying expensive data mirroring/backup strategies.

End-to-end access control is more scalable compared with existing centralized one. Currently, there is an emerging trend of storing data in a distributed fashion across many servers spanning different administration domains and geographic locations. Replicating data across several locations have advantages in availability, reliability and disaster tolerance. What is more, for online service providers, abandoning a primary data server and allowing access to different replicas based on geographic location, load balancing status is also very promising because of performance (balancing infrastructure workload and reducing service latency) and security reasons (i.e., mitigating DOS attacks). To realize this goal using traditional way, access control is enforced by employing a centralized while trusted server. This server itself can be a bottleneck of scalability. Apart from this, cross domain/geographic location centralized access control needs to be much more complex in logic and thus easier to be error-prone. When data is stored at several locations, the chances that one of them get compromised increases dramatically. With a “perfect” end-to-end access control mechanism, this problem can be solved. Data servers can be dumb and simply serve bits, at the same time, all the access control enforcement can be done at client side.

Ciphertext-Policy Attribute Based Encryption (CP-ABE) is a promising cryptographic algorithm and an initial attempt that allows end-to-end access control of shared data. Its simplified key distribution and expressive access control logic make building secure collaborative/sharing intensive applications to be possible. However, current CP-ABE is far from being “perfect”. First of all, it is hard to achieve provable security, expressiveness of access logic and efficient encryp-
tion/decryption at the same time. Secondly, mapping to asymmetric encryption, there is no corresponding “attribute based signature” which is both practical and secure to use. Thirdly, attribute revocation is extremely hard to realize in an efficient while reliable way. Current work such as [12] focuses on solving the first problem, and second one can sometimes be overcame using existing alternatives. The third one, attribute revocation, is intrinsically hard because each attribute is conceivably shared by multiple users and only a subset of them need to be revoked. Thus revocation should not prevent valid users with same attributes from accessing shared data. Proposed solutions such as associating secret key with timestamps [28] or introducing another centralized proxy for re-encryption [31] are not promising at all. The former needs to weigh the lifetime of timestamps (thus the complexity of re-issuing secret key) against confidentiality of protected data and introduce massive useless effort to re-validate access of users whose roles stay unchanged. The later introduces another centralized authority and vanishes the advantage of fully decentralized access control.

In this paper, we try to solve the key revocation problem of CPABE by using a seamless combination of existing mature technologies (TPM, virtualization, etc). We assure strong, light weight key revocation of CP-ABE with a trusted key server executing simple management logic. In our model, identity check does not have to be in-lined with each access request, however, we rely on client side cryptography to enforce access control to encrypted data. We illustrate our solution using a prototype file system called Attribute Based File System (ABFS for short). Based on our current knowledge, ABFS is the first file system using ABE to deploy end-to-end access control. We further evaluate the performance of ABFS and prove that it is both secure and “performance acceptable” for practical deployment.

1.1 Background

In this section, we provide background information on the technologies leveraged by this thesis.
1.1.1 Attribute Based Encryption

Attribute Based Encryption enables fine-grained access control towards confidential data. It is conceptually close to traditional role-based access control schema. Each user in the system is assigned with a unique ABE secret key, which is associated with his roles. ABE public key is non-secure, and can be distributed to everyone in the system for encryption use. Each encryption must specify an access structure, e.g., a logical expression over attributes, and send it along with ABE public key to ABE encryption engine. Only users whose roles integrated inside their ABE secret key meet the access control logic can decrypt the cipher. ABE is also collusion resistant, which means it is impossible to combine several ABE secret keys to form a new key to expand its power. ABE master key should be kept in a super secure way, since it can be used to generate all the keys involved in this system.

ABE makes offline key distribution much easier than before. Although initial key distribution is still needed to let each client get his key according to his role, this can be done during the machine provisioning phase and further distribution is no longer needed if this role doesn’t change. This feature is especially helpful when dealing with sharing intensive workloads, in which case an alternative solution based on symmetric and asymmetric encryptions needs massive offline key distributions and authentications.

1.1.2 TPM Based Attestation and Sealed Storage

TPM is a trusted co-processor implemented as a chip physically attached to a platform’s motherboard. Before each TPM being shipped, an endorsement key (EK) is generated and burned inside TPM’s NVRAM by the manufacturer. It is used to prove to a second party that a key generated in the TPM was generated in a genuine TPM. Upon activating the TPM, a 2048-bit RSA key called Storage Root Key (SRK) is created. SRK is used to store all the other keys generated by TPM and thus acts as the root of all the key chains. SRK itself is also stored in the non-volatile storage inside the TPM chip. Each TPM chip provides a set of Platform Configuration Registers(PCRs) that can be used to attest the state of the platform. TPM based attestation is executed following a “bottom-up” order of software stack. For
instance, TPM first takes measurements of BIOS and bootloader and then transfers control to them to measure the initial kernel code. Afterwards, it enables the kernel to measure changes to itself (loading of kernel modules and patches) as well as user level applications. Each executable is measured during the load time, and is reduced to a 160-bit hash value using the build-in SHA-1 function of TPM. The only way for software to change the value of a PCR is by invoking the TPM extend primitive:

\[
PCRExtend(index, data) \tag{1.1}
\]

\[
PCR_{index} \leftarrow H(PCR_{index} \| data) \tag{1.2}
\]

This operation updates the value of the indexed PCR with a SHA-1 hash of the previous value concatenated by the data provided. This property is important, since there is no way for malicious users to write desired values to indexed PCRs because of the irreversible nature of SHA-1 hash, and further cheat TPM to reveal the sealed data or proof unauthentic measurements to remote trusted attestation servers. TPM v1.2 specification allows static and dynamic PCRs and only a system reboot can reset the value in a static PCR. At boot time, all PCRs are initialized to a known value, i.e., 0 for static PCRs (range from 1-16) and -1 for dynamic PCRs (range from 17-23). Apart from attestation, the following functionalities provided by TPM are also helpful to ABFS:

- **Non-Migratable Key.** Non-Migratable keys are keys generated inside the TPM and cannot be transferred from one platform to another. In this case, the plaintext of non-migratable key never leaves TPM.

- **Sealed Storage.** TPM offers two primitives, i.e. seal and unseal, to encrypt and decrypt secrets. “Seal” encrypts the input data using Storage Root Key (SRK), who never leaves TPM. The sealed data can be further bound to a particular software state, as defined by the contents of various PCRs. This allows the “unseal” primitive to validate the integrity of software platform before unsealing the data. However, since SRK is specific to TPM ownership, thus sealed data is bound to a particular TPM chip and cannot be unsealed on other platforms. This inflexibility is cumbersome for some ap-
applications, but it is particularly useful to store platform specific identity keys. NVRAM provides a solution to store data inside TPM even during system power-off. The build-in encryption engine implements several cryptographic functions and allows sealed data to be unsealed using TPM.

Apart from the above benefits, TPM has the following limitations. First, TPM only provides load time measurement and attestation, thus it cannot prevent an attacker to launch in memory attack by exploiting backdoors and flaws in currently loaded software. Meanwhile, TPMs are inefficient. TPMs do not support concurrency and a thorough attestation will take more than one second to finish. This inconvenience hampers TPM from frequent invocation for performance consideration, and might even open new avenues for denial of service attacks.

1.1.3 Xen and Virtualization

Xen hypervisor is running between hardware and guest operating systems. It provides both an abstraction to model and emulate a physical machine and strong isolations between control VM and Guest VMs and also among guest VMs. Our work bases on the assumption that after a trusted boot of virtual machine monitor (VMM), there is no way for one guest VM to escape the isolation and tamper with control VM and other guest domains. Since Xen is running on the most privileged ring of CPU and has the capability to schedule the CPU between domains, filtering network packets, and enforcing memory protection and access control when reading data blocks from desk, it is ideal for protecting ABFS domain from runtime malicious attacks.

1.2 Roadmap

The rest of this thesis is organized as follows:

Chapter 2 introduces the general system architecture, as well as the assumptions we made and threat model we defended against. Chapter 3 discusses the basic ABFS file format and the file level access control derived from it. Chapter 4 introduces how we integrate TPM to our system in order to provide stronger security of client side software stack and protect ABE secret key. Implementation details and performance evaluation will be presented in the following two chapters.
Chapter 7 presents the related work to this thesis. And Chapter 8 summarizes this thesis and highlights the future work which need to be done in order to make ABFS a better system.
Chapter 2

System Overview

2.1 System Overview

Figure 2.1 shows an overview of ABFS. The whole system is built on top of Xen and using TPM as a hardware “root-of-trust”. Since domain0 does not support file system logic based on its original design, we thus create a separate storage domain to run ABFS. Because of isolation reason, only domain0 has direct unchecked access to physical disks, and sharing files on disk between domains intrinsically violate the isolation rule of Xen, thus, we exploit Ethernet interface to pass decrypted files from ABFS domain to guest VM. Here, we use “guest VM” to refer to the system that clients directly work with, and thus it is much easier to be exposed to potential malicious attacks. We don’t run ABFS as a software in guest VM because having it run in a separate domain can help us isolate ABFS’s memory space using Xen virtualization, thus protect the plaintext of ABE secret key as well as the confidential files from potential memory/buffer based attacks. Another reason of creating a separate domain is for usability consideration. Frequent patches and updates in guest VM will make practical TPM based attestation impossible. However, based on our design, we can keep a stable version of ABFS domain, and give clients strong freedom to configure and manage their own working environments.

The security of client machine builds on top of a trust chain. TPM attests the authenticity and integrity of each system component during the booting phase, and then transfers control to attested system software stack. Afterwards, we depend
on the trusted system to isolate memory space of storage VM from guest VM and protect user’s attribute based secret key as well as important file metadata. The detail of file metadata will be presented in Chapter 3 and building trust chain will be discussed in Chapter 4.

In our “proof-of-concept” implementation, ABFS is built using FUSE. Thus, we have all crypto libraries running in the user space. To allow ABFS communicate with remote file server, we do a NFS mount of ABFS. Thus, they can pass files through NFS’s RPC/XDR interface. We have TPM chips plugged in the motherboard of machines, and rely on a modified BIOS, boot loader as well as Xen hypervisor to support boot time attestation.

2.2 Assumptions
The security of ABFS is based on the following assumptions:

*Perfect Crypto Libraries.* We assume that TPM implements a SHA-1 hash function that is totally collision resistant. Although every hash function with more
inputs than outputs will consequently generate collisions, we assume it is computationally difficult to find two inputs that hash to the same output within a given period of time. A minor difference in inputs will also generate significantly different fingerprints. We also assume that CP-ABE, symmetric encryption (i.e., AES-256-CBC) and asymmetric encryption (i.e., PKCS#11) are resistant to chosen-ciphertext attack. For example, adversaries gathering information (such as any ciphertext and corresponding decryption) cannot recover the hidden secret key within a finite time.

*Perfect TPM.* We assume that TPM provides perfect protection over its Endorsement Key(EK) and Storage Root Key(SRK). It also protects its non-migratable keys perfectly as described in the specification, which means plaintext of these keys will never leave TPM. Sealed storage and sealed NVRAM also perform correctly that unseal will never happen if PCRs do not meet the requirement. What is more, we also suppose that TPM device driver as well as TCG software stack(TSS) both function correctly so that each high level command issued by authorized entity will be executed with strong assurance. Last but not the least, we also assume that physical attacks towards TPM are impossible.

*Perfect Xen.* We assume that Xen hypervisor successfully partition the physical machine that multiple operating systems running on top of it have no way to interfere with each other. This requires strict isolation on memory space, i/o buffer and even on disk. Apart from security, we also need Xen to schedule each domain “smartly” that performance of each domain can be optimized.

All the assumptions we made above comply to the design concepts of related technologies, and we don’t require more than those.

## 2.3 Deployment Model

Figure 2.1 shows the deployment model of ABFS. In our model, enterprises have freedom to decide whether to keep a primary data server or not. It is always a choice that enterprises eliminate the centralized data storage permanently, and result in a fully peer-to-peer fashion. However, enterprises need to be very careful
when make this choice since P2P has its significant downside. It is a poor fit for applications in which a user needs a timely notification that their operations have been committed successfully and will not be overridden by any others. An intermediate choice would be having a server which implemented a traditional file system interface to address access node and record commit history. A third choice is to employ ABFS for purely data mirroring/backup purpose. The design in this aspect is quite free and we make it as a open discussion. In this thesis, we focus on the design and implementation of the client side software stack.

However, in our model, we do have no trust assumptions built for this centralized server. Apart from serving bits, there is no access control enforcement need to be performed on the server. Thus enterprises can choose to outsource their server to potentially scalable, reliable but untrusted cloud service providers, or keep it privately. Key server (revocation server) is separated from data server, and is responsible for key revocation. It communicates with a trusted certificate authority (CA) in order to verify the identity of guest machines. We do require that key server and CA need to be kept securely and managed correctly inside the enterprises. In ABFS, clients have more freedom to place their machines. They can connect to the

![Deployment model](image-url)
data servers through laptops from public internet or through desktops inside enterprises. However the laptops need to be provisioned before shipping to the clients, which is the common case for almost all the modern enterprises.
Chapter 3

File Level Access Control

Different from depending on centralized server to enforce access control through checking users’ group membership, ABFS migrates cryptographic and key management operations as well as access control to the clients, and the server incurs very little cryptographic overhead. Client side file level access control list is defined by file owners following ABFS file format and enforced by cryptographic methods and TPM. In this section, we will introduce ABFS file format. Similar to SiRiUS, we use a file meta-file to store file metadata.

3.1 Terms

The following terms are frequently used in this chapter.

- *File owner* is who creates the file and defines access policies of the file.
- *Readers* are who can read the file.
- *Writers* are who can write the file.
- *ABE* stands for attribute based encryption.
- *APK* stands for ABE public key.
- *ASK* stands for ABE secret key.
- *BSK* stands for block signature key.
3.2 File Format

Regarding design goals, in ABFS we need techniques to 1) keep confidential files away from unwanted readers; 2) differentiate between readers and writers on file access; 3) make valid readers and writers easy to detect destructed data. To achieve these goals, we design ABFS file format as shown in Figure 3.1. Other goals also critical to secure file system, such as 1) strong access revocation; 2) prevent valid readers/writers from caching plaintext of ASK as well as the plaintext of any other encrypted keys from file metadata, are together achieved by combining TPM with virtual machine isolation, which will be discussed in next chapter.

3.2.1 File Metadata

File metadata contains the access control information that can only be modified by file owners. APK is used to encrypt most fields of file metadata and can be distributed to anyone inside the system. Readers’ and writers’ attributes are used to differentiate readers from writers. Block metadata pointer points to the metadata of the blocks. Owner’s signature is a signature of all the other components in file metadata, so that only the owner can change each field of the file metadata. Owner’s public key is also contained in file metadata in order to allow everybody

Figure 3.1: File format

- BVK stands for block verification key.
- File Encryption Key (FEK) is used to encrypt the real data of a file in a symmetric manner.
to verify the integrity of the file metadata.

- $\{\text{FEK}\}_{\text{APK},\text{writers' attributes}}$: File encryption key encrypted with writers’ attributes, is used by writers to encrypt modified data blocks.
- $\{\text{BSK}\}_{\text{APK},\text{writers' attributes}}$: Block signature key encrypted with writers’ attributes, is used by valid writers to sign modified data blocks.
- Block Signature Key is the public part of BSK and is used to verify the signature of writers.

### 3.2.2 Block Metadata

File data are stored into blocks, each with a predefined size (e.g. 4KB). In order to ensure that only the valid writers can modify the block, the block data is signed using the plaintext of BSK and the fingerprint is stored in per block signature. In this case, we restrict write access to valid writers only (since BSK is encrypted with writers’ attributes), and anyone inside the system can verify the signature (since BVK is kept in plaintext). Block data pointer points to the encrypted data blocks. As for the encrypted fields,

- $\{\text{FEK}\}_{\text{APK},\text{readers' attributes}}$: File encryption key encrypted with readers’ attributes, is used by readers to decrypt modified data blocks.
- $\{\text{Block Data}\}_{\text{FEK}}$: Each data block is encrypted with FEK.

### 3.3 Read, Write, Share and Verification

Valid readers can read the files. In order to read the authentic files, they first need to verify the integrity of file metadata as well as data blocks. Then they can decrypt the files follow the following steps:

1. Fetch the block metadata pointers from file metadata and further locate the block metadata.
2. Decrypt the FEK located in the block metadata using valid ASK.
3. Use the FEK to further decrypt the block data.

When a writer wants to write a modified block, the following steps need to be performed:

1. The writer encrypts the modified blocks using FEK.
2. The writer decrypts BSK from file metadata and signs the modified block using BSK.
3. The writer encrypts FEK with readers’ attributes only if it get changed, and places it into block metadata.
4. The writer sends write request to the storage server along with the data blocks.

When an file owner changes the field in file metadata, the following steps need to be done.

1. The owner signs the file metadata with his public key after modifying the file metadata.
2. The owner commits the change to the storage server.

File owners can create and share files, with preferred access control logic. To create and share a file, file owners will:

1. Create a symmetric FEK for file encryption and an asymmetric BSK-BVK pair for signature and verification.
2. Construct file metadata following [3.1], and sign it with his own private key.
3. Perform per block encryption using FEK and signature using BSK.
4. Commit files to storage server for further broadcasting.

Anyone can check the integrity of a file by performing the following steps. Suppose the block data, file metadata and block metadata have already been cached in client side.
1. The client uses owner’s public key to verify the signature of file metadata.

2. If the verification fails, the client reports a broken data. Otherwise, the client continues doing the following steps.

3. The client fetches BVK from file metadata, and further verify the signature of each data block.

4. If the verification of current data block fails, the client reports a broken data. Otherwise, the client continue verifying data blocks left.

### 3.4 Attribute Revocation

Key revocation and attribute revocation are treated differently in ABFS. User who no longer holds the attributes (e.g., leave his position or change his role) should not keep his old ASK. In another word, we need to keep each user’s ASK consistent with his roles from time to time in order to avoid frequent file re-encryption. This consistency is enforced by remote key revocation server along with client side TPM. The detail of ASK revocation will be presented in next chapter. Attribute revocation happens when file owners decide to revoke access to files from certain roles (change read access of a file from "Sys_admin AND NSS_lab” to ”NSS_lab” only). Although this might be a rare case, we still need mechanism to deal with it. We propose two different ways to solve attribute revocation. The easiest way is to re-encrypt the whole file and broadcast updates aggressively to valid readers and writers. Afterwards, only the most up-to-date version of the file will be read and written. If attribute revocation is truly infrequent and occasional re-encryption won’t cause too much overhead, re-encryption is a good choice because of its simplicity in logic. However, since we are lack of real file system trace to prove this hypothesis, we thus propose a lazy revocation schema as an alternative.

#### 3.4.1 Lazy Revocation

In lazy revocation, only the file metadata need to be updated by file owners. Data blocks as well as block metadata stay unchanged. From previous discussion, it is clear to us that read access is controlled by \( \{\text{FEK}\}_\text{APK,reader's attributes} \) placed in
block metadata since data block is directly encrypted with FEK. At the same time, write access is restricted by \( \{BSK\}_{APK, \text{writers' attributes}} \) since only valid writers have access to BSK and any one with BVK can verify the signature of each data block. Thus they are the subjects to update during lazy revocation. In lazy revocation schema, we introduce Block Encryption Key (BEK), for per block encryption. We further update our file format to Figure 3.2.

In order to avoid file re-encryption when revoking read, we rely on valid writers overwriting block metadata, encrypting and signing modified block with updated keys to get new access control information propagated. When “read” revocation is needed, the file owner first modifies the “Readers’ Attributes” field with new readers’ list, and choose another File Encryption Key encrypted with APK and writers’ attributes. File owner further signs the new file metadata with his private key. Afterwards, for each new write, valid writer will encrypt the modified block with new FEK and replace the old \( \{BEK\}_{APK, \text{readers' attributes}} \) with new \( \{FEK\}_{APK, \text{updated readers' attributes}} \). In this case, revoked readers still have access to the stale data blocks since they haven’t been re-encrypted yet, however, they cannot get access to updated blocks.

When “write” revocation is needed, the file owner modifies the “Writers’ Attributes” field in the file metadata, selects a new BSK-BVK pair and encrypts BSK with updated writers’ attributes. The owner further signs the updated file metadata with his own private key. Afterwards, the valid writers will use new BSK to sign each modified block before commitment. Write revocation is more problematic because: 1) revoked writers may launch rollback attack and mislead users into accessing stale data, and 2) since we only update file metadata when revoking...
writes and some data blocks may not yet being re-encrypted and still signed with old BSK, this inconsistency between file metadata and block metadata may disturb users (especially those who haven’t cached the old file metadata) from verification.

First problem comes from the possibility that a revoked writer may replace the new file metadata with a stale version he cached, and regain access to this file. Since the old version of file metadata still confirm to the checking policy we talked about, it is impossible to be marked as “out-of-data” without additional evidence. He can also update data blocks signed with old BSK to mislead valid readers and writers into accepting unauthorized content since the file may be partially updated and some blocks are still signed with old BSK. To overcome this attack, we require file owners aggressively broadcast changed file metadata to all valid readers/writers through a secure communication channel. Readers and writers can further query a trusted CA in the enterprise to verify file owner’s identity (each identity in the system is associated with a AIK stored inside TPM, thus it is quite straightforward to do the attestation). After receiving updates, valid readers and writers will no longer accept updates in old version. In this case, revoked writers cannot fraudulently update stale file metadata to regain access. A file owner can further exploit a Paxos liked protocol to make sure his updates have been accepted by most of the valid recipients.

Figure 3.3 and 3.4 show a scenario of our second concern over lazy revocation. As shown in 3.3, the file owner creates the file at time 0. At this point, all the data blocks are signed with BSK0 and can be verified through BVK0. At time 1, writer A writes the block0 and uses BSK0 to sign the modified block. At time 2, file owner revokes some writers’ write access and further updates a new pair of BSK1-BVK1 to file metadata. At time 3, writer B writes block1 and signs the modified block with BSK1. At time 4, reader C reads both modified blocks, and the state of current blocks is shown in 3.4. At this time, block0 and block1 are signed with totally different keys and cannot be verified only through BVK1 placed in file metadata.

To solve this “inconsistent keys” problem, we exploit a key rotation schema similar to Plutus [15]. Using key rotation, an authorized reader can generate all previous versions of the key from current version, yet has no way to generate the future version. In our example, reader C with key rotation schema can derive previous key pair BSK0-BVK0 from BSK1-BVK1, but has no way to guess future
key pair BSK$_2$-BVK$_2$. In our case, we only need BVK to be rotatable in order to let readers verify data blocks prior to re-encryption, since writers can always fetch new BSK from fresh file metadata for signing and thus have no motivation to recover old BSKs. Since we make no contribution to pervious work conducted on key rotation, we simply choose Plutus's key rotation schema for convenience and further details can be found in paper [15].

**Discussion**

Lazy revocation works well for preventing revoked readers from accessing data that has been updated. However, the problem with revoked writers is more severe since revoked writers can still update stale blocks and cheat readers. For instance, a revoked reader may collude with storage server and update block signed with BSK$_0$ even if the block has already signed with BSK$_1$. To detect this rollback attack, we may introduce client side state machine to record block versions and offline method to collaborate valid readers to defeat this attack. In next section, we will introduce server-verified writes as a stronger alternative to revoke write access.

Note that the above discussion is based on the assumption that a valid writer
may not have the read access to the file, which is identical to Linux style file level access control policy. However, in the real world, file owners always assign valid writers with read access, in order to make each write meaningful. Based on this assumption, our design can be simpler. Another assumption for ABFS is, all the writers will execute the right behavior. In our future work, we will explore a solution to a stronger threat model than this.

### 3.4.2 Server-verified Writes

Although server-verified writes will introduce another trusted party, it can intrinsically help us prevent unauthorized writers from making authentic changes to the persistent store. In ABFS, server-verified writes is much simpler in logic compared with other file systems. Since we rely on TPM to always keep each user’s ASK consistent with his roles, the only duty for server is to make sure potential writers’ attributes confirm to the access control logic defined in file metadata. Figure 3.5 shows the protocol we deployed for ABFS’s server-verified writes mechanism.

As soon as the server receives the write request along with object’s file metadata, server read writers’ attributes field from file metadata and encrypts a random nonce with APK and writers’ attributes. Writer with valid ASK can quickly decrypt the nonce and send the nonce in plaintext back through a secure channel (such as
SSL). The server further accepts or denies write request based on the challenge result. After a predefined timeout, if the server cannot receive desired response from the writer, server will also close the session and deny the write request. In this case, we allow data server to verify whether a user has required authorization without keeping any state on the server side as well as revealing secrets of confidential files.

Besides, in order to defeat replay attack launched by revoked writers, server should be smart enough to identify the freshest version of file metadata. In another word, server should only allow valid file owners to update file metadata, and forbids everyone else from modifying it. File metadata check can be made through checking hashed values of the whole block, and the problem left to be how to identify owner’s identity as well as file’s ownership.

### 3.4.3 Owner’s Identity and File’s Ownership

By owner’s identity, we mean the ownership of the public key that is used to verify the signature of file metadata. Without this attestation, a malicious writer can pick up his own public-private key pair and forge the signature of valid file owners’. In ABFS, we deploy traditional public key certificate scheme to certify the
ownership of a public key. A certificate authority stores the public keys as well as their owners’ information, such as unique ID and email address, and issues digital certificates based on these information. Any subject in the system can verify the certificates and further verify the ownership of the public key. With this scheme, any file metadata update signed with fake private key (the public part used to verify the signature cannot be certified) will be discarded by file server and thus only the file owner can update file metadata (since she is the only person who has the private key).

By file’s ownership, we mean the actual mapping between files as well as their owners. Without this mapping, a malicious writer may claim the ownership of a file that doesn’t belong to him and further mislead others by using his own certified public key to verify the signature. Afterwards, he can manipulate every field in file metadata including access logic. In ABFS, we assign each file a unique “uri”, which is constructed by using owners’ unique ID appending with file’s absolute path, as the logic path of a file. This “uri” will also be placed in file metadata, and signed together with owner’s private key. File server uses this “uri” to finally locate requested files. In this case, we bind the ownership of files to file metadata and forbid malicious writers from claiming fake ownership. The final version of file metadata is shown in Figure 3.6.

3.5 Future Work

3.5.1 Attribute Based Signature

In attribute based signature, a signature attestation does not aim to reveal the identity of the individual who signed the message, but a claim regarding the attributes the underlying signer possesses. Ideally, user cannot forge signatures with attributes he does not own even through collusion. Attribute based signature can
help us revoke writers’ privilege under lazy revocation schema, and thus get rid of trusted storage server. Besides, attribute based signature can help us design a much cleaner file metadata format, and simplify our identity checking logic. With attribute based signature, writers can also generate their own symmetric keys to encrypt modified blocks, which can be verified by each valid reader. In this case, ABFS can achieve higher key diversity and survive under stronger threat model.

3.5.2 Merkle Hash Tree
Since per block signature is expensive in practice, Merkle hash tree [20] can be used to consolidate all the hashes, with only the root being signed. This is especially helpful when a writer modifies a large number of data blocks in a huge file. It also makes readers easier to certify the digital signature generated by writers. In this paper, we haven’t implemented merkle hash tree so far. In our future work, this will be implemented as a necessary optimization.

3.6 Weakness
Although we impose strong effort on detecting malicious manipulation of users’ data, yet given the fact that data server is intrinsically untrusted, it is impossible for us to defeat all kinds of attacks from the client side. One possible attack launched by data server is Denial of Service (DoS). Although data servers cannot snoop users’ secrets, they can simply delete those content from stable storage and make all further access impossible. Also, our work relies on the assumption that the data servers will function correctly, which means they will execute each valid command issued by valid commanders accurately. This is also not always the case. In ABFS, we don’t deal with these issues. We assume that it is possible for users to detect these attacks through offline communication. Afterwards they can simply switch to another storage service provider to get better service.
Chapter 4

TPM based Attestation and Key Revocation

A major challenge of migrating access control to the clients is offline security. A diligent and curious user can break through client side software stack by all means, and exceeds the barrier he should never cross. In this chapter, we discuss the mechanisms we adopted to provide strong offline security of client side software stack. Our discussion based on assumptions that a user without corresponding ABE secret key cannot decrypt files, and for those files he has access to, there is no motivation for him to pry into the memory space used to cache the plaintext of the file because he can “copy-paste” the plaintext in a much easier way. However, a user does have motivations to steal the plaintext of his ABE secret key. In this case, he can still get access to the files as well as their updates even after revocation. Thus we hone our argument down on how to protect the plaintext of ABE secret key and ensure remote revocation of ASK. We further dispatch our attentions into boot time protection and runtime protection. In general, our solution takes advantage of TPM based attestation and sealed storage so that revealing ASK’s plaintext to the main memory will only happen after successful attestation of an authentic kernel and ABFS software stack, and we further rely on virtual machine and ABFS domain to protect and remove the plaintext of ASK securely.
4.1 Design Criteria

The current state of TPM specification Version 1.2\(^1\) depresses our design into the following aspects:

- PCRs are always in volatile storage, and each system reboot will trigger a reset of all PCRs. Thus they are not candidates to store ASK since ASK should be protected within persistent storage intermediate.

- TPM has no ABE engine implemented so far. Thus attribute based encryption and decryption must be done outside TPM using third party ABE libraries. This fact makes loading decrypted ASK into main memory inevitable.

- Non-volatile storage provided by TPM is a very limited resource, and calls for efficient use.

Although TPM supports non-migratable keys, yet by default, these keys are encrypted and stored on disk. Thus they can be the targets of offline attacks since TPM has no protection over their encrypted bulbs. The details of how to overcome these difficulties will be covered later in this chapter.

4.2 Xen Boot Sequence

Since we use Xen to ensure runtime security of our software stack, it is essential for us to understand Xen boot sequence in order to perform accurate boot time attestations. Similar to all kinds of systems, when a computer is turned on, it first loads BIOS (Basic Input/Output System) from non-volatile storage on the motherboard. BIOS is a very low-level, hardware oriented application that does some basic hardware initialization, testing, and configuration work. TPM also need to be enabled here to so that it can be accessible to the rest of the system. Afterwards, BIOS loads another program, called Master Boot Record(MBR), into memory from a predefined location on the disk. MBR is a boot sector consists of 512 bytes of data and is located on the first sector of the hard disk. It contains a small program that

\(^1\)http://www.trustedcomputinggroup.org/resources/tpm_main_specification
copies additional code (includes boot loader) from the storage device into memory. Similar to booting Linux, the standard Linux boot loader (i.e., GRUB) is a key component of installing and booting Xen hypervisor. It loads Xen hypervisor using its kernel command, and subsequently identify dom0 kernel and initial RAM disk or file system and then transfers control to Xen hypervisor, and let it continuously execute chain-booting of dom0 Xen kernel.

The Xen hypervisor further probes and initializes system’s hardware so that it can correctly map and handle incoming requests from the actual device drivers used by dom0 kernel as well as other paravirtualized domains. It also creates its own memory map for managing memory use by various domains. Finally, it loads the Linux kernel that it should boot for Dom0 and transfers control to it. Dom0 then can be bootstrapped following typical Linux kernel booting routines and ABFS domain can be further created by Dom0. Until here, the full system boot is finished. The guest OS will be run in other guest domain on the machine.

4.3 Boot Time Integrity

In ABFS, before unseal the ASK, TPM need convictive evidence to ensure that current loaded code and data maintain the trust into the overall software stack of the whole system. We rely on building a trust chain from TPM based hardware “root-of-trust” into the current system runtime to assure the secure executing environment, and unseal ASK only if the current state of software stack meets predefined “security requirements”. The boot sequence of Xen talked about in last section forms the base of our measurement. Specifically, we need to make sure that the whole system, including BIOS, bootloader, Xen hypervisor, domain0 as well as the ABFS storage domain, are loaded with trusted executables in a correct order.

A TPM measurement is a $SHA-1$ hash computed over the file that contains data or executables loaded into the runtime. A slight difference in the file will result in a distinguished fingerprint and hence, variations in executables are easily detected by differing measurement values. The correctness of loading order is enforced by concatenating the existing measurements with the new one, since a different ordered concatenations will also result in totally different hashing results. Assum-
ing that the system administrators have profound knowledge of guest machines’ configuration, and thus they can determine a trusted boot during the machine provisioning phase. Then the steps to achieve boot time integrity of guest machines are as follows:

1. Measure the boot of BIOS, bootloader, Xen hypervisor, Dom0 and ABFS domain using dedicated PCR and then transfer control to ABFS.

2. ABFS loads its viewing program, takes a hash over this viewing program and concatenates the result with previous PCR and transfers control to the viewing program.

3. Viewing program unseals the ASK if current measurements meet the integrity requirement, and then extends measurement PCR so that all the following loaded executables cannot tamper with ASK. All the further decryption of secrets using ASK is done in ABFS’s local memory, isolated and protected by Xen.

4. If the measurement results after loading viewing program is different from a trusted boot (because of the compromising of software stack’s integrity or a bad booting order), ASK cannot get unsealed by TPM.

4.4 System Provisioning and ASK Revocation

Sealed storage provides a perfect property to bind decryption of secrets with the integrity of the software platform, however, it is not enough for revocation. Revocation needs to assure the number of copies and exact locations of target elements, and erase them confidently. However, apart from Endorsement Key(EK), only the Storage Root Key(SRK) is guaranteed to be always kept inside TPM. Nothing else processes the similar property. Non-migratable property is not a perfect match because although the plaintext of the key will never leave TPM (the private part is encrypted using SRK before getting stored during generation phase), yet, TPM has no control over the encrypted bulb. The storage of encrypted bulb can be placed anywhere on disk. Even if it is inside TPM, we need to make sure that nobody can copy it to somewhere else. Thus the challenges of design include: 1) The secret
should be kept inside TPM with a fixed index and it should be prevented from getting snooped by malicious executables; 2) Secret need to be sealed so that integrity attestation is needed before using the secret 3) NV storage in TPM is a very limited resource and data stored in this area should be as small as possible. To fulfill these goals, we associate NVRAM area with PCR values through \textit{Tspi_NV\_DefineSpace} command, so that TPM can enforce read or write based on current state of loaded software stack. Similar to what we talked about in last section, after integrity check, we depend on the correctness of viewing program of ABFS to protect secrets from getting copied.

Before the machines get shipped to the clients, trusted system administrators in the enterprise will run the \textit{tpm\_takeovership} command to take the ownership of the TPM, and afterwards, only the valid owners have the rights to read/write and grant access to the NVRAM area. Here, we exploit the ownership of TPM to set access policy of NVRAM and use TCG Software Stack(TSS commands) to issue commands. During the provisioning phase, system administrators need to:

1. Generate ASK securely using ABE master key, APK and candidate’s attributes.

2. Create a non-migratable asymmetric key pair using targeted TPM, and fetch the public portion (NM\_Pub).

3. Generate a symmetric key (ASK\_EK) to encrypt ASK, and use NM\_Pub to encrypt ASK\_EK.

4. Create a NVRAM data object to encapsulate \{(ASK\_EK)\}\_NM\_Pub with access policies.

5. Associate NVRAM data object with NV index by using \textit{Tspi\_SetAttribute} command.

6. Apply for storage in the NVRAM of TPM and bind this area with PCR values using \textit{Tspi\_NV\_DefineSpace}.

7. Send \{(ASK)\}\_ASK\_EK to the client and delete plaintext of ASK\_EK, ASK.
After finishing these steps, we actually sealed an index in NVRAM to protect the content associated with the index from getting snooped. If a system boot compromises the integrity of the whole software stack, unseal the NVRAM area will be forbidden. At the same time, if the TPM owner (here, system administrator) erases \( \{ \text{ASK}_E\}_\text{NM}_\text{Pub} \) in the non-volatile RAM, decrypting ASK will be made impossible either. And this forms the base of our ASK revocation.

Revoking an ASK is very similar to system provisioning. However, it requires a secure communication channel to allow revocation server to talk to client machines. Implementation of this channel will be talked about later. Apart from this difference, the owner of TPM need to erase \( \{ \text{ASK}_E\}_\text{NM}_\text{Pub} \) from TPM’s NVRAM, and replace it with a new one (if necessary) with high confidence. In this case, we must keep NM_Pub which is used to encrypt ASK_EK secure inside enterprise, and limit the number of people who can access it.

Since loading the decrypted version of ASK into main memory happens after trusted boot and successful attestation of Xen hypervisor and client side software stack, we thus ensure the boot time security. We further rely on Xen to isolate each domain and rely on ABFS domain to mark the memory space storing the plaintext of ABE secret key as “protected”, and provide an isolated and secure executing space of client side software stack. For security consideration, the plaintext and encrypted version of ASK should never be written to the disk.

Since SRK is the always kept inside TPM and used to encrypt all the other non-migratable keys, a more aggressive way to do key revocation is to erase the SRK permanently. Although there is no client side command to erase SRK, since each SRK is associated with a TPM ownership, we can indeed use `TPM_ForceClean` command to remove the old ownership and implicitly clean the SRK permanently. However, all the non-migratable keys in the TPM chip will get lost afterwards because removing SRK means destroying all the key chains having dependencies with SRK. And in fact, almost all the keys protected by TPM use SRK for encryption. This method can simplify our design of key revocation, yet this will collapse all the other applications using the same TPM. We thus don’t recommend this solution.
4.5 Runtime Integrity

After establishing the trust chain, the challenge turns to be *how to maintain the same level of integrity continuously throughout the lifetime of the hypervisor*. Since verification of a kernel is extremely hard (interested readers can refer to [17]), and it took a professional team eight years to verify a microkernel, the integrity of the hypervisor can still get compromised by software bugs and backdoors. We rely on existing methods to protect the lifetime integrity of Xen hypervisor. For example, HyperSafe[30] provides a non-bypassable memory lockdown to protect hypervisor’s code and state and a restricted pointer indexing to enhance the control flow integrity. Xoar[7] further breaks the control VM into single purpose components and make attestation much easier and break through away harder. Other works on reducing the TCB size and enforce isolation will also help to us. In general, ABFS can benefit from all the existing technologies that improve the lifetime integrity of hypervisor control flow.

4.6 Secure Channel

Key server need a secure channel to communicate with client machines for key revocation. A SSL connection may be enough to establish an end-to-end connection, however, there are at least two questions to answer. 1) Is the machine connected to the right one key server wants to talk to? 2) Is the end point secure enough to apply high confidential key revocation?

To answer the first question, we assume that key server holds a valid and trusted certificate binding with a public RSA identity key $AIK_{pub}$ of the client’s TPM. Since AIK is non-migratable, there is no way to forge an AIK and snoop the identity of others. To answer the second question, key server can require a signed measurement list from client machine, and depend on attestation to verify the correctness of guest machine.

4.7 Usability

One usability problem concerned with sealed storage is frequent system patching and updates. Regardless users’ preferences on installing different updates and
patching, even the order in which patches are applied can result in a combinatorial explosion of distinct configurations for a single application, and each configuration requiring a distinct reference value for attestation purposes. Indeed, this fact leads to problems with TPM based attestation. In ABFS, we argue that the subjects of our attestations are only Xen hypervisor, domain0 and ABFS storage domain, and even the device driver running in guest domain is beyond our measurement. Enterprise’s IT department will be responsible to maintain the above attestation subjects, and users are kept free to update software in their guest VM domain on their will.

4.8 Discussions

Xen hypervisor is not the only candidate to secure the execution of client side software stack in ABFS. Capabilities offered by AMD’s Secure Virtual Machine (SVM) extension allow late launch of a hypervisor or Security Kernel at an arbitrary time with build-in protection against software-based attacks. The trusted software running in the ring 0 of the CPU can issue the *SKINIT* command to allocate a physical memory address for its own. The *SKINIT* command will disable the DMA to the physical memory pages as well as the interrupts that may cause the untrusted software to gain the control. Intel TXT technology provides similar functionalities. Thus we can run the client side software stack directly on those hardware. Software like Security Kernel and microkernel can also help to provide strong isolation of our client side software.
Chapter 5

Implementation

In this chapter, we will introduce the proof-of-concept prototype of ABFS we built for experimental use.

5.1 File System Frontend

Client side ABFS software stack is built on top of FUSE. Each time when ABFS is mounted, we create a context object to store the root directory of file system. Using this context object, we can track file/directory node in the directory tree. A file node may be opened many times, but only one instance per file is kept. We are doing a global level reference counting for file node, and a instance will be released only if the counting turns to be zero. Apart from storing root directory of ABFS, we also use this data structure to store ASK and APK of the client. The viewing program is invoked during the mounting time. It unseals the ASK and load ASK as well as APK into context object for further use. Afterwards, it calls the TPM_PcrExtend program to extend indexed PCRs and thus forbids further unsealing of ASK intended by malicious executables. During the lifetime of ABFS, we maintain a “Cipher” object. It contains all the methods for cryptographic use, such as attribute based encryption/decryption, symmetric encryption/decryption, sign/verification and so on. The final read and write to data blocks are performed on a 4KB granularity.
Figure 5.1: Code for extending indexed PCR’s value

5.2  TPM Based Attestation and Sealed Storage

Trust Computing Group (TCG) provides a well designed software stack (TSS) for programmers to manipulate TPM and the high level interface of TSS is TCG Service Provider Interface (Tspi). In ABFS, we let viewing program use Tspi to communicate with TPM, verify the attestation results, unseal ASK, and extend PCRs to disable further access of sealed NVRAM. Note that the following code is open sourced and originally created in trouSer test suite version 0.3\footnote{http://sourceforge.net/projects/trousers/files/TSS%20API%20test%20suite/0.3/} and modified by the author for experimental use.

5.2.1  Extend PCRs

After successfully loading the ASK, what the viewing program need to do immediately is to extend the dedicated PCRs so that all other codes running in ABFS domain and Guest VM cannot read the indexed area of NVRAM. Figure 5.1 shows
this piece of code. A very good property provided by TPM is, although we pass
the value we plan to extend PCRs with (here, the newPerValue object) to the func-
tion, TPM will take a hash of it implicitly before actually concatenate it with the
old PCR values. This forbids anyone from writing desired values to PCRs. The
reason for it is because of the “one-wayed” nature of SHA-1 hash, it is impossible
to deduce a source value of hash function by using its hashed result.

5.2.2 Seal Indexed NVRAM Area

Code on sealing an indexed NVRAM area is shown in Figure 5.2. One interest-
ing trick need to be mentioned is that although defining a new space in NVRAM
need to be owner authorized, there is no TPM object associated with NV Tspi
APIs to receive the owner’s authorization. In order to overcome this problem,
Tspi_NV_DefineSpace transparently inherits the policy object associated with the
TSS_HNVSTORE object passed in, and uses the owner authorization data from
that policy object.

In this code, we set the permissions to require owner authorization to write to
the NVRAM area. However, read is associated with PCRs and will not need any
further authorization. This is exactly the solution we deployed to achieve ASK re-
vocation while at the same time still grant users read access to ASK after successful
system attestation.
function TPM_SealIndexedNVRAM(BYTE * data, UINT32 dataLen) {
    /* Define Variables */
    TSS_HCONTEXT hContext;
    TSS_HTPM hTPM;
    TSS_HPOLICY hOwnerPolicy;
    TSS_HNVSTORE hNVStore;
    TSS_RESULT result;

    /* Create TPM context */
    result = Tspi_Context_Create(&hContext);

    /* Connect to TPM context */
    result = Tspi_Context_Connect(hContext, NULL);

    /* Get TPM object */
    result = Tspi_Context_GetTpmObject(hContext, &hTPM);

    /* Get policy object */
    result = Tspi_GetPolicyObject(hTPM, TSS_POLICY_USAGE, &hOwnerPolicy);

    /* Create the NVRAM object */
    Tspi_Context_CreateObject(hContext, TSS_OBJECT_TYPE_NV, 0, &hNVStore);

    /* Set related attributes:
        Set the attribute in the NVRAM object so that NV index 0x8 (a random choice) is used.
        Set the permissions to require authorization to write to the NVRAM area.
        Set the size of the area we are about to define. */
    Tspi_SetAttribUint32(hNVStore, TSS_TSPATTRIB_NV_INDEX, 0, 0x8);
    Tspi_SetAttribUint32(hNVStore, TSS_TSPATTRIB_NV_PERMISSIONS, 0, TPM_NV_PER_AUTHWRITE);
    Tspi_SetAttribUint32(hNVStore, TSS_TSPATTRIB_NV_DATASIZE, 0, dataLen);

    /* Call down to TPM to define the space. We associate this NVRAM space
     * with 10th PCR */
    Tspi_NV_DefineSpace(hNVStore, 10, 0);

    /* Write the data into the defined area */
    Tspi_NV_WriteValue(hNVStore, 0, dataLen, data);
}

Figure 5.2: Code for sealing indexed NVRAM space
Chapter 6

Performance Evaluation

The design of our system includes several factors that we expect to impose considerable overheads on performance. In this part, we seek to answer the following questions: How much is the overhead of ABFS? What are the factors trigger this overhead? How to mitigate (if cannot avoid) it? We run our client on a desktop with Intel Core i5 2.80GHZ quad-core CPU, 4GByte of RAM and Intel 82578DC GbE network interface. Our server filer is a machine with Intel Xeon E5420 2.50GHZ quad-core CPU, 4GByte of RAM and the same network interface. We further separate our tests from boot time tests to run time tests in order to make our evaluation more clear and efficient.

We evaluate our system against Encfs, an encrypted file system using symmetric encryption and running in user space using FUSE. For local tests, we run ABFS against EncFS and local Ext4 mount. For tests on network filer, we run ABFS against EncFS and NFS. We further mount ABFS and EncFS on NFS in order to use NFS’s RPC/XDR interface for remote communication. We want to understand the performance disparity of ABE based file system against symmetric encrypted and non-encrypted file systems. However, a considerable performance penalty may be caused by FUSE, which cannot be shown based on our implementation.
6.1 Boot Time Overhead

By boot time overhead, we mean the inevitable initialization routines starting from system power on to the end of ABFS mount. These routines only happen once during the lifetime of a desktop instance. The boot time overhead includes trusted boot, software stack measurements, unseal ABE secret key, protected memory allocation and ABE public/secret keys loading. Since each client is associated with a unique ABE secret key, we make ABE public/secret key pair loading as a boot time routine. The client daemon use the same data structure that maintains the file system context to store the key pair. Thus, further ABE key loading issued by file system operations will be served from the memory instead of TPM.

In this thesis, we didn’t evaluate a full version of TPM based attestation. Previous works such as [30], [25] and [27] have done a profound performance evaluation on TPM based kernel hash, PCR extend and data sealing and unsealing. Summarizing from paper [30], performance of a prevailing v1.2 Broadcom BCM0102 TPM is shown in table 6.1.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCR Extend</td>
<td>1.2</td>
</tr>
<tr>
<td>Hash of Kernel</td>
<td>22.0</td>
</tr>
<tr>
<td>Unseal data</td>
<td>898.3</td>
</tr>
</tbody>
</table>

Table 6.1: TPM centric overhead

Open source package such as Trusted GRUB\(^1\) covers modified bootloader and can be setup easily to do boot time attestation. IBM IMA\(^2\) [25] is also open sourced and contains a full implementation of trusted boot. Thus in this chapter, we concentrate on evaluating the runtime overhead and storage consumption of ABFS.

6.2 Runtime Overhead

Runtime overhead refers to the overhead incurred when the file system has been mounted and client side daemon has been launched. Essentially this overhead is caused by file system supported operations, i.e., open, read, write, etc. Compared

\(^1\)http://trousers.sourceforge.net/grub.html
with boot time overhead, runtime overhead is a more likely to impact user experience and thus we did a more sufficient work on evaluating runtime overhead. In the following tests, we investigate more on ABFS’s sequential and random I/O performance. For each test, we present the mean of 6 out of 10 runs, ignoring the top and bottom two outliers.

### 6.2.1 Sequential I/O performance

We use Linux tool *dd* to transfer 512 MB raw data with block size of 4KB each into and out of a single file from local and remote file server separately. Since asynchronized mount enables write system call to be returned to caller before the data has indeed been written to the stable storage, we set both Ext4 and NFS to be mounted as synchronized to see the real IO impact. For each read test, we clear the system buffer cache by using Linux *drop_caches* command before running in order to make sure all read results serving from the disk. NFS is also mounted with “noac” mode so that client side attributes, metadata and file cache will be disabled. In this case, each NFS read will be served by direct I/O read from remote file.

Right now, our system only supports synchronized mode. As shown in Figure 6.1 and Figure 6.2 our system as well as Encfs suffer from reasonable performance...
penalty compared with Ext4 and NFS. We believe that this is mainly caused by overhead introduced by FUSE and per block signature and encryption. Since we only use ABE once for decryption in sequential read and write, its influence can be ignored.

6.2.2 Postmark Benchmark

Postmark is a benchmark designed to be resource intensive and non-deterministic in order to portray performance in the ephemeral small file regime used by Internet, such as Email, news feed, web-based commerce and so on. It models a heavy workload placed on many small files, and thus gives us a sense of the performance of random file access. Postmark generates an initial pool of random text files ranging in size from a configurable low bound to high bound. The file pool is of configurable size and can be located on any accessible file system. Once the pool has been created, a specified number of transactions occurs. Each transaction consists a subset of \{create file, delete file, read file, append file\}. We configure the Postmark benchmark following table 6.2.

Figure 6.3 and Figure 6.4 show the results of running PostMark on ABFS on local and remote servers, against EncFS, Ext4 and NFS. In this case, the per file
Figure 6.3: Postmark benchmark on local machine

Figure 6.4: Postmark benchmark on network filer
Table 6.2: Postmark configuration

<table>
<thead>
<tr>
<th>Items</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>500000</td>
<td>500000</td>
</tr>
<tr>
<td>transactions</td>
<td>1024</td>
<td>262144</td>
</tr>
<tr>
<td>subdirectories</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

ABE decryption becomes the bottleneck that downgrades the performance of our system. This further gives us an intuition to improve the performance under high random file access workloads by reducing the frequency of invoking ABE encryption / decryption engine.

6.3 Storage Overhead

Apart from the time consumption, the design of ABFS also incurs considerable space overhead for storing encrypted files. This fact not only encourages us to look for better solutions to shrink the on disk encrypted files, but also drives us to make a smarter use of TPM’s NVRAM. Table 6.3 shows the storage related information of ABFS based file abstractions.

Table 6.3: Storage overhead for ABFS file abstraction

<table>
<thead>
<tr>
<th>Items</th>
<th>Size</th>
<th>After ABE encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABE public key</td>
<td>888 bytes</td>
<td>-</td>
</tr>
<tr>
<td>ABE master key</td>
<td>156 bytes</td>
<td>-</td>
</tr>
<tr>
<td>Kevin’s ABE secret key</td>
<td>47798 bytes (47 KB)</td>
<td>-</td>
</tr>
<tr>
<td>Signature key (in PEM)</td>
<td>887 bytes</td>
<td>15630 bytes</td>
</tr>
<tr>
<td>Verification key (in PEM)</td>
<td>273 bytes</td>
<td>-</td>
</tr>
<tr>
<td>File encryption key</td>
<td>32 bytes</td>
<td>14782 bytes</td>
</tr>
<tr>
<td>Block encryption key</td>
<td>32 bytes</td>
<td>14782 bytes</td>
</tr>
<tr>
<td>Policy file</td>
<td>314 bytes</td>
<td>-</td>
</tr>
<tr>
<td>Signature</td>
<td>256 bytes</td>
<td>-</td>
</tr>
<tr>
<td>File meta-file</td>
<td>31528 bytes (31K)</td>
<td>-</td>
</tr>
<tr>
<td>Block meta-data</td>
<td>15038 bytes (15K)</td>
<td>-</td>
</tr>
</tbody>
</table>

In Table 6.3 we create a ABE secret key named “Kevin’s Secret Key” with
attributes “sysadmin AND it_department AND ‘office = X410D’ AND ‘hire_date =’ ‘data + %s’ ”. We define readers’ attributes as “(sysadmin and (hire_date < 946702800 or security_team)) or (business_staff and 2 of (executive_level ≥ 5, audit_group, strategy_team))” and writers’ attributes with the same configuration. Signature key and verification key are RSA asymmetric key pair generated using OpenSSL lib and further converted into PEM format. File encryption key is a symmetric key generated using the same lib with AES-256-CBC mode. Since we finally use symmetric key to encrypt block data, there is no incremental storage overhead for it.

Table 6.3 inspires us into two dimensions. First, as to the NVRAM of TPM, ASK is huge. This is the reason why we seal a symmetric key which is used to encrypt the ASK instead of sealing the ASK directly. Secondly, per file meta-file and per block metadata are big storage overhead that we need to avoid as much as possible. As a solution, we collect file/block with same access information into groups to reduce the redundancy. The details will be presented in next section.

### 6.4 Runtime Speedup

**Client Side Caching.** As inspired by NFS, caching data in buffer cache as well as on disk will improve the read throughput dramatically. Reasonable amount of read-ahead will also help. However, flushing the in memory buffer cache is necessary when new updates are pushed to the local disk by remote filers for consistency reasons.

**ABE Accelerations.** Since we rely on TPM to enforce the key revocation, it is possible to allow file owners to use the same symmetric keys to share the files with same ACLs to the recipients. In this case, we introduce the concept of sharing group. Sharing group is defined on a per file owner base. A file owner can group the file that he wants to share with the same readers’ and writers’ lists into a sharing group and encrypted these files with same symmetric key. Similar to userspace file system daemon which maintains a map list of active file node and their data structure in memory during the lifetime of the daemon, we can further maintain a
map list of file owner as well as his sharing group on the client side. Each item will be associated with a symmetric key. Thus, when ABE decryption is called, client side daemon will first look up the file owner as well as the ACLs inside the memory. If there is a hit, the symmetric key will be fetched directly. Otherwise, an ABE decryption call will be invoked to fetch the symmetric key from file metadata. Valid readers can also cache block encryption key to facilitate block level decryption. Since in most cases, FEK is identical to BEK. Thus caching BEK will save readers effort in decrypting BEK on per block base using ABE.
Chapter 7

Related Work

7.1 Access Control and Secure Storage

Traditional authentication and authorization rely on maintaining a centralized database of user identities in order to perform role based access control, making it difficult to authenticate users in a different administrative domain and make itself a target of single point of failure. NFS [3], AFS [13] and CIFS [22] belong to this category, and a Kerberos like protocol is used for authentication. The Self-Certifying File System [19] provides authentication across domains and channel security for accessing remote file systems. Furthermore, it is extended to provide a decentralized access control mechanism, and allowed to grant access to users and groups from different administrative domains without pre-existing administrative relationships [16]. However, access control in SFS still relies on all file requests passing through the trusted SFS servers. Thus, it cannot support offline access control efficiently. SNAD [23] requires strong authentication of users along with user trust in the server to enforce access control. SUNDR [18] and FAUST [6] use fork consistency techniques to implement storage protocols on top of untrusted servers and enable end-to-end detection of forking groups. However, both of these works are based on the assumption that forking groups can be detected easily offline, which is not always the case. This is particularly harmful to time-critical tasks which require malicious manipulations to be detected immediately.

Plutus [15] and SiRiUS [10] are particularly designed for end-to-end secure file
sharing using untrusted storage. Plutus encrypts each file block with a unique file block key, and encapsulates file block keys belonging to the same sharing group into file lockboxes. File-lockbox keys need to be shared securely among group members. Thus, it calls for a secure offline key distribution channel and will generate a large amount of keys when system is growing. SiRiUS enable secure file sharing by encrypting file encryption key by all of the readers and writers’ public keys, and thus the size of file metadata will grow rapidly if large amount of people are sharing the same file. Furthermore, the above two systems need file owner to be able to enumerate all the target recipients, which is not always possible in an enterprise. Moreover, these two system are particularly awkward in key management and “lazy revocation” adopted by these two papers are not suitable for enterprise scenario, which calls for strong assurance for its key management.

7.2 TPM Based Attestation

TPM [14] has been a mature technology and commodity hardware to help establishing trust between entities using a separate trusted coprocessor whose state cannot be compromised by potentially malicious host system software. Preliminary works such as [25], [24] and [26] illustrate how to use TPM to establish a trust chain from the boot time of the attested machine and thus allow remote trust server to perform attestation. However, the time-of-use and time-of-attestation discrepancy remains to be addressed, since the code may be correct at the time of attestation and can be changed or tampered with on the fly during the runtime. Associating with AMD Secure Kernel [2] , BIND [27] measures a piece of code immediately before it is executed and uses a sand-boxing mechanism to protect the execution of the attested code. It also ties the code attestation with the data that the code produces so that we can pinpoint what code has been run to generate that data. Current commercial products such as Bitlocker [1] exploit TPM to protect the trusted boot pathway and thus help to provide high assurance drive encryption on commodity operating systems.

In this work, we try to extend TPM to enforce key management inside enterprise and thus enable strong assurance on end-to-end access control. With a secure communication channel, a trusted attestation server can remotely force a revoked
user to remove her identity key and detect the client side misbehavior.

7.3 Attribute Based Encryption (ABE)

In ABE [11], a user is able to access data only if he possesses a certain set of credentials or attributes. CP-ABE [5] binds ciphertexts to access structures while secret keys contain attributes. In this case, ciphertexts can be decrypted with a key contains a set of attributes that satisfies the access structure defined in the ciphertexts themselves. This helps to eliminate the centralized trust server which stores the data and mediates access control in order to enforce Role-Based Access Control (RBAC) [9] and thus allows end-to-end access control. [29] introduces a tiered architecture to improve the performance of ABE so that it can scale to millions of users. Persona [4] hides user data with ABE, allowing users to apply fine-grained policies over who can view their data on online social networks. [12] transfers ABE ciphertext satisfied by user’s attributes into a constant-size El Gamal-style ciphertext without revealing any part of the user’s messages and thus helps to mitigate the overhead of ABE (the size of ciphertext and the time required to decrypt it).

However, based on our knowledge, there is no concrete file system based on ABE built for enterprise so far. We think the major problem forbid ABE to be used for building high assurance file system is its inconvenience in key revocation. Thus, we try to solve this problem in our work.
Chapter 8

Conclusions and Future Work

8.1 Conclusions
In this thesis, we argue that a seamless combination of attribute based encryption, TPM and virtual machine based isolation can help to build a client side access control model with high security assurance. The weakness of ASK revocation is overcame by introducing TPM and client side software stack is secured by using Xen. Sealed storage helps to make sure that confidential ASK will be revealed to the main memory only if the whole software stack is secure and TPM based attestation helps to make sure client side software is trustable. We further build a proof-of-concept prototype called “ABFS” and evaluate the performance of this file system. We demonstrate that ABFS is both secure and efficient for practical deployment.

8.2 Future Work
In the future, we plan to do a large scale deployment of ABFS, study its performance, scalability and explore the possibility of using ABFS to replace the centralized access control, and even more aggressive, to replace the whole primary filer using a decentralized solution. Current study[21] (appeared in year 2011) shows a mean file system utilization of 43% after studying 857 desktop computers in a major IT company. A much earlier study[8] (in the year of 1999) shows a simi-
lar result (57% average utilization) after examining nearly five thousand machines. Based on these work, we are more confident to conclude that most organizations already have ample resources available on client desktop, and these resources are currently underutilized. We argue that providing durability and role based access control at the centralized filer is not necessary and a decentralized alternative is more scalable, durable and cost-effective. However a pure peer-to-peer design maybe less promising because of its poor fit for applications in which a user needs a timely notification that their operations have been committed successfully and will not be overridden by any others. Thus, we will further explore the balance between centralized and decentralized solutions, and plan to come up with a product meets commercial requirements.
Bibliography

[1] Bitlocker driver encryption. → pages 46


fine-grained access control of encrypted data. In ACM Conference on
Computer and Communications Security, 2006. \(\rightarrow\) pages 47

ciphertexts. In USENIX Security Symposium, August 2011. \(\rightarrow\) pages 3, 47

Technical Conference, February 1998. \(\rightarrow\) pages 45

Trusted Platform Module Main Specification, version 1.2 edition, October
2003. \(\rightarrow\) pages 46

Scalable secure file sharing on untrusted storage. In USENIX File and
Storage Technologies (FAST), 2003. \(\rightarrow\) pages 19, 20, 45

user authentication in a global file system. In ACM Symposium on Operating
System Principles, 2003. \(\rightarrow\) pages 45

D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch,
and S. Winwood. sel4: Formal verification of an os kernel. In 22nd ACM
Symposium on Operating Systems Principles, 2009. \(\rightarrow\) pages 31

repository. In 6th Usenix Symposium on Operating Systems Design and
Implementation, 2004. \(\rightarrow\) pages 45

management from file system security. In ACM Symposium on Operating
System Principles, 1999. \(\rightarrow\) pages 45

function. In CRYPTO, 1987. \(\rightarrow\) pages 24

USENIX Conference on File and Storage Technologies, 2011. \(\rightarrow\) pages 48

URL http://www.ubiqx.org/cifs/. \(\rightarrow\) pages 45


