Efficient Trusted Cloud Storage Using Parallel, Pipelined Hardware

by

Hsin-Jung Yang

B.S. in Electrical Engineering
National Taiwan University, 2010

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2012

Certified by .....
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Nickolai Zeldovich
Associate Professor of Electrical Engineering and Computer Science
Thesis Co-supervisor

Accepted by .....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Efficient Trusted Cloud Storage Using Parallel, Pipelined Hardware

by

Hsin-Jung Yang

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Cloud storage provides a low-cost storage service with high efficiency and global accessibility via the Internet, but it also introduces security risks. One major security concern is the integrity and freshness of data stored on the cloud, that is, whether a storage provider can guarantee that the data received by its clients is always correct and up-to-date.

Recent studies have focused on data integrity and freshness guarantees. However, systems that solely rely on cryptography are not able to immediately detect data freshness violations, while systems using resource-constrained trusted hardware are impractical due to long latency and low throughput.

In this thesis, we describe a prototype of a trusted cloud storage system that efficiently ensures data integrity and freshness by attaching a piece of high-performance trusted hardware to an untrusted server. We propose a write access control scheme to prevent unauthorized writes and ensure all writes are fresh. We also introduce a crash-recovery mechanism to protect our prototype system from crashes and power loss events. In addition, we minimize the system overhead by (1) parallelizing and pipelining the operations that are carried out on the server and the trusted hardware and (2) judiciously partitioning the operations across the trusted and untrusted components. The throughput and latency of our prototype system are analyzed to provide customized solutions to performance-focused and budget-focused cloud storage providers. We believe this work takes a major step in making trusted cloud storage practical from an efficiency and cost standpoint.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Thesis Co-supervisor: Nickolai Zeldovich
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

First, I would like to express my sincere gratitude to my advisor, Professor Srinivas Devadas. His inspiring guidance and full support has helped me to complete this project successfully and develop myself as an independent researcher. His extreme enthusiasm for various research topics has deeply influenced me. Working with him has been a matter of great pleasure for me and has enabled me to experience the excitement of doing research.

I would like to profoundly thank my thesis co-supervisor, Prof. Nickolai Zeldovich, for his guidance and patience. This thesis project cannot be completed without his insights, constructive suggestions, and helpful feedback.

I want to extend my gratitude to my labmates, Victor and Mieszko, for providing invaluable resources on this project. I would like to thank Victor for guiding me into this field, and he is the co-author of this project. I am especially grateful to Mieszko for his patience and significant help. He is also a great mentor in my research and graduate life. I would also like to acknowledge Owen for his great suggestions on my research and his willingness to help me out with all technical and non-technical questions.

I greatly appreciate my friends, Yin-Wen, Owen, Annie, Ann, Hao-Wei, Yichang, and Yu-Hsin, for sharing wonderful times with me at MIT. I would also like to thank Kuri, my dear friend in Taiwan, for her encouragement and always being there for me no matter how far we are apart.

Finally, I am deeply grateful to my parents and family. No words can express my gratitude for their unconditional love and full support throughout my life. Without them, I would not have been able to go my own way during these two years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Cloud computing is an emerging web-based computing model that provides users with storage, computational resources, and software applications as services. It offers a cost-effective solution to satisfy cloud users' various computing needs through cloud service platforms, such as Amazon Elastic Compute Cloud (EC2) [1], Google App Engine [2], and Windows Azure [3]. Cloud users are able to access high computing power and data storage on demand through the Internet using light-weight portable devices, while cloud service providers can achieve better resource utilization via multiplexing the workloads. This new computing model moves data and computing tasks from local computers and portable devices into large data centers, bringing cloud users great cost reduction, scalability, and accessibility.

Cloud storage is one of the most prominent cloud-based services as it can be used to not only store files but also support cloud-based applications. For example, Amazon Simple Storage Service (S3) [4], Google Storage [5], and Azure Storage are well known cloud storage providers offering scalable storage services to end users, enterprises, web application developers, and other cloud services providers. Based on these storage platforms, Dropbox (relying on S3) [6], Microsoft SkyDrive [7], Apple iCloud [8], and the recently launched Google Drive [9] further provide file sharing and synchronization services among multiple devices and multiple users. Figure 1-1 represents an overview of a cloud storage system. Cloud storage users are able to back

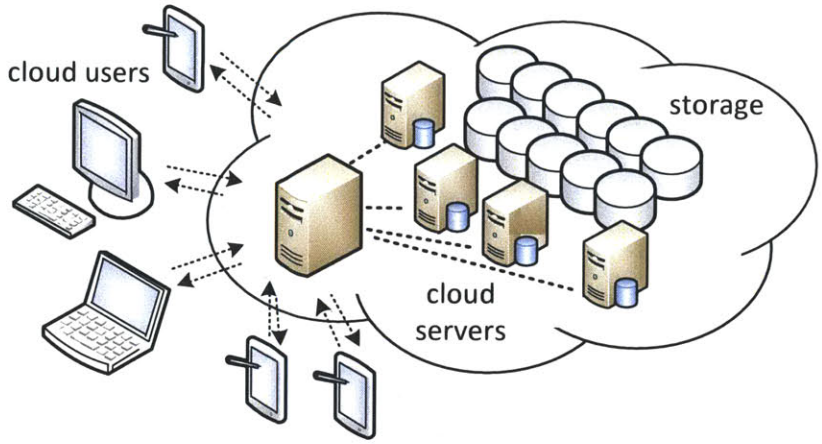


Figure 1-1: Cloud storage overview

up their data remotely, access the data from any connected device, and collaborate within groups to work on the shared data. This outsourced data storage service offers great convenience and global data accessibility to the users at low cost, and the users no longer need huge local data storage nor do they need to worry about data maintenance.

However, outsourcing data introduces security risks, such as privacy and data integrity losses. These risks are primarily from resource sharing: hardware resources are shared on the cloud among various users and thus vulnerable to intruder attacks. Adversaries may penetrate the cloud storage provider and control the cloud server to leak its users' confidential information, corrupt the stored data, or present inconsistent data to different users. In addition, cloud servers may be attacked by malicious insiders or experience some accidental failures due to software bugs, crashes, hardware failures, or power loss events, and therefore cause data loss or inconsistency. As a result, cloud users cannot trust cloud servers to store and manage their data, especially when the data are used for sensitive computations, e.g., financial or medical transactions.

The main security concerns about current cloud storage providers are guarantees of confidentiality, availability, integrity, and freshness. Under these guarantees, the data can only be accessed by authorized users, is accessible at all times, and is always correct and up-to-date. Most of these concerns can be resolved by software: confidentiality by encryption, availability by appropriate data replication, and integrity

by digital signatures together with message authentication codes (MACs). On the other hand, freshness is difficult to ensure especially when multiple cloud clients are involved.

Freshness is guaranteed if the data read from a cloud server has exactly the content that is written by the latest update. Recent studies have focused on freshness guarantees. In a single-client situation, a malicious server can perform a replay attack by answering a client's read request with properly authenticated but stale data. This misbehavior can be detected if the client is aware of the last operation he or she has performed [10]. In a group collaboration scenario, a malicious server can perform a forking attack by showing the group members divergent histories to hide their updates from each other. This misbehavior, however, cannot be immediately detected by software-based schemes.

To ensure freshness by detecting forking attacks, while software-based solutions [11, 12, 13, 14] require user-to-user communication and therefore cannot achieve immediate detection, hardware-based solutions [15, 16, 17] add a small piece of trusted hardware to the system. This piece of trusted hardware, which is also called the trusted computing base (TCB), is used as a secure log device or a monotonic counter, preventing a malicious server from reversing the system state to its previous value or presenting different valid system states to different users.

However, today's trusted hardware is resource-constrained, and this makes the TCB the system bottleneck in throughput and latency. To improve the performance while keeping the overall cost low, Costan et al. in a position paper [18] proposed high-level concepts of splitting the functionality of the TCB into two chips: a P (processing) chip with high computing power, and an S (state) chip with secure non-volatile memory (NVRAM). The P chip performs sensitive computations, and the S chip securely stores the system state. Neither results nor implementation were provided in [18].

1.2 Contribution

In this work, we propose a detailed design of an efficient trusted storage system to prove that the concept of splitting the TCB into two chips is practical, in terms of

security and performance. In addition to adopting the existing memory authentication technique, we propose a write access control scheme to prevent unauthorized writes and ensure that all writes are fresh. We also introduce a recovery mechanism to ensure that we can recover the system state from crashes to be consistent with the state stored on the S chip, making our storage system robust against power loss events and accidental/malicious crashes.

We implement our prototype system using an FPGA board and a Linux server. To maximize the performance of our prototype system, we parallelize and pipeline the operations that are carried out on the server and the P chip, and we judiciously partition the functionality across the trusted and untrusted components. We evaluate the system performance with synthetic benchmarks, focusing on the throughput that our system can support, the latency for processing each request, and the performance overhead introduced by performing security checks on trusted hardware. Based on the performance evaluation, we provide customized solutions to performance-focused and budget-focused cloud storage providers by showing the performance that the system can achieve given different hardware requirements. For performance-focused storage providers, our solution can achieve 2.4 GB/s system throughput. For budget-focused storage providers, we provide a single-chip solution that can achieve 377 MB/s system throughput, which is much higher than that of other existing single-chip solutions such as [17]. This single chip needs to run at around 125 MHz and requires some RAM and a hash engine on top of secure NVRAM smart card chip functionality. We believe this work takes a major step in making trusted cloud storage practical from an efficiency and cost standpoint.

The main contributions of this work are summarized below:

- We prove that the concept of splitting the TCB into two chips is practical.
- We provide detailed design, implementation, and evaluation of an efficient trusted storage system with integrity and freshness guarantees.
- We propose a write access control scheme to ensure that all writes are fresh and from authorized writers.
- We propose a crash-recovery mechanism to protect the system from power loss events or crashes.

- Finally, we arrive at a single-chip solution that achieves much higher throughput than existing single-chip solutions.

1.3 Thesis Structure

The rest of this thesis is organized as follows. In Chapter 2, we discuss security issues in cloud storage systems and summarize the related work. We also introduce the related trusted hardware and the memory authentication techniques adopted in our system. In Chapter 3, we describe our prototype system design, introducing the functionality of each system component and explaining how we guarantee integrity and freshness while maintaining high performance. In Chapter 4, we discuss the implementation details, different implementation options, and how we improve the system performance. In Chapter 5, we evaluate the throughput and latency of our prototype system and measure the system overhead using synthetic benchmarks. We also provide customized solutions by showing hardware requirements and estimated performance. In Chapter 6, we conclude our work by describing the objectives we have achieved and discussing future work.

Chapter 2

Background and Related Work

In this chapter, we provide the background information for the understanding of the design and implementation presented in this thesis. We first discuss the desirable security properties of cloud storage systems and the existing mechanisms to achieve these properties. Then, we introduce different types of trusted hardware, including smart cards, the Trusted Platform Module (TPM) [19], and the two-chip model proposed in [18]. Finally, we introduce the memory authentication techniques we adopt in our system design.

2.1 Cloud Storage Security Requirements

Hardware resources on the cloud are shared between multiple users and thus vulnerable to attacks from both outside and inside the cloud. Cloud users, ranging from individuals to enterprises, outsource their data to storage providers and no longer have physical possession of the data. To securely store data and run sensitive computations on the cloud, a cloud storage system should provide following security guarantees:

- Confidentiality: Also known as privacy. The cloud storage provider and other unauthorized users cannot identify the contents of the user's data.
- Availability: The data is accessible from any connected device at all times.
- Integrity: Only the data owner or the authorized users who share the data can modify the data. The data read from the cloud server should be consistent

with an update from authorized users. Any unauthorized modification should be detected by the user or the cloud storage provider.

- Freshness: The data read from the cloud server should have exactly the content that is written by the latest update from the authorized users.

However, except for availability, current cloud storage services do not provide other security guarantees in their Service Level Agreements (SLAs). For example, Amazon S3's SLA [20] and Windows Azure's SLA [21] only guarantee that clients can receive reimbursement when availability falls below 99.9%. This problem is addressed in [22], where a proof-based system is proposed to enable security guarantees in the SLAs of current storage providers.

In this section, we introduce existing schemes that provide each of these security guarantees.

2.1.1 Confidentiality

Encryption is commonly used to preserve data confidentiality. The general concept is that the data owner encrypts the data content before sending it to an untrusted cloud server and discloses the decryption key only to the authorized users.

To make this concept practical in a current cloud storage system, performance issues need to be addressed. One performance issue is how to manage access control and key distribution without introducing a high complexity on computation and communication. There has been a lot of research focusing on developing efficient and fine-grained access control schemes on an untrusted server [23, 24, 25, 26, 27, 28]. In particular, Goyal et al. proposed the Key-Policy Attribute-Based Encryption (KP-ABE) scheme, which is based on the concepts of Attribute-Based Encryption (ABE) proposed in [29], for fine-grained sharing of encrypted data. In a KP-ABE system, each ciphertext is labeled with a set of descriptive attributes, and each private key is associated with an access structure. A user is able to decrypt a ciphertext only if the attributes associated with the ciphertext satisfy the access structure of the user's private key. Yu et al. [28] further combined KP-ABE with the techniques of proxy re-encryption [30] and lazy re-encryption [31], and delegated most computations to cloud

servers while preserving confidentiality, making a KP-ABE system more applicable to cloud storage services.

Another performance issue is that storing encrypted data on the cloud introduces more difficulty to keyword search. A naïve solution is to download all the encrypted data, decrypt it, and search locally. This solution is not practical because it introduces a huge amount of bandwidth cost in a cloud storage system. Researchers have been working on privacy-preserving and effective search services over encrypted data and have proposed different types of searchable encryption schemes using symmetric searchable encryption (SSE) [32, 33, 34] or asymmetric searchable encryption (ASE) [35, 36, 37]. Recent works are focusing on multi-keyword search that enables conjunctive or disjunctive search formulas [37, 38, 39], and ranked search that sends back only the most relevant data to eliminate unnecessary network traffic [40, 41].

In addition to the performance issues, there is another security concern. Although having the users encrypt the data before sending to the cloud can prevent the cloud from learning information from the encrypted data, the cloud can gain information from the users' access patterns. This problem was first addressed in [42], where Private Information Retrieval (PIR) was proposed as a primitive for accessing data from a database without the database learning any information about the retrieved item. However, PIR solutions introduce high computational complexity. Researchers have been working on improving the communication complexity of PIR schemes [43, 44, 45, 46] but have not yet found efficient protocols that are applicable to current cloud storage systems.

2.1.2 Availability

Current cloud storage systems are often implemented with complex, multi-tiered distributed systems on clusters of multiple commodity servers and disk drives. Data unavailability can be caused by failures in any of these layers, such as software bugs, crashes, system planned and unplanned reboots, hardware failures, and power loss events [47]. For example, Amazon S3 experienced an over seven-hour downtime in 2008 [48]. Gmail outage [49] is another example.

Data backup, recovery, and some redundant data storage are needed to reduce the probability of any type of data loss. In distributed file systems, data is divided into

chunks and spread across servers with redundancy to tolerate a fraction of servers' failures and support data recovery. Existing works use two types of redundancy schemes: replication [50] and erasure encoding techniques [51, 52, 53].

A number of works focus on proving retrievability of outsourced data, allowing storage servers to provide availability and integrity guarantees [54, 55, 56]. Juels and Kaliski first proposed the notion of proofs of retrievability (POR) [54]. A POR is a challenge-response protocol that enables a storage provider to prove to a client that a target file is intact, i.e., recoverable without any loss or corruption, with high probability. The basic idea is that a user first encodes some additional information with the file before the file is sent to the server. Then, the user can verify the integrity of the file by challenging the server for a set of data blocks within the file and checking the encoded information from the server's response. In HAIL [52], the POR scheme is combined with data replication and further extended to work on multiple servers. Note that the POR scheme is used for verify the integrity of the user's own data and is not suitable for a multi-client setting.

2.1.3 Integrity

Cloud users no longer have physical possession of data when they outsource the data to storage providers. Therefore, an efficient scheme is required to assure the users that their data stored at remote servers has not been corrupted. This scheme should only allow the authorized users to modify the stored data. Any modification from unauthorized users or storage providers should be detected.

To detect unauthorized data modification, cryptographic hashes, message authentication codes (MACs), and digital signature schemes are commonly adopted in current systems [23, 31, 22]. In addition, a fine-grained access control is needed to separate the writers from the readers in the same file. For example, in Plutus [31], each file is associated with a public/private key pair to differentiate read/write access. For each file, a private key (referred as a file-sign key) is handed only to the writers, while the readers have the corresponding public key (referred as a file-verify key). When updating the file, an authorized writer recomputes the hash of the file (which is the root hash calculated from the block hashes using the Merkle tree technique [57]), signs the hash using the file-sign key, and places the signed hash in the header

of the file. Then, readers can check the integrity of the file by using the file-verify key to verify the signed hash.

2.1.4 Freshness

Freshness verification of outsourced storage is a challenging problem, especially when serving a large number of clients. When a client issues a read request to a cloud server, he or she cannot detect the server's misbehavior using the signature verification scheme mentioned earlier if the server performs a replay attack by maliciously sending the stale data with a valid signature from an authorized user. This kind of attack can cause freshness violations.

In a single-client setting, a replay attack can be detected if the client is aware of the latest operation he or she has performed. Cryptographic hashes can be used to guarantee both integrity and freshness. A naïve approach is to store a hash for each memory block in the client's local trusted memory and verify the retrieved data against the corresponding hash value. For large amounts of data, tree-based structures [57, 58, 59] have been proposed to reduce the memory overhead of trusted memory to a constant size. In tree-based approaches, the tree root represents the current state of the entire memory, and it can be made tamper-resistant and guaranteed to be fresh if stored in trusted memory. The trusted memory can be the client's local memory in this case. For example, the Merkle tree technique [57] is commonly used in outsourced file systems, such as [60] and [61], to reduce the storage overhead at the client-side to a constant. In our design, we also apply the Merkle tree technique but store the root hash securely at the server-side.

In a multi-client system, ensuring freshness is more difficult. In a group collaboration scenario, a cloud server can maliciously prevent each group member from finding out the other has updated the data by showing each member a separate copy of data. This kind of replay attack is also called a forking attack, which was first addressed by Mazières and Shasha in [62, 63]. Figure 2-1 describes a simple forking attack example. Suppose user A and user B are sharing a file and each modifies the file in turn. The server stores a complete history of all operations performed by A and B. Each operation is attached with a signature (σ) of this current operation and the complete history of the previous operations. The signature is signed by the user

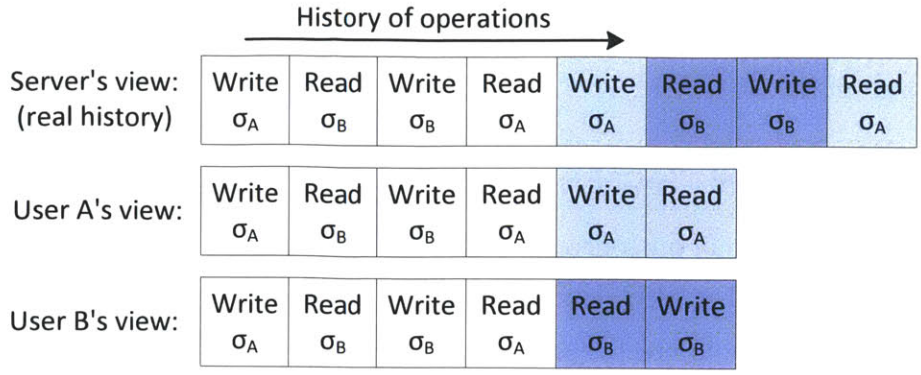


Figure 2-1: A forking attack example

who performs this operation. The user first downloads the complete history from the server, validates the latest signature for each user, and checks whether his or her last operation is in the history. If the history is valid, the user appends the new operation to the history, signs the new history and sends it to the server. In this way, each user is always aware of his or her own previous operations. However, the server can still lie to the users. For example, in Fig. 2-1, the server hides A's second write operation and sends an old history to B. B signs the new read and write operations without detecting the server's misbehavior. After this, A requests a read operation. The server cannot send A the history that contains the two new signatures from B, because these signatures enable A to figure out that B did not see A's previous operation. Therefore, once the server starts to lie, it must ensure that the users can only see divergent histories and cannot see each other's operations again; otherwise, the users can detect this misbehavior.

Mazières and Shasha introduced the forking consistency condition in [63], showing that a forking attack can be detected unless clients cannot communicate with each other and can never again see each other's updates. The SUNDR system [10] was the first storage system using forking consistency techniques on an untrusted server, and there were subsequent fork-based protocols, such as [64] and [65]. User-to-user communication is required to detect server misbehavior: for example, FAUST [11] and Venus [12] allowed clients to exchange messages among themselves. To improve the efficiency, FAUST weakened the forking consistency guarantee, and Venus separated the consistency mechanism from storage operations and operated it in the background. Two recent systems, Depot [13] and SPORC [14], further supported disconnected

operations and allowed clients to recover from malicious forks. In addition to storage services, forking consistency has been recently applied to a more general computing platform [66].

Software solutions mentioned above assured totally untrusted servers and relied on end-to-end checks to guarantee integrity. Although some software solutions can detect and even recover from servers' malicious forks, they require communication among clients and cannot detect attacks immediately. Attaching an additional trusted component to the system can solve this problem.

To ensure trustworthiness, critical functionality is moved to a Trusted Computing Base (TCB). The Trusted Platform Module (TPM) [19], a low-cost tamper-resistant cryptoprocessor introduced by the Trusted Computing Group (TCG), is an example of such trusted hardware. Since the TPM became available in modern PCs, many researchers have developed systems that use the TPM to improve security guarantees.

Attested append-only memory (A2M) proposed by Chun et al. [15] provided the abstraction of a trusted log that can remove equivocation and improve the degree of Byzantine fault tolerance. Van Dijk et al. used an online untrusted server together with a trusted timestamp device (TTD) implemented on the TPM to immediately detect forking and replay attacks [17]. Levin et al. proposed TrInc [16], which is a simplified abstraction model and can be implemented on the TPM. In both TrInc and TTD, monotonic counters were used to detect conflicting statements sent from the untrusted sever to different clients. But, unlike TrInc, in which each user is asked to attach a trusted component to his computer, TTD is at the server side to manage counters for multiple clients.

In our system, we also placed the trusted components at the server side to immediately detect forking attacks as well as to minimize the clients' workload. However, today's trusted hardware is slow, which affects the throughput and latency of the whole system. To solve this problem and enhance efficiency, as suggested in [18], we split the TCB's functionality into a P chip with high throughput and an S chip with secure NVRAM. As a result, we can significantly reduce overheads caused by security checks on trusted hardware. More importantly, we can increase the capabilities of trusted storage systems, e.g., the number of clients and bandwidth, significantly beyond [17, 16].

2.2 Trusted Hardware

Hardware-based security models use trusted hardware as root of trust, providing stronger security guarantees compared to software-only approaches and simplifying software authentication schemes. As mentioned in the previous section, in order to guarantee data freshness and consistency when multiple cloud users are involved, a piece of trusted hardware is required to immediately detect forking attacks. This piece of trusted hardware is used as the trusted computing base (TCB). The concept of a TCB is defined in [67]: the TCB consists of all system elements that are critical for the security of the system and needed to be trusted to protect computation or storage. System elements not included in the TCB need not be trusted to maintain security guarantees. The TCB, which may include hardware, firmware, and software, should be as simple as possible and consistent with the functions it is required to perform. In this section, we introduce some related trusted hardware and the S-P chip model [18] that can be used as a TCB for the system.

2.2.1 Smart Card

A smart card [68] is a microcontroller, embedded in a credit-card-sized plastic card with a set of metal contacts or an antenna, which provides a secure platform for storage, authentication, and cryptographic operations (e.g., encryption, decryption, and digital signing) at low cost. Smart cards can be classified into two groups that differ in both functionality and price: memory cards for storage and processor cards for security applications. In addition, there are two data transmission methods: contact smart cards accessed through smart card readers, and contactless smart cards accessed through an RF interface. The ISO/IEC 7816 family of standards specify the fundamental properties and functions of smart cards, such as the physical shape, numbers of electrical contacts, voltages accepted by the contact points, clock frequency (1–5 MHz), and data transmission protocols. Due to chip-area and cost restrictions, computational and storage resources in a smart card chip are limited. Therefore, it is difficult to use a smart card chip for high-complexity applications.

The key component of a smart card is the embedded microcontroller under the contacts. Figure 2-2 shows the architecture of a smart card microcontroller. It consists

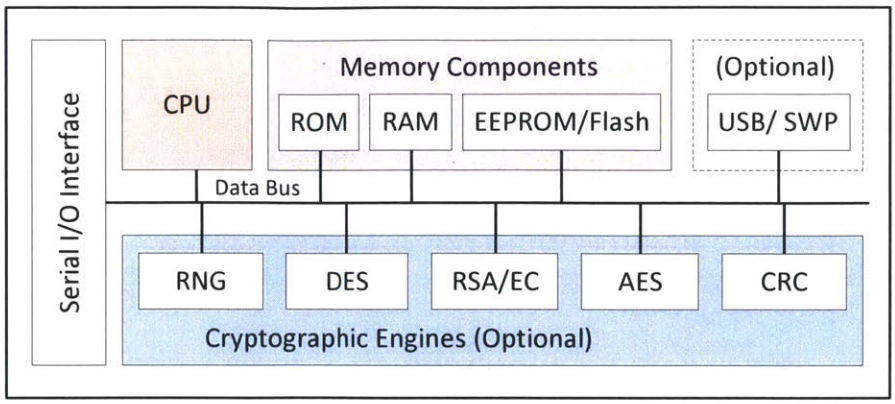


Figure 2-2: Smart card microcontroller architecture

of a processor (CPU) surrounded by data buses, functional blocks, and various types of memory (RAM, ROM, and EEPROM/flash). The ROM stores the chip's operating system fixed at manufacturing time, and the RAM is the processor's working memory. The EEPROM, the smart card's non-volatile memory, provides a small amount of secure and tamper-resistant storage for the user's data such as certificates and private keys, but it has some limitations such as a limited number of write/erase cycles (around 10^5–10^6) as well as relatively long write/erase times (around 1 ms/byte). Some smart cards use flash memory (with 10^5 write/erase cycles) instead for their non-volatile storage. In recent smart card standards, a USB interface is specified as the new I/O interface for high data transmission rates (12 Mbit/s for recent smart cards). In addition, the smart card CPU ranges from a simple 8-bit CPU to a 32-bit RISC architecture, depending on the application and the required processing power. For example, SLE 88CFX4001P, a smart card design in 0.13 µm CMOS technology and released in 2011 for highly secure applications, digital signatures, and access control [69], has a 32-bit RISC CPU, 400 kByte flash, 16 kByte RAM, and a 1 MHz–10 MHz clock, providing 3DES, RSA (up to 2048-bit), and ECC functionality. The semiconductor technology currently used for the fabrication of smart card microcontrollers lies in the range of 0.18 µm to 90 nm.

2.2.2 Trusted Platform Module (TPM)

The Trusted Platform Module (TPM) [19] is a widely available, low-cost, and tamper-resistant cryptoprocessor that resides in most PCs. The TPM, specified by the

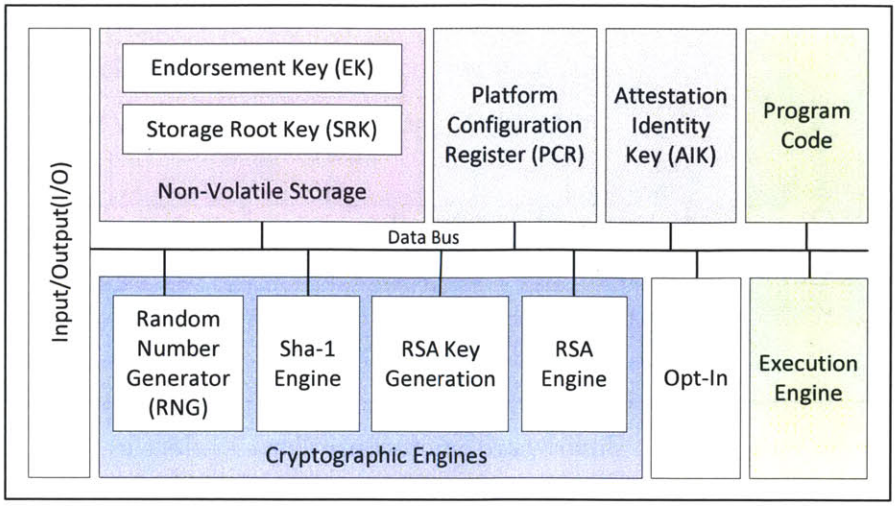


Figure 2-3: TPM component architecture

Trusted Computing Group (TCG), is designed for secure key generation, cryptographic operations, sealed storage, user authentication, and remote attestation of the platform status. Figure 2-3 represents the architecture of a TPM chip, which is usually mounted on the motherboard of a computer and connected to the system via the Low Pin Count (LPC) bus. A TPM chip consists of some non-volatile memory storing cryptographic keys and authorization data, some volatile memory, some cryptographic engines, the platform configuration registers (PCRs) recording the current state values, and a small number of monotonic counters.

Trusting the TPM

During manufacturing time, an endorsement key pair ($EK = (PubEK, PrivEK)$) is generated and stored in the TPM's non-volatile memory, and the endorsement private key ($PrivEK$) is never exposed outside the TPM. The manufacturer generates the endorsement certificate to certify that EK is unique, securely generated and stored in the TPM. A pair of asymmetric keys called attestation identity keys ($AIK = (PubAIK, PrivAIK)$) is generated in the TPM and certified by the privacy CA to represent EK. The attestation identity private key ($PrivAIK$) is protected by the TPM and never exposed. Due to privacy concerns, $PrivAIK$ instead of $PrivEK$ is used as the TPM's signing key to sign the messages that are generated inside the TPM such as the PCR values or the keys generated inside the TPM. The signing key

PrivAIK can be trusted because it is securely generated, verified, and stored inside the TPM. The message signed by *PrivAIK* can be protected from tampering and therefore can also be authenticated.

TPM Limitations

One primary goal of a TPM is to assure the integrity of the whole platform by the authenticated boot process. The TPM, together with the BIOS, forms a root of trust and proves to a third party that only an unaltered trusted OS is loaded during the boot process and is running on the PC that the TPM is bound to. The authenticated boot process has been used to allow web-servers [70] and peer-to-peer systems [71] to provide stronger security guarantees. However, this authenticated boot process cannot prevent bugs in the authenticated software and is vulnerable to physical attacks because the LPC bus, which connects the TPM to the host computer, is not completely secure. As a result, researchers have reduced the TCB to only a single TPM chip without the trusted software [17, 16]. However, the TPM also limits system performance, because it has limited computational capabilities and is connected to the host computer via the slow LPC bus, which is a 4-bit wide bus running at only 33.3 MHz clock frequency.

2.2.3 S-P Chip Model

Trusted hardware designed for secure storage applications requires NVRAM for long-term storage as well as control logic, data transmission logic, and cryptographic engines for encryption, decryption, or authentication. However, it is difficult to achieve high-performance computation while keeping cost low by combining all the building blocks on a single chip, because the process for the NVRAM and the process for high-performance computational logic are different. Combining two processes on a single chip is not practical due to high complexity and low wafer yield, while under an NVRAM-only process, the transistors are much slower than that in the regular CMOS process. To avoid this problem, Costan et al. proposed the concepts of splitting the functionality of the TCB into two chips: a P (processing) chip with high computing power but only volatile memory, and an S (state) chip with secure non-volatile mem-

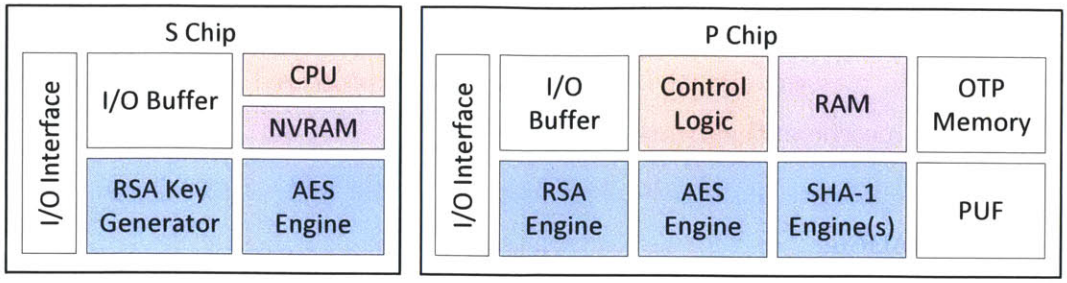


Figure 2-4: S and P chip functional units

ory (NVRAM) [18]. Figure 2-4 shows the functional units of a P chip and an S chip. The P chip and S chip should be securely paired in order to serve as a single TCB. A Physical Unclonable Function (PUF) is used to bind the P chip to the S chip. We first introduce the PUF, and then describe the pairing scheme.

Physical Unclonable Function (PUF)

A Physical Unclonable Function (PUF) is a function that is embodied in a physical structure and maps a set of challenges to a set of responses [72]. The mapping is static but random, and it should not be replicable. PUFs can be easily implemented with integrated circuits (ICs): no two ICs even with the same layouts have identical timing and delay responses due to the manufacturing process variations.

Suh and Devadas discussed how to use PUFs for low-cost authentication of ICs and for cryptographic key generation [73]. Figure 2-5 shows how to use PUFs to generate volatile cryptographic keys. In the initialization step, an error correcting syndrome is generated from the PUF circuit output, using functions such as a BCH code. The syndrome is public information, and it needs to be saved, either on-chip or off-chip. In the re-generation step, the syndrome is used to correct any changes in the PUF circuit output so that the PUF can re-generate the same output as the output generated in the initialization step. This output can be directly used as a symmetric key, or it can be used as a static but random seed to an asymmetric key generation algorithm.

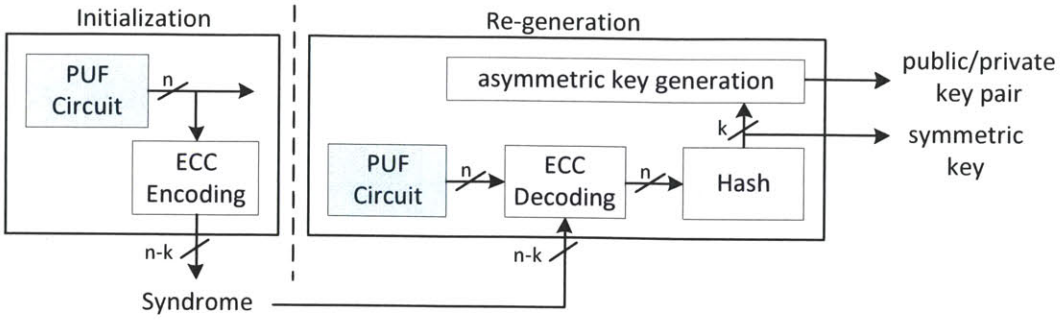


Figure 2-5: Cryptographic key generation with PUFs

S-P Pairing Scheme

The S and P chips are securely paired via an untrusted channel at manufacturing time. The pairing relies on a PUF on the P chip to generate a symmetric key. The key generation process is the same as Fig. 2-5. The symmetric key is shared and stored in the S chip's NVRAM. The syndrome for re-generating the same PUF output (and re-generating the P chip's symmetric key) is stored in plain text and requires an integrity check during a future booting process.

The pairing scheme is described as follows:

1. The P chip uses its on-chip PUF to generate a symmetric key SK and a syndrome ECC.

2. The S chip generates an endorsement key pair ($PubEK$, $PrivEK$), stores the key pair, and outputs $PubEK$.

3. The manufacturer signs $PubEK$, generates $ECert$, and sends $ECert$ (containing $PubEK$) to the P chip.

4. The P chip verifies $ECert$ by checking the manufacturer's CA key against the key in its ROM, then encrypts SK with $PubEK$.

5. The P chip outputs encrypted SK, ECC, and $HMAC_{SK}(ECC)$.

6. The manufacturer stores ECC and provides the P chip's output to the S chip.

7. The S chip decrypts the encrypted SK with $PrivEK$ and stores SK in its NVRAM. Then, the S chip uses SK to verify $HMAC_{SK}(ECC)$, and outputs

the signature of ECC, $\sigma_{PrivEK}(ECC)$, if the verification is successful.

8. The manufacturer packages the S and P chips with the public state: $ECert$, ECC, and $\sigma_{PrivEK}(ECC)$.

Note that the manufacturing process requires integrity guarantees in the channel between the S chip and the manufacturer for issuing the endorsement key certificate $ECert$, similar to the TPM model. The channel between the S chip and P chip can be completely untrusted.

During manufacturing time, the S chip generates $(PubEK, PrivEK)$ and then stores $PrivEK$ as well as the P chip's symmetric key SK in its NVRAM. After the S and P chip are securely paired, this chip pair can be attached to the cloud server and serve as a single TCB to store the system state. When the cloud server boots, the P chip re-generates its symmetric key SK, and the system state as well as $PrivEK$ are transmitted from the S chip to the P chip. The boot process is described as follows:

1. The server presents $ECert$, ECC, and $\sigma_{PrivEK}(ECC)$ to the P chip.

2. The P chip verifies $ECert$ against the key in its ROM, verifies ECC against $\sigma_{PrivEK}(ECC)$ using $PubEK$ in $ECert$.

3. The P chip re-generates SK if the verification is successful.

4. The P chip generates a boot nonce n, outputs n and $HMAC_{SK}(n)$.

5. The server provides the P chip's output to the S chip.

6. The S chip verifies n against $HMAC_{SK}(n)$, then outputs the system state s, $PrivEK$ encrypted under SK, and $HMAC_{SK}(s||n)$.

7. The server presents $ECert$ and the S chip's output to the P chip.

8. The P chip decrypts the encrypted $PrivEK$ using SK, checks that $PrivEK$ corresponds to $PubEK$, checks s against $HMAC_{SK}(s||n)$, and stores $PrivEK$ and system state s in its RAM.

2.3 Memory Authentication Techniques

Memory authentication can be defined as the ability to verify that the data read from memory at a given address is the data written most recently at this address [74]. In order words, memory authentication is used to verify integrity and freshness of the data stored in the memory. Integrity trees are commonly used for memory authentication [57, 58, 59]. The general concept of these tree-based methods is to split the memory to be protected into multiple equal-sized blocks, then apply a function f called the authentication primitive to each memory block to generate the leaf nodes of a balanced A-ary integrity tree. The remaining tree levels are created by recursively applying f to the A-sized groups of tree nodes starting from leaves until a single node, the root of the tree, is generated. The root node captures the current state of the memory space. The root node needs to be stored in the trusted memory to protect against replay attacks or data corruption, while other tree nodes can be stored in the untrusted memory, which is larger and cheaper. The root node is made tamper-resistant by being stored in the trusted memory; therefore, any data corruption in the memory space can be detected by the tree authentication procedure.

Tree Authentication Procedure. To authenticate a memory block fetched from untrusted memory, the root node is recomputed by recursively applying f to the tree nodes (stored in untrusted memory) on the path from the corresponding leaf to the root together with their siblings. If the memory block and all the tree nodes used in re-computation were not tampered with, the recomputed root node must match the one stored in the trusted memory.

Tree Update Procedure. Each time a memory block is modified, the root node should be updated to reflect the change in the state of the memory space. When the memory block is modified, the tree nodes on the path from the corresponding leaf to the root together with their siblings should be first verified by the tree authentication procedure. Then, the leaf node is updated by applying f to the modified memory block, and all the nodes on the path (including the root node) are also recomputed using f. The root node is stored back in the trusted memory, and the rest of the nodes that have been updated are stored back in the untrusted memory.

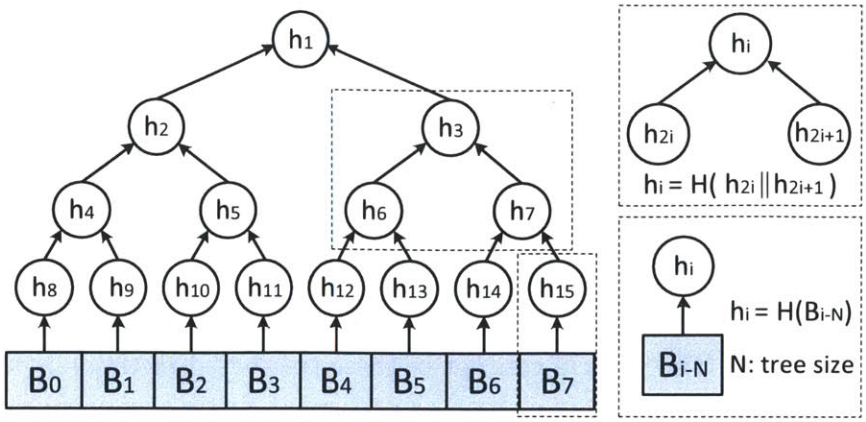


Figure 2-6: A Merkle tree example for a disk with 8 blocks

2.3.1 Merkle Tree

In our prototype system, we built a Merkle tree [57] to authenticate the disk space. Merkle trees were first introduced by Merkle [57], and Blum et al. used it to check integrity of memory contents [75]. In a Merkle tree, the authentication primitive f is a cryptographic hash function. Figure 2-6 gives an example of a binary Merkle tree over 8 memory blocks, whose leaves are hashes of memory blocks, and internal nodes are hashes of their children. The Merkle tree authentication and update procedures follow the general procedures described above. The Merkle tree's root hash represents the current state of the memory space because of the hash function's collision resistance property, i.e., any bit change in the memory space produces a different root hash in practice.

2.3.2 Integrity Tree Caching

In the original tree-based schemes, only the root node is stored in trusted memory. The direct implementation introduces large overhead in terms of memory bandwidth and execution time, because each time a memory block is authenticated, all the tree nodes on the path from the corresponding leaf to the root together with their siblings need to be read from external memory and checked by recursively executing the authentication primitive f. To improve efficiency, Gassend et al. proposed to cache some tree nodes in trusted memory [76]. They demonstrated the concept by storing some tree nodes of a binary Merkle tree in the on-chip L2 cache, assuming the on-chip

cache is trusted. The main idea is that once a tree node is authenticated and cached on-chip, it can be seen as a local tree root. Therefore, the authentication procedure can be ended as soon as it reaches a cached tree node, reducing the original $\log N$ overhead in terms of memory bandwidth and execution time, where N is the number of memory blocks.

Chapter 3

System Design

3.1 Design Goals

To build a practical cloud storage system that can immediately detect integrity and freshness violations, our system design should achieve the following goals:

1. Integrity and freshness guarantees: any integrity or freshness violations such as unauthorized writes or forking attacks should be immediately detected by our system or by the clients.

2. Simple tasks done by clients: data checking and management done by clients should be simple. Communication between clients is not required except for initially sharing keys used for write access control.

3. Simple API: the API between the server and its clients should be simple. Each read or write operation requires only one request/response transaction. The clients should be separated from the back-end of the system so that any future modification of the system will not change the API.

4. Minimal local storage: the storage requirement at the client side should be minimal. This would be beneficial for thin clients.

5. Acceptable overhead: our system should maintain high performance despite adding integrity and freshness guarantees. The performance overhead should be acceptable compared to the cloud systems without these security guarantees.

6. Acceptable cost: to achieve above goals, our system should not add too much cost to storage providers compared to the existing hardware-based solutions.

7. Customized solutions: based on our prototype system, storage providers should be able to adjust their systems according to the performance and cost trade-off.

3.2 System Overview

To build a trusted cloud storage system that efficiently guarantees integrity and freshness of the data stored on the cloud, we attach a piece of trusted hardware to an untrusted server and use the S-P chip model mentioned in Section 2.2.3 as the trusted hardware; that is, the functionality of the trusted hardware is split into S and P chips. The P chip, which can be an FPGA board or an ASIC, has high computing power but only volatile memory, while the S chip, which can be a smart card, has secure NVRAM but only constrained resources.

Figure 3-1 represents the system model. For simplicity, we make the assumption that a single-server system provides its clients with a block-oriented API to access a single large virtual disk. The clients access the cloud storage service via the Internet; the untrusted server is connected to the disk and the trusted S-P chip pair. To access/modify the data stored on the cloud, the clients send read/write requests, wait for the responses, and use the responses to check data integrity and freshness. The untrusted server schedules requests from the clients, handles disk I/O, and controls the communication between the P chip and S chip. On the other hand, the S-P chip pair shares a unique and secret HMAC key with each client, and thus essentially becomes an extension of the clients. The S-P chip pair is trusted to update and store the system's state, manage write access control, verify data integrity, and authenticates the responses sent to the client using the HMAC key. More specifically, the P chip does all sensitive computations and verifications and stores the system's state when the system is powered; the S chip is responsible for securely storing the system's state across power cycles. This scheme simplifies the computation and verification that need to be done by clients in software-based solutions, and abstracts away the design and implementation details and complexity.

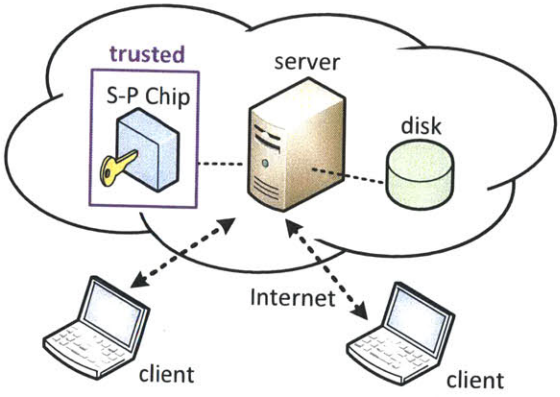


Figure 3-1: System model

3.2.1 Threat Model

In our simplified system model shown in Fig. 3-1, there is one cloud server, one large disk, one S-P chip pair, and multiple clients. The cloud server is untrusted; it may answer the clients' read requests with stale or corrupted data. Furthermore, the cloud server may pretend to be a client and overwrite the client's data. The disk is vulnerable to attackers and hardware failures, so the data stored on the disk may not be correct. The connections between the cloud server and the other system components (the S-P chip pair, the disk, and the clients) as well as the communication between the S and P chips are also untrusted. Any message traveling on these channels may be altered to an arbitrary or stale value. A client is trusted with the data he/she is authorized to access, but the client may try to modify the data outside the scope of his/her access privilege.

3.2.2 Chain of Trust

The S chip and P chip are securely paired during manufacturing time and thus can be seen as a single TCB. The details of the S-P chip pairing procedure is described in Section 2.2.3. Similar to the TPM model described in Section 2.2.2, we use the S-P chip pair as the root of trust and establish the chain of trust, allowing clients to trust the computation and verification performed by our storage system.

During manufacturing time, the P chip generates a symmetric encryption key SK; the S chip generates an endorsement key pair $(PubEK, PrivEK)$ and stores the

P chip's SK. The manufacturer, who can be seen as a CA, signs $PubEK$ and produces the endorsement certificate ($ECert$) to promise that $PrivEK$ is only known to the S-P pair. After the two chips are securely paired, the P chip re-generates SK, and the S chip uses SK to share the system's state as well as $PrivEK$ with the P chip. When a client connects to the cloud server, ($ECert$) is presented to the client, and the CA key is verified by the client's software against a list of trusted CAs. If the verification is successful, which means $PubEK$ can be trusted, the client can secretly share an HMAC key with the S-P chip pair attached to the server by encrypting the HMAC key under $PubEK$. The S-P chip pair can then use the HMAC key to authenticate the response messages sent to the client.

In this work, we also provide a single chip solution where the S chip and P chip can be integrated into an ASIC. This chip can be viewed as a smart card running at a higher frequency with additional logic for data hashing. The detailed specification is described in Section 5.3.2. In this solution, the S-P chip pair becomes a single chip, the communication between the S and P chips becomes on-chip and thus can be trusted. Therefore, the pairing scheme is not required. This single chip also generates an endorsement key pair ($PubEK$, $PrivEK$) and a symmetric encryption key SK during manufacturing time, and follows the same chain of trust model described above.

3.3 Notation

Table 3.1 lists the symbols used to describe our design concepts and protocols.

3.4 System Essentials

We implement a prototype system to prove practicality of the concept of splitting the TCB into two chips, to analyze performance factors, and to further customize solutions for performance-focused and budget-focused cloud storage providers based on our evaluation.

Figure 3-2 represents our prototype system architecture, which consists of two parts: an untrusted server with an untrusted disk, and a trusted pair consisting of an S chip and a P chip. In this section, we introduce the characteristics of our

Table 3.1: Notation

Notation	Description
$PubEK$	the public key of the endorsement key pair
$PrivEK$	the private key of the endorsement key pair
$ECert$	the endorsement certificate
SK	the symmetric key generated by the P chip
ECC	the syndrome used to re-generate SK
$\sigma_{PrivEK}(ECC)$	the signature of ECC
H_X	the hash value of X
$\{M\}_K$	the encryption of message M with the encryption key K
$HMAC_K(M)$	the HMAC of message M generated by the key K
MT_{XYN}	the message type used to indicate that a message is sent from X to Y, where X and Y can be C (client), P (the P chip), or S (the S chip), and N is a number that indicates the sub-type of the message
n	nonce
s	the system's state, which is the root hash in our system
S_{id}	the session ID
S_C	the session cache entry used to store a session key ($Skey$)
$Skey$	the HMAC key (session key) shared between a client and the S-P chip
$Pkey$	the processed key, an HMAC key encrypted under SK, $\{Skey\}_{SK}$
$Wkey$	the write access key associated with a data block
V_{id}	the revision number associated with a data block
W	$Wkey \| \| V_{id}$, the write access information
$data$	the data of a data block
B_{id}	the block number of a data block
N_{id}	the node number of a tree node (in BFS-traversal order)
$leaf$	the value of a Merkle tree leaf node, which is also called a leaf hash
$leaf_{arg}$	the argument of a leaf hash ($H_{data} \| \| V_{id} \| \| H_{Wkey}$)
V	the valid bit of a tree cache entry
L	the left child valid bit of a tree cache entry
R	the right child valid bit of a tree cache entry
C	the tree cache entry in which a Merkle tree operation is performed
C_P	the parent tree cache entry in a VERIFY operation
C_L	the left child tree cache entry in a VERIFY operation
C_R	the right child tree cache entry in a VERIFY operation
H	the value of a tree node
$C_{P_{old}}$	the parent tree cache entry of an evicted tree node
C_{Path}	the tree cache entries on an update path from a tree leaf to the root
C_{Sibs}	the tree cache entries of the sibling nodes along an update path
LV	the height of the Merkle tree
H_{zero}	the leaf hash value of the initial Merkle tree

prototype system and how we achieve the security and performance goals mentioned in Section 3.1. The detailed hardware techniques we used for performance enhancement are discussed in Chapter 4.

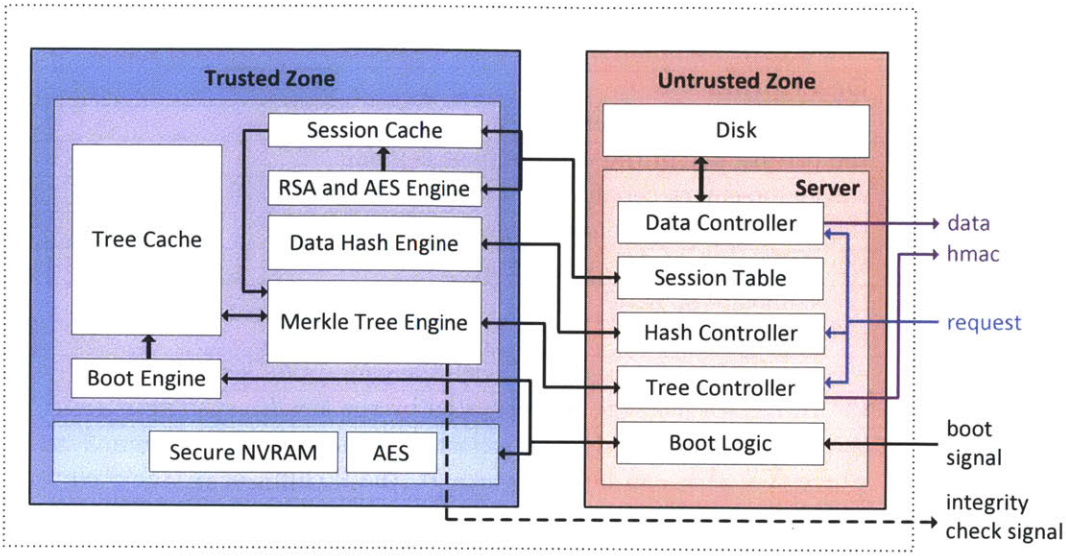


Figure 3-2: System overview

3.4.1 Memory Authentication

To verify the integrity and freshness of the data stored on the disk, we build a Merkle tree [57], which is a binary hash tree described in Section 2.3, on top of the disk. The hash function's collision resistance property allows the Merkle tree root, also called the root hash, to represent the current state of the entire disk. The root hash is calculated, updated, and stored in the S-P chip pair, so it can be trusted against any corruption or replay attacks. The root hash is always fresh, and leaf hashes are verified by the S-P chip pair to be consistent with the root hash and sent to the clients in the response messages, which are authenticated using HMACs. Therefore, a client can detect any data corruption or forking attack by verifying the received data against the received leaf hash. In this scheme, there is no need to communicate with other clients and check the consistency of the data.

To improve efficiency of the Merkle tree authentication, we let the P chip cache some of the tree nodes. The caching concept is similar to what Gassend et al. proposed in [76]: once a tree node is authenticated and cached on-chip, it can be seen as a local tree root. While Gassend et al. use the secure processor's L2 cache, which is on-chip and assumed to be trusted, to cache tree nodes, we cache the tree nodes on the P chip and let the software running on the untrusted server OS (on a commodity PC) control the caching policy. The caching policy is controlled by the untrusted

server instead of the trusted hardware because software can easily switch between different caching policies to match the data access patterns requested by different cloud-based applications.

In our prototype system, the entire Merkle tree is stored on the untrusted server. The P chip's Merkle tree engine (shown in Fig. 3-2) updates the tree to reflect the write operations and verifies the tree nodes to authenticate read operations. The Merkle tree stored on the server is also updated during each write operation. The P chip updates and stores the root hash when the system is powered; the S chip maintains the root hash value across power cycles and sends it to the P chip when the system boots. The P chip's Merkle tree engine is also responsible for computing the first root hash when the disk is initially empty and the S chip has not stored any root hash yet.

Under our Merkle tree caching protocol, the P chip caches tree nodes in its tree cache. Each entry in the tree cache contains the cached node's node number, its hash value, its valid bit (V) indicating whether the hash is verified to be correct or not, and its child nodes' valid bits (L and R). The Merkle tree engine manages the cached nodes according to the cache management commands sent from the server's tree controller, which controls the caching policy.

There are three cache management commands: (1) the LOAD command asks the Merkle tree engine to load a certain tree node and evict a cached node if necessary; (2) the VERIFY command asks the Merkle tree engine to authenticate two child nodes against their parent node; (3) the UPDATE command asks the Merkle tree engine to calculate and update the tree nodes on a certain path from a leaf node to the root. Note that these commands are sent from the untrusted server via an untrusted connection, therefore, the P chip should do additional checks against each cache management command to prevent attacks from violating the integrity and freshness guarantees. If any check or any tree node verification fails, the Merkle tree engine raises the integrity check signal and reports the error to the system manager. Table 3.3 has a more detailed description of the operations and checks done by the Merkle tree engine for each command.

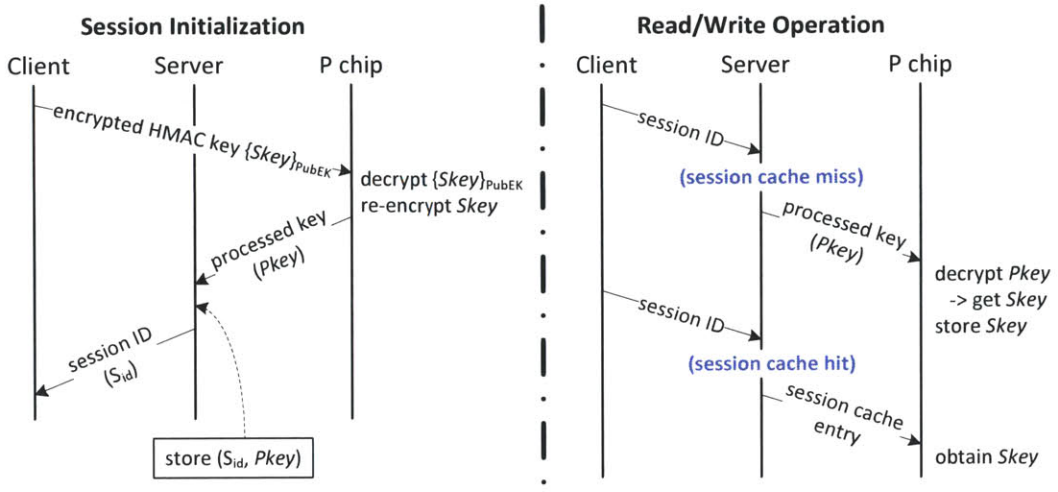


Figure 3-3: HMAC key management protocol

3.4.2 Message Authentication and HMAC Key Management

As mentioned in the threat model, the connection between each client and the trusted S-P chip pair is untrusted. To prevent message corruption or replay attacks, we create an authenticated channel over the untrusted connection using the client's HMAC key. A client's HMAC key should be kept secret, only known to the client and the S-P chip pair. When sending a request, the client needs to send an HMAC along with the request; similarly, the trusted S-P chip pair needs to generate an HMAC over its response. When receiving a client's request, the P chip checks the request against the HMAC from the client and rejects any invalid request. When receiving an HMAC from the S-P chip pair, the client uses it to authenticate the read/write operation. A more detailed description of how the S-P chip/the client generates and checks HMACs can be found in Section 3.5, Table 3.2, and Table 3.3.

Figure 3-3 describes how we securely share the HMAC key between a client and the S-P chip pair with minimal performance overhead even when the storage system serves multiple clients. The communication between the client and the server is based on a session-based protocol.

Each time a client connects to the server, the client first requests a session for future communication. Each session has a unique HMAC key, so an HMAC key is also called a session key. To share the HMAC key with the S-P chip, the client encrypts the HMAC key with *PubEK* and sends the encrypted key along with the

request for the new session. When receiving the client's session-request, the server assigns a new session ID to the client and forwards the encrypted key to the P chip. The P chip can decrypt the encrypted HMAC key using $PrivEK$, which is only known to the S-P chip pair. To eliminate the need for key transmission in future read/write operations, the server stores HMAC keys in the encrypted version along with the corresponding session IDs in its session table, and the P chip's session cache stores a subset of the HMAC keys in plain text to reduce the number of decryption operations. When the client issues a read/write request, the client simply sends the session ID instead of the encrypted session key, and the server can find the corresponding key using its session table. If the key is not cached on the P chip, the P chip will decrypt the encrypted key sent from the server. To reduce the performance overhead when handling read/write requests, we do not store the original encrypted key sent by the client on the server; instead, we let the P chip generate the processed key by re-encrypting the HMAC key using the P chip's symmetric key SK, because symmetric key decryption is much faster than public key decryption.

3.4.3 Write Access Control

We let the S-P chip pair manage the write access control to ensure fresh writes and prevent unauthorized writes from the server and clients. Under our write access control, no unauthorized user or malicious server can overwrite a block without being detected by the S-P chip pair or an authorized user. In addition, all writes are ensured to be fresh; that is, an old write from an authorized user cannot be replayed by the server. Note that we do not focus on read access control in our storage system, because a client can prevent unauthorized reads by encrypting the data locally, storing the encrypted data on the cloud, and sharing the read access key with authorized users without changing the system design.

To manage a situation where a data block has multiple authorized writers, we assume a coherence model in which each user should be aware of the latest update when requesting a write operation. For example, considering the following operation sequence on a certain block: "A read, B read, A write, B write", B should be informed that A has updated the data. Under this coherence model, we achieve write access control and fresh write guarantees as follows. To distinguish authorized writers from

others, each set of blocks with the same authorized writers has a unique write access key ($Wkey$), which is only known to the authorized writers and the S-P chip pair. In addition, to protect data against replay attacks, each block is associated with a revision number (V_{id}), which increases during each write operation, and each leaf node of the Merkle tree should reflect the change of the associated $Wkey$ and V_{id}. In this way, any change of $Wkey$ and V_{id} in any data block would change the root hash, and therefore cannot be hidden by the untrusted server. In the following paragraphs, we describe this write access control scheme in more detail.

For each data block, in addition to the data itself, the server also stores the block's write access information, which consists of the hash of the write key (H_{Wkey}) and the revision number (V_{id}). To guarantee that the write access information stored on the server is correct and fresh, we slightly modify the original Merkle tree by changing the function used to compute each leaf node to reflect any change of the write access information. The new formula to compute each leaf node is shown in Equation 3.1, where H refers to the cryptographic hash function used in the Merkle tree. It is similar to adding an additional layer under the bottom of the Merkle tree. Each leaf node in the original Merkle tree now has three children: the original leaf hash (H_{data}), the write key (H_{Wkey}), and the revision number (V_{id}). We refer the children of each leaf node to $leaf_{arg}$.

$$leaf = H(H_{data}||V_{id}||H_{Wkey}) = H(leaf_{arg}) \tag{3.1}$$

Figure 3-4 describes the concept of how the P chip manages the write access control, and the exact API is described in Table 3.2 and Table 3.3. When a client reads a certain block, the server sends the latest revision number (V_{id}) along with the original response. On the next write to the same block, the client encrypts the write key ($Wkey$) and the new revision number (V_{id}+1) under the HMAC key ($Skey$), then sends the encrypted message ($\{W\}_{Skey}$) as well as the hash of the new write key (H_{Wkey^*}) along with the write request. The new write key ($Wkey^*$) is different from the original write key ($Wkey$) only if the client wants to change the access information, e.g., revoking a certain user's write access. $\{W\}_{Skey}$ can only be decrypted by the P chip. The P chip first authenticates the access information stored on the server by checking it against the verified leaf node. Then, the P chip checks the access

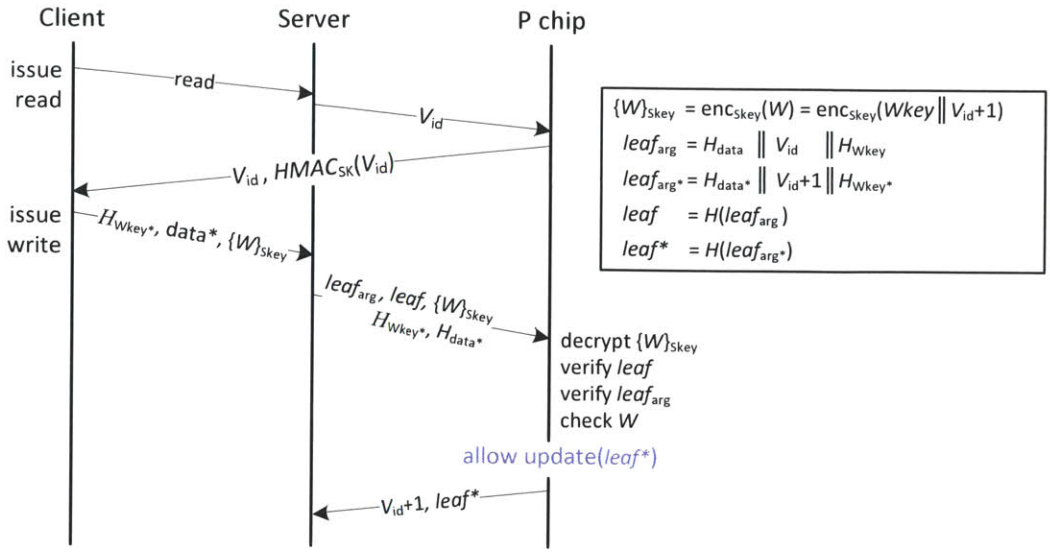


Figure 3-4: Write access control example

information provided by the client against the one stored on the server. If the write keys are not consistent, the P chip rejects the write request directly. If the new revision number provided by the client is not larger than the one stored on the server by 1, the P chip sends the client the correct revision number (the one stored on the server) to inform the client that some other authorized users have already updated the block and the client's write request needs to be re-issued. If all verification is successful, the P chip generates the new leaf value to reflect the change of the access information and performs tree updates. In this scheme, only the users with correct *Wkey* can increase the revision number and send a valid $\{W\}_{Skey}$. The server cannot perform replay attacks because it cannot re-use $\{W\}_{Skey}$ from an authorized user and modify the revision number without knowing the user's *Skey*, and it cannot generate a valid $\{W\}_{Skey}$ (using its own *Skey*) without knowing the correct *Wkey*. In addition, the access information for each block (H_{Wkey} and V_{id}) is protected under the Merkle tree scheme, so it can be guaranteed to be correct and fresh.

The write access control scheme described above is used after the client has generated a write key and stored H_{Wkey} on the server. The P chip should be able to deal with the first write to a data block if the write key of the block has not been yet established since the system is initially set up. As described in Section 3.4.1, when the disk is initially empty, each leaf node of the Merkle tree is assigned to a special

value, and the P chip generates the first root hash based on this value. The P chip does not check the access of the first write to each data block. After the first write, the write key has been established, and the P chip starts to check subsequent writes following the write access control scheme. In a real cloud storage case, when a client requests to have a chunk of data blocks, the server can first establishes a write key for these data blocks and shares the write key with the client. Then, the client can overwrite the write key to prevent the server from modifying the data.

3.4.4 System State Protection against Power Loss

In our prototype system, while the S chip is responsible for storing the root hash, which is the system's state, across power cycles, the P chip computes and updates the root hash in its volatile memory (the tree cache), in which the data stored is vulnerable to power loss. To prevent the server from maliciously or accidentally interrupting the P chip's supply power and losing the latest system state, the P chip should keep sending the latest root hash to the S chip and delay the write responses to be sent to the clients until the latest root hash is successfully stored on the S chip. When a client receives a write response, the system guarantees that the system state that can reflect this write operation is securely stored in the NVRAM. Considering that the S chip has long write times (around 1 ms/byte for smart cards as mentioned in Section 2.2.1), in order to maintain high throughput of the system, the P chip deals with new requests from the clients but stores the responses in an on-chip buffer while waiting for the S chip's acknowledgement for successfully saving the root hash.

Figure 3-5 illustrates our root hash storage protocol. After a Merkle tree update, the P chip generates an HMAC to authenticate the write operation, stores the HMAC in the on-chip buffer instead of sending to the client immediately. When receiving a getRootP() request from the server, the P chip sends the current root hash (s) and a random nonce (n) to the server. Then, the server passes the information to the S chip. While waiting for the acknowledgement from the S chip, the P chip keeps working on clients' requests and generates responses ($HMAC_{wi}$ and $HMAC_{ri}$). The P chip stores the responses that are used to authenticate write operations or read operations that access the same blocks written by buffered write operations. The P chip releases the responses only if it receives a valid acknowledgement from the

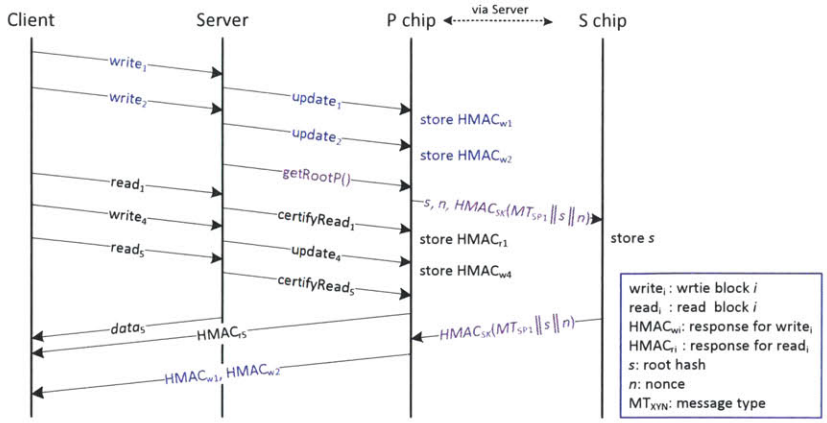


Figure 3-5: Root hash storage protocol

S chip indicating that the corresponding root hash has been successfully stored. In the two-chip prototype system, the communication between the S chip and P chip is untrusted. In order to securely store the current root hash back on the S chip, the P chip sends $HMAC_{SK}(MT_{PS1}||s||n)$ along with the root hash and a nonce, and the S chip uses $HMAC_{SK}(MT_{SP1}||s||n)$ as the acknowledgement to protect against forging and replay attacks, where MT_{PS1} and MT_{SP1} are message types used to differentiate between the HMACs sent by the P chip and by the S chip so that the server cannot maliciously acknowledge the P chip. On the other hand, the communication between the S and P chips becomes trusted in a single chip solution, and therefore the HMACs for the root hash storage protocol are no longer needed.

3.4.5 Crash-Recovery Mechanism

Under our root hash storage protocol, when a client receives a write response, the system guarantees that the root hash stored on the S chip reflects the client's write operation. Even if the power of the P chip is maliciously or accidentally interrupted, the server cannot perform replay attacks without being detected by the client. In this section, we discuss how to guarantee that even if the server crashes (either accidentally or maliciously), our system is able to recover from the crash; that is, the data stored on the disk can be recovered to the state that is consistent with the root hash stored on the S chip. This is essential for a robust storage system in order to maintain service against crashes or unplanned re-boots.

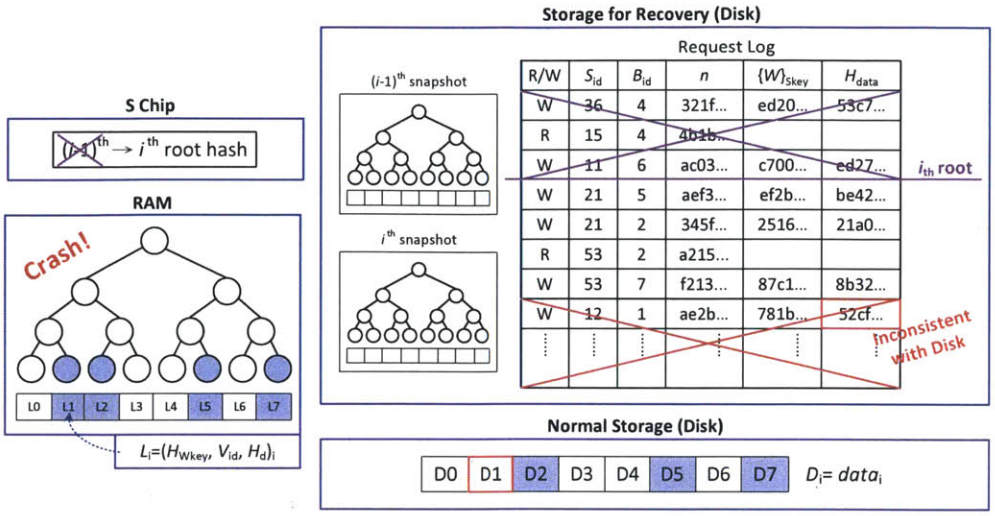


Figure 3-6: Crash-recovery mechanism

There are two possible scenarios in which the disk state is not consistent with the root hash stored on the S chip after the server re-boots from a crash. One happens when the the server crashes after the root hash is stored on the S chip but the data has not yet been stored on the disk. The other one happens when the server crashes after the data is stored on the disk but the corresponding root hash has not yet been stored on the S chip. To prevent the first scenario, the server should first flush the data into disk before it passes the root hash to the S chip, eliminating the possibility that the root hash is "advanced" to the disk state. To recover from the second scenario, we keep a request log on the disk where we save a snapshot of the Merkle tree and the leaf arguments of each block $(H_{Wkey}, V_{id}, H_{data})$.

Figure 3-6 shows how the recovery scheme works. When the server sends a getRootP() request to the P chip and obtains the latest root hash (the i^{th} root hash), it flushes all the data into the disk, takes a snapshot of the current Merkle tree and write access information for each block (the i^{th} snapshot), and stores the snapshot on the disk. After the data and the snapshot are stored on the disk, the server sends the root hash (as well as the nonce) to the S chip, and continues to work on new requests from clients. The Merkle tree and the access information stored in the RAM are updated by new write requests. The information in each new write request except for the write data (shown in Fig. 3-6) is stored in the request log on the disk. The read request that reads the data from a write request in the request log is also stored.

Algorithm 1 Recovery Procedure

```
1: procedure RECOVERSYSTEM
2:     Re-boot the P chip
3:     Ask the S chip to share the root hash (s) and PrivEK with the P chip
4:     Reload the snapshot that contains the root hash (s) into RAM
5:     i ← 1, done ← false
6:     while i ≤ N and done ≠ true do          ▷ N equals the depth of the request log
7:         Read request i from the request log
8:         if Request i is a read request then
9:             Re-perform request i
10:        else
11:            Read D_{B_id} from the disk
12:            if Hash of D_{B_id} ≠ H_data then      ▷ Request i is newer than the disk state
13:                done ← true                                  ▷ Recovery completes
14:            else
15:                Re-perform request i
16:            end if
17:        end if
18:        i ← i+1
19:    end while
20:    s, n ← getRootP()                  ▷ Ask the P chip to return the latest root hash
21:    Take a snapshot of the current Merkle tree and access information
22:    Store the snapshot on the disk
23:    HMAC_SK(MT_PS1||s||n) ← storeRoot(s, n)          ▷ Send the root hash to the S chip
24:    Clear the request log
25:    Send HMAC_SK(MT_SP1||s||n) to the P chip, and the P chip releases all HMACs
26: end procedure
```

In short, the request log buffers all the requests whose responses are buffered by the P chip. Note that we keep the previous snapshot on disk so that the system is able to recover from a crash that happens after the server sends the root hash but before the root hash is successfully stored. When the server receives the acknowledgement from the S chip saying that the i^{th} root hash is successfully stored, it clears all the requests that are not newer than the i^{th} root hash from the request log. Under this scheme, the system is able to recover from the server's crash following the recovery procedure described in Algorithm 1. After the server reboots, it re-boots the P chip and obtains the stored root hash from the S chip. The server re-loads the snapshot that is consistent with the S chip's root hash, and re-performs the requests in the request log until the root hash is consistent with the disk state. In our recovery mechanism, we assume that each write of a data block is atomic; that is, the file system guarantees that writing the whole amount of data within one data block is not interrupted.

3.5 Trusted Storage Protocol

In this section, we describe how the components in our system work together to provide a trusted storage service. Table 3.2 shows the API between each client and the server, and Table 3.3 describes how the server controls the S chip and the P chip and how the two chips perform trusted storage and computation according to each command from the server.

When the cloud server boots, the server's boot logic re-pairs the S chip and P chip following the procedure described in Section 2.2.3. The server executes the recovery procedure (see Algorithm 1) if the server re-boots from a crash. After boot or recovery, the server is ready to provide a trusted storage service. When a client requests a new session for future communication, the server assigns a new session ID to the client and stores the client's HMAC key in the encrypted version as described in Section 3.4.2. After the session is created, the client uses the session to communicate with the cloud storage system, sending read/write requests to access/modify the data stored on the cloud. Algorithm 2 and Algorithm 3 show how the system handles each read and write request. The P chip obtains the client's HMAC key from its session cache or from the encrypted version stored on the server. For a read request, the server reads the data from the disk and asks the P chip to verify the Merkle tree nodes and generate an HMAC to authenticate the read operation. As described in Section 3.4.4, the P chip buffers the HMAC if the client tries to access the data that is not yet reflected by the root hash stored on the S chip. For a write request, the P chip checks the client's write access and only allows authorized users with a correct revision number to update the Merkle tree (see Section 3.4.3). The server writes the data into the disk and asks the P chip to send the latest root hash to the S chip. The P chip buffers the HMAC for the write operation until the root hash is successfully stored on the S chip. At the same time, the server stores the required information on the disk as described in Section 3.4.5 so that the system is able to recover from an accidental or malicious crash.

Algorithm 2 Read Operation

1: **procedure** READBLOCK(S_{id}, B_{id}, n, $HMAC_{Skey}$)
2: $Pkey \leftarrow$ sessionTable(S_{id})
3: **if** session cache miss **then**
4: loadSession(S_C, $Pkey$) ▷ To obtain $Skey$
5: **end if**
6: The P chip checks the client's request ▷ See checkReqR() in Table 3.3
7: The server rejects the invalid request
8: $data \leftarrow$ readDisk(B_{id}) ▷ Read data from disk
9: **if** tree cache miss **then**
10: The P chip LOADs required but uncached tree nodes
11: The P chip VERIFYs newly cached tree nodes
12: **end if**
13: $HMAC_{Skey}(MT_{PC0}||B_{id}||n||H_{data}||V_{id}) \leftarrow$ readCertify(S_C, n, C, $leaf_{arg}$)
14: **return** $data$, V_{id}, $HMAC_{Skey}(MT_{PC0}||B_{id}||n||H_{data}||V_{id})$
15: **end procedure**

Algorithm 3 Write Operation

1: **procedure** WRITEBLOCK(S_{id}, B_{id}, n, $data^*$, $\{W\}_{Skey}$, H_{Wkey^*}, H_{data^*}, $HMAC_{Skey}$)
2: $Pkey \leftarrow$ sessionTable(S_{id})
3: **if** session cache miss **then**
4: loadSession(S_C, $Pkey$) ▷ To obtain $Skey$
5: **end if**
6: $H^*_{data^*} \leftarrow$ dataHash($data^*$)
7: The P chip checks the client's request ▷ See checkReqW() in Table 3.3
8: The server verifies $data^*$ by checking $H^*_{data^*}$ against H_{data^*}
9: The server rejects the invalid request
10: **if** tree cache miss **then**
11: The P chip LOADs required but uncached tree nodes
12: The P chip VERIFYs newly cached tree nodes
13: **end if**
14: The P chip checks $\{W\}_{Skey}$ ▷ See checkWrite() in Table 3.3
15: **if** valid $Wkey$ and valid V_{id} **then**
16: writeDisk(B_{id}, $data^*$) ▷ Write data to disk
17: $HMAC_{Skey}(MT_{PC1}||B_{id}||n||H_{data^*}) \leftarrow$ The P chip UPDATEs the tree nodes
18: **return** $HMAC_{Skey}(MT_{PC1}||B_{id}||n||H_{data^*})$
19: **else if** valid $Wkey$ **then**
20: **return** correct V_{id}, $HMAC_{Skey}(MT_{PC2}||B_{id}||n||V_{id})$
21: **else**
22: **return** $Invalid$
23: **end if**
24: **end procedure**

Table 3.2: API between client and server

Command	Arguments and Operation Semantics
connect()	**Args:** None **Return:** $PubEK$, $ECert$
createSession()	**Args:** $\{Skey\}_{PubEK}$ **Return:** S_{id} **Action:** The server assigns S_{id}, calls processKey($\{Skey\}_{PubEK}$) to obtain $Pkey$, and stores $(S_{id}, Pkey)$.
readBlock()	**Args:** S_{id}, B_{id}, n, $HMAC_{Skey}(MT_{CP0}\|\|B_{id}\|\|n)$ **Return:** $data$, V_{id}, $HMAC_{Skey}(MT_{PC0}\|\|B_{id}\|\|n\|\|H_{data}\|\|V_{id})$ **Check:** The client re-computes the hash of $data$, H^*_{data}, and checks H^*_{data} and V_{id} against $HMAC_{Skey}(MT_{PC0}\|\|B_{id}\|\|n\|\|H_{data}\|\|V_{id})$. **Action:** See Algorithm 2.
writeBlock()	**Args:** S_{id}, B_{id}, n, $data^*$, $HMAC_{Skey}(MT_{CP1}\|\|B_{id}\|\|n\|\|H_{data^*}\|\|H_{Wkey^*})$, H_{data^*}, H_{Wkey^*}, $\{Wkey\|\|V_{id}\}_{Skey}$ **Return:** $HMAC_{Skey}(MT_{PC1}\|\|B_{id}\|\|n\|\|H_{data^*})$ if successful write, or V_{id}, $HMAC_{Skey}(MT_{PC2}\|\|B_{id}\|\|n\|\|V_{id})$ if invalid V_{id} **Check:** The client checks the received HMAC, and in addition, checks V_{id} if the write is not successful.
closeSession()	**Args:** S_{id} **Action:** The server releases the corresponding entry in its session table.

Table 3.3: API between server and S-P chip

Command	Arguments and Operation Semantics
Boot Process Commands (P Chip)	
genSK()	**Args:** $ECert$, ECC, $\sigma_{PrivEK}(ECC)$ **Return:** nonce(n), $HMAC_{SK}(MT_{PS0}\|\|n)$ **Action:** The P chip verifies $ECert$, checks ECC against $\sigma_{PrivEK}(ECC)$, and re-generates SK if the verification is successful.
reloadRoot()	**Args:** s, $HMAC_{SK}(MT_{SP0}\|\|s\|\|n)$ **Action:** The P chip clears its tree cache (if necessary), checks the given root hash (s) against $HMAC_{SK}(MT_{SP0}\|\|s\|\|n)$, and stores s.

Table 3.3: (continued)

Command	Arguments and Operation Semantics
reloadKey()	**Args:** $\{PrivEK\}_{SK}$, $ECert$ **Action:** The P chip decrypts $\{PrivEK\}_{SK}$ using SK, checks $PrivEK$ against $PubEK$, and stores $PrivEK$.
initRoot()	**Args:** H_{zero}, LV **Action:** The Merkle tree engine initializes the tree root using the given leaf hash (H_{zero}) and the tree height (LV). (Note: This command is issued when the system boots for the first time.)
Boot Process Commands (S Chip)	
getRootS()	**Args:** n, $HMAC_{SK}(MT_{PS0}\|\|n)$ **Return:** root hash (s), $HMAC_{SK}(MT_{SP0}\|\|s\|\|n)$ **Action:** The S chip verifies the nonce (n) against $HMAC_{SK}(MT_{PS0}\|\|n)$.
getKey()	**Args:** None **Return:** $\{PrivEK\}_{SK}$ **Action:** The S chip encrypts $PrivEK$ under SK.
Root Hash Storage (P Chip)	
getRootP()	**Args:** None **Return:** s, n, $HMAC_{SK}(MT_{PS1}\|\|s\|\|n)$ **Action:** The P chip stores the current root hash (s) and a nonce (n).
Root Hash Storage (S Chip)	
storeRoot()	**Args:** s, n, $HMAC_{SK}(MT_{PS1}\|\|s\|\|n)$ **Return:** $HMAC_{SK}(MT_{SP1}\|\|s\|\|n)$ **Action:** The S chip checks s and n against $HMAC_{SK}(MT_{PS1}\|\|s\|\|n)$, stores s, and generates $HMAC_{SK}(MT_{SP1}\|\|s\|\|n)$.
Request Check Commands (P Chip)	
checkReqR()	**Args:** S_c, B_{id}, n, $HMAC_{Skey}(MT_{CP0}\|\|B_{id}\|\|n)$ **Action:** The P chip verifies B_{id} and n against $HMAC_{Skey}(MT_{CP0}\|\|B_{id}\|\|n)$
checkReqW()	**Args:** S_c, B_{id}, n, $HMAC_{Skey}(MT_{CP1}\|\|B_{id}\|\|n\|\|H_{data^*}\|\|H_{Wkey^*})$, H_{data^*}, H_{Wkey^*} **Action:** The P chip verifies B_{id}, n, H_{data^*}, and H_{Wkey^*} against $HMAC_{Skey}$

Table 3.3: (continued)

Command	Arguments and Operation Semantics
HMAC Key Management Commands (P Chip)	
processKey()	**Args:** $\{Skey\}_{PubEK}$ **Return:** $Pkey$ ($Pkey = \{Skey\}_{SK}$) **Action:** The P chip uses $PrivEK$ to decrypt the encrypted HMAC key ($\{Skey\}_{PubEK}$), then uses SK to re-encrypt $Skey$ and obtains the processed key ($Pkey$).
loadSession()	**Args:** S_C, $Pkey$ **Action:** The P chip uses SK to decrypt the given processed key ($Pkey$) and obtain the HMAC key ($Skey$). Then, the session cache stores $Skey$ in the given entry (S_C).
Data Hashing Command (P Chip)	
dataHash()	**Args:** $data$ **Return:** H_{data} **Action:** The data hash engine calculates the hash value of $data$.
Write Access Control Command (P Chip)	
checkWrite()	**Args:** S_C, $\{W\}_{Skey}$, C, H_{Wkey}, V_{id}, H_{data} **Action:** The P chip verifies H_{Wkey}, V_{id}, H_{data} against the leaf hash stored in the given tree cache entry (C). Then, the P chip decrypts $\{W\}_{Skey}$ and obtains $Wkey^*$ and V_{id}^*. The P chip checks $H(Wkey^*)$ against H_{Wkey} and checks V_{id}^* against $V_{id}+1$. (Note: If the leaf hash equals the H_{zero}, then all checks are ignored.) **Check:** The leaf node stored in C should already be verified.
Tree Cache Management Commands (P Chip)	
LOAD()	**Args:** C, N_{id}, H, $C_{P_{old}}$ **Action:** The Merkle tree engine loads the node number (N_{id}) and the hash (H) of the new node into the given cache entry (C). If the node being evicted is verified, the L or R bit stored in its parent's cache entry ($C_{P_{old}}$) will be cleared to reflect the eviction. **Check:** To avoid multiple caching of a single node, the evicted node should not have cached children.

Table 3.3: (continued)

Command	Arguments and Operation Semantics	
VERIFY()	**Args:**	C_P, C_L, C_R
	Action:	The Merkle tree engine checks the hashes in the given child entries (C_L and C_R) with the hash in their parent entry (C_P) and sets the corresponding valid bits.
	Check:	The node in C_P should already be verified, and its L and R bits should match the valid bits in C_L and C_R.
UPDATE()	**Args:**	C_{Path}, C_{Sibs}, $\{W\}_{Skey}$, H_{data^*}, S_C, n, H_{Wkey^*}
	Return:	$HMAC_{Skey}(MT_{PC1}\|\|B_{id}\|\|n\|\|H_{data^*})$
	Action:	The Merkle tree engine generates new *leaf* using Equation 3.1 and updates the tree nodes specified in C_{Path} using *leaf* and the sibling nodes specified in C_{Sibs}. Then, the Merkle tree engine calls writeCertify(S_C, n, C, $leaf_{arg}$) to obtain the HMAC for the write operation, and stores the HMAC in the on-chip buffer until the root hash is stored on the S chip. (Note: writeCertify() is similar to readCertify() but called inside the P chip.)
	Check:	H_{data^*}, H_{Wkey^*}, and $\{W\}_{Skey}$ should already be verified by checkReqW() and checkWrite(). C_{Path} should form a valid update path. Nodes stored in C_{Path} and C_{Sibs} entries should be verified.
Response Command (P Chip)		
readCertify()	**Args:**	S_C, n, C, $leaf_{arg}$
	Return:	$HMAC_{Skey}(MT_{PC0}\|\|B_{id}\|\|n\|\|H_{data}\|\|V_{id})$
	Action:	The P chip reads the HMAC key (*Skey*) from its session cache entry (S_C). Then, the Merkle tree engine reads the leaf node out of the given tree cache entry (C), checks H_{data} and V_{id} in $leaf_{arg}$ against the leaf hash, generates $HMAC_{Skey}(MT_{PC0}\|\|B_{id}\|\|n\|\|H_{data}\|\|V_{id})$, and releases the HMAC if the HMACs in the on-chip buffer do not have the same block number (B_{id}).(Note: n signifies a nonce, and B_{id} can be derived from N_{id} in C.)
	Check:	The tree node stored in C should already be verified.

Chapter 4

Implementation

Our implementation focuses on performance, which is determined by the processing time for each incoming request. The processing time is directly affected by the latency and the throughput of the disk, the controllers on the server, the data hash engine and the Merkle tree engine on the P chip, as well as by the connection between the server and the P chip. On the other hand, the S chip and the P chip's boot engines only affect booting time, and the P chip's session cache as well as its cryptographic engines are assumed to introduce constant and negligible overhead. The root hash storage protocol can be done asynchronously with normal write operations, so it has no impact on throughput and only introduces constant latency overhead. In the write access control scheme, the decryption and checks required on the P chip also can be done in parallel with the data hashing operation and therefore introduce negligible overhead. The recovery mechanism only has little memory bandwidth and storage overhead, which is negligible compared to the data access and storage. To validate the prototype system in an efficient way, we only implemented in hardware (i.e., the FPGA) the parts that are directly related to the processing time while using software to simulate the functionality of the remaining parts. More specifically, in our implementation and experiments (Section 5.2), we send back HMACs immediately without waiting for the latest root hash to be stored on the S chip and the data to be stored on the disk. We will discuss the overhead introduced by the root hash storage protocol, the write access control scheme, and the recovery mechanism in Section 5.4.

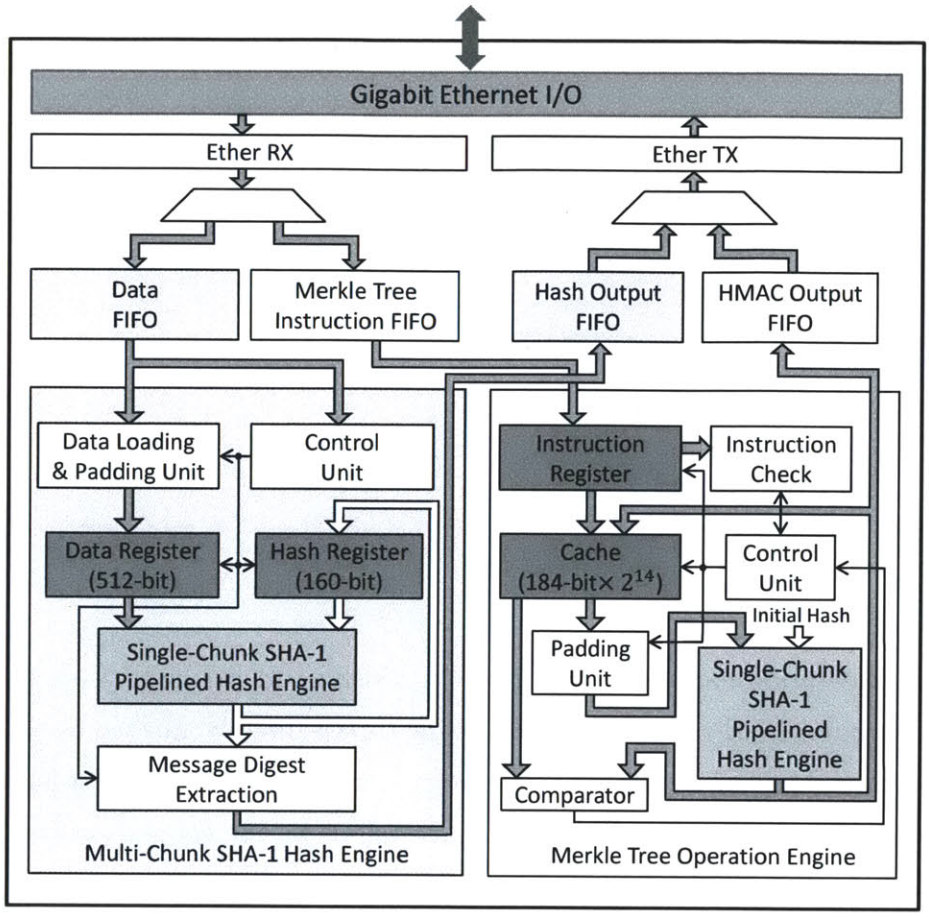


Figure 4-1: Block diagram of P chip

4.1 P Chip Implementation

We implemented the P chip on a Xilinx Virtex-5 FPGA, using Gigabit Ethernet to connect with the server. As shown in Fig. 4-1, two computational logic modules, the data hash engine and Merkle tree operation engine, are independent but share a single Gigabit Ethernet I/O. Because these two engines are independent, they can be executed in parallel. Asynchronous FIFOs are placed at the input and output sides of each engine to parallelize the input data, received from the Ethernet receiver, and to serialize the output data that is going to be sent to the Ethernet transmitter. Table 4.1 shows a summary of the resources used by the P chip.

When receiving instructions from the server, the Ethernet receiver sends the instructions to the corresponding input FIFOs according to their types. Each engine

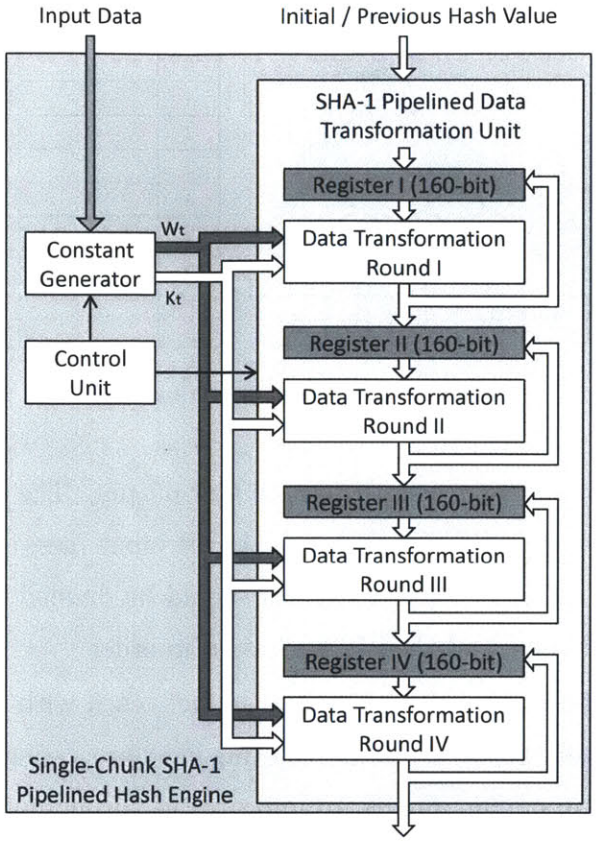


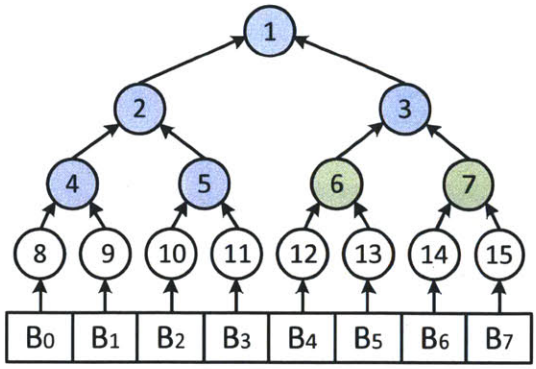
Figure 4-2: Four-stage pipelined SHA-1 engine

will start executing whenever its input FIFO becomes non-empty. Next, we describe the implementation details of the two engines.

4.1.1 Data Hash Engine

The data hash engine is designed to generate the hash value for any given data block. In our storage system, the data hash engine is used to verify the integrity of data sent from a client. As described in Algorithm 3, the server asks the data hash engine to re-compute the hash value of the data, and checks the re-computed hash against the hash sent from the client. We implemented a hardware hash engine for performance consideration. This hash operation can also be executed on the server using a software hash function. We describe our software hash function in Section 4.2, and the performance comparison can be found in Section 5.2.

We implemented SHA-1 [77], which is the most commonly used hash function,



Entry (C)	Node Number (N_{id})	Hash (H)	V	L	R
0	1	321f...	Y	Y	Y
1	2	ed20...	Y	Y	Y
2	5	52cf...	Y	N	N
3	4	be42...	Y	N	N
4	3	345f...	Y	N	N
5	7	21a0...	N	N	N
6	6	ae2b...	N	N	N
⋮	⋮	⋮	⋮	⋮	⋮

Figure 4-3: A tree cache example

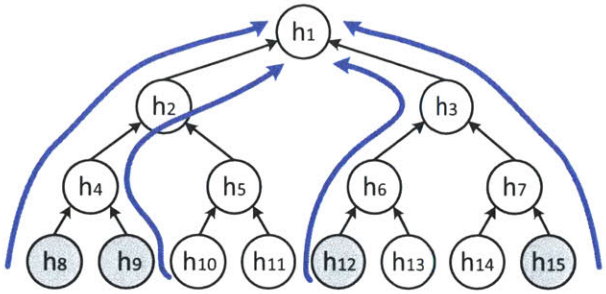
mapping an input of arbitrary length to a 160-bit output. The basic core is a single chunk SHA-1 hash engine, which accepts a 512-bit input message and generates the 160-bit hash output. Longer input messages should be divided into 512-bit message blocks, also called chunks, and then be processed in order.

SHA-1 requires four data transformation rounds, each with a distinct non-linear function applied 20 times in each round, resulting in 80 processing steps in total. Since SHA-1 comprises four similar rounds, to improve the throughput, a pipeline stage is assigned to each round as proposed in [78]. The resulting single chunk pipelined SHA-1 engine is shown in Fig. 4-2.

4.1.2 Merkle Tree Operation Engine

The Merkle tree engine performs operations on the tree cache, according to the tree cache management commands sent from the server's tree controller. The functionality of each operation is described in the Section 3.4.1 and Table 3.3.

We implement the tree cache using the FPGA's block RAM. Figure 4-3 shows how the tree cache stores the information of a tree node. As described in Section 3.4.1, each tree cache entry (C) stores the cached node's node number, its hash value, and three valid bits. The node number is assigned according to the BFS-traversal order, starting at 1 from the root, so that the Merkle tree engine can easily determine the relationship (the parent, the left child, or the right child) between two tree nodes. In Fig. 4-3, the colored nodes (node 1–7) are stored on the tree cache, where the blue ones (node 1–5) are verified and the green ones (node 6–7) are not verified yet.



(a) Update paths

Operation	Start Cycle	End Cycle
$h_4 = H(h_8 \| \| h_9)$	0	80
$h_6 = H(h_{12} \| \| h_{13})$	20	100
$h_7 = H(h_{14} \| \| h_{15})$	40	120
$h_2 = H(h_4 \| \| h_5)$	80	160
$h_3 = H(h_6 \| \| h_7)$	120	200
$h_1 = H(h_2 \| \| h_3)$	200	280

(b) Timing table

Figure 4-4: An example of updating four leaf nodes

To verify or update hashes of the cached tree nodes, we put an identical single-chunk pipelined SHA-1 engine inside the Merkle tree engine. We also use this SHA-1 engine to generate the HMAC. To maximize the utilization of the embedded hash engine, the Merkle tree engine can potentially process up to four VERIFYs, UPDATEs, and HMAC operations (readCertify() or writeCertify() in Table 3.3) simultaneously. Note that under the root hash storage protocol (see Section 3.4.4 and Table 3.3), the HMAC generation command for a write operation (writeCertify()) should be called inside the P chip. In Chapter 4 and Chapter 5, we separate writeCertify() from the UPDATE command for convenience. We combine readCertify() or writeCertify() as a single CERTIFY command that asks for HMAC generation, and the UPDATE command only asks the Merkle engine to execute the tree update procedure.

Our system only supports instructions with multiple UPDATEs. This is because the HMAC generation operation has little performance impact on the system, requiring only one hash operation per client's request. Although the Merkle tree engine does support multiple VERIFY operations, only nodes that are on the independent paths can be verified together. In order to avoid dependencies, more complicated software than currently implemented is needed to schedule the instructions across multiple requests. On the other hand, each UPDATE command contains successive hash operations from a leaf to the root. For instance, if we build a hash tree over a 1TB disk that is divided into 2^{20} megabyte data blocks, each UPDATE consists of 20 successive hash functions. Moreover, it is easy for the software tree controller to send multiple UPDATEs by collecting UPDATE commands from multiple requests

and sending them as a single instruction. Therefore, compared with the other two operations, supporting multiple UPDATEs to improve the system's throughput is most cost-effective.

Figure 4-4 gives an example of how combining multiple UPDATEs improves the system's throughput. Suppose we need to update h_8, h_9, h_{12}, and h_{15}. The update paths are shown in Fig. 4-4a, each requiring 3 hash operations. Without supporting multiple UPDATEs, we update one path at a time, and we need $3 \times 80 \times 4 = 960$ cycles to complete all UPDATE operations. On the other hand, if we combine multiple UPDATEs by utilizing the SHA-1 pipeline stages, we only need 280 cycles. Figure 4-4b shows the detailed timing table for combining multiple UPDATEs. In addition, we merge the hash operations performed by nodes that are siblings to avoid computing hashes that will be overwritten soon. In our example, this reduces the number of hash operations from 12 to 6, eliminating unnecessary power consumption.

4.1.3 Resource Usage Summary

Table 4.1 shows the summary of the resource used by the P chip.

Table 4.1: P chip implementation summary

Modules	FFs	LUTs	Block RAM/FIFO
Data Hash Engine	4408	5597	0 kB
Merkle Tree Engine	4823	9731	2952 kB
Ethernet Modules	1130	1228	144 kB
Total	10361	16556	3096 kB

4.2 Server Implementation

We build the server on the Linux platform. As shown in Fig. 3-2, the data controller handles disk accesses; the hash controller and the tree controller send commands through the Ethernet controller, which is a Gigabit Ethernet interface, to control the computation on the P chip. The server schedules all the operations, following the trusted storage procedure described in Section 3.5, Algorithm 2, and Algorithm 3.

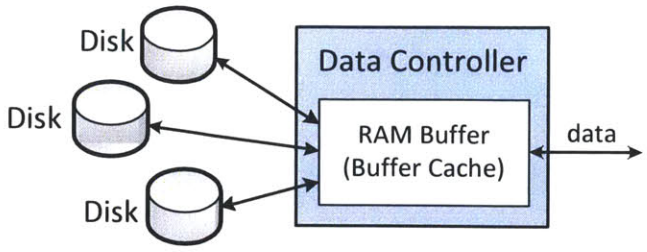


Figure 4-5: Data controller

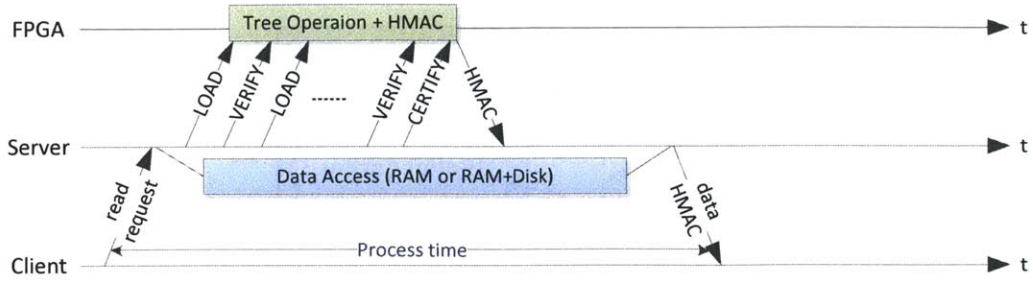
4.2.1 Data Controller and Request Handling Timelines

To speed up data access time, most operating systems have buffer caches, which store a subset of the disk data in the RAM. The data read from the disk is stored in the buffer cache until evicted; therefore, reading the same part of the disk in a short period of time requires only one disk access. The data to be written to the disk is first stored in the buffer cache, so the operating system can handle the write from the buffer cache to the disk in the background, without slowing down other running programs.

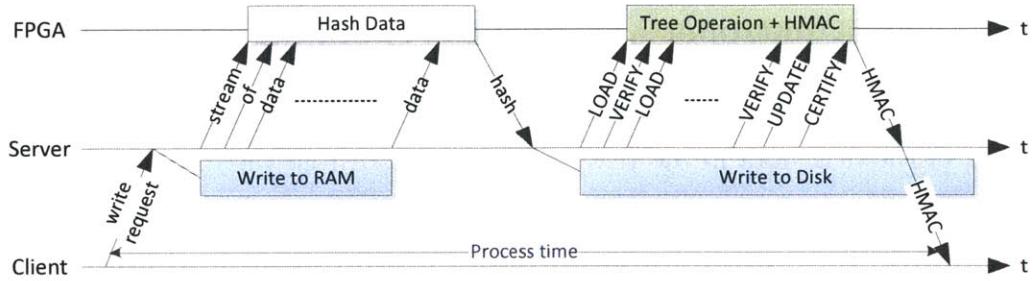
In our prototype system, the data controller (see Fig. 4-5) handles data accesses to/from one or multiple hard disks through the operating system. When handling a read operation, the data controller reads the data from the buffer cache if there is a cache hit; otherwise, the operating system first reads the data from the disk and stores the data in the buffer cache. When handling a write operation, the data controller first writes the data to the buffer cache, and the operating system writes the data that are in the *dirty* state to the disk at a later time. To achieve parallel execution, we put the data controller on another thread.

Figure 4-6a shows the timeline of a read operation. When receiving a read request, the server reads the data from the disk or the buffer cache. At the same time, the server sends the tree operation commands to the FPGA and asks the FPGA to generate an HMAC for authentication. After the server receives the data and the HMAC, the server sends the data and the HMAC to the client and starts to handle the next request.

As described in Algorithm 3, when handling a write operation, the server should first check the integrity of data sent from the client by re-computing the hash of data and checking against the hash sent from the client before the server writes the data



(a) Timeline of a read operation



(b) Timeline of a write operation

Figure 4-6: Trusted storage timelines

to the disk. To minimize the system latency, we perform speculative writes; that is, when the data hash engine is computing the hash value of the data, the server writes the data into the buffer cache at the same time. If the verification fails, the data in the buffer cache should be discarded. The operating system should be modified so that it only writes data blocks that are verified to be correct to the disk. Figure 4-6b shows the timeline of a write operation. When receiving a write request, the server sends the data to the FPGA to compute the hash; at the same time, the server writes the data into the buffer cache. After the server obtains the hash from the FPGA and verifies the data, the server starts to send tree cache commands to the FPGA and send the data to the disk. Since the disk write controlled by the operating system can be done in the background, the server (as well as the FPGA) can start to handle the next request once the server finishes writing the data to the buffer cache and the FPGA finishes generating the HMAC. Therefore, in general, the disk write time does not affect the system throughput. Under the root hash storage protocol and crash-recovery mechanism, however, the disk write time affects the system latency because the P chip buffers the HMAC until the data is stored on the disk (and the

root hash is stored on the S chip). The disk write time may also affect the system throughput if the P chip's on-chip buffer is full and forces the server to stop handling new requests.

One of our main implementation goals is to evaluate the maximum throughput that our system can achieve. The throughput and latency of the buffer cache can be seen as the optimal performance we can achieve, since all data sent from the server or sent from the disk must pass through the buffer cache. Note that the disk read/write throughput can be increased by using smart caching policies, multiple disks, or faster data storage devices such as SSDs. We assume that we have a perfect caching scheme and a large enough number of hard disks so that the data access (read) time is close to the RAM access time, and disk write time does not affect the system throughput even under the root hash storage protocol and crash-recovery mechanism but only introduces a constant latency. Therefore, to simplify our implementation and evaluation, instead of measuring both the RAM and disk read/write time, we only measure the data access time to/from the buffer cache. In our implementation, instead of modifying the operating system, we allocate a RAM buffer in the data controller (see Fig. 4-5) to store all the test data. The RAM buffer mimics the buffer cache with a 100% hit rate, and the data controller controls the data access to/from the RAM buffer. In addition, as mentioned in the beginning of this chapter, we do not include the root hash storage protocol and the crash-recovery mechanism in our implementation; that is, the FPGA does not buffer the HMAC until the data is stored on the disk (and the root hash is stored on the S chip) but sends the HMAC immediately to the client (see Fig. 4-6b). The overhead introduced by the root hash storage protocol and the recovery mechanism is discussed in Section 5.4.

4.2.2 Hash Controller

When handling a write request, the hash controller packs the data block, which is typically one megabyte long, into hundreds of Ethernet frames, and waits for the result from the P chip. We also provide an alternative software hash function to generate hash values for data blocks when hardware resources are limited. Note that replacing the hardware hash function with software does not degrade the security level, because the computed hash ($H^*_{data^*}$ in Algorithm 3) is only used to verify the integrity of data

sent from the client. The hash used to update the Merkle tree (H_{data^*} in Algorithm 3) is included in the HMAC sent by the client, so this hash can be authenticated by the P chip. If the server mis-computes $H^*_{data^*}$ and allows the wrong data to be stored on the disk, the inconsistency between the data and H_{data^*} can be detected by the client on the next read to the same block.

4.2.3 Tree Controller and Request Queue

The tree controller maintains a Merkle tree over the disk and controls the caching policy of the P chip's tree cache. The Least-Recently-Used (LRU) caching policy is implemented. Additionally, the caching policy can be easily switched to other policies to match different access patterns.

To support the P chip's pipelined data hash engine and the Merkle tree engine that allows multiple UPDATEs, the server is also designed to process up to four requests simultaneously. A request queue is set up to store incoming requests. The server keeps checking the request queue and starts processing one or multiple requests if it finds that the queue is not empty. Since hardware optimization techniques across multiple requests are only used for handling write requests, the server takes up to four requests at a time only when they are successive write requests.

Chapter 5

Evaluation

This chapter evaluates the throughput and latency of our prototype system, using synthetic benchmarks to simulate requests from the cloud clients. The performance overhead introduced by providing integrity and freshness guarantees is also analyzed. Finally, we study the effects of different architectural parameters on system performance, and then provide suggestions on hardware requirements according to the cloud storage providers' different demands.

Our experimental setup consists of an Intel Core i7-980X 3.33 GHz processor with 6 cores and 12 GB of DDR3-1333 RAM, which is used as the untrusted server, and a Xilinx Virtex-5 XC5VLX110T FPGA board connected to the server via Gigabit Ethernet. For the purpose of performance evaluation, we assume that there is a 1 TB disk on the server and the disk is divided into 2^{20} 1 MB-long data blocks, which fixes the depth of the Merkle tree at 20.

5.1 Throughput Estimation

To evaluate the throughput of our system, we first estimate the throughput of each system component (see Table 5.1). Table 5.2 shows how we calculate the worst-case throughput of the Merkle tree engine. The worst case happens when handling write requests with an empty tree cache, because the maximum number of LOAD and VERIFY operations are needed when there are no tree nodes cached, and UPDATE is only needed when handling write requests. In our system settings, N equals 20,

Table 5.1: Throughput estimation

Component		Estimated Throughput (MB/s)
Hard Disk (7200 rpm)		100
RAM (DDR3-1333)		10,240
Server-P Chip Data Bus	Gigabit Ethernet	125
	PCI Express x16	4,096
Data Hash Engine	non-pipelined	100
	pipelined	400
Merkle Tree Engine		37,650

Table 5.2: Merkle tree engine throughput estimation

Commands	Throughput (block/s)	# cycles / operation	# operations / block
LOAD	$125{,}000{,}000/2N$	1	$2N$
VERIFY	$1{,}562{,}500/N$	80	N
UPDATE	$1{,}562{,}500/N$	$80N$	1
CERTIFY	1,562,500	80	1
Total	$125{,}000{,}000/(162N+80)$ block/s		

Operating frequency = 125 MHz
$N = \log_2$(total data blocks over the disk)

and each block represents 1 MB of data. Each component in the system can be seen as a hardware pipeline stage. Therefore, to maximize the system's throughput while efficiently utilizing hardware resources, the throughput of each component should be balanced.

As mentioned in Section 4.2.1, all data sent from the server (for write operations) or sent from the disk (for read operations) must pass through the buffer cache (the RAM buffer in our implementation); therefore, the RAM limits the system's throughput. In Table 5.1, except for the Merkle tree engine which already has much higher throughput than the RAM, the throughput of other components (including the hard disk) can be increased by using multiple engines or buses in parallel.

5.2 Experimental Results

To analyze the overhead introduced by our memory authentication scheme, we built another system assuming that the server is trusted, which we refer to as Baseline. Figure 5-1 illustrates the comparison of the two schemes' timelines for handling a write request. In the baseline scheme, because of the trusted server assumption, all

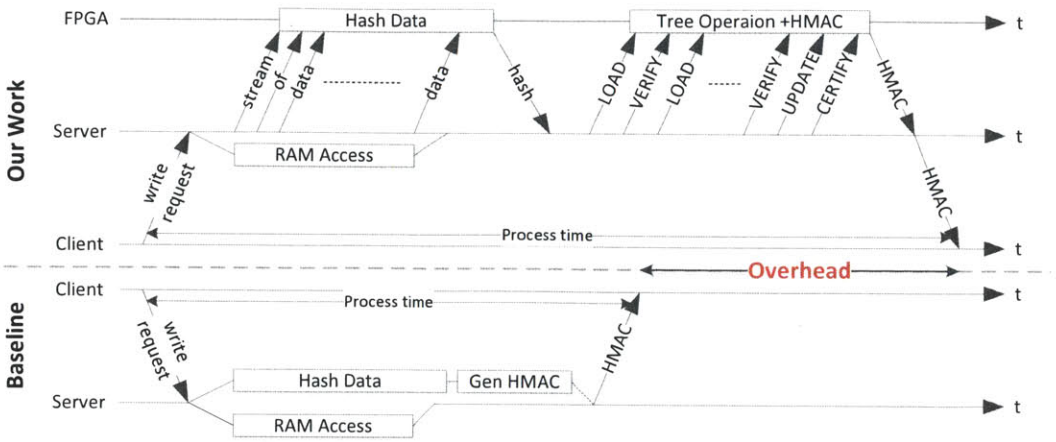


Figure 5-1: Timeline comparison of a typical write request

Table 5.3: Synthetic benchmarks

Benchmark Type	**Description**
read only cont	continuously read 2 GB data
read only period	repeatedly read 256 MB data
read only random	randomly read 1 MB data blocks
write only cont	continuously write 2 GB data
write only period	repeatedly write 256 MB data
write only random	randomly write 1 MB data blocks
random read write	randomly read/write 1 MB blocks (read probability = 0.8)

computation can be done on the server. The HMAC is generated by the server to prevent attacks between the server and its clients. The timeline for handling a read request is similar except that there is no hash operation. Note that we ignore the disk access time in Fig. 5-1, because the disk access time does not affect the maximum system throughput as discussed in Section 4.2.1. Instead of measuring both the RAM and disk access time, we only measured the RAM access time by storing all the test data in the RAM buffer and measuring the data access time to/from the RAM buffer. Keeping in mind the size of the server's RAM, we built a 2 GB RAM buffer and set our working set size as 2 GB.

We compare the performance of the two schemes in terms of average processing time, using several synthetic benchmarks listed in Table 5.3. We measured the processing time of each request as the period starting from when the server dequeues the request and ending when the server finishes processing it. In Table 5.3, in addition to random accesses, the benchmarks for continuous data accesses simulate backup appli-

Table 5.4: System overhead

Benchmark Type	Our work (software hash)	Our work (hardware hash)
read only cont	0.03%	-0.22%
read only period	1.18%	1.09%
read only random	-0.75%	1.01%
write only cont	2.25%	302.71%
write only period	4.00%	307.83%
write only random	3.38%	309.21%
random read write	2.06%	212.58%

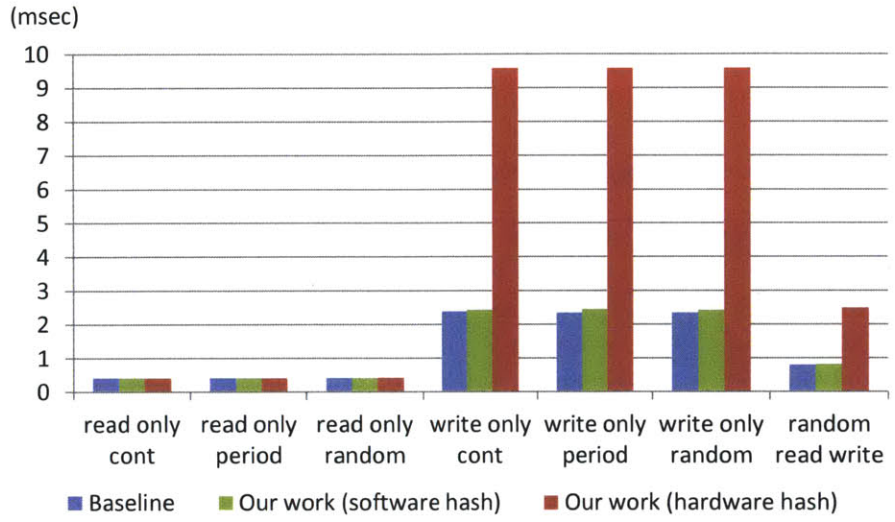
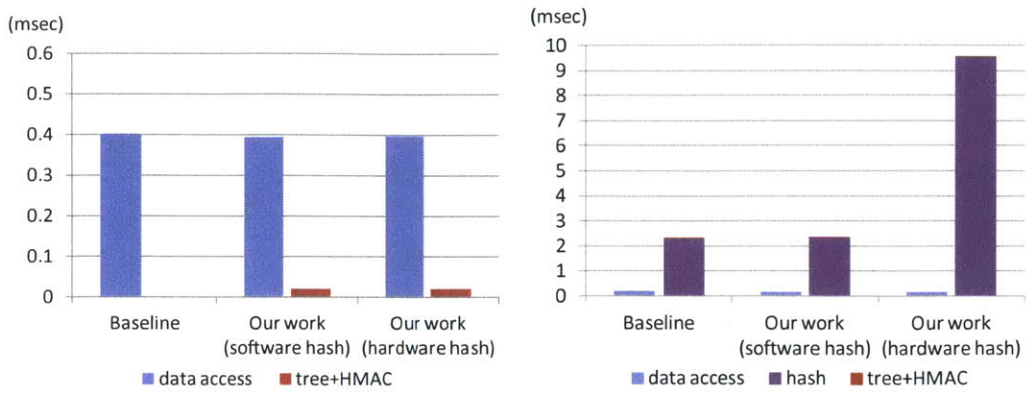


Figure 5-2: Average processing time comparison (2048 operations on 1 MB data blocks with tree cache size = 2^{14})

cations for a single client, while the benchmarks for repeated data accesses simulate group collaboration on the same chunk of data.

Figure 5-2 and Table 5.4 show the comparison between the baseline scheme and the two schemes we implemented, with a single hardware pipelined hash engine and a software hash. The three schemes have the same performance when handling read requests, while our hardware hash engine scheme is four times slower when handling write requests. To understand which component slows down the system, we did detailed timing analysis shown in Fig. 5-3 and Table 5.5. For read requests, the processing time is equal to the data access time. The latency of tree operations and HMAC generation is completely hidden because they can be executed in parallel with data access. On the other hand, the hash operation dominates the processing time when handling write requests. In our software hash implementation scheme, the



(a) read only random

(b) write only random

Figure 5-3: Detailed timing analysis

Table 5.5: Detailed timing analysis

Benchmark		Data Access (msec)	Hash (msec)	Tree + HMAC (msec)
read only random	Baseline	4.01E-1	0	1.69E-3
	Our work (software hash)	3.93E-1	0	2.02E-2
	Our work (hardware hash)	3.97E-1	0	2.04E-2
write only random	Baseline	1.76E-1	2.29	2.40E-3
	Our work (software hash)	1.74E-1	2.33	3.72E-2
	Our work (hardware hash)	1.69E-1	9.51	3.69E-2

overhead for writes is small. In our hardware hash engine scheme, the large overhead for processing write requests is due to the fact that our single hardware pipelined data hash engine is running at a much lower clock frequency compared to the software hash function and that its throughput is limited by the Gigabit Ethernet. We will have a more detailed discussion later.

Table 5.6 shows the throughput and latency of the two schemes we implemented. The throughput and latency for handling write requests is improved by replacing the single hardware hash engine with the software hash function. An alternative would have been to replicate the hardware hash engine. To achieve greater throughput than what can be achieved with the software hash, more parallelism is required in hardware or software.

Table 5.1 shows that when using the non-pipelined data hash engine, computing the hash value is the bottleneck of the whole hash operation, which includes sending

Table 5.6: Performance summary

Scheme	Randomly Write		Randomly Read	
	Throughput	Latency	Throughput	Latency
Our work (software hash)	411 MB/s	2.4 msec	2.4 GB/s	0.4 msec
Our work (hardware hash)	104 MB/s	12.3 msec		

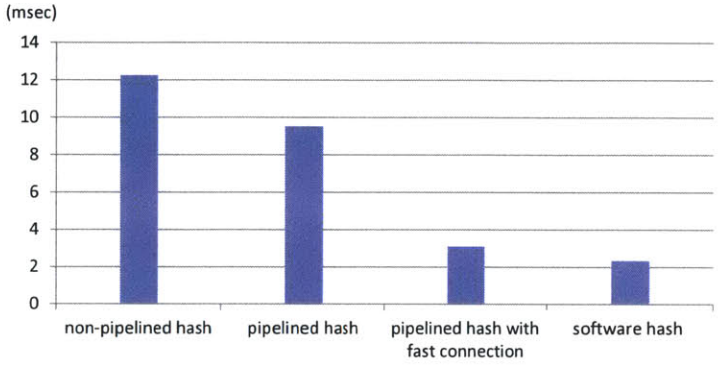


Figure 5-4: Comparison of hash execution time

data packets to the P chip, computing the hash value, and sending back the result. When using the pipelined hash engine, the Gigabit Ethernet becomes the bottleneck. To verify this, when using the pipelined hash engine, we mimic a 4× faster data bus by only sending a quarter of each 1 MB block to the P chip and expanding it at the P chip's side. Figure 5-4 shows the comparison of the average hash execution time using different hash operation schemes on a workload of randomly writing 2,048 1 MB data blocks. With a fast data bus, the throughput of the pipelined hash engine is 4× as the throughput of the non-pipelined hash engine, which is consistent with Table 5.1. One might ask why we are interested in a pipelined hash engine when the software hash is faster. Although the throughput of the pipelined hash engine (running at 125 MHz) with a 4× faster data bus is still smaller than that of the software hash function (running at 3.33 GHz) because of the much lower clock frequency in our prototype hardware, it is cheaper and more energy-efficient to have multiple hash engines in hardware to further improve the throughput as long as the throughput of the data bus is large enough. (As we will indicate in Section 5.3, if we can build a large enough number of hardware hash engines, we can support performance-focused cloud storage providers.) Moreover, the clock frequency running at our prototype hardware can also be increased to improve the throughput of the hash engine.

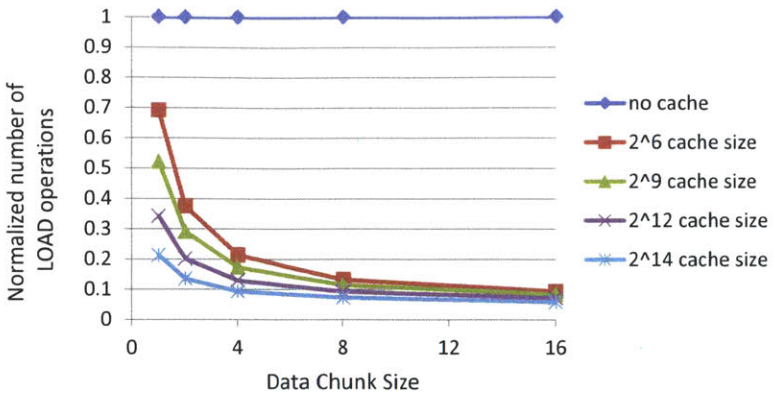


Figure 5-5: Performance comparison between different cache sizes

As mentioned previously, the Merkle tree engine has a higher throughput than the RAM, so there is no need to improve the throughput of the Merkle tree engine. The latency of the Merkle tree engine can be reduced by carefully caching tree nodes in the tree cache. If the hit rate of the tree cache is high, it can significantly reduce the number of required LOAD operations as well as the number of corresponding VERIFY operations, saving hardware computation time as well as instruction transmission time. The hit rate is determined by the tree cache size, the caching policy, and the request patterns. Figure 5-5 gives an example of how we can choose an appropriate tree cache size given request patterns. (In Figure 5-5, the number of LOAD operations is normalized with the maximum number of LOAD operations when there is no tree cache existed.) To approach multi-client settings, we enlarged the working set to 64 GB and issued 6144 requests to randomly access chunks of data of fixed size, under the assumption that there are multiple clients accessing data of fixed size, such as photos or mp3 files. The result shows that if the caching policy works well, there is no need to have a large tree cache, while with random access patterns, a larger tree cache is needed. If the request pattern is known, cloud storage providers can determine an appropriate tree cache size by considering the tradeoff between area and performance. In addition, we also simulated the case without a tree cache by flushing our tree cache between handling each request. We observed that the time spent on tree operations was ten times larger than with a large tree cache, but there was no significant overhead in terms of system latency. This is because the latency caused by tree operations is small compared to the latency caused by other components. Therefore, if resources are limited, the tree cache can be removed without a large performance degradation.

Table 5.7: Hardware requirements

Demand	Connection	Hash Engine	Tree Cache
Performance-focused	PCIe x16(P)/USB(S)	8 + 1 (Merkle tree)	large
Budget-focused	USB	0 + 1 (Merkle tree)	none

Table 5.8: Estimated performance

Demand	Randomly Write		Randomly Read		# HDDs supported
	Throughput	Latency	Throughput	Latency	
Performance-focused	2.4 GB/s	12.3 msec	2.4 GB/s	0.4 msec	24
Budget-focused	377 MB/s	2.7 msec	2.4 GB/s	0.4 msec	4

5.3 Suggestions on Hardware Requirements

Based on the experimental results of our prototype system, we can provide different hardware suggestions to cloud storage providers with different needs. In Table 5.7, we list the different hardware requirements for performance-focused and budget-focused storage providers, and the estimated performance is listed in Table 5.8. The estimated performance is derived from our experimental results with a 1 TB disk divided into 1 MB-long blocks and a typical DDR3-1333 RAM.

5.3.1 Performance-focused Solution

If a storage provider focuses on performance, the system can achieve a throughput as high as its RAM throughput, which is 2.4 GB/s, using multiple hardware data hash engines and a fast data bus. The throughput of a pipelined hash engine, 330 MB/s, is measured using a fast data bus as shown in Fig. 5-4. Therefore, in order to achieve 2.4 GB/s throughput, we need 8 pipelined hash engines to perform 1 MB data hashing as well as a PCI Express x16 link, which can support up to 4.0 GB/s. Figure 5-6 shows the functional units of a S-P chip pair required by a performance-focused cloud storage provider. A typical performance-focused solution is an ASIC paired with a smart card chip. If the P chip runs at a clock frequency higher than 125 MHz, which is easy for an ASIC to achieve, the number of hash engines required can be further reduced.

5.3.2 Budget-focused Solution

If a storage provider has limited resources, a solution with a software hash function and without a tree cache can be chosen to reduce the cost while maintaining the sys-

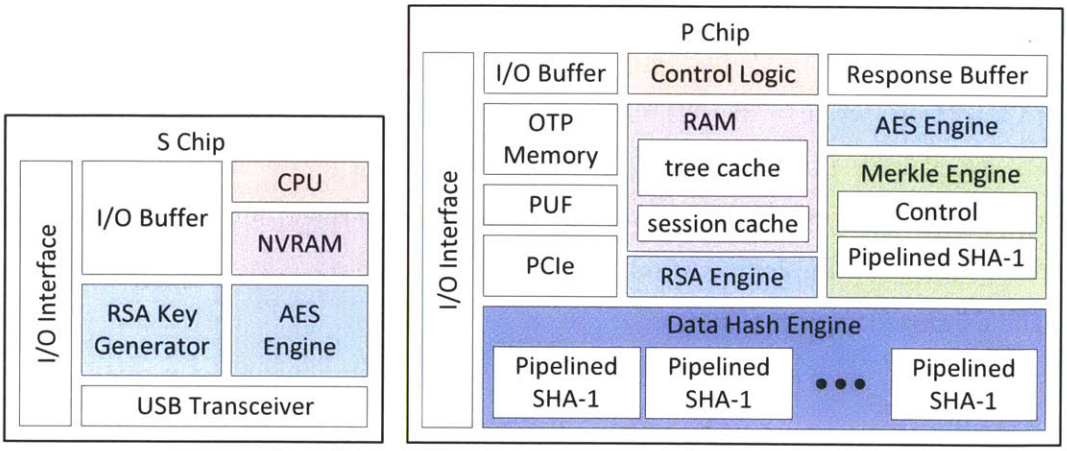


Figure 5-6: Performance-focused (two chip) solution

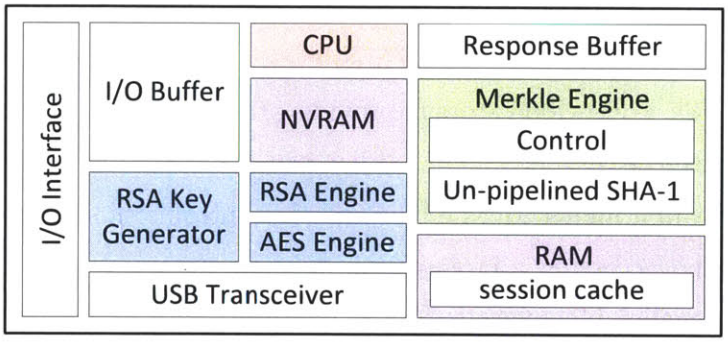


Figure 5-7: Budget-focued (single chip) solution

tem's throughput around 400 MB/s for write requests and 2.4 GB/s for read requests (shown in Table 5.8). In a budget-focused design, many functional units and on-chip storage required by the original P chip are removed. Therefore, we can combine the functionality of the original two chip solution and build a single chip as shown in Figure 5-7. This single chip can be imagined as a typical smart card chip running at 125 MHZ (or slightly less than 125 MHz) with an additional hardware SHA-1 engine as well as some control logic and a on-chip buffer. In addition, after the two chips are merged into a single chip, the communication between the two chips become trusted, so the HMACs between the two chips are no longer needed to protect messages against forging or replay attacks, and updating the root hash in the NVRAM becomes easier. Under today's NVRAM process, this single chip solution is feasible in terms of chip area and speed. Therefore, this represents a cheap solution for trusted cloud storage, and yet is significantly more efficient than, for example, the solution of [17].

5.4 Discussion

In this section, we first discuss the reasoning behind our design parameter choices. Then, we evaluate our write access control scheme and the root hash storage protocol qualitatively to prove that they have fixed or little overhead and thus can be safely excluded from our implementation. We also discuss the memory overhead introduced by the crash-recovery mechanism.

5.4.1 Design Parameter Choices

In our experiment, we fix the block size as 1MB, which is close to current cloud storage systems. For example, Dropbox uses 4 MB blocks, and the Google File System uses 64 MB chunks [79]. For a cloud storage provider, the best choice of block size depends on the access patterns from its clients. When the disk size is fixed, choosing a smaller block size results in a higher Merkle tree and thus introduces larger overhead on tree operations. When accessing a large chunk of data, the overhead introduced by the scheme with a smaller block size is larger than that with a larger block size due to more requests required as well as more communication overhead. However, when accessing a chunk of data that is smaller than a single data block, the whole data block needs to be retrieved from the disk, and thus the scheme with a smaller block size is more efficient. Therefore, a cloud storage provider should choose the block size based on the statistics of its clients' access patterns, and the storage provider should optimize the data hashing scheme when choosing a larger data block size, because data hashing (as well as the data access time) would dominate the processing time.

We assume one terabyte disk in our experiment, where one terabyte is a typical size of a commodity hard disk. When the block size is fixed, the size of the Merkle tree is determined by the size of the disk. A larger disk results in a larger Merkle tree and thus more tree operations for each read/write operation. However, as shown in Table 5.1, the throughput of the Merkle tree engine for a 1 TB disk is much higher than that of RAM. Therefore, increasing the disk size would have little impact on system throughput. For example, we can increase the disk size to 16 TB without system throughput degradation.

5.4.2 Write Access Control

As mentioned in Section 3.4.3, in order to prevent unauthorized writes as well as replay attacks, the P chip needs to decrypt $\{W\}_{Skey}$, which is the client's encrypted access information, verify with the access information stored on the server, and generate the new leaf value using Equation 3.1. In other words, to manage the write access control on the P chip, we need to add one symmetric decryption and two hash operations to our hardware implementation. However, these additional operations do not require additional hardware resources and only add negligible performance overhead. For the hardware resource issue, we can simply re-use the AES engine, which is already required by the P chip's boot process and the processed key decryption, to decrypt $\{W\}_{Skey}$, and re-use the SHA-1 hash engine in the Merkle tree operation engine to perform the hash operations. For the performance issue, the AES decryption can performed in parallel with the 1 MB data hashing, which is performed either on the server or on the P chip. According to the FPGA implementation and evaluation results in [80], the decryption of $\{W\}_{Skey}$, which is the decryption of two 128-bit data blocks, only requires around 100 clock cycles and can be run above 125 MHz, which is the clock frequency we use in the prototype system. Therefore, the latency of the AES decryption can be easily hidden behind the 1 MB data hashing operation. The two additional hash operations, which are both single-chunk hash operations, need to be performed in serial with the Merkle tree operations (after the tree verification and before the tree update), adding no throughput overhead and negligible latency overhead because the Merkle tree engine has higher throughput than the RAM on the server. Therefore, we can safely omit the implementation of the write access control scheme, because it has little impact on system performance.

5.4.3 Root Hash Storage Protocol

Our root hash storage protocol (mentioned in Section 3.4.4) requires the P chip to buffer the responses of the write operations and the related read operations until the latest root hash is stored on the S chip. This protocol does not degrade the system throughput and only introduces a constant latency overhead if the on-chip buffer is large enough. The latency overhead depends on the issuing rate of the

server's getRootP() requests and the round-trip time, which consists of the time the server spends on flushing the data to disk (around 10 ms for 1 MB data), the time the S chip spends on storing the 20-byte root hash (around 20 ms as mentioned in Section 2.2.1), and the time the S chip spends on checking $HMAC_{SK}(MT_{PS1})$ and generating $HMAC_{SK}(MT_{SP1})$ (around 2 ms if using a 32-bit RISC CPU or less time if using a SHA-1 hardware engine). Therefore, the minimum latency introduced is around 32 ms. To maintain the throughput of the system, the performance-focused solution needs a 2k-byte on-chip buffer to store up to 100 responses, while the budget-focused solution, which is a single chip solution, needs a 300-byte on-chip buffer to store up to 15 responses. Note that the EEPROM/flash on a smart card has limited number of write/erase cycles (around 10^5 as mentioned in Section 2.2.1). In order to extend the lifetime of the S chip, we divide the NVRAM into 20-byte long chunks and switch to a different chunk once the root hash cannot be successfully stored on the previous chunk. Recent smart card chips, which have above 100 kB EEPROM/flash storage, can serve for around 5 years with the maximum issuing rate of the getRootP() requests. The S chip can serve for a longer time with a lower issuing rate, but the P chip requires a larger on-chip buffer to maintain the system throughput.

5.4.4 Crash-Recovery Mechanism

To recover from accidental or malicious crashes, the server needs to store the required information, including a snapshot of the Merkle tree and the leaf arguments $(H_{Wkey}, V_{id}, H_{data})$ as described in Section 3.4.5. A straightforward approach to save a snapshot on the disk is to overwrite the entire tree and all leaf arguments, resulting in a large memory bandwidth overhead (around 84 MB for a 1 TB disk). To reduce the memory overhead, when saving the i^{th} snapshot, we only overwrite the parts that are different between the i^{th} and $(i-2)^{th}$ snapshots and only take a snapshot of the leaf nodes and leaf arguments. The entire tree can be built after the server re-boots from a crash. As mentioned in the previous section, during a 32 ms time interval, the storage system can process up to 100 requests. Therefore, the number of leaf nodes (and their arguments) that are different in the $(i-2)^{th}$ and i^{th} snapshots is at most 200, resulting in total 12.8 KB memory bandwidth overhead, which is negligible compared to a 1 MB data block. Therefore, in the previous section, we omit the time the

server spends on saving the snapshot when calculating the required round-trip time to store the root hash on the S chip. The memory storage overhead introduced by storing snapshots and request logs is less than 65 MB, which is very small compared to a 1 TB disk.

Chapter 6

Conclusion

6.1 Thesis Summary

In this thesis, we have detailed the design of a cloud storage system that ensures data integrity and freshness while maintaining high-performance by attaching a trusted pair of chips to an untrusted server. We proposed a write access control scheme to prevent unauthorized writes and ensure all writes are fresh. We also introduced a crash-recovery mechanism to protect our prototype system from crashes and power loss events. To prove our prototype design is practical, we implemented the system using an FPGA and a Linux server. We analyzed our prototype system and showed that even with limited resources, the storage system can achieve 2.4 GB/s (as high as the server's RAM throughput) for handling read requests and 377 MB/s for handling write requests. One can easily imagine building a single chip that is not appreciably more expensive than current smart card chips and can support this level of throughput. If more hardware resources are available, the throughput for handling write requests can be increased to 2.4 GB/s.

6.2 Future Work

In this work, we implemented the data hash engine, the RAM buffer, and the Merkle tree operation engine, which are the most essential parts of the system that directly affect the system throughput and latency. We omitted the implementation of the

write access control scheme, the root hash storage protocol, and the recovery mechanism. We adopted theoretical results and qualitatively proved that these schemes only introduced negligible or constant overhead. However, to completely catch the interaction between different components and the effectiveness of various optimization schemes, an implementation of the full system is necessary. In addition, evaluating with real workloads and taking the network latency between the client and the server into consideration would make the experimental results of our system more practical.

Bibliography

[1] Amazon. Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[2] Google. Google app engine. https://developers.google.com/appengine/.

[3] Microsoft. Windows azure. http://www.windowsazure.com/en-us/.

[4] Amazon. Amazon simple storage service. http://aws.amazon.com/s3/.

[5] Google. Google cloud storage. https://developers.google.com/storage/.

[6] Dropbox. Dropbox. https://www.dropbox.com/.

[7] Microsoft. Windows live skydrive. http://skydrive.live.com/.

[8] Apple. icloud. http://www.apple.com/icloud/.

[9] Google. Google drive. https://drive.google.com/.

[10] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, 2004.

[11] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, 2009.

[12] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security*, 2010.

[13] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *OSDI, Oct*, 2010.

[14] A.J. Feldman, W.P. Zeller, M.J. Freedman, and E.W. Felten. Sporc: Group collaboration using untrusted cloud resources. *OSDI, Oct*, 2010.

[15] B.G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[16] D. Levin, J.R. Douceur, J.R. Lorch, and T. Moscibroda. Trinc: small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[17] M. van Dijk, J. Rhodes, L.F.G. Sarmenta, and S. Devadas. Offline untrusted storage with immediate detection of forking and replay attacks. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, 2007.

[18] V. Costan and S. Devadas. Security challenges and opportunities in adaptive and reconfigurable hardware. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2011.

[19] Trusted Computing Group. Trusted Platform Module (TPM) Specifications. https://www.trustedcomputinggroup.org/specs/TPM/.

[20] Amazon. Amazon s3 service level agreement. http://aws.amazon.com/s3-sla/.

[21] Microsoft. Windows azure service level agreements. http://www.windowsazure.com/en-us/support/sla/.

[22] R.A. Popa, J.R. Lorch, D. Molnar, H.J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. *Microsoft TechReport*, 2010.

[23] E.J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proceedings of NDSS*, 2003.

[24] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, 2006.

[25] S.D.C. Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Over-encryption: management of access control evolution on outsourced data. In *Proceedings of the 33rd international conference on Very large data bases*, 2007.

[26] M. Chase. Multi-authority attribute based encryption. *Theory of Cryptography*, 2007.

[27] M. Chase and S.S.M. Chow. Improving privacy and security in multi-authority attribute-based encryption. In *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[28] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, 2010.

[29] A. Sahai and B. Waters. Fuzzy identity-based encryption. *Advances in Cryptology–EUROCRYPT*, 2005.

[30] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. *Advances in CryptologyŮEUROCRYPT*, 1998.

[31] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[32] D.X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, 2000.

[33] Y.C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security*, 2005.

[34] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, 2006.

[35] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology-Eurocrypt*, 2004.

[36] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, and H. Shi. Searchable encryption revisited: Consistency properties, relation to anonymous ibe, and extensions. In *Advances in Cryptology–CRYPTO*, 2005.

[37] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. *Theory of Cryptography*, 2007.

[38] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology*, 2008.

[39] E. Shen, E. Shi, and B. Waters. Predicate privacy in encryption systems. *Theory of Cryptography*, 2009.

[40] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou. Secure ranked keyword search over encrypted cloud data. In *IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*, 2010.

[41] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. In *INFOCOM*, 2011.

[42] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science*, 1995.

[43] C. Aguilar-Melchor and P. Gaborit. A lattice-based computationally-efficient private information retrieval protocol. In *Western European Workshop on Research in Cryptology (WEWoRCŠ2007), Bochum, Germany. Book of Abstracts*, 2007.

[44] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2007.

[45] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[46] F. Olumofin and I. Goldberg. Revisiting the computational practicality of private information retrieval. *Financial Cryptography and Data Security*, 2012.

[47] D. Ford, F. Labelle, F.I. Popovici, M. Stokely, V.A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[48] Amazon S3. Amazon s3 availability event. http://status.aws.amazon.com/s3-20080720.html.

[49] Google. Current gmail outage. http://googleblog.blogspot.com/2009/02/current-gmail-outage.html, 2009.

[50] E. Sit, A. Haeberlen, F. Dabek, B.G. Chun, H. Weatherspoon, R. Morris, M.F. Kaashoek, and J. Kubiatowicz. Proactive replication for data durability. In *5rd International Workshop on Peer-to-Peer Systems (IPTPS 2006)*, 2006.

[51] R. Rodrigues and B. Liskov. High availability in dhts: Erasure coding vs. replication. *Peer-to-Peer Systems IV*, 2005.

[52] K.D. Bowers, A. Juels, and A. Oprea. Hail: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[53] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. Racs: a case for cloud storage diversity. In *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.

[54] A. Juels and B.S. Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[55] H. Shacham and B. Waters. Compact proofs of retrievability. *Advances in Cryptology-ASIACRYPT*, 2008.

[56] K.D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, 2009.

[57] R.C. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, 1980.

[58] W. Hall and C. Jutla. Parallelizable authentication trees. In *Selected Areas in Cryptography*, 2006.

[59] R. Elbaz, D. Champagne, R. Lee, L. Torres, G. Sassatelli, and P. Guillemin. Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks. *Cryptographic Hardware and Embedded Systems-CHES*, 2007.

[60] A. Heitzmann, B. Palazzi, C. Papamanthou, and R. Tamassia. Efficient integrity checking of untrusted network storage. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, 2008.

[61] M. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. *Information Security*, pages 80–96, 2008.

[62] D. Mazires and D. Shasha. Don't trust your file server. In *Hot Topics in Operating Systems*, 2001.

[63] D. Mazieres and D. Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, 2002.

[64] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, 2007.

[65] C. Cachin and M. Geisler. Integrity protection for revision control. In *Applied Cryptography and Network Security*, 2009.

[66] C. Cachin. Integrity and consistency for untrusted services. *SOFSEM 2011: Theory and Practice of Computer Science*, pages 1–14, 2011.

[67] D.C. Latham. Department of defense trusted computer system evaluation criteria. *Department of Defense*, 1986.

[68] W. Rankl and W. Effing. *Smart card handbook*. Wiley, 2010.

[69] Infineon. Infineon sle 88cfx4001p. http://www.infineon.com/cms/en/product/index.html.

[70] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium*, volume 13, pages 16–16, 2004.

[71] S. Balfe, A.D. Lakhani, and K.G. Paterson. Securing peer-to-peer networks using trusted computing. *Trusted Computing*, pages 271–298, 2005.

[72] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, 2002.

[73] G.E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th Annual Design Automation Conference*, pages 9–14, 2007.

[74] R. Elbaz, D. Champagne, C. Gebotys, R. Lee, N. Potlapally, and L. Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. *Transactions on Computational Science IV*, pages 1–22, 2009.

[75] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2):225–244, 1994.

[76] B. Gassend, E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and merkle trees for efficient memory authentication. In *Proceedings of 9th International Symposium on High Performance Computer Architecture*, 2003.

[77] F.P.U.B. NIST. 180-1: Secure hash standard, 1995.

[78] N. Sklavos, E. Alexopoulos, and O. Koufopavlou. Networking data integrity: High speed architectures and hardware implementations. *Int. Arab J. Inf. Technol*, 1:54–59, 2003.

[79] S. Ghemawat, H. Gobioff, and S.T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, 2003.

[80] P. Bulens, F.X. Standaert, J.J. Quisquater, P. Pellegrin, and G. Rouvroy. Implementation of the aes-128 on virtex-5 fpgas. *Progress in Cryptology–AFRICACRYPT*, 2008.