Securing Digital Archiving Systems Against Mass Breaches and Long-Term Security Degradation

Securing Digital Archiving Systems Against Mass Breaches and Long-Term Security Degradation

By Moe Sabry, M.Eng.

A Thesis Submitted to the Department of Computing & Software in the Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

McMaster University

Doctor of Philosophy (2023)

Hamilton, Ontario (Computing and Software)

TITLE: Securing Digital Archiving Systems Against Mass Breaches and Long-Term Security Degradation

AUTHOR: Moe Sabry (McMaster University)

SUPERVISORS: Dr. Reza Samavi Dr. Emil Sekerinski Dr. Douglas Stebila

NUMBER OF PAGES: xv, 145

Lay Abstract

In this thesis, we address three challenges faced in securing digital archives. The first challenge is how to protect digital archives against security information leakage leading to mass data breaches. We developed an anti mass-leakage archiving system that eliminates the need for managing large sets of secret keys and preventing an adversary from gaining immediate and unlimited access to all archives if a key is compromised. The second challenge is how to keep these archives secure in the long-term despite the advancement of computational powers and cryptanalysis techniques. We developed a secure archiving framework guaranteeing secure long-term confidentiality and integrity protection. The third challenge is to construct an efficient and simple way to protect the integrity of the archives in the long-term. We developed the Hybrid Merkle Tree, a succinct updatable data structure based on Merkle trees.

Abstract

Every year the amount of digitally stored sensitive information increases significantly. Due to the digitization of such information, adversarial attacks on digital archiving systems have increased significantly as well. In this thesis, we address two areas of digital archiving systems security, mass data breaches and long-term security. Mass data breaches—mass leakage of stored information—are a major security concern. Encryption can provide confidentiality, but encryption depends on a key which, if compromised, allows the attacker to decrypt everything, effectively instantly. Security of encrypted data thus becomes a question of protecting and managing the encryption keys. For long-term security, cryptographic schemes based on single computational assumptions are not guaranteed to stay secure for such long periods so they cannot be used for this purpose. Current state-of-the-art systems providing long-term confidentiality and integrity rely on information-theoretic techniques, such as multi-server secret sharing and commitments. These systems achieve the desired results; however, establishing private channels for secret sharing is costly and requires a complex setup.

This thesis provides solutions for both mass data breaches and long-term security. First, we propose using *keyless encryption* to construct ArchiveSafe, a mass leakage resistant archiving system, where decryption of a file is only possible after the requester, whether an authorized user or an adversary, solves a cryptographic puzzle. This proposal is geared towards protection of infrequently accessed archival data, where any one file may not require too much work to decrypt but decryption of a large number of files—mass leakage—becomes increasingly expensive for an attacker.

Secondly, we present ArchiveSafe LT, a framework for digital archiving systems aiming to provide long-term confidentiality and integrity. The framework relies on

using multiple computationally secure schemes to form robust combiners, with a design that plans for agility and evolution of cryptographic schemes. ArchiveSafe LT is efficient and suitable for practical adoption as it eliminates the need for private channels compared to its counterparts.

Finally, we present the Hybrid Merkle Tree. An authenticated data structure based on the Merkle tree. It supports evolving to a secure hashing function if its hashing function becomes insecure, making it suitable for integrity schemes used by secure long-term digital archiving systems. We show how it can be integrated in ArchiveSafe LT as an example.

Due to the recent increase in digitally stored sensitive information, digital archiving systems have become a crucial part in the information systems space, and we believe their importance will continue to grow in the near future. This research contributes towards the goal of improving the security of these systems in the short and long term.

Acknowledgements

I sincerely thank my supervisors, Dr. Reza Samavi, Dr. Emil Sekerinski and Dr. Douglas Stebila for their support, patience, and guidance throughout my thesis. It has been a pleasure and privilege to have the opportunity to learn from you.

I also thank the members of my thesis committee, Dr. Fei Chiang and Dr. Jelle Hellings who supported my research.

I thank Dr. Ken Baker who appraised my work and provided valuable comments that improved my thesis.

Dedication. . .

In memory of my mother, whose love continues to sustain propelling me forward to this day.

To my children, Salma, Layla and Hussein, thank you for your understanding when I was distracted or not fully present for you while pursuing this dream. I hope the sacrifices you have endured for me to pursue this dream will be reciprocated with abundant opportunities for happiness and accomplishment in your own futures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Digital archiving systems are used to store information that is rarely accessed but must be retained. If the information to be stored is sensitive, the archiving system must be secure, that is, provide confidentiality and integrity protection.

In the past few decades, efforts have been underway to digitize massive amounts of documents containing sensitive information. One example is documents containing personal information such as identification data, pictures, movies and music. Another example is documents containing commercial information such as banking records, invoices and various business transactions. Documents containing critical information such as governmental and legal documents, healthcare, and tax records are examples of documents that have been massively digitized recently and are required to be securely archived for decades to comply with various laws and regulations.

The significant increase in digitally archived information types coincides with the rise of policies and regulations requiring the security of such digital archives to be maintained during their life cycles spanning from years to decades. For example, digital information such as government documents, legal contracts, health, and tax

records are required to be stored securely for several years or decades in some cases where the information is needed for the lifespan of the corresponding individuals or organizations [19].

Since these documents are rarely accessed but must be securely retained for future reference or regulatory compliance, they are stored and maintained by digital archiving systems. As a result, attacks on digital archiving systems have become increasingly common. Whatever the attack vector, a frequent outcome is a *data breach*, in which sensitive information is stolen from the victim organization. Archival data has been targeted in many data breaches [12, 14, 33], leading to loss of privacy, loss of reputation, business setbacks, and costly remediation.

The two main reasons the attacks on digital archiving systems succeed are either the leakage of security information, such as an encryption key or a password or the compromising of a cryptographic scheme due to the advancement of computational powers and cryptanalysis techniques.

1.1 Challenges & Motivation

Digital archiving systems face two major security challenges. The first challenge is the mass information breaches due to a security information leakage, where the adversary gets immediate and complete access to archived information. The second challenge is the failure of traditional security schemes to stay secure in the long-term due to the advancements in computational powers and cryptanalysis techniques.

The first challenge is a result of the current storage and archiving systems focus on

defense-in-depth only in encrypting the data at rest to support confidentiality. However, this approach, even when implemented using secure and carefully implemented algorithms, is typically all-or-nothing: if the key is compromised, the attacker can decrypt everything with minimal overhead. Hardware-assisted cryptography, such as hardware security modules (HSMs), trusted computing, or secure enclaves like Intel SGX[1] or ARM TrustZone[2] may prevent keys from leaking if decryption is only ever done inside a trusted module, but many IT systems remain software-only without the use of these technologies. Security of encrypted data in the current systems is a question of protecting and managing the encryption key. Using a large pool of keys to remediate this problem is problematic. The keys are either carried by the data collector, which is impractical due to cost and complexity or, if a key management system is used, access to this system becomes a single point of failure.

The first objective of this thesis is to find a solution that prevents mass data breaches without the risk of depending on a master encryption key or having to manage a large number of keys.

The second challenge is a result of the evolving nature of standard cryptographic schemes. These schemes are insufficient to accommodate the security of long-term archives since they could fail during the life cycle of the archived documents due to advancements in both computational power and cryptanalysis methods. Current state-of-the-art systems providing long-term confidentiality and integrity such as LINCOS [8], PROPYLA [23], ELSA [35] and SAFE [10], rely on information-theoretic techniques, such as multi-server secret sharing and commitments. These systems achieve the desired results; however, establishing private channels for secret sharing

[1]https://software.intel.com/en-us/sgx

[2]https://developer.arm.com/ip-products/security-ip/trustzone

is costly and requires a complex setup. Additionally, this approach requires heavy data processing, which slows down the archiving operations and by-design requires an inflated storage space since each archive share is the same size as the whole archive.

The second objective of this thesis is to find a solution that guarantees long-term security while decreasing the cost and complexity required by state-of-the-art systems in addition to improved performance.

1.2 Contributions & Thesis Outline

In Chapter 2, we present a review of the literature related to the research work presented in this thesis.

In Chapter 3, we address our first objective by presenting ArchiveSafe, a digital archival system that utilizes cryptographic puzzles to slow down mass data leakages and prevent the attacker from having immediate and unconditional access to all the archived files by compromising a key. ArchiveSafe utilizes disposable keys that are used once and never stored anywhere, so they cannot be leaked. We propose using *keyless encryption*, where decryption of a file is only possible after the requester, whether an authorized user or an adversary, solves a cryptographic puzzle to get the key. This approach makes the decryption of a large number of files—mass leakage—increasingly expensive for an attacker, but to a legitimate digital archiving system user, who rarely accesses limited number of files, the processing needed to solve the puzzles is tolerable.

We present a prototype implementation of ArchiveSafe realized as a user-space file system driver for Linux. We report experimental results of system behavior under different file sizes and puzzle difficulty levels. The results show that ArchiveSafe adds

a slight overhead when writing a file, and a customizable overhead when reading a file depending on the difficulty level chosen. Adding computational overhead at read time is exactly our goal, so an adversary who obtained full access to the system cannot extract all the stored information without paying a large processing price. Choosing the difficulty level depends on the tolerable cost for honest users to access the data, the perceived risk of a data breach, and the anticipated value of the information to an adversary, is a calculation that is left to the adopter.

Our keyless encryption technique can be added as a layer on top of traditional encryption: together, they provide strong security against adversaries.

In Chapter 4, we address our second objective by presenting ArchiveSafe LT, a framework for archival systems providing long-term confidentiality and integrity. ArchiveSafe LT addresses the challenge by utilizing evolving *Robust Combiners* instead of the information-theoretic methods used by the current state-of-the-art systems. A robust combiner combines multiple cryptographic schemes into one, so the resulting scheme is robust to the failure of any of the combined ones. In the long term, any individual computationally secure cryptographic scheme may be broken and deemed insecure; ArchiveSafe LT relies on the combiners to continually be updated to mitigate this risk. ArchiveSafe LT evolves the schemes used in the robust combiner in case any of its schemes are deemed insecure by adding another secure scheme to the combiner. This evolution capability allows ArchiveSafe LT based systems to stay secure in the long term.

Using robust combiners eliminates the need for the costly and complex setup required by these systems, but it comes at the cost of sacrificing information-theoretic security for computational assumptions. We mitigate this risk by utilizing robust

combiners and the novel evolution protocol.

We present in this chapter two system designs based of the ArchiveSafe LT framework. One design is geared towards trusted storage providers and the other is geared towards untrusted ones.

In order to ensure the coverage of all scenarios and usage paths, we use an automatic prover to prove the security of one of the ArchiveSafe LT designs. The second design shares the same main structure as the first one and its security proof should follow the same construction.

We develop an evaluation experiment for one of the designs to measure its performance against state-of-the-art systems. The results show significant improvement in performance and space utilization. ArchiveSafe LT requires only 14% to 33% of the time needed by the current systems to process the same archives' sizes and utilizes less than one-third of the storage space required by these systems.

In Chapter 5, we introduce an improvement to the way ArchiveSafe LT manages long-term integrity. Instead of utilizing multiple Merkle trees and building a new one for every system evolution, we introduce the *Hybrid Merkle Tree*.

The Hybrid Merkle tree is an authenticated data structure based on the Merkle tree. It supports evolving to a secure hashing function if its hashing function becomes insecure, making it suitable for integrity schemes used by long-term secure digital archiving systems. We start by presenting *sStruct*, a succinct updatable proof data structure to model authenticated data structures we use in our archiving integrity schemes. We study the Merkle tree as one instantiation of this structure and prove its security. Next, we present *esStruct*, an evolving version of *sStruct* that is capable of

utilizing multiple compression functions at the same time. *esStruct* has the capability of evolving when the compression function it uses becomes insecure. We study the Hybrid Merkle tree as one instantiation of *esStruct* and prove its security.

Finally, we update the ArchiveSafe LT framework to utilize the Hybrid Merkle tree in its long-term integrity scheme to improve its robustness and effectiveness. We rerun one of the evaluation experiments we used to measure ArchiveSafe LT performance. The results show the expected reduction in tree update time due to the use of a single Hybrid Merkle tree instead of two Merkle trees.

Chapter 6 concludes this thesis by summarizing the results and discussing future research opportunities.

The work presented in Chapter 3 [40] and Chapter 4 [39] is published. These publications are made by the author of this thesis as the lead author in collaboration with his supervisors. The work presented in Chapter 5 has only been published in this thesis.

Chapter 2

Literature Review

In this chapter, we start by introducing the cryptographic concepts and components used throughout this thesis in Section 2.1. Next, we present a literature review of methods and systems providing protection against mass data leakage related to our work in Chapter 3, followed by a literature review of systems providing long-term security protection related to our work in Chapter 4, and finally, a literature review of authenticated data structures used in data integrity protection related to our work in Chapter 5.

2.1 Cryptographic Concepts and Building Blocks

Throughout the thesis, we rely on some cryptographic concepts and use some cryptographic components to build the solutions we propose. In this section, we cover these concepts and components along with some background on their security and attacks.

2.1.1 Cryptographic Concepts

For a cryptographic scheme to be usable, it must be secure. Cryptographic security can have different forms; we introduce some of them here:

Indistinguishability: The definition of indistinguishability is based on an experiment $\text{Exp}_S^{\mathsf{ind}}(\mathcal{A})$, where a passive adversary is given a ciphertext and trying to guess which of two possible messages corresponds to the ciphertext. The indistinguishability experiment is defined as follows:

1. The adversary $\mathcal{A}$ generates two plaintexts m_0 and m_1.
2. A key k is generated and a uniform bit b is chosen.
3. The ciphertext of m_b is computed and given to the adversary.
4. The adversary wins if they can guess b correctly and the experiment outputs 1.

Perfect Security: A perfectly secure scheme does not leak any information about the plaintext or the keys to an eavesdropper. In other words, the advantage of any adversary $\mathcal{A}$ in an indistinguishability experiment $\text{Exp}_S^{\mathsf{ind}}()$ on a scheme S is the same as flipping a fair coin:

$$\text{Adv}_S^{\mathsf{ind}}(\mathcal{A}) = \Pr\left[\text{Exp}_S^{\mathsf{ind}}(\mathcal{A}) \Rightarrow 1\right] - \frac{1}{2}$$

The limitation of this notion according to Shannon's theorem, is that it requires the key space to be at least equal to the message space $|\mathcal{K}| \geq |\mathcal{M}|$, which is practically infeasible. Under perfect security, the adversary is assumed to have unbounded resources.

Computational Security: This is the basis of modern cryptography. While perfect

security requires that absolutely no information about an encrypted message is leaked, even to an eavesdropper with unlimited computational power, it is not required in computational security. A cryptographic scheme is considered computationally secure if it only leaked a negligible amount of information to an eavesdropper with bounded computational power. Under the computational security definition, we have two assumptions:

- The adversary's resources are polynomially bounded.
- The adversary's advantage to win is more than half by a negligible value.

The advantage of any adversary $\mathcal{A}$ in an indistinguishability experiment $\text{Exp}_S^{\mathsf{ind}}()$ on a computationally-secure scheme S is:

$$\text{Adv}_S^{\mathsf{ind}}(\mathcal{A}) = \Pr\left[\text{Exp}_S^{\mathsf{ind}}(\mathcal{A}) \Rightarrow 1\right] - \frac{1}{2} = \epsilon$$

Provable Security: To prove a cryptographic scheme to be secure, we utilize the *Provable Security* methodology. Using this methodology, the scheme's security requirements must be stated formally through an adversarial model. The assumptions regarding the adversary powers must be clearly stated. The proof of the scheme's security is executed through reducing the problem of the scheme's security to a certain computational hard problem. This approach is also called *Reductionist Security*.

Attacks: Encryption schemes attacks fall into four main categories:

- *Ciphertext-Only Attack:* Where the adversary is able to obtain a set of ciphertexts.

- *Known-Plaintext Attack:* Where the adversary is able to obtain a set of pairs of ciphertext and corresponding plaintext.
- *Chosen-Plaintext Attack (CPA):* Where the adversary can choose a number of messages and obtain their corresponding ciphertexts.
- *Chosen-Ciphertext Attack (CCA):* The adversary can choose a number of ciphertexts and obtain their corresponding plaintexts.

Digital signing schemes attacks fall into three main categories:

- *Key-Only Attack:* Where the adversary is able to obtain the verification key.
- *Known-Message Attack:* Where the adversary is able to obtain a list of messages and their corresponding signature.
- *Chosen-Message Attack (CMA):* The adversary can choose a number of messages and obtain their corresponding signatures.

2.1.2 Cryptographic Components

We present here the definitions of the cryptographic components used throughout this thesis.

Symmetric Encryption: A symmetric encryption scheme uses the same key for both encryption and decryption of data. The key is considered a shared secret between the parties involved in securing and sharing the data. The symmetric encryption key size is usually small and the processes of encrypting and decrypting the data is faster compared to asymmetric (public key) encryption.

A symmetric encryption scheme Π with secret key length λ, key space $\mathcal{K} = \{0,1\}^{\lambda}$ and message space $\mathcal{M}$ consists of three algorithms:

- $\Pi.\text{Gen}() \xrightarrow{\$} k$: A (probabilistic) key generation algorithm that generates the key used for encryption and decryption.
- $\Pi.\text{Enc}(k, msg) \xrightarrow{\$} c$: A (probabilistic) encryption algorithm that takes a key $k \in \mathcal{K}$ and a message $msg \in \mathcal{M}$ as input and outputs a ciphertext c.
- $\Pi.\text{Dec}(k, c) \rightarrow msg$: A deterministic decryption algorithm that takes as input a key $k \in \mathcal{K}$ and a ciphertext c and outputs a message $msg \in \mathcal{M}$ or an error $\perp \notin \mathcal{M}$.

The desired security property for a symmetric encryption scheme is indistinguishability under chosen ciphertext attack (IND-CCA). Experiment Figure 2.1 represents the indistinguishable security for a symmetric encryption scheme Π.

$\underline{\text{Exp}^{\text{ind-cca}}_{\Pi}(\mathcal{A})}$:

1. $k \leftarrow_{\$} \Pi.\text{Gen}()$
2. $(m_0, m_1, st) \leftarrow_{\$} \mathcal{A}^{\Pi.\text{Enc}(k,\cdot),\Pi.\text{Dec}(k,\cdot)}()$
3. $b \leftarrow_{\$} \{0, 1\}$
4. $c \leftarrow_{\$} \Pi.\text{Enc}(k, m_b)$
5. $b' \leftarrow_{\$} \mathcal{A}^{\Pi.\text{Enc}(k,\cdot),\Pi.\text{Dec}(k,\cdot \neq c)}(c, st)$
6. return $(b' = b)$

FIGURE 2.1: Security experiment for symmetric encryption scheme Π under IND-CCA

There are two types of symmetric encryption algorithms, stream ciphers and block ciphers. A stream cipher algorithm uses a pseudorandom stream of bits as the key. Plaintext digits are encrypted one at a time using one bit of the stream. *ChaCha20* is an example of a stream cipher algorithm. A block cipher algorithm encrypts a block of plaintext digits at once, each block is encrypted using the full key. *DES, 3DES* and *AES* are examples of block cipher algorithms.

Asymmetric Encryption: An asymmetric encryption scheme uses two keys, a public key to encrypt the data and a private key to decrypt it. An advantage of

this scheme is the elimination of the need for a shared secret between the parties. However asymmetric encryption schemes are generally slower than symmetric schemes in encrypting and decrypting data. An asymmetric encryption scheme Σ with secret key length λ, key space $\mathcal{K} = \{0,1\}^\lambda$ and message space $\mathcal{M}$ consists of three algorithms:

- $\Sigma.\text{Gen}() \xrightarrow{\$} (k_{priv}, k_{pub})$: A (probabilistic) key generation algorithm that generates a key pair (k_{priv}, k_{pub}) used for decryption and encryption respectively.
- $\Sigma.\text{Enc}(k_{pub}, msg) \xrightarrow{\$} c$: A (probabilistic) encryption algorithm that takes the public key $k_{pub} \in \mathcal{K}$ and a message $msg \in \mathcal{M}$ as input and outputs a ciphertext c.
- $\Sigma.\text{Dec}(k_{priv}, c) \rightarrow msg$: A deterministic decryption algorithm that takes the private key $k_{priv} \in \mathcal{K}$ and a ciphertext c as input and outputs a message $msg \in \mathcal{M}$ or an error $\perp \notin \mathcal{M}$.

The desired security property for an asymmetric encryption scheme is indistinguishability under chosen ciphertext attack (IND-CCA). Experiment Figure 2.2 represents the indistinguishable security for an asymmetric encryption scheme Σ.

$\underline{\text{Exp}^{\text{ind-cca}}_{\Sigma}(\mathcal{A})}$:

1. $(k_{priv}, k_{pub}) \leftarrow_\$ \Sigma.\text{Gen}()$
2. $(m_0, m_1, st) \leftarrow_\$ \mathcal{A}^{\Sigma.\text{Dec}(k_{priv}, \cdot)}(k_{pub})$
3. $b \leftarrow_\$ \{0,1\}$
4. $c \leftarrow_\$ \Sigma.\text{Enc}(k_{pub}, m_b)$
5. $b' \leftarrow_\$ \mathcal{A}^{\Sigma.\text{Dec}(k_{priv}, \cdot \neq c)}(c, st)$
6. return $(b' = b)$

FIGURE 2.2: Security experiment for asymmetric encryption scheme Σ under IND-CCA

RSA, Diffie-Hellman and *ElGamal* are examples of asymmetric encryption algorithms.

Hash Functions: A hash function H maps an input string of arbitrary length to an output string of fixed length. A hash function H is represented as: $H : 0,1^* \rightarrow 0,1^n$, where n is the fixed hash value length.

A secure hash function must be:

- Collision Resistant: It is hard to find two input m_0 and m_1 such that $H(m_0) = H(m_1)$.
- Preimage Resistant: Given $h \in \{0,1\}^n$, it is hard to find a value m such that $H(m) = h$.
- Second Preimage resistant: Given m_0, it is hard to find a value $m_1 \neq m_0$ such that $H(m_0) = H(m_1)$.

The collision resistance of a hash function H is defined by the experiment in Figure 2.3.

$\underline{\mathrm{Exp}^{\mathsf{coll}}_H(\mathcal{A})}$:

1. $(m_0, m_1) \leftarrow_{\$} \mathcal{A}()$
2. return $(m_0 \neq m_1) \wedge (H(m_0) = H(m_1))$

FIGURE 2.3: Collision resistance experiment for a hash function H

Message Authentication Code (MAC): To verify the authenticity of data message, a MAC could be used. A compression function is used to map the message to a compressed data value, the MAC. The MAC, also called the *tag* sometimes, is used to verify the message authenticity. A MAC scheme Γ with secret key space $\mathcal{K} = \{0,1\}^\lambda$ and message space $\mathcal{M}$ consists of three algorithms:

- $\Gamma.\mathrm{Gen}() \xrightarrow{\$} k$: A (probabilistic) key generation algorithm that generates the key to be used in signing and verifying the data.

- $\Gamma.\text{Sign}(k, msg) \xrightarrow{\$} tag$: A (probabilistic) compression algorithm that takes as input a key k and a message msg and outputs a tag.
- $\Gamma.\text{Verify}(k, msg, tag) \rightarrow \{0, 1\}$: A deterministic verification algorithm that takes as input the key k, a message msg and a signature tag and outputs 1 if the verification is successful and 0 otherwise.

The desired security property for a MAC scheme is existential unforgeability against a chosen message attack (EUF-CMA). Experiment Figure 2.4 represents the indistinguishable security for a MAC scheme Γ.

$\text{Exp}_{\Gamma}^{\mathsf{forge}}(\mathcal{A})$

1 : $I \leftarrow \emptyset$
2 : $k \leftarrow_{\$} \Gamma.\text{Gen}()$
3 : $(msg', tag') \leftarrow \mathcal{A}^{\mathsf{OSign},\mathsf{OVerify}}$
4 : **return** $((tag' = \Gamma.\text{Verify}(k, msg')) \wedge (msg' \notin I))$

$\mathsf{OSign}(msg)$

1 : $tag = \Gamma.\text{Sign}(msg)$
2 : $I \leftarrow I \cup msg$
3 : **return** tag

$\mathsf{OVerify}(msg, tag)$

1 : **return** $(tag = \Gamma.\text{Verify}(msg, tag))$

FIGURE 2.4: Security experiment for existential unforgeability under chosen message attack for a MAC scheme Γ

Digital Signing: A digital signing scheme is the public-key equivalent to a MAC scheme. Using a digital signing scheme, the owner party uses a private key to generate a digital signature of the stored data, a value that is hard to forge. The signature could be used to verify the authenticity of the stored data by any interested party using a public key. A digital signing scheme Δ with secret key space $\mathcal{K} = \{0, 1\}^{\lambda}$ and message space $\mathcal{M}$ consists of three algorithms:

- $\Delta.\text{Gen}() \xrightarrow{\$} (k_{sig}, k_{ver})$: A (probabilistic) key generation algorithm that generates a pair of keys. k_{sig} for signing the data and k_{ver} to verify the data.
- $\Delta.\text{Sign}(k_{sig}, msg) \xrightarrow{\$} s$: A (probabilistic) signing algorithm that takes as input the signing key k_{sig} and a message msg and outputs a signature s.
- $\Delta.\text{Verify}(k_{ver}, msg, s) \rightarrow \{0, 1\}$: A deterministic verification algorithm that takes as input the verification key k_{ver} and signature s and outputs 1 if the verification is successful and 0 otherwise.

The desired security property for a digital signing scheme is existential unforgeability against a chosen message attack (EUF-CMA). Experiment Figure 2.5 represents the indistinguishable security for a digital signing scheme Δ.

$$\underline{\text{Exp}^{\mathsf{forge}}_{\Delta}(\mathcal{A})\text{:}}$$

1. $(k_{sig}, k_{ver}) \xleftarrow{\$} \Delta.\text{Gen}()$
2. $(m, s) \xleftarrow{\$} \mathcal{A}^{\Delta.\text{Sign}(k_{sig}, \cdot \neq m)}(k_{ver})$
3. return $\Delta.\text{Verify}(k_{ver}, m, s)$

FIGURE 2.5: Security experiment for existential unforgeability under chosen message attack for digital signing scheme Δ

2.2 Mass Data Leakage

In this section, we present a background and literature review related to our ArchiveSafe system presented in Chapter 3. We review the related literature on systems providing mass data leakage protection to databases and filesystem encryption in Section 2.2.1. We present a literature review of cryptographic puzzles systems in Section 2.2.2 and the application of cryptographic puzzles to support confidentiality in Section 2.2.3.

2.2.1 Database and Filesystem Protection

File and database systems have been encrypted traditionally by using one of two methods: encrypting the entire system using a single master key or using multiple keys to secure different parts of the system. The first method introduces the risk of having the entire system unsecured if the single master key is compromised. The second method introduces the complexity of managing keys and, in some cases, the cumbersomeness of diminished usability of entering multiple keys by a user in order to access the secured information.

There are several systems offering full or partial cryptographic services such as encryption, decryption, signing and verification of files. Blaze [6] introduced the Cryptographic File System (CFS), which provides a virtual filesystem as a layer between the user and the operating system; cryptographic services are a basic functionality of the virtual filesystem. CFS uses a different key for each directory, and the user is required to enter the key for the directory in every session to access the directory and its contents. The Transparent Cryptographic File System (TCFS) proposed by Cattaneo et al. [13] tied directory encryption keys to a single master encryption key (which could compromise the whole system) and implemented TCFS as a modified kernel-mode version of Sun's Network File System (NFS) client. Subsequent proposals of operating systems providing sorts of cryptographic services include Cryptfs [47] and Ncryptfs [46].

In recent years, encrypted filesystems have become widespread, and all major operating systems (OS) provide cryptographic services implementations, often enabled by default. Examples of OS providing cryptographic services are FileVault on Apple's

macOS[1] and BitLocker on Microsoft Windows[2], in addition to a range of options on Linux such as Linux Unified Key Setup (LUKS)[3]. The common practice in these technologies is to use a single master key from which multiple keys are derived per-file, per-directory, or per-sector; the master key is usually stored on the device itself and encrypted under the user's password. Once the user has logged in, the filesystem transparently and automatically decrypts files for all applications.

In addition to filesystem encryption, a broad range of database encryption systems exist. The general approach in database encryption systems is applying traditional symmetric or asymmetric encryption algorithms to entire databases and tables, rows, or even individual fields. Assuming key management is successful and usable, this approach provides strong confidentiality against adversaries who obtain the encrypted database, but legitimate users lose the ability to perform certain types of queries without decrypting the entire table.

Over the past decade, there has been much research on encrypted databases that retain some functionality for legitimate users, for example using order-preserving encryption so that sorting a column of ciphertexts yields approximately the same order as if the plaintexts were sorted. This increased functionality comes at the cost of information leakage.

Some examples of such proposals are CryptDB [37], Arx [36] and PuzzleDB [34]. CryptDB works by executing SQL queries over encrypted data using a collection of efficient SQL-aware encryption schemes. Arx, is another database encryption system

[1]https://support.apple.com/en-ca/HT204837

[2]https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview

[3]https://guardianproject.info/archive/luks/

which encrypts the data with semantically secure encryption schemes. In PuzzleDB, the database records are protected by client puzzles.

Although our system, presented in Chapter 3, utilizes the same cryptographic components as the aforementioned proposals such as symmetric encryption and key wrapping, it targets a different goal. Our system's goal is to protect the filesystems from an adversary who has already obtained full access to the filesystems as opposed to protecting the systems from an adversary who is trying to get access. Moreover, our keyless wrapping approach eliminates the need for key management by using a unique encryption key for each file in the system compared to using a master key for encryption as presented in [13].

2.2.2 Cryptographic Puzzles Systems

Dwork and Naor [18] introduced client puzzles to control junk email: recipients would only accept emails if the sender was able to solve a puzzle. It should be "moderately hard" for the sender to solve the puzzle, but easy for the recipient to check whether a solution is valid. This was the first example of a cryptographic puzzles system, which in general grants access to a resource dependent on the requester being able to demonstrate proof that they have performed some work, typically in the form of solving a puzzle. Client puzzles were for many years suggested as a means to prevent denial of service attacks in a range of contexts but have seen renewed interest as a building block for cryptocurrencies and blockchains. Aura et al. [3] presented a DOS-resistant authentication with client puzzles was presented. Juels and Brainard [28] presented a cryptographic countermeasure against connection depletion attacks. Dean and Stubblefield [15] presented a system where client puzzles were used to protect Transport Layer Security (TLS).

There is a range of client puzzles in the literature based on different computational problems, and having a range of characteristics. One classification of client puzzles is whether the limiting factor in the ability to solve the puzzle quickly is the CPU speed or memory access time, typically called CPU-bound versus memory-bound, respectively. Another classification is whether solving the puzzle is a parallelizable operation where the work needed could be split between parallel processes; or a sequential operation where every step of the solution depends on the result of the previous one, so the processing to solve the puzzle has to be carried out in order.

The simplest CPU-bound puzzles are based on cryptographic hash functions, such as: finding a preimage of a hash given a hint (for example, a part of the preimage); or finding an input whose hash starts with a certain number of zero bits. These types of cryptographic hash functions are easily parallelizable. Additionally, there is usually a high amount of variance in their solution time, since each guess is essentially independent and equally likely to be correct.

Non-parallelizable CPU-bound puzzles often rely on theoretic approaches [38, 45]. For example, [38] uses repeated squaring modulo an RSA modulus, for which the best known technique to compute the solution (without the factorization of the modulus) is repeated squaring, but can be verified efficiently using a trapdoor (the factorization of the modulus). Often these types of puzzles have low variance in their solution time, making them more amenable to scenarios where it is desirable that the solving time for puzzle instances be predictable.

Memory-bound puzzles [1, 17] use techniques for which the best known solving algorithm involves a large number of memory accesses; it is argued that memory access time varies less than CPU speed between small and large computing platforms, and

that building customized hardware is more expensive for memory-bound puzzles.

In our research, the goal is to enforce a set amount of computing on the requester regardless of whether or not the computing could be parallelized. The idea is to make the process of compromising the system highly expensive for mass leakage attempts by adversaries but still within the acceptable user experience range for an honest user. Additionally, an adversary targeting mass information leakage, could parallelize solving puzzles for multiple files instead of parallelize solving a single file's puzzle, rendering using a sequential puzzle pointless. For these reasons and our need to measure and compare the efforts needed to perform other tasks in the system such as decryption, encryption and puzzle creation, we chose to use CPU bound puzzles to implement our proof of concept.

2.2.3 Cryptographic Puzzles Systems for Confidentiality

Time-lock encryption was proposed by Rivest, Shamir, and Wagner [38] as a way of "sending information into the future", and focused specifically on hiding keys or data in a cryptographic puzzles system that had a predictable wall-clock time for solving, thus focusing on puzzles for which the best known solving algorithm is inherently sequential. In 2018, Vargas et al. [44] described a database encryption system called "Dragchute" based on time-lock encryption, aiming to provide both confidentiality and the ability to demonstrate compliance with mandatory retention laws for data. In their system, records in a database are encrypted using time-lock encryption, so anyone who wants to access the data – legitimate user or attacker – must solve a non-parallelizable task for a predictable amount of time. Each ciphertext in their system is accompanied by an authentication tag which contains a non-interactive zero-knowledge proof that the ciphertext properly escrows the data (that is, solving

the task will indeed yield a valid decryption key for the ciphertext); moreover, that proof can be checked much more efficiently than the full work required to solve and decrypt the ciphertext. A simpler database encryption scheme relying on hash-based client puzzles, without any efficient verification of well-formedness, was proposed by Moghimifar in a Master's report [34].

In some sense, password hashing functions also use cryptographic puzzles to enhance confidentiality. Best-practices for storing password data to verify logins involve storing salted hashes generated using a *slow* hash function: a hash function is used many times sequentially to hash the password (and salt) when it is stored, and the verifier must repeat the same sequence of hashes to check a provided password against the hash during login. Confidentiality of stored password data is enhanced in the sense that an attacker who obtains the database of stored hashes can only obtain information about a password by doing a fresh brute force search against each stored hash, and each guess requires evaluating the slow hash function. One notable difference between password hashing and client puzzles is that most client puzzles can be verified without having to repeat the whole solving process. Quick verification of the hash is why client puzzles are the best choice for our work. In our proposed system, the user must be able to verify a solution for the puzzle with negligible effort leaving the main solving effort to the puzzle guessing.

2.3 Long-Term Security

In this section, we present a background and literature review related to secure long-term digital archiving systems and our system presented in Chapter 4. The current state-of-the-art digital archiving systems providing long-term confidentiality utilize

secret sharing techniques [41] in conjunction with information-theoretic secure key agreements such as Quantum Key Distribution (QKD), key agreements through noisy channel models, the Bounded Storage Model and the limited access model [9]. For integrity, these systems utilize variations of commitment schemes.

In 2017, Braun et al. presented a secure storage system, LINCOS [8], which provides long-term protection of confidentiality and integrity. LINCOS uses proactive secret sharing for confidential storage of secret data and information-theoretic hiding commitments for confidentiality preserving integrity. Within the proactive secret sharing protocol, LINCOS uses quantum key distribution and one-time pad encryption for information-theoretic private channels. The system was implemented on the Tokyo QKD network for experimental evaluation. LINCOS addresses a simplified all-or-nothing storage scenario, where only the whole archive can be accessed and/or verified. To overcome this problem, Geihs et al. introduced PROPYLA [23], a privacy-preserving long-term secure storage, which was designed using the same concepts as LINCOS but combined information-theoretic secret sharing, renewable timestamps, and renewable commitments with an information-theoretic oblivious random-access machine. PROPYLA is able to verify parts of the archive without accessing the whole archive.

In 2018, Muth et al. presented ELSA: Efficient Long-Term Secure Storage of Large Datasets [35]. ELSA provides long-term confidentiality and integrity protection of large datasets by utilizing proactive secret sharing and renewable vector commitments in combination with renewable timestamps. The new commitment approach is designed to overcome the slow performance of LINCOS when working on datasets that hold a large number of relatively small data items.

In 2020, Buchmann et al. introduced SAFE: Secure and Efficient Long-Term Distributed Storage System [10], using a different approach for key distribution and secret sharing. SAFE uses a secret sharing method similar to the previous systems but utilizes a trusted execution environment (TEE) for secret calculation and distribution. SAFE requires having a trusted TEE provider in addition to multiple secure communication channels as in the previous systems.

All systems described above are similar in terms of using proactive secret sharing for long-term confidentiality which increases the size of the archive significantly and risks having the data kept in plain text during shares renewal. Key generation is another challenge for these approaches. On average, the QKD key supply rate drops significantly for distances more than 100 km [2]. The best rate LINCOS could achieve was 40 Kb/s. Thus a 158 GB archive with secret sharing would require 2.3 days for key generation and transmission. These systems also require secure channels for one-time pads or QKD; both are challenging to be realized practically due to the complicated logistics of sharing One Time Pads (OTP) or dedicated hardware for QKD. Other limitations of these systems are the need for an information-theoretically secure channel between any two nodes involved in the process of share renewal and that long-term confidential commitment schemes are computationally impractical for large files.

Among the systems described, only SAFE's implementation eliminates the complexity of the OTP and QKD channels and the risk of having the data in a plain state during shares renewal. However, the rest of the challenges still persist in SAFE. Additionally, SAFE requires the involvement of a third-party TEE provider, which adds to the complexity of the system setup and increases the risk of trusting an additional party. Utilizing TEEs could implicitly introduce computational assumptions because

the use of computationally secure encryption schemes to encrypt the contents of its memory.

2.4 Authenticated Data Structures & Merkle Trees

In this section, we present a background and literature review related to authenticated data structures and our structures presented in Chapter 5. Authenticated Data Structures (ADS) are used in many areas of computer science [11] [22] [30]. They are used to verify if a data object belongs to an ordered list $\mathcal{D}$ without revealing or exchanging the data object itself. An ADS model consists of two main actors, a data collector who owns the original data objects in $\mathcal{D}$ and a requester who sends verification requests to the data collector. In many cases, a third party verifier is involved to handle the verification requests. The verifier owns the ADS and uses it to answer the verification requests but does not have access to the data objects. An ADS implementation using skip lists was introduced [24] and a hash tree implementation called Merkle trees was introduced by RC Merkle [32].

Merkle trees are binary trees where the leaves are the hash values of the data objects. Each internal node is the hash of its two child nodes. In the tree shown in Figure 2.6, in order to verify d_4, the verification process requires the following nodes to be provided: h_3, h_4, $h_{3,4}$ and the root along with the data item to be verified, d_4. The tree root and the nodes connecting it to the leaf to be verified are used to verify whether a data object belongs to the ordered list represented by the tree or not. The verification process follows the path from the leaf to the root by recalculating the hashes of the internal nodes up to the root, then comparing the result to the stored tree root. If the two values match, then the data object is verified. Otherwise, the

verification fails.

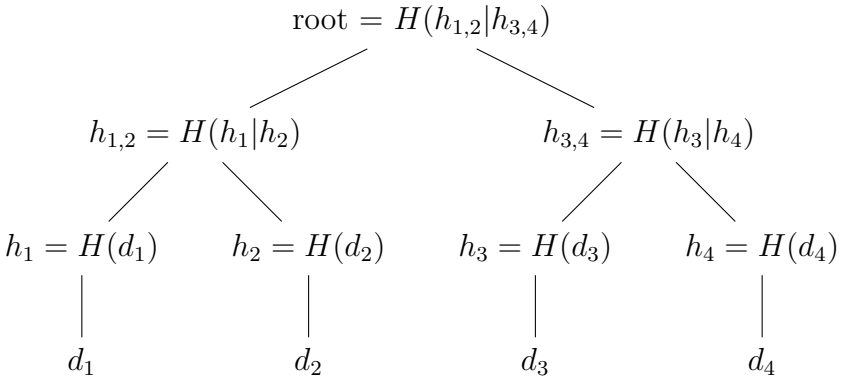

FIGURE 2.6: Merkle tree example

Chapter 3

ArchiveSafe: Mass-Leakage-Resistant Storage from Client Puzzles

In this chapter we present ArchiveSafe, a *mass leakage resistant archiving system* with the goal of enhancing defense-in-depth for encryption. We aim to preserve confidentiality even in the presence of an adversary with full access to the system, including ciphertexts and decryption keys. While no system can provide full cryptographic security in the face of such a well-informed adversary, our goal is to increase the economic cost of *mass leakage*, which for our purposes is defined as an adversary obtaining the plaintexts of a large number of files or database records, not just one.

Unlike most applications of cryptography, we do not aim to achieve a difference in work factor between honest parties and adversaries. Rather, we assume that honest parties and adversaries have different goals, and we aim to change the economics of data breaches by achieving a difference in the cost of honest parties and adversaries

achieving their goals. In our scenario, honest parties need to store a large number of files, but only access a small number of them. Consider for example a tax agency: after processing millions of citizens' tax returns each year, those files must be stored for several years in case an audit or further analysis is required, but only a small fraction of those records will end up actually being pulled for analysis. In contrast, an adversary breaching the tax agency's records may want to read a large number of files to identify good candidates for identity theft or other criminal actions.

We highlight that ArchiveSafe is meant to add defense-in-depth to confidentiality; one would typically not rely on ArchiveSafe alone, but combine it with traditional encrypted file system or database encryption. ArchiveSafe main component is a difficulty-based keyless encryption scheme *DBKE* which is similar to a symmetric encryption scheme, except that no secret key is kept for use between the encryption and decryption algorithm.

In this combination, traditional encryption using strong algorithms and keys provides a high level of security if the keys are not compromised, but we still have the difficulty-based keyless encryption of ArchiveSafe as a bulwark if the keys are compromised. To succeed under this setup, the adversary must compromise the traditional encryption keys in addition to solving a large number of DBKE puzzles corresponding to the files in the archive.

In this research, we build a prototype implementation showing the use of ArchiveSafe on a local computer. Our prototype is implemented as a filesystem-in-userspace (FUSE) driver on Linux. A FUSE driver can be used to intercept I/O operations in certain directories (mount points) before reading/writing to disk. This allows us to implement ArchiveSafe in a manner that is transparent to the application, as well as transparent

to the underlying storage mechanism, which could be a local disk (with normal disk encryption enabled or not), or a network share mounted locally. We validate the performance of our prototype implementation, focusing primarily on ensuring that write operations incur minimal overhead. Since system administrators can set policies with puzzle difficulties requiring seconds or minutes of computational effort to solve, slow read performance is intended, and there is little sense in performance measurements on reads, beyond checking that they scale as intended with no unexpected overhead. We envision that, when used on a local computer, ArchiveSafe would be applied only to a subset of the directories on the computer. One might use ArchiveSafe to protect documents created by the user more than a certain number of days ago, but would not use it on system libraries and executables.

We start by presenting an overview of ArchiveSafe in the next section followed by the system's requirements in Section 3.2. Next, we introduce our Difficulty-Based Keyless Encryption scheme used in ArchiveSafe in Section 3.3, the experimental evaluation of the system in Section 3.4 and finally a summary of the work in Section 3.6.

3.1 ArchiveSafe Overview

We design *ArchiveSafe*, where access to a resource is only possible after the requester—whether an honest user or an adversary—has expended sufficient computational effort, in the form of solving a "moderately hard" cryptographic puzzle [18]. Since we will not rely on the access control system nor any keys to be uncompromised, the decryption operation itself must be tied to the cryptographic puzzle. In our approach, while a proper cryptographic key is used to encrypt a file, the encryption key is not stored, even for legitimate users. Instead, the key is wrapped in a client puzzle based encryption

scheme with a desired difficulty level, and all users—adversarial or honest—must solve the puzzle to recover the key and then decrypt the file.

Our main technical tool for building of ArchiveSafe is a new cryptographic primitive that we call difficulty-based keyless encryption (DBKE), which is an encryption scheme that does not make use of a stored key. We give a generic construction for DBKE from a standard symmetric encryption scheme and a new tool called difficulty-based keyless key wrap, which wraps the symmetric encryption key in an encapsulation that can only be unwrapped by performing a sufficiently high number of operations, as in a client puzzle based scheme. Difficulty-based keyless key wrap can be achieved from many types of cryptographic puzzles, and we show one example based on hash function partial pre-image finding [28, 27]. One interesting feature of using this form of hash-based puzzle, which to our knowledge is a novel observation on hash-based puzzles, is that the puzzle and ciphertext can be *degraded*—that is, turned into a harder one—essentially for free. We use the reductionist security methodology to formalize the syntax and security properties of difficulty-based keyless encryption and keyless key wrap and show that our hash-based construction achieves these properties.

Figure 3.1 gives a high-level overview of how an application interacts with the ArchiveSafe system. The two main operations performed by the ArchiveSafe system are (i) creating a puzzle and encrypting during writes, and (ii) solving the puzzle and decrypting during reads. ArchiveSafe could be used in a variety of data storage architectures: on a local computer; on a file server; or in a cloud architecture. In a file server or cloud scenario, an IT system may be set up so the file server enforces that all files are protected by ArchiveSafe during writes by centralizing puzzle creation and encryption, but leaves puzzle solving and decryption to clients. Since puzzle creation and encryption in our system is cheap, this avoids bottlenecks on the file

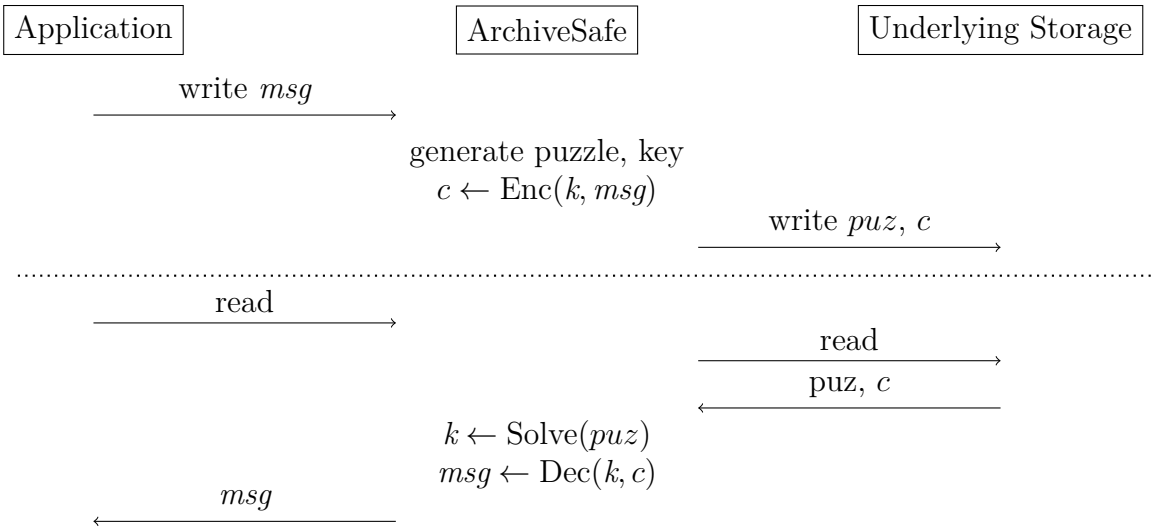

FIGURE 3.1: High-level overview of ArchiveSafe, showing a write followed by a read

server. Individual client applications occasionally reading a small number of files have to do a moderate, but not prohibitive, amount of work to solve the puzzle to obtain the key to decrypt.

3.2 Requirements

In this section, we discuss the functionality and security requirements for a mass leakage resistant archiving system, which informs our construction and evaluation in subsequent sections.

3.2.1 Design Criteria

Confidentiality in the face of compromised keys. The system should achieve some level of confidentiality even if all stored keys are compromised. This means we assume that an adversary can learn a symmetric key or a private key corresponding to a public key stored for later use in decrypting a ciphertext, even if the key is stored in a

separate key management service, trusted computing or secure enclave environment, or separate tamper-resistant device.

Cooperation with traditional encryption. It should be possible to use the system in conjunction with the traditional encryption mechanisms applied to storage systems (folder/disk encryption, database encryption, etc.), so that strong confidentiality is achieved if keys are not compromised, but some confidentiality is retained in the face of compromised keys.

Reliance on industry standard cryptographic algorithms. Deployed IT systems should rely only on well-vetted, standardized cryptographic algorithms. But all such algorithms for achieving confidentiality—public key or symmetric—require a secret key, seemingly conflicting with the first design criteria of confidentiality in the face of compromised keys. Our construction builds a mechanism for confidentiality without keys while still relying on standard cryptographic algorithms like AES for symmetric encryption: while a proper cryptographic key is used to encrypt data, that key is not kept, even by authorized users. Instead, the key is wrapped in a client puzzle based encryption scheme with a desired difficulty level, and users must solve the puzzle to recover the key and then decrypt the data. We introduce difficulty-based keyless encryption in Section 3.3 which formalizes this idea and generically construct it from standard cryptographic algorithms such as AES and Argon2.

Imposing a significant cost to access a large number of files while maintaining acceptable cost to access one file. Since we do not have a key that gives honest users an advantage over the adversary, we should look at things from the viewpoint of typical honest behaviour—periodically accessing a small number of files—versus adversary behaviour—accessing a large number of files in a data breach. Proof-of-work and related techniques

have long been used to achieve security goals from that viewpoint, whether in password hardening or client puzzles for denial of service resistance.

Customizing file access cost. It should be possible for a system administrator or user to control the cost incurred by the adversary or honest user for accessing a file. This may be set as a system-wide policy or a file-by-file basis, depending on the desired access control paradigm. This is achieved in our system by varying the difficulty level of the puzzle wrapping the decryption key.

A related design criteria is the ability to customize file access cost over time. Demand for access to records may change over time; for example, records older than 5 years may be accessed much less frequently than more recent records. Our system allows the file access cost to be increased with minimal effort, through a process we call *puzzle degradation*, that could be performed as part of regular system maintenance. This is a novel feature available from some types of puzzle constructions but not others, and in particular not from the number-theoretic repeated squaring non-parallelizable constructions used in time-lock puzzles [38] and the Dragchute database encryption system [44].

3.2.2 Choice of Puzzle

One of the major design decisions for our system is which type of puzzles to use: sequential versus parallelizable, and CPU-bound versus memory-bound.

As our design criteria focus on mass leakage adversaries trying to decrypt many files, and since we think of cost in a general economic sense, we do not have to restrict to client puzzles that are sequential/non-parallelizable. Concerned with an adversary trying to decrypt many files who has parallel computing resources available to them, it

does not matter whether they choose to deploy their parallel resources to sequentially decrypt each file quickly or in parallel decrypt many files more slowly. Overall, they will decrypt the same number of files with the same resources. Additionally, using a sequential puzzles does not prevent the adversary from decrypting large number of files in parallel, which void the point of using sequential puzzles. We also need not worry about the variability of puzzle solving time for individual instances, only the expected puzzle solving time for many instances. These design choices are, for example, significantly different from those of the Dragchute system for database confidentiality and integrity from proof-of-work. Moreover, parallelization permits honest users to reduce the latency in occasional access of files by taking advantage of short, on-demand use of cloud servers (see Table 3.5).

Whereas sequential versus parallelizable puzzles is a qualitative choice for our scenario, CPU-bound versus memory-bound is a quantitative choice with respect to the economic cost. To achieve a given dollar-cost-for-adversary, it is possible to pick appropriate parameters for both CPU-bound and memory-bound puzzles under appropriate cost and puzzle-solving assumptions. So, a priori, either can be used in our constructions. For our prototype we choose simple hash-based CPU-bound puzzles because puzzle creation is cheaper (thereby achieving extremely low overhead on write operations) and because they allow us to obtain novel useful functionality such as puzzle degradation (Section 3.3.4), but with the hash function being Argon2 which is designed to be resistant to GPU and ASIC optimization. Picking appropriate difficulty levels for puzzles is something an adopter must do as a function of the tolerable cost for honest users to access data, the perceived risk of a data breach, and the anticipated value of the information to an adversary. We do not aim to study such economic calculations exhaustively, but we provide one worked example in Section 3.4.4 and

Table 3.5.

3.2.3 Threat Model

ArchiveSafe is a software system with one target asset, the data files. The security goal for the target asset is confidentiality. As shown in Figure 3.1, information flows from the user application through the ArchiveSafe driver to the underlying storage during writes, and in the reverse direction during reads.

An adversary could access the system either via the same mechanism as an honest user application (that is, mediated by the ArchiveSafe driver), or may have direct access to the underlying storage. We aim to achieve confidentiality against a strong adversary that can bypass the ArchiveSafe driver during read operations (e.g., because they are untrusted server administrators, or because they have compromised the kernel using privilege escalation), or who can directly read from the underlying storage (e.g., an untrusted cloud storage provider, or physical theft of a hard drive). We do not consider in our threat model an adversary who undermines the write operation to intercept data during a write operation or who prevents the ArchiveSafe technique from being applied when saving files. We assume operations by honest parties are performed on a trusted and uncompromised system that faithfully deletes keys from memory once an operation is completed.

3.2.4 Limitations

ArchiveSafe does not manage interrelationships between files. Related files such as spousal tax or medical records, payroll records for employees with similar roles are treated independently by the system. The responsibility of setting their puzzle difficulty levels based on their relationship is left to the system administrator.

Protecting archive and files metadata and files access patterns is outside of the scope of this work.

3.3 Difficulty-Based Keyless Encryption

A difficulty-based keyless encryption scheme is similar to a symmetric encryption scheme, except that no secret key is kept for use between the encryption and decryption algorithm.

Definition 1 (Difficulty-Based Keyless Encryption)**.** *A difficulty-based keyless encryption (DBKE) scheme Δ for a message space $\mathcal{M}$ with maximum difficulty $D \in \mathbb{N}$ consists of two algorithms:*

- $\Delta.\text{Enc}(d, msg) \xrightarrow{\$} c$*: A (probabilistic) encryption algorithm that takes as input difficulty level $d \leq D$ and message msg and outputs ciphertext c.*
- $\Delta.\text{Dec}(c) \rightarrow msg'$*: A deterministic decryption algorithm that takes as input ciphertext c and outputs message msg' or an error $\perp \notin \mathcal{M}$.*

A DBKE scheme Δ is *correct* if, for all messages $m \in \mathcal{M}$ and all difficulty levels $d \leq D$, we have that $\Pr[\Delta.\text{Dec}(\Delta.\text{Enc}(d, msg)) = msg] = 1$, where the probability is taken over the randomness of Δ.Enc.

The desired security property for a DBKE is semantic security in the form of ciphertext indistinguishability. Since there is no persistent secret key, there is no need to consider security notions incorporating chosen plaintext or chosen ciphertext attacks: each plaintext is protected by independent randomness. The security experiment $\text{Exp}^{\mathsf{db\text{-}ind}}_{\Delta,d}(\mathcal{A})$ for an adversary $\mathcal{A}$ trying to break indistinguishability of DBKE scheme Δ at difficulty level d is shown in Figure 3.2. The difficulty level d ranges between 0

$\underline{\mathrm{Exp}^{\mathsf{db\text{-}ind}}_{\Delta,d}(\mathcal{A})}$:	$\underline{\mathrm{Exp}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{A})}$:
1. $(m_0, m_1, st) \leftarrow\!\!\$\ \mathcal{A}(1^d)$	1. $(k_0, w) \leftarrow\!\!\$\ \Sigma.\mathrm{Wrap}()$
2. $b \leftarrow\!\!\$\ \{0,1\}$	2. $k_1 \leftarrow\!\!\$\ \mathcal{K}$
3. $c \leftarrow\!\!\$\ \Delta.\mathrm{Enc}(d, m_b)$	3. $b \leftarrow\!\!\$\ \{0,1\}$
4. $b' \leftarrow\!\!\$\ \mathcal{A}(c, st)$	4. $b' \leftarrow\!\!\$\ \mathcal{A}(w, k_0, k_1)$
5. return $(b' = b)$	5. return $(b' = b)$

FIGURE 3.2: Security experiments for (*left*) indistinguishability of difficulty-based keyless encryption scheme Δ at difficulty level d; and (*right*) indistinguishability of difficulty-based keyless key wrap scheme Σ with keyspace $\mathcal{K}$ and difficulty level d

where $\mathcal{A}$ does not have to solve a puzzle and the maximum difficulty $D \in \mathbb{N}$ where $\mathcal{A}$ has to guess the whole key.

We define the advantage of such an adversary in the security experiment as

$$\mathrm{Adv}^{\mathsf{db\text{-}ind}}_{\Delta,d}(\mathcal{A}) = \left| 2 \cdot \Pr\left[\mathrm{Exp}^{\mathsf{db\text{-}ind}}_{\Delta,d}(\mathcal{A}) \Rightarrow \mathtt{true} \right] - 1 \right|.$$

Useful forms of $\mathrm{Adv}^{\mathsf{db\text{-}ind}}_{\Delta,d}(\mathcal{A})$ will relate the amount of work done by the adversary, the difficulty level, and the adversary's success probability.

3.3.1 Generic Construction of DBKE

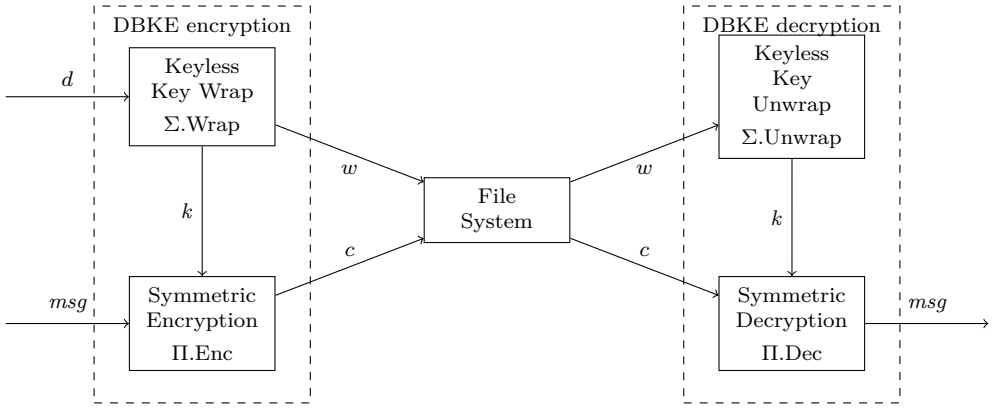

FIGURE 3.3: Architectural diagram for generic construction of a difficulty-based keyless encryption scheme $\Gamma = \Gamma[\Pi, \Sigma]$ from a difficulty-based keyless key wrap scheme Σ and a symmetric encryption scheme Π

Our main construction of DBKE, as shown in Figure 3.3, generically combines a traditional symmetric encryption scheme with a *keyless key wrap* which is a difficulty-based form of key wrapping: there is no master key wrapping the session key, instead the session key is recovered via some difficulty-based operation. In this section we present the generic building blocks we use to construct DBKE. In Section 3.3.2 we show how to instantiate the keyless key wrap.

Definition 2 (Keyless key wrap scheme)**.** *A* keyless key wrap scheme Σ *for a key space* $\mathcal{K} = \{0,1\}^\lambda$ *with maximum difficulty level* $D \in \mathbb{N}$ *consists of two algorithms:*

- $\Sigma.\text{Wrap}(d) \xrightarrow{\$} (k, w)$*: A (probabilistic) key wrapping algorithm that takes as input difficulty level* $d \leq D$ *and outputs key* $k \in \mathcal{K}$ *and wrapped key* w*.*
- $\Sigma.\text{Unwrap}(w) \rightarrow k$*: A deterministic key unwrapping algorithm that takes as input wrapped key* w *and outputs key* $k \in \mathcal{K}$ *or an error* $\perp \notin \mathcal{K}$*.*

Correctness, again, is defined in the natural way: applying Unwrap to a wrapped key w output by Wrap should yield, with certainty, the same key k as originally output by Wrap.

The desirable security property for a keyless key wrap scheme will be indistinguishability of keys: given the wrapped key, can the adversary learn anything about the key within it? The key indistinguishability security experiment $\text{Exp}^{\mathsf{key\text{-}ind}}_{\Sigma,d}$ for an adversary $\mathcal{A}$ trying to break key indistinguishability of a keyless key wrap scheme at difficulty level d is shown in Figure 3.2. We define the advantage of such an adversary in the security experiment as

$$\text{Adv}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{A}) = \left| 2 \cdot \Pr\left[\text{Exp}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{A}) \Rightarrow \texttt{true}\right] - 1 \right|.$$

Γ.Enc(d, msg):	Γ.Dec$((c, w))$:
1. $(k, w) \leftarrow_{\$} \Sigma$.Wrap$(d)$	1. $k' \leftarrow \Sigma$.Unwrap(w)
2. $c \leftarrow_{\$} \Pi$.Enc$(k, msg)$	2. $msg' \leftarrow \Pi$.Dec(k', c')
3. return (c, w)	3. return msg'

FIGURE 3.4: Generic construction of a difficulty-based keyless encryption scheme $\Gamma = \Gamma[\Pi, \Sigma]$ from a difficulty-based keyless key wrap scheme Σ and a symmetric encryption scheme Π

As with DBKE security, useful forms of $\mathrm{Adv}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{A})$ will relate the amount of work done by the adversary, the difficulty level, and the adversary's success probability.

As noted above, we generically construct a difficulty-based keyless encryption scheme by combining a traditional symmetric encryption scheme with a keyless key wrap scheme, as outlined in Figure 3.3. Let Π be a symmetric encryption scheme with key space $\mathcal{K} = \{0,1\}^{\lambda}$, and let Σ be a keyless key wrap scheme for key space $\mathcal{K}$ with maximum difficulty level D. Construct the difficulty-based keyless encryption scheme $\Gamma[\Pi, \Sigma]$ from Π and Σ as outlined in Figure 3.3 and specified in Figure 3.4.

The security proof for DBKE proceeds as a sequence of games. For Game G_i, let S_i denote the event that game G_i outputs true. Let $\mathcal{K}$ be the key space of the symmetric encryption scheme Π, which is also the key space of the keyless key wrap scheme Σ.

First we show in Claim 1 that Game 0 and Game 1 are indistinguishable under the assumption that the key wrapping scheme is secure. Then we argue in Claim 2 that breaking Game 1 corresponds to breaking the indistinguishability of the symmetric key encryption scheme.

Game 0. Denoted G_0, Game 0 as shown in the left side of Figure 3.5 is the db-ind experiment from Figure 3.2 with construction $\Gamma = \Gamma[\Pi, \Sigma]$ inline. Thus,

$$\Pr\left[\mathrm{Exp}^{\mathsf{db\text{-}ind}}_{\Gamma,d}(\mathcal{A}) \Rightarrow \mathtt{true}\right] = \Pr[S_0] \ . \tag{3.1}$$

Game 1. In this game, the challenger generates two symmetric encryption keys k and k'; it uses k in the key wrapping scheme, but k' in the symmetric encryption scheme. This is shown in Game G_1 in the right side of Figure 3.5.

Game G_0:	Game G_1:
1. $(m_0, m_1, st) \leftarrow\!\!\$\ \mathcal{A}(1^d)$	1. $(m_0, m_1, st) \leftarrow\!\!\$\ \mathcal{A}(1^d)$
2. $b \leftarrow\!\!\$\ \{0,1\}$	2. $b \leftarrow\!\!\$\ \{0,1\}$
3. $(k, w) \leftarrow\!\!\$\ \Sigma.\mathrm{Wrap}(d)$	3. $(k, w) \leftarrow\!\!\$\ \Sigma.\mathrm{Wrap}(d)$
4. $c \leftarrow\!\!\$\ \Pi.\mathrm{Enc}(k, m_b)$	4. $k' \leftarrow\!\!\$\ \mathcal{K}$
5. $b' \leftarrow\!\!\$\ \mathcal{A}((c, w), st)$	5. $c \leftarrow\!\!\$\ \Pi.\mathrm{Enc}(k', m_b)$
6. return $(b' = b)$	6. $b' \leftarrow\!\!\$\ \mathcal{A}((c, w), st)$
	7. return $(b' = b)$

FIGURE 3.5: Sequence of games for proof of Theorem 1. Changes between games are highlighted

Claim 1. *Let $\mathcal{B}_1$ be the algorithm shown in Figure 3.6, which is an adversary against the key indistinguishability of keyless key wrap scheme Σ. Then*

$$|\Pr[S_0] - \Pr[S_1]| \leq \mathrm{Adv}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{B}_1^{\mathcal{A}}) \ . \tag{3.2}$$

Proof:

$\mathcal{B}_1$'s input is a challenge (w, k_0, k_1) from a challenger for the key indistinguishability of difficulty-based keyless key wrap scheme Σ. This means that w is the wrapping of either k_0 or k_1, chosen by a random hidden bit b in $\mathrm{Exp}^{\mathsf{key\text{-}ind}}_{\Sigma,d}$. When the hidden bit $b = 0$, and hence when w is the wrapping of k_0, then, in the ciphertext (c, w) that $\mathcal{B}_1$ gives to $\mathcal{A}$, the key used in the key wrapping is the same as the key used in

the symmetric encryption scheme, so $\mathcal{B}_1$ exactly simulates Game 0 to $\mathcal{A}$. When the hidden bit $b = 1$, and hence when w is the wrapping of k_1, then, in the ciphertext (c, w) that $\mathcal{B}_1$ gives to $\mathcal{A}$, the key used in the key wrapping is different from the key used in the symmetric encryption scheme, so $\mathcal{B}_1$ exactly simulates Game 1 to $\mathcal{A}$. Thus, if $\mathcal{A}$ outputs $\hat{b}'$ with different probabilities in Game 0 compared to Game 1, then $\mathcal{B}_1^{\mathcal{A}}$ outputs $\hat{b}'$ with different probabilities when the hidden bit b is 0 or 1. This shows that eq. (3.2) holds. □

Claim 2. *Let $\mathcal{B}_2$ be the algorithm shown in Figure 3.6, which is an adversary against the one-time indistinguishability of symmetric encryption scheme* Π*. Then*

$$\Pr[S_1] \leq \Pr\left[\mathrm{Exp}_{\Pi}^{\mathsf{ind}}(\mathcal{B}_2^{\mathcal{A}}) \Rightarrow \mathtt{true}\right] \quad . \tag{3.3}$$

Proof:

$\mathcal{B}_2^{\mathcal{A}}$ is an adversary in the security experiment $\mathrm{Exp}_{\Pi}^{\mathsf{ind}}(\mathcal{B}_2^{\mathcal{A}})$ for the one-time indistinguishability of symmetric encryption scheme Π. When we inline the code of $\mathcal{B}_2^{\mathcal{A}}$ in $\mathrm{Exp}_{\Pi}^{\mathsf{ind}}(\mathcal{B}_2^{\mathcal{A}})$, we see that it performs the same tasks as Game 1, except some lines are reordered, and some variables are named differently. In particular, $\mathcal{A}$ is run with a ciphertext c that is the encryption of either m_0 or m_1 under the key k from the symmetric encryption experiment, but this key is different from the key $\hat{k}$ that is in the wrapped key w that $\mathcal{A}$ is provided with. Thus, eq. (3.3) holds. □

$\underline{\mathcal{B}_1^{\mathcal{A}}(w, k_0, k_1)}$:	$\underline{\mathcal{B}_2^{\mathcal{A}}()}$:	$\underline{\mathcal{B}_2^{\mathcal{A}}(c, st)}$:
1. $(m_0, m_1, st) \leftarrow_{\$} \mathcal{A}(1^d)$	1. $(m_0, m_1, st) \leftarrow_{\$} \mathcal{A}(1^d)$	1. $(\hat{k}, w) \leftarrow_{\$} \Sigma.\mathrm{Wrap}(d)$
2. $\hat{b} \leftarrow_{\$} \{0, 1\}$	2. return (m_0, m_1, st)	2. $\hat{b}' \leftarrow_{\$} \mathcal{A}((c, w), st)$
3. $c \leftarrow_{\$} \Pi.\mathrm{Enc}(k_0, m_{\hat{b}})$		3. return $\hat{b}'$
4. $\hat{b}' \leftarrow_{\$} \mathcal{A}((c, w), st)$		
5. if $(\hat{b}' = \hat{b})$ return 0		
6. else return 1		

FIGURE 3.6: Reductions for the proof of Theorem 1

Theorem 1 proves that our DBKE scheme Γ is secure in the sense of difficulty-based indistinguishability as shown in Figure 3.2, under the assumption that the building blocks are secure. The proof follows from a straightforward game-hopping argument.

Theorem 1. *If Σ is a key-indistinguishable difficulty-based keyless key wrap scheme, and Π is a one-time indistinguishable symmetric encryption scheme, then $\Gamma[\Pi, \Sigma]$ is a secure difficulty-based keyless encryption scheme. More precisely, let $d \leq D$ and let $\mathcal{A}$ be a probabilistic algorithm. Then there exists algorithms $\mathcal{B}_1$ and $\mathcal{B}_2$, such that $\mathrm{Adv}^{\mathsf{db\text{-}ind}}_{\Gamma,d}(\mathcal{A}) \leq 2 \cdot \mathrm{Adv}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{B}_1^{\mathcal{A}}) + \mathrm{Adv}^{\mathsf{ind}}_{\Pi}(\mathcal{B}_2^{\mathcal{A}})$. Moreover, $\mathcal{B}_1^{\mathcal{A}}$ and $\mathcal{B}_2^{\mathcal{A}}$ have about the same runtime as $\mathcal{A}$.*

Proof of Theorem 1:

By substitution from equations (3.1), (3.2), and (3.3), we get

$$
\begin{aligned}
\mathrm{Adv}^{\mathsf{db\text{-}ind}}_{\Gamma,d}(\mathcal{A}) &= \left| 2 \cdot \Pr\left[\mathrm{Exp}^{\mathsf{db\text{-}ind}}_{\Gamma,d}(\mathcal{A}) \Rightarrow \mathtt{true} \right] - 1 \right| \\
&= |2 \cdot \Pr[S_0] - 1| && \text{(by (3.1))} \\
&= |2 \cdot (\Pr[S_0] - \Pr[S_1] + \Pr[S_1]) - 1| \\
&\leq 2\,|\Pr[S_0] - \Pr[S_1]| + |2 \cdot \Pr[S_1] - 1| \\
&\leq 2\mathrm{Adv}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{B}_1^{\mathcal{A}}) + |2 \cdot \Pr[S_1] - 1| && \text{(by (3.2))} \\
&\leq 2\mathrm{Adv}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{B}_1^{\mathcal{A}}) + \left| 2 \cdot \Pr[\mathrm{Exp}^{\mathsf{ind}}_{\Pi}(\mathcal{B}_2^{\mathcal{A}}) \Rightarrow \mathtt{true}] - 1 \right| && \text{(by (3.3))} \\
&= 2 \cdot \mathrm{Adv}^{\mathsf{key\text{-}ind}}_{\Sigma,d}(\mathcal{B}_1^{\mathcal{A}}) + \mathrm{Adv}^{\mathsf{ind}}_{\Pi}(\mathcal{B}_2^{\mathcal{A}})
\end{aligned}
$$

which is the desired result.

The $\mathcal{B}_1^{\mathcal{A}}$ and $\mathcal{B}_2^{\mathcal{A}}$ algorithms consists of the $\mathcal{A}$'s algorithm plus a minimal cost of either encryption or wrapping. Thus, $\mathcal{B}_1^{\mathcal{A}}$ and $\mathcal{B}_2^{\mathcal{A}}$ runtimes is the runtime of $\mathcal{A}$. $\square$

3.3.2 Hash-Based Construction of Difficulty-Based Keyless Key Wrap

We now show how to construct our difficulty-based keyless key wrap using a hash-based puzzle. The idea is simple: a random seed r is chosen, and the key and a checksum of the seed are derived from the seed using hash functions. The wrapped key consists of the checksum of the seed and the seed with some of its bits removed; the number of bits removed corresponds to the difficulty of the puzzle. This is similar to the sub-puzzle construction of Juels and Brainard [28] or partial inversion proof of work by Jakobsson and Juels [27]. Such a puzzle is solved by trying all possibilities for the missing bits, in any order and with or without using parallelization.

In particular, let $\lambda \in \mathbb{N}$, and let $H_1, H_2 : \{0,1\}^\lambda \to \{0,1\}^\lambda$ be independent hash functions. Define keyless key wrap scheme $P = P[H_1, H_2]$ as in Figure 3.7. The notation $r[\lambda - d : \lambda]$ on line 2 of P.Wrap denotes taking the substring of r corresponding to indices $\lambda - d$ up to λ, removing the first d bits of r.

P.Wrap(d):	P.Unwrap($w = (h, \overline{r})$):
1. $r \leftarrow\!\!\$\ \{0,1\}^\lambda$	1. $d \leftarrow \lambda - \lvert\overline{r}\rvert$
2. $\overline{r} \leftarrow r[\lambda - d : \lambda]$	2. for $i \in \{0,1\}^d$:
3. $h \leftarrow H_1(r)$	3. $\quad r' \leftarrow i \Vert \overline{r}$
4. $k \leftarrow H_2(r)$	4. $\quad h' \leftarrow H_1(r')$
5. $w \leftarrow (h, \overline{r})$	5. $\quad$ if $h' = h$:
6. return (k, w)	6. $\quad\quad k \leftarrow H_2(r')$
	7. $\quad\quad$ return k
	8. return $\perp$

FIGURE 3.7: Keyless key wrapping scheme construction from hash functions H_1, H_2

3.3.3 Security of Hash-Based Keyless Key Wrap Scheme P

The following theorem shows the key indistinguishability security of our hash-based keyless key wrap scheme P in the random oracle model. The proof consists of a query counting argument in the random oracle model.

Theorem 2. *Let H_1 and H_2 be random oracles. Let $\lambda \in \mathbb{N}$ and let $d \leq \lambda$. Let $P = P[H_1, H_2]$ be the keyless key wrap scheme from Figure 3.7 (left). Let $\mathcal{A}$ be an adversary in key indistinguishability experiment against P which makes q_1 and q_2 distinct queries to its H_1 and H_2 random oracles, respectively. Then* $\mathrm{Adv}^{\mathsf{key\text{-}ind}}_{P,d}(\mathcal{A}) \leq \frac{q_1}{2^{d-1}} + \frac{2}{2^d - q_1}$.

Proof of Theorem 2:
Let k_0, k_1, and w be as in $\mathrm{Exp}^{\mathsf{key\text{-}ind}}_{P,d}$ in Figure 3.2 for keyless key wrap scheme P, so that r is the seed behind k_0 and w.

Let W be the event that $\mathrm{Exp}^{\mathsf{key\text{-}ind}}_{P,d}(\mathcal{A})$ outputs `true`. Let E_i be the event that $\mathcal{A}$ queries r to random oracle H_i, for $i = 1, 2$. Our task is to bound $\Pr[W]$, which we do using the following application of the law of total probability:

$$\begin{aligned}\Pr[W] = &\Pr[W|\neg E_2] \cdot \Pr[\neg E_2] \\ &+ \Pr[W|E_2 \wedge E_1] \cdot \Pr[E_2 \wedge E_1] + \Pr[W|E_2 \wedge \neg E_1] \cdot \Pr[E_2 \wedge \neg E_1]\end{aligned}$$

If E_2 does not occur, then, since $k_0 = H_2(r)$, $\mathcal{A}$ has no information about k_0 and thus has no advantage in distinguishing k_0 from k_1, so $\Pr[W|\neg E_2] = \frac{1}{2}$.

Next, we observe that $\Pr[E_2 \wedge E_1] \leq \Pr[E_1]$. The only way $\mathcal{A}$ can learn information about r (and hence k) is by querying values to H_1, and since H_1 is a random oracle,

the adversary can rule out at most one guess for the missing d bits of r with each query to H_1. Thus $\Pr[E_1] \leq \frac{q_1}{2^d}$.

Now we observe that $\Pr[E_2 \wedge \neg E_1] = \Pr[E_2|\neg E_1] \Pr[\neg E_1] \leq \Pr[E_2|\neg E_1]$. Since the q_1 queries to H_1 could have ruled out q_1 candidate values for the missing bits of r, we have that $\Pr[E_2|\neg E_1] \leq \frac{q_2}{2^d - q_1}$. Additionally, we note that, when $E_2 \wedge \neg E_1$ occurs, $\mathcal{A}$ has no information to help it determine which of its q_2 queries to H_2 caused E_2 to occur, so $\Pr[W|E_2 \wedge \neg E_1] = \frac{1}{q_2}$.

Substituting the above observations into the expression for $\Pr[W]$, and bounding all other probabilities by 1, we get

$$\Pr[W] \leq \left(\frac{1}{2} \cdot 1\right) + \left(1 \cdot \frac{q_1}{2^d}\right) + \left(\frac{1}{q_2} \cdot \frac{q_2}{2^d - q_1}\right) = \frac{1}{2} + \frac{q_1}{2^d} + \frac{1}{2^d - q_1} \ ;$$

substituting into the advantage expression yields the desired result. □

Puzzle granularity. The partial pre-image puzzle construction used in Figure 3.7 does not allow for fine-grained control of difficulty: removing each additional bit increases the expected computational cost by a factor of 2. Higher granularity can be achieved similar to how the puzzle difficulty in Bitcoin is set, by giving a hint that narrows the range of data from 2^d to some smaller subset.

3.3.4 Puzzle Degradation

We now introduce an additional feature of difficulty-based keyless encryption that emerges naturally from our hash-based keyless key wrap construction: *puzzle degradation*. Abstractly, puzzle degradation is a process that takes a DBKE ciphertext and

increases the difficulty of decrypting it, preferably without needing to decrypt and then re-encrypt at a higher difficulty level.

In the context of the ArchiveSafe long-term archiving system, this may be used to gradually increase the difficulty of accessing the files that have not been accessed for a certain period of time. For example, a monthly maintenance process could apply degradation to stored files to gradually increase the cost (to both an attacker and an honest party) of accessing increasingly older files.

The DBKE system Δ from Definition 1 is augmented with the algorithm:

- $\Delta.\text{Degrade}(c, d') \xrightarrow{\$} c'$: A (possibly probabilistic) algorithm that takes as input ciphertext c and target difficulty level $d' \leq D$, and outputs updated ciphertext c'.

Correctness is extended to demand that a ciphertext output by Δ.Enc then degraded any number of times is still correctly decrypted by Δ.Dec (although decryption may take longer).

Security with the degraded algorithm included should mean, intuitively, that a ciphertext degraded any number of times can be decrypted only using the required amount of work at the new difficulty level.

We capture both correctness and security of degradation formally by demanding that, for all $d \leq d' \leq D$ and all $msg \in \mathcal{M}$, we have that $\Delta.\text{Enc}(d', msg) \equiv \Delta.\text{Degrade}(d', \Delta.\text{Enc}(d, msg))$; in other words: the distribution of ciphertexts produced by encrypting at difficulty d' is identical to the distribution of ciphertexts produced by encrypting at difficulty d and then degrading to difficulty d'.

$\Gamma[\Pi, P]$.Degrade$(\hat{c}, d')$:
1. parse $\hat{c}$ as $(c, w = (h, \overline{r}))$
2. $d \leftarrow \lambda - |\overline{r}|$
3. abort if $d' < d$
4. $\overline{r}' \leftarrow \overline{r}[d' - d : |\overline{r}|]$
5. $w' \leftarrow (h, \overline{r}')$
6. return (c, w')

FIGURE 3.8: Degradation algorithm for DBKE $\Gamma = \Gamma[\Pi, P]$

We can achieve degradation in DBKE $\Gamma = \Gamma[\Pi, P]$ constructed from our hash-based keyless key wrap P in a trivial way: by removing $(d' - d)$ more bits from the puzzle hint $\overline{r}$. This clearly requires no decryption and re-encryption, only a constant-time edit to the metadata stored containing the wrapped key. The procedure Γ.Degrade is stated in Figure 3.8. Degraded ciphertexts are identically distributed to ciphertexts freshly generated at the target difficulty level, as removing additional bits of the partial seed $\overline{r}$ is associative. An adversary who possess a copy of the metadata from an earlier version of the archive prior to degradation can solve puzzles and decrypt at the earlier, non-degraded difficulty level.

3.3.5 Additional Considerations

Outsourcing Puzzle Solving. The generic DBKE construction Γ of Figure 3.4 allows the key unwrapping and ciphertext decryption to be done separately, so the expensive key unwrapping could be outsourced to a cloud server. In the example of the hash-based keyless key wrap scheme P of Figure 3.7, the user could give the wrapped key $w = (h, \overline{r})$ to the cloud server who unwraps and returns the key k, which the user then locally uses to decrypt the ciphertext c.

This does mean that the cloud server learns the encryption key k. However, this can be avoided with the following adaption to the construction P of Figure 3.7. During

wrapping, the algorithm generates an additional *salt* value $s \leftarrow\!\!\$\ \{0,1\}^{\lambda}$ and computes $k \leftarrow H_2(r\|s)$; s is stored in the wrapped key w. When outsourcing the unwrapping to the cloud server, the user only sends h and $\overline{r}$, but not s. The cloud server is still able to use the checksum h with the partial seed $\overline{r}$ to recover the full seed r, but lacks the salt s and thus the cloud server alone cannot compute the decryption key k. Theorem 2 still applies to this adaptation.

Combining Keyless and Keyed Encryption. As previously mentioned, our keyless encryption approach can (and should) be used in conjunction with traditional keyed encryption mechanisms using a different set of keys. Traditional keyed encryption gives honest parties a (conjecturally exponential) work factor advantage over adversaries if keys remain uncompromised, while keyless encryption slows adversaries if the traditional encryption keys are compromised. The two schemes can be layered in one of two ways: first applying keyless encryption DBKE and encrypting the result using keyed symmetric encryption Sym, that is, $c \leftarrow \text{Sym.Enc}(k, \text{DBKE.Enc}(d, m))$ or in the order, with keyless encryption on the outer layer, that is, $c \leftarrow \text{DBKE.Enc}(d, \text{Sym.Enc}(k, m)))$. Either approach yields robust confidentiality, but we recommend the latter method as it facilitates the puzzle degradation process described in Section 3.3.4.

Hash Function Long-Term Security. Due to the continuous advancement in computation power and cryptanalysis techniques, the hash functions used in ArchiveSafe could become insecure in the future rendering the system vulnerable to security attacks. One way of mitigating this risk is to implement an *evolution* process where the system adopts a new secure hash function to replace the one deemed insecure. The evolution process could be implemented in different ways, either by replacing all the puzzles affected by the insecure hash function at once, or gradually replacing the puzzles

starting with the files containing more sensitive information. The process could also implemented by adopting the new secure hash function for any new file creation or update operations and gradually replace the rest of the files over a preset period of time.

3.4 Evaluation

We evaluate ArchiveSafe by measuring its performance against other systems through real life experiment. The goals of the experiment are to: (1) measure the overhead ArchiveSafe introduces on adversaries and honest users, and (2) verify that puzzle solving difficulty scales according to the theoretical system design.

3.4.1 Prototype Implementation

To run the evaluation experiment, we implemented a prototype of ArchiveSafe. In terms of instantiating the difficulty-based keyless encryption using the generic construction from Section 3.3.1, our proof-of-concept uses AES-128 in CBC mode for the symmetric encryption scheme. The hash functions H_1 and H_2 in the hash-based keyless key wrap scheme are both instantiated with Argon2id [5] with a prefix byte acting as a domain separator between H_1 and H_2, with the following parameters: parallelism level: 8; memory: 102,400 KiB; iterations: 2; output length: 128 bits. We did not parallelize puzzle solving in Unwrap to avoid locking other system operations, but it is easily parallelized.

The ArchiveSafe prototype is implemented as a Linux Filesystem in Userspace (FUSE) using a Python toolkit[1] to simplify implementation. Our Python FUSE driver

[1] `https://github.com/skorokithakis/python-fuse-sample`

relies on the OpenSSL library for encryption and decryption, and Ubuntu's `argon2` package. In a real deployment in the context of a filesystem, ArchiveSafe would be implemented as a kernel module, likely written in C, for improved performance and reliability.

Our prototype has a tuneable difficulty level, which we label in this section as D1, D2, D3, etc. Difficulty Dx corresponds to hash-based keyless key wrap scheme P of Figure 3.7 with difficulty parameter $d = 4x$; in other words, D1 removes 4 bits of the seed, D2 removes 8 bits of the seed, etc. We chose a 4-bit step between difficulty levels to focus on how system behaviour scales across difficulty levels; finer gradations could be chosen by users.

3.4.2 Experimental Setup

The experiment measures ArchiveSafe's performance at three difficulty levels (D1, D2, D3) compared to an unencrypted file system (denoted UN) and Linux's built-in folder encryption using eCryptfs[2] (denoted FE) and disk encryption (denoted DE) on read and write tasks at different file sizes. When running the ArchiveSafe experiments, the ArchiveSafe FUSE driver was writing its files to an unencrypted file system.

Measurements. For each storage system being evaluated, we measure read and write times for files of sizes 1 KB, 100 KB, 1 MB, 10 MB, and 100 MB. Performance is measured at the application level, from the time the file is opened until the time the read/write operation is completed. For folder and disk encryption, this includes the filesystem's encryption operations. For ArchiveSafe, we instrumented the driver to

[2]https://www.ecryptfs.org/

record the total time as well as the times for different sub-tasks (encryption, puzzle solving, decryption, file system I/O).

Test environment. Measurements were performed on a single-user Linux machine with no other processes running. The computer was a MacBook Pro running Ubuntu Linux 18.04 LTS with an 4-core Intel Core i7-4770HQ processor with base frequency 2.2 GHz, bursting to 3.4 GHz. The computer had 16 GiB of RAM. The hard drive was a 256 GiB solid state drive with 512-byte logical sectors and 4096-byte physical sectors. The disk encryption was done using Linux Unified Key Setup system version 2.0, and folder encryption was done using the Enterprise Cryptographic Filesystem (eCryptfs) version 5.3.

Execution. For each storage system and file size, we performed many repetitions of the following tasks. A file was created with randomly generated alphanumeric characters using a non-cryptographic random number generator, the time for this step is discarded. Read and write operations were measured as indicated above. For file sizes of 1 KB, 100 KB, 1 MB, and 10 MB, we collected data for 1000 writes and reads; for 100 MB files, we ran 200 writes and reads, due to extensive time of operations at this size.

3.4.3 Results

Table 3.1 and Table 3.2 show average read and write times respectively for the file systems under consideration at different file sizes. Since read operations in the ArchiveSafe system become increasingly expensive with difficulty, we show in Table 3.3 the average time of sub-tasks of ArchiveSafe read operations at different file sizes and difficulties: the puzzle solving time (which should scale with puzzle difficulty), the

File system	File Size				
	1 KB	100 KB	1 MB	10 MB	100 MB
Unencrypted (UN)	0.526	0.550	1.70	10.1	110
Disk Encryption (DE)	0.737	0.924	3.15	10.5	160
Folder Encryption (FE)	0.737	0.961	3.42	10.9	190
ArchiveSafe D1	630	630	630	650	860
ArchiveSafe D2	7070	7080	7310	7180	7290
ArchiveSafe D3	112140	111760	107390	114530	107630

Table 3.1: Average read time comparison in milliseconds

File system	File Size				
	1 KB	100 KB	1 MB	10 MB	100 MB
Unencrypted (UN)	0.07	0.25	0.83	6.76	97.82
Disk Encryption (DE)	0.08	0.25	0.85	6.63	97.97
Folder Encryption (FE)	0.12	0.50	3.31	29.07	319.88
ArchiveSafe D1	114.05	141.67	146.09	221.73	848.30
ArchiveSafe D2	114.25	141.43	145.08	223.50	847.02
ArchiveSafe D3	114.01	140.98	145.74	222.40	846.06

Table 3.2: Average write time comparison in milliseconds

system file read time plus decryption time (which should scale with file size), and the overhead from other file system driver operations (which includes puzzle read and system file open times). As the partial pre-image puzzle used in ArchiveSafe leads to highly variable solving times, Table 3.4 shows the average time and standard deviation for puzzle solving at difficulties D1, D2, and D3.

3.4.4 Discussion

The results show consistent behavior across different file sizes. The larger files consumed more time in decrypting and reading. We also observed that the time consumed is

Diff.		1 KB	100 KB	1 MB	10 MB	100 MB
D1	Puzzle Solve	510	510	510	510	500
	Decryption	5.42	5.71	7.25	20	150
	Other	0.387	0.373	0.378	0.384	0.363
D2	Puzzle Solve	6960	6980	7210	7050	6930
	Decryption	5.58	6.12	7.89	20	140
	Other	0.357	0.373	0.376	0.374	0.335
D3	Puzzle Solve	112040	111730	107280	114410	107270
	Decryption	5.56	5.94	7.96	20	140
	Other	1.075	1.216	0.971	1.195	1.045

TABLE 3.3: ArchiveSafe read sub-tasks average time
Read sub-tasks average time in milliseconds

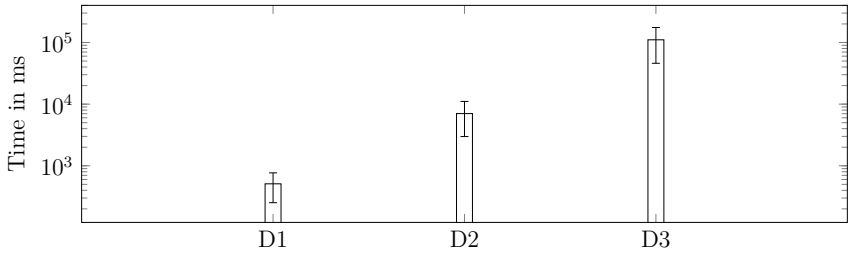

TABLE 3.4: ArchiveSafe Puzzle solving time
Puzzle solving time in milliseconds (average, standard deviation)

roughly the same for smaller file sizes (1 KB and 100 KB) where operation cost is dominated by overhead.

As expected, the read speeds decrease with the difficulty level because the system must solve the puzzle before reading the file and the puzzle solving effort scales with the difficulty level. As per Table 3.3, puzzle solve times on average scale by a factor of 13.6–14.1× between D1 and D2 and a factor of 14.9–16.2× between D2 and D3, roughly in line with the theoretical scaling factor of 16×.

Evaluating the overhead added by ArchiveSafe for write operations, we see in Table 3.1 that ArchiveSafe incurs a baseline overhead related to setting up the puzzle

(which involves 2 Argon2 calls), then scales with the file size due to the cost of AES encryption and writing. Note that ArchiveSafe uses a different encryption library (user-space calls to OpenSSL) compared with disk and file encryption (kernel encryption via dm-crypt), so symmetric encryption/decryption performance is not directly comparable, but we see similar scaling.

The short summary of performance is that ArchiveSafe adds a 140–520 ms overhead when writing a file, and a customizable overhead when reading a file, ranging from 510 ms at difficulty D1, 7 seconds at D2, or 110 seconds at D3. But recall that adding computational overhead at read time is exactly the purpose of ArchiveSafe! What an acceptable difficulty level—and hence acceptable computational overhead at read time for honest users—is a policy choice by the system administrator. As noted earlier, choosing the difficulty level depends on the tolerable cost for honest users to access data, the perceived risk of a data breach, and the anticipated value of the information to an adversary, and is a calculation that must be left to the adopter. Note that honest users need not solely rely on sequential operations on their own computer: as described in Section 3.3.5 an ArchiveSafe installation could be configured so that honest users offload their puzzle solving tasks to private or commercial clouds which are spun up on demand with large amounts of parallelization to reduce the wall clock time before they can access a file.

Table 3.5 shows examples of costs at higher difficulty levels. To provide further interpretation to these costs, we look not only at the computation time required for an honest user on our test platform to decrypt a file, but also at the real-world cost for an adversary, based on the cost of renting computation time on Amazon Web Services (AWS) Elastic Cloud Compute (EC2) platform. EC2 has many machine types available; Argon2 is designed to not be substantially accelerated by more sophisticated

	D3	D4	D5	D6
Honest user decrypting 1 file				
Local machine, threaded 4 cores, 2.2 GHz	0.5 min.	7.3 min.	2 hrs.	31 hrs.
Cloud server `c5.metal`, spot pricing	$\ll$\$0.01	$<$\$0.01	\$0.05	\$0.73
Adversary decrypting 1 million files				
Cloud server `c5.metal`	8 days	130 days	5.7 yrs.	91.4 yrs.
Cloud server `c5.metal`, spot pricing	\$178	\$2,852	\$45,648	\$730,364

TABLE 3.5: Dollar cost and computation time required to unlock ArchiveSafe files

architectures, GPUs, or ASICs. As such we choose for our pricing example an EC2 instance that minimizes cost per core-GHz-hour; the `c5.metal` EC2 instance type has 96 Intel Xeon cores running at 3.6 GHz at a cost of USD\$0.9122 per hour using Amazon's cheapest spot pricing model.[3]

We can see, for example, that at difficulty D5, an honest user can unlock an archived file with about 2 hours of work on a local machine, or about 3 minutes of `c5.metal` rental costing 4.5 cents at spot pricing (20 cents on-demand pricing). However, an adversary trying to decrypt 1 million such files from a data breach would need 5.7 years of `c5.metal` rental at a spot pricing cost of USD\$45,648.

3.5 Use Cases

ArchiveSafe could be used alone or in conjunction with a standard secure archiving system. In this use case, the files will be secured by ArchiveSafe first before being encrypted and stored by the secure archiving system. Retrieving the files is done by first decrypting them by the archiving system then have their puzzles solved by ArchiveSafe before passing them to the requesting application.

[3] `https://aws.amazon.com/ec2/instance-types/`,`https://aws.amazon.com/ec2/spot/pricing/`; prices as of April 23, 2020.

Another use case for ArchiveSafe is to have it as a part of an operating system's kernel. In this use case, non-data files such as executables and system files must be excluded from ArchiveSafe protection by setting their difficulty level to 0. Files containing sensitive information must be secured with a more difficult puzzles than other files. The system will have to maintain a *mapping* table linking each file to its puzzle difficulty level.

3.6 Summary

ArchiveSafe, using difficulty-based keyless encryption, can add defense-in-depth to confidentiality of archived data and change the economics of mass leakage attacks via data breaches. We expect that most uses of ArchiveSafe would be in addition to, not as a replacement for, traditional keyed encryption; full cryptographic security would be achieved if encryption keys are properly managed and kept safe, but ArchiveSafe provides a residual level of protection if traditional encryption keys are also breached. This means the key management service is no longer a single point of failure.

One target application is IT systems which retain large amounts of archival data, most of which will be rarely or perhaps never again accessed by legitimate users. Although honest users have no advantage in difficulty-based decryption compared to an adversary on a file-by-file basis, if their operational goals are different—an honest user decrypting 1 file occasionally, versus an adversary decrypting thousands or millions of files quickly—their costs are different.

Our approach can be applied in a variety of system architectures: local storage and execution (as demonstrated by our prototype), local storage with private or public cloud assistance for puzzle solving, or remote (file server / cloud) storage with local or

assisted puzzle solving. Our approach can also apply to different storage paradigms, including file systems, cloud "blob" storage, and databases.

Puzzle difficulty can be set as a system-wide policy or with higher granularity based individual records' sensitivity. A novel features of our construction is the ability to degrade puzzle difficulty effectively for free, which could be built into periodic maintenance or through a heuristic system based on suspicious activity.

In the next chapter, we will study in-depth the challenges facing the adoption of current state-of-the-art long-term secure archiving systems and propose a solution if possible.

Chapter 4

ArchiveSafe LT: Secure Long-term Archiving System

In this chapter, we introduce ArchiveSafe LT, a framework for secure long-term digital archiving systems that provide long-term confidentiality and integrity protection. It uses a different approach utilizing robust combiners [42] [26] for confidentiality and multiple secure integrity schemes based on authenticated data structures to ensure the integrity of the archive in case a scheme is compromised.

ArchiveSafe LT provides long-term confidentiality and integrity that is practically feasible to implement due to its low setup cost and superior performance compared to the other systems based on the secret-sharing approaches such as LINCOS [8], PROPYLA [23], ELSA [35] and SAFE [10]. We eliminate the need for private channels and significantly decrease the required storage space by using robust combiners rather than information-theoretic secret sharing. ArchiveSafe LT adopts a cryptographic agile approach. The approach is built on the idea of utilizing a pool of secure symmetric and asymmetric encryption schemes and hashing functions to do multi-layer data

encryption and integrity verification. If a cryptographic component is deemed insecure, the system initiates an evolution process that wraps the secured data and related integrity information in an additional secure layer of encryption or integrity protection. ArchiveSafe LT decreases the time required for key generation from days in the case of the QKD key generation to seconds by utilizing standard symmetric and asymmetric key generation methods.

As a digital archiving framework, ArchiveSafe LT considers two main actors, a data collector and a storage provider, in addition to the adversaries. Similar to current archiving systems, we assume the storage provider is cloud-based. However, trust in the storage provider is not presumed. The archive will stay secure even if the storage provider exhibits adversarial behavior.

We present two designs based on the ArchiveSafe LT framework covering two types of storage providers, trusted and untrusted non-malicious. In the first design ASLT-D1, we assume the storage provider either cannot be trusted with performing the evolution process or is incapable of performing data processing. This design utilizes symmetric encryption schemes and the data collector performs all processing needed for all archiving operations. The second design ASLT-D2 assumes the storage provider can be trusted to perform the evolution process and is capable of performing data processing. The storage provider is trusted to have at most one key and for a short period of time, it happens during the evolution process. At this time, and aside from the scheme that the provider has its key, the archive is still protected by at least one more secure scheme, so the provider cannot access the archive's plaintext. ASLT-D2 utilizes hybrid encryption schemes where the data collector securely sends the information needed for the evolution process to the storage provider who performs the process.

We develop protocols for the archiving processes carried out by ArchiveSafe LT. We analyze the security of ArchiveSafe LT using an automatic prover to verify its confidentiality and integrity. We design an experiment to analyze and evaluate the performance of one of two designs, ASLT-D1. The second design ASLT-D2 is more performant by design since most of the processing is offloaded to the storage provider.

We start by presenting a background on robust combiners in the next section then we present the ArchiveSafe LT framework in Section 4.2, and the two designs ASLT-D1 and ASLT-D2 are presented in Section 4.3. We present the system evaluation in Section 4.4 and finally the summary and future work in Section 4.5.

4.1 Robust Combiners

A robust combiner combines multiple cryptographic schemes into one so the resulting scheme is robust to the failure of any of the combined ones. Rather than using information-theoretically secure schemes as used in LINCOS, ELSA, PROPYLA and SAFE, one can utilize robust combiners to reduce the risk of relying on any single computational assumption.

A (k, n) robust combiner for a cryptographic primitives is a construction where combining n primitives is secure if at least k primitives are secure. Robust combiners are secure against IND-CCA[16].

Definition 3 (Robust Combiner(k,n)). *A (k, n) robust combiner for a cryptographic primitive P is a probabilistic polynomial time machine that gets n candidate schemes as input and implements P such that:*

- *The combiner is secure if at least k of the candidates securely implement P.*

- *The combiner running time is polynomial in a security parameter n.*

Robust combiners are not new; Shannon introduced the idea of encrypting a message multiple times using different keys to increase confidentiality protection [42]. Herzberg studied robust combiners [26] and how they can form a tolerant cryptographic scheme that remains secure even if some subsets of its cryptographic components assumptions underlying its security become invalid. Robust combiners for symmetric schemes were studied by Even and Goldreich [20] to answer the question of whether cascading any cipher systems yields a stronger cipher system or not. They proved cascading ciphers yield stronger cipher systems in certain cases where compromising a scheme does not lead to compromising the rest which is the case with robust combiners.

4.2 ArchiveSafe LT Framework

ArchiveSafe LT is a framework for archiving applications providing long-term confidentiality and integrity using robust combiners and Merkle trees respectively. The framework considers two main actors, the *data collector* and the *storage provider*. The data collector is able to do local processing but has limited local storage space. The storage provider provides storage space in addition to optional processing capability that could be utilized by the data collector to offload some of the data processing needed for the encryption, decryption or hashing. We study ArchiveSafe LT in the presence of adversaries who are capable of controlling the network, controlling the storage provider's environment or both. The adversaries' powers are detailed in Section 4.2.3.

4.2.1 Protocols

The ArchiveSafe LT framework divides archiving systems functionality into six main protocols, *Initialize*, *Update*, *Retrieve*, *Delete*, *Evolve Confidentiality*, and *Evolve Integrity*. In this section, we present what these protocols are meant to accomplish.

Initialize is a protocol initiated by the data collector to create a new archive. The data collector starts by identifying the data to be archived and then initializing the archive on its own system. The data collector sends a request containing the archive's metadata (such as, size and encoding) to the storage provider to initiate the archive on its side as well. The storage provider sends an acceptance response if the accommodation is confirmed. Otherwise, the request fails.

Update is a protocol initiated by the data collector when the contents of an archive file on the data collector's system have changed from the ones securely stored on the storage provider's system. The protocol starts with the data collector sending a request containing the contents of the changed file and its metadata to the storage provider to update the file's contents and its local state with the new associated integrity information. The storage provider sends an acceptance response if the accommodation is confirmed. Otherwise, the request fails. If the request succeeds, the data collector updates the local state of its archive with the new integrity information associated with the changed file.

Retrieve is a protocol initiated by the data collector when they need to retrieve the contents of an archive file. The protocol starts with the data collector sending a request to the storage provider containing the identifier for the needed file. If the storage provider can accommodate the request, it sends the file contents to the data collector

along with its associated integrity information. Otherwise, the request fails. The data collector verifies the integrity of the received file using the integrity information received from the storage provider and the corresponding integrity information stored in its local state before accepting the file.

Delete is a protocol initiated by the data collector when they need to delete a file from the archive. It starts with the data collector identifying the file to be deleted. The data collector's system sends a request containing the file identifier to the storage provider. The storage provider sends an acceptance response if the accommodation is confirmed. Otherwise, the request fails. If the storage provider can accommodate the request, it deletes the file and updates the archive's integrity information. The storage provider sends the new integrity information to the data collector to update its integrity information as well.

Evolve Integrity is a protocol initiated by the data collector when they flag an integrity scheme or a key, as insecure. The protocol starts with the data collector identifying and flagging the insecure component. The data collector's system selects a replacement for the flagged component from a local pool. The replacement component is then used to secure the integrity information of the archive. The integrity information and local states on both the data collector and storage provider sides are updated accordingly.

Evolve Confidentiality is a protocol initiated by the data collector when they flag a confidentiality component such as an encryption scheme or a key, as insecure. The protocol starts with the data collector identifying the insecure component. The data collector's system selects a replacement for the flagged component from a local pool. The replacement component is then used to secure the archive. The integrity

information is updated to reflect the new secured data, and local states on both the data collector and storage provider sides are updated accordingly.

4.2.2 ArchiveSafe LT Specifications

ArchiveSafe LT-based systems consist of a set of cryptographic suites $\mathcal{E}$, an integrity object I and a pair of local states for each archive: a local state on the data collector's system and another on the storage provider's system.

Notation: The notation $\langle x\ ;\ y\rangle \leftarrow P(\langle u\ ;\ v\rangle)$ denotes the execution of two-party protocol P, where the first party has input u and receives output x, and the second party has input v and receives output y. Additionally, we use the parameters with a subscript o or p to indicate whether the source or destination of the parameter is the data collector or the storage provider, respectively.

An ArchiveSafe LT-based system consists of the following components:

- $\mathcal{E} = \{\S_1, \ldots, \S_n\}$: A set of cryptographic suites, where at any point in time at least two of them are deemed secure.
- $LS = (ls_o, ls_p)$: A pair of local states containing confidentiality and integrity details of the archive, where ls_o represents the data collector's version and ls_p the storage provider's version, respectively.
- $I = (I_v, I_d)$: Integrity information corresponding to the secured data is stored in I_d, typically stored on the storage provider's side and used by the data consumer to request integrity verification of the data. I_v contains the integrity verification values for the secured data, it is usually stored on the data collector's side and used to verify the data using I_d. I is part of the local state LS.

Generally, a cryptographic suite § supports four operations, `KeyGen`, `LockConf`, `LockInt` and `Unlock`. Both `LockConf`, `LockInt` and `Unlock` utilize symmetric encryption and MAC schemes (Section 2.1.2) together. The operations are defined as follows:

- $(k_C, k_I) \leftarrow$ §.`KeyGen`(): generates a key pair that is used by the cryptographic suite in locking and unlocking data.
- $(SecuredData) \leftarrow$ §.`LockConf`$(k_C, \Pi, Data)$: secures the confidentiality of a data object using a key k_C and a symmetric encryption scheme Π. It outputs the secured data.
- $(I') \leftarrow$ §.`LockInt`$(k_I, \Delta, Data, I)$: secures the integrity of a data object using a key k_I and a MAC scheme Δ. It outputs the updated integrity data object I'.
- $Data \leftarrow$ §.`Unlock`$(k_C, \Pi, SecuredData)$: retrieves a secured data object to its original state using the key k_C and scheme Π. The operation returns an error if it fails for any reason.

The ArchiveSafe LT framework utilizes the operations described above in its APIs, which we present next. We start by presenting the parameters used by the APIs:

- *fc*: The contents of the file to be updated or retrieved.
- *fn*: The name of the file to be updated or retrieved.
- *arch*: The stored archive in its secured state.
- ls_o: The data collector's local state of the archive.
- ls_p: The storage provider's local state of the archive.
- $algs_{old}$: A set of cryptographic schemes used to secure the archive's confidentiality and integrity that has been deemed insecure.

- $algs_{new}$: A set of cryptographic schemes used to secure the archive's confidentiality and integrity that is deemed secure and to be used in the evolution process.
- $algs_{cur}$: A set of cryptographic schemes currently deemed secure and are used by the ArchiveSafe LT implementation.
- $algs_{dep}$: A set of cryptographic schemes currently deemed insecure and are not used by the ArchiveSafe LT implementation.

We now present the APIs:

- $\langle ls_o \ ; \ ls_p, arch \rangle \leftarrow$ `Initialize`(): This API is called to perform the initialization tasks on all parties. It takes no input parameters. The output consists of the created stored archive *arch*, the storage provider's local state ls_p, and the data collector's local state ls_o.
- $\langle ls'_o \ ; \ ls'_p, arch' \rangle \leftarrow$ `Update`$(\langle ls_o, fn, fc \ ; \ ls_p, arch \rangle)$: This API is called to perform the update task on an archive. It takes the local state ls_o and the file name *fn* to be updated along with the new content *fc* from the data collector, the local state ls_p from the storage provider, and the archive *arch*. The API's output is the updated archive $arch'$, the updated storage provider's local state ls'_p, and the updated data collector's local state ls'_o.
- $\langle fc \ ; \ \rangle \leftarrow$ `Retrieve`$(\langle ls_o, fn \ ; \ ls_p, arch \rangle)$: This API is called to perform the tasks to retrieve a file in the archive to its original plain data state. It takes from the data collector the file's name to be retrieved *fn*, the currently stored archive *arch* from the storage provider and the local states from both sides ls_o, ls_p. It outputs the retrieved file contents *fc* to the data collector.
- $\langle ls'_o \ ; \ ls'_p, arch' \rangle \leftarrow$ `Delete`$(\langle ls_o, fn \ ; \ ls_p, arch \rangle)$: This API is called to perform the

tasks required to delete a file from the archive. It takes the file name to be deleted *fn* and the local state ls_o as input from the data collector and the storage provider's local state ls_p. The outputs are the updated local state to the data collector ls'_o, the local state ls'_p and the updated archive *arch*' to the storage provider.

- $\langle ls'_o \ ; \ ls'_p \rangle \leftarrow$ `EvolveInt`$(\langle ls_o, algs_{old}, algs_{new} \ ; \ ls_p, arch \rangle)$: This API is called to perform all the tasks required to evolve the archive's integrity to eliminate the risk of the compromised cryptographic suite. The API takes as input from the data collector: the compromised cryptographic suite $algs_{old}$ and its secure replacement $algs_{new}$, and the local state ls_o. The API takes as input from the storage provider: the stored archive *arch* and local state ls_p. The output consists of the updated local state ls'_o to the data collector and updated local state ls'_p to the storage provider.
- $\langle ls'_o \ ; \ ls'_p, arch' \rangle \leftarrow$ `EvolveConf`$(\langle ls_o, algs_{old}, algs_{new} \ ; \ ls_p, arch \rangle)$: This API is called to perform all the tasks required to evolve the archive's confidentiality to eliminate the risk of the compromised cryptographic suite. The API takes as input from the data collector: the compromised cryptographic suite $algs_{old}$ and its secure replacement $algs_{new}$, and the local state ls_o. The API takes as input from the storage provider: the stored archive *arch* and local state ls_p. The output consists of the updated local state ls'_o to the data collector, the evolved stored archive $arch'$ and updated local states ls'_p to the storage provider.

4.2.3 Threat Model

To build our threat model, we start by defining the powers available to the adversaries, followed by the construction of the adversaries in terms of these powers. An adversary may have one or more of these powers. The powers are:

P_1 - *Compromise a Scheme or Keys:* This power enables an adversary: 1) to obtain

a set of keys that are used to secure the archive, 2) to be able to deterministically decrypt any ciphertext produced by a scheme actively used by the archiving system, and 3) to alter the contents of an archive while still passing integrity verification of an integrity scheme actively used by the archiving system. The collection of all the compromised keys and schemes granted by this power cannot comprise the entirety of the two most recent $Lock()$ processes.

P_2 - *Control Communication Channel:* This power enables an adversary to control the communication channel between the data collector and the storage provider, listening and writing messages to the channel for both entities.

P_3 - *Control Storage Provider:* This power enables an adversary to control the storage provider's environment, including being able to read, write and alter files on the storage.

Our threat model considers the adversary to be active, computationally bound and has P_1, P_2, and P_3. The goal of the adversary is either to obtain the contents of the whole content of the archive or parts of it in clear text form or to alter the contents of the archive without the data collector detecting it through integrity checks. The model is chosen to address two common real-life attack scenarios on archiving systems and their adversarial goals: obtaining an archive's data or altering it.

4.2.4 Limitations

ArchiveSafe LT does not cover the case of an adversary aiming to obtain the plaintext, who is active and have access to unevolved archives for a long enough time until they can compromise all the encryption schemes.

Protecting the archive and files metadata and files access patterns is outside of the scope of this work.

4.2.5 Security

In this section, we define the security experiments for confidentiality and integrity to formalize our threat models described in Section 4.2.3.

Confidentiality: The security experiment for confidentiality shown in Figure 4.1, is based on the adversary's ability to distinguish between two update operations applied to the same archive. We define the advantage of such adversary in the security experiment using an ArchiveSafe LT-based system scheme S as

$$\mathrm{Adv}_{S}^{\mathsf{ind}}(\mathcal{A}) = \Pr\left[\mathrm{Exp}_{S}^{\mathsf{ind}}(\mathcal{A}) \Rightarrow 1\right] - \frac{1}{2}$$

The term $\mathrm{Adv}_{S}^{\mathsf{ind}}(\mathcal{A})$ represents the indistinguishability ind advantage of the adversary $\mathcal{A}$ given the scheme S. This advantage is defined by the probability of the experiment to output 1 -indicating the success of the adversary- more than 50% of the times.

The confidentiality experiment takes n cryptographic schemes $\S_1, \ldots, \S_n$ as input. The archive is initialized and updated with a file fn^* provided by the adversary and then secured by n cryptographic suites. The adversary generates two different updates m_0 and m_1 to the fn^* contents. The experiment randomly picks one of the updates to be executed and updates the archive state st. The adversary succeeds if they can correctly guess which update was executed.

$\text{Exp}^{\mathsf{ind}}_{n,\S_1,\ldots,\S_n}(\mathcal{A})$

1 : $\langle ls_o \ ; \ ls_p, arch\rangle \leftarrow \texttt{Initialize}()$
2 : $(m, st) \leftarrow \mathcal{A}^{\mathsf{OAll}}()$
3 : $\langle ls_o \ ; \ ls_p, arch\rangle \leftarrow \texttt{Update}(\langle ls_o, fn^*, m \ ; \ ls_p, arch\rangle)$
4 : **for** $i = 1, \ldots, n$
5 : $\quad \langle ls_o \ ; \ ls_p, arch\rangle \leftarrow \texttt{EvolveConf}(\langle ls_o, algs_{old}, algs_{new} \ ; \ ls_p, arch\rangle)$
6 : $algs_{dep} \leftarrow \emptyset$
7 : $corr \leftarrow \emptyset$
8 : $(fn^*, m_0, m_1, st) \leftarrow \mathcal{A}^{\mathsf{OAll}}()$
9 : $b \leftarrow\!\!\$ \ \{0, 1\}$
10 : $\langle ls_o \ ; \ ls_p, arch\rangle \leftarrow \texttt{Update}(\langle ls_o, fn^*, m_b \ ; \ ls_p, arch\rangle)$
11 : $algs^* \leftarrow algs_{cur}$
12 : $b' \leftarrow \mathcal{A}^{\mathsf{OAll}}(st, arch)$
13 : **if** $(|m_0| \neq |m_1|) \vee (algs^* \subseteq corr)$ **then**
14 : $\quad r \leftarrow\!\!\$ \ \{0, 1\}$; **return** r
15 : **else return** 1 **if** $b' = b$ **else** 0

FIGURE 4.1: Confidentiality experiment for indistinguishability of ArchiveSafe LT
*OAll represents access to all oracles

For the security experiments, we provide the adversary with the following oracles shown in Figure 4.2:

- OUpdate(fn, fc): The oracle allows the adversary to request to update a file fn in the archive with new content fc.
- ORetrieve(fn): The oracle allows the adversary to request the retrieval of file fn from the archive in its original insecure state.
- ODelete(fn): Allows the adversary to delete a file fn from the archive.
- OEvolveInt($algs_{old}$): Allows the adversary to initiate an evolution on the archive for a compromised integrity component. The oracle takes as input, the set of cryptographic schemes used to secure the archive's confidentiality and integrity

that has been deemed insecure $algs_{old}$.

- OEvolveConf($algs_{old}$): Allows the adversary to initiate an evolution on the archive for a compromised confidentiality component. The oracle takes as input, the set of cryptographic schemes used to secure the archive's confidentiality and integrity that has been deemed insecure $algs_{old}$.
- OCorruptKey(i): Allows the adversary to obtain the encryption key used by scheme i.
- OGetArchive(): Allows the adversary to obtain a copy of the stored secured archive.

OUpdate(fn, fc)

if $fn = fn^*$ **then** $fn^* \leftarrow \perp$
$\langle ls_o \ ; \ ls_p, arch \rangle \leftarrow$
$\quad$ `Update`$(\langle ls_o, fn, fc \ ; \ ls_p, arch \rangle)$
$Files \leftarrow (fn, fc)$
return $\perp$

ORetrieve(fn)

$\langle fc \ ; \ \rangle \leftarrow$ `Retrieve`$(\langle ls_o, fn \ ; \ ls_p, arch \rangle)$
if $fn = fn^*$ **then return** $\perp$
else return fc

ODelete(fn)

$\langle ls_o \ ; \ ls_p, arch \rangle \leftarrow$
$\quad$ `Delete`$(\langle ls_o, fn \ ; \ ls_p, arch \rangle)$
return $\perp$

OCorruptKey(i)

$corr \leftarrow corr \cup \{i\}$
return $(k_o^{(i)}, k_p^{(i)})$

OEvolveConf($algs_{old}$)

$\langle ls_o \ ; \ ls_p, arch \rangle \leftarrow$
$\quad$ `EvolveConf`$(\langle ls_o, algs_{old}, algs_{new} \ ; \ ls_p, arch \rangle)$
$algs_{dep} \leftarrow algs_{dep} \cup algs_{old}$
$algs_{cur} \leftarrow (algs_{cur} \cup algs_{new}) \setminus algs_{dep}$
return $\perp$

OEvolveInt($algs_{old}$)

$algs \leftarrow algs_{dep}, algs_{old}, algs_{cur}, algs_{new}$
$\langle ls_o \ ; \ ls_p, arch \rangle \leftarrow$
$\quad$ `EvolveInt`$(\langle ls_o, algs_{old}, algs_{new} \ ; \ ls_p, arch \rangle)$
$algs_{dep} \leftarrow algs_{dep} \cup algs_{old}$
$algs_{cur} \leftarrow (algs_{cur} \cup algs_{new}) \setminus algs_{dep}$
return $\perp$

OGetArchive()

return $arch$

FIGURE 4.2: ArchiveSafe LT oracles available to the Adversary

In this experiment, OAll = ($\mathsf{OUpdate}$, $\mathsf{ORetrieve}$, $\mathsf{ODelete}$, $\mathsf{OEvolveInt}$, $\mathsf{OEvolveConf}$, $\mathsf{OCorruptKey}$, $\mathsf{OGetArchive}$). In some of the oracles, we use the notation fn^* to represent the name of the file used to challenge the adversary.

Based on our threat model defined in Section 4.2.3, the adversary's goal is to obtain the contents of the archive in clear text format. The experiment is designed to check the adversary's ability to achieve their goal. The experiment is not designed to check the ability of the adversary to obtain the archive's metadata or access patterns.

Integrity: The security experiment for integrity shown in Figure 4.3, is based on the adversary's ability to change the contents of one or more files in the archive while keeping the stored local state valid. We define the advantage of such an adversary in the security experiment using an ArchiveSafe LT scheme S as:

$$\mathrm{Adv}_S^{\mathsf{forge}}(\mathcal{A}) = \Pr\left[\mathrm{Exp}_S^{\mathsf{forge}}(\mathcal{A}) \Rightarrow 1\right]$$

The term $\mathrm{Adv}_S^{\mathsf{forge}}(\mathcal{A})$ represents the forging advantage of the adversary $\mathcal{A}$. The advantage is defined as the probability of the experiment to output a 1, indicating the success of the adversary.

The experiment takes n cryptographic schemes $\S_1, \ldots, \S_n$ as input. The archive is initialized and updated with a file fn^* provided by the adversary and then secured by n cryptographic suites. The adversary obtains the stored archive and forges it by changing the contents of file fn^* using the provided oracles. The adversary succeeds if they can successfully change the contents of fn^* while still matching the stored local states to complete the retrieve process.

$\text{Exp}^{\mathsf{forge}}_{n,\S_1,\ldots,\S_n}(\mathcal{A})$

1 : $\langle ls_o \ ; \ ls_p, arch\rangle \leftarrow \texttt{Initialize}()$
2 : $(fc^*, st) \leftarrow \mathcal{A}^{\mathsf{OAll}}()$
3 : $\langle ls_o \ ; \ ls_p, arch\rangle \leftarrow \texttt{Update}(\langle ls_o, fn^*, fc^* \ ; \ ls_p, arch\rangle)$
4 : **for** $i = 1, \ldots, n$
5 : $\quad \langle ls_o \ ; \ ls_p\rangle \leftarrow \texttt{EvolveInt}(\langle ls_o, algs_{old}, algs_{new} \ ; \ ls_p, arch\rangle)$
6 : **endfor**
7 : $algs_{dep} \leftarrow \emptyset$
8 : $corr \leftarrow \emptyset$
9 : $arch' \leftarrow \mathcal{A}^{\mathsf{OAll}'}(st, arch)$
10 : $algs^* \leftarrow algs_{cur}$
11 : $fc^{*'} \leftarrow \texttt{Retrieve}(\langle ls_o, fn^* \ ; \ ls_p, arch'\rangle)$
12 : **if** $algs^* \subseteq corr$ **then**
13 : $\quad$ **return** 0
14 : **else return** 1 **if** $fc^{*'} \neq fc^*$**else** 0

FIGURE 4.3: ArchiveSafe LT Integrity Experiment

4.3 System Designs

In this section, we present two system designs based on the ArchiveSafe LT framework to cover the two types of storage providers, trusted and untrusted. The first design $ASLT-D1$ utilizes a non-malicious storage provider who either cannot be trusted with performing the evolution process or is incapable of performing complex data processing operations such as encrypting large data files. The second design $ASLT-D2$ utilizes a storage provider who is both capable of doing complex data processing operations and trusted to perform the evolution processes. In both designs, we utilize cascade combiners, which is a sequential application of two or more cryptographic schemes. This combiner is robust for block ciphers against message recovery attacks [20].

In both designs, the data collector is responsible for all the processing required in initially securing the archive and retrieving it. However, these two designs are

different in taking ownership of performing the evolution process. In $ASLT - D1$, the data collector performs the evolution process work; however, in $ASLT - D2$, the data collector offloads the evolution process work onto the storage provider. In both designs, the storage provider is unaware of the real names of the files comprising the archive. The data collector keeps a map array *MapFile* that maps each file in the archive to a code called *fcode*. The data collector also holds a policy file *PolicyFile* containing the encryption schemes used for each layer of encryption applied to the archive along with their corresponding keys and the two hashing functions used in the two integrity schemes and their corresponding keys. In both designs, the storage provider is responsible for providing the storage space for storing the archive. The storage space must be secured by the provider against unauthorized physical and network access. We also assume that the communications between the data collector and the storage provider are carried over a secure channel, that is, authenticated and encrypted.

Since ArchiveSafe LT allows for individual files update, addition and deletion, depending on the time a file was added to the archive or updated, it could have a different number of evolution processes applied to it than the rest of the archive files. To address this, the *MapFile* stores the number of the evolution processes applied to each file, that is, the number of encryption layers l. The structure of the *MapFile* records is (*fn*, *fcode*, l). The structure of the *PolicyFile* is $(((\Pi_1,k_{C_1}),\ldots,(\Pi_n,k_{C_n})), ((\Delta_a,\ k_{I_a}),\ (\Delta_b,k_{I_b})))$ where (Π_i,k_{C_i}) is a symmetric scheme and its key, and $(\Delta_j,\ k_{I_j})$ is a MAC scheme and its key.

In both designs, we assume the data collector keeps a list of secure encryption and MAC schemes. Whenever a new secure scheme emerges, the data collector adds it to the list. The data collector initiates the system by populating the *PolicyFile* with two

secure cryptographic schemes $\S_1$ and $\S_2$.

$ASLT-D1$ utilizes symmetric encryption schemes, and the data collector performs all processing needed for all archiving processes. Since the data collector is responsible for all data processing because of the storage provider's limitations, the entire archive needs to be downloaded from the storage provider to be evolved on the data collector's side and then uploaded back to the storage provider, which consumes possibly ample resources. This drawback is addressed in $ASLT-D2$.

$ASLT-D2$ utilizes hybrid encryption schemes where the data collector securely sends the information needed for the evolution process to the storage provider who performs the process. The storage provider maintains a public key encryption scheme in which the data collector uses the public key k_{pub} to encrypt the evolution information and sends it to the storage provider, who uses the private key k_{priv} to decrypt the information and uses it to perform the evolution process. Although ASLT-D2 utilizes a hybrid encryption scheme to exchange the new evolution private key, the communication of this key could also be done over a confidential channel if available.

To achieve long-term confidentiality, ArchiveSafe LT keeps the archive encrypted by n encryption schemes at all times where at least two of these n schemes are deemed secure. ArchiveSafe LT maintains this state by using the evolution process. To achieve long-term integrity, ArchiveSafe LT maintains two integrity data objects $I_a = (I_{v_a}, I_{d_a})$ and $I_b = (I_{v_b}, I_{d_b})$ each of which is built using a different secure hash function. When one of the used hash functions becomes compromised, its corresponding Merkle tree is dropped and a new one is generated using a secure hash function through the evolution process. I_d contains the archive integrity data generated by secure integrity schemes, and I_v contains the information required to verify the integrity of the archive against

I_d.

In these two designs, we utilize Merkle trees [32] as the integrity data objects. The hash values of the archive files are the tree leaves, I_d is the tree, and I_v is the tree root. For any file in the archive, we define $arch_{fcode}$ to be the secured data of the file corresponding to *fcode* and $I_{d_{fcode}}$ to be the set of nodes connecting the tree root to the leaf representing this file and their siblings.

In both designs, the hash values are calculated based on the encrypted version of the files to provide immediate verification of the files without the need to download the file and decrypt all the layers.

The two designs utilize standard Merkle tree algorithms for maintaining the tree. We use `UpdIntObj` for updating the tree and its root after a leaf has been added or changed, `ExtIntObj` to extracts the nodes in the path from a certain tree leaf to the root, and `VfyIntObj` to verify the integrity of the tree by recalculating the root using the nodes provided and comparing it to the stored root value.

- $I_{d_{fcode}} \leftarrow$ `ExtIntObj`$(fcode, I_d)$: The function takes a file code *fcode* and the Merkle tree I_d as input and returns an array of nodes $I_{d_{fcode}}$ connecting the *fcode* leaf to the tree root.
- $(I'_v, I'_{d_{fcode}}) \leftarrow$ `UpdIntObj`$(I_{d_{fcode}}, fcode, tag, k, H)$: The function takes the array of nodes $I_{d_{fcode}}$ connecting the *fcode* leaf to the tree root, the file code *fcode*, the new integrity tag *tag*, a hash function H and its key k as input. It recalculates the hash values of the nodes in $I_{d_{fcode}}$ and the root using H and k. It outputs the updated root I'_v and nodes $I'_{d_{fcode}}$.
- $\{0, 1\} \leftarrow$ `VfyIntObj`$(I_v, I_{d_{fcode}}, k, H)$: The function takes the current values of the Merkle tree root I_v, the array of nodes $I_{d_{fcode}}$ connecting the *fcode* leaf to

the tree root, a hash function H and its key k as input. It outputs true if the tree integrity check passes or false if it does not.

In the next section, we present the detailed implementation of the two designs, their security analysis and how they provide long-term confidentiality and integrity.

4.3.1 ASLT-D1

In this design, the data collector side is responsible for all data processing. The archive's initial securing, evolution and retrieval processing are all done on the data collector's side. The data collector's implementation of the archive's local state ls_o consists of the policy file, the integrity verification data object I_v, and the map file. The ls_o structure is $\{PolicyFile,\ MapFile,\ I_v\}$. The ls_p structure is $\{I_d\}$. The implementations of the archiving protocols in $ASLT - D1$ are described in Figures 4.4, 4.5, 4.6, 4.7, 4.8, and 4.9.

Collector		**Provider**
$\langle\rangle$		$\langle\rangle$
	$\xrightarrow{Initialize}$	
$(k_{C_1}, k_{I_1}) \leftarrow \S_1.\texttt{KeyGen}()$		$arch \leftarrow \varnothing$
$(k_{C_2}, k_{I_2}) \leftarrow \S_2.\texttt{KeyGen}()$		
$PolicyFile \leftarrow ((\S_1.\Pi, k_{C_1}), (\S_2.\Pi, k_{C_2}),$ $(\S_1.\Delta, k_{I_1}), (\S_2.\Delta, k_{I_2}))$		
$I_v \leftarrow \varnothing$		$I_d \leftarrow \varnothing$
$MapFile \leftarrow \varnothing$		$ls_p \leftarrow I_d$
$ls_o \leftarrow (PolicyFile, MapFile, I_v)$		
$\langle ls_o \rangle$		$\langle ls_p, arch \rangle$

FIGURE 4.4: $ASLT - D1$ - Initialization Protocol

Collector		**Provider**
$\langle ls_o, fn \rangle$		$\langle ls_p, arch \rangle$
$(PolicyFile, MapFile, I_v) \leftarrow ls_o$		$I_d \leftarrow ls_p$
$fcode, l \leftarrow MapFile[fn]$		
$((k_{C_{n-l+1}}, \Pi_{n-l+1}), ..., (k_{C_n}, \Pi_n)),$		
$(k_{I_{curr}}, \Delta_{curr}) \leftarrow PolicyFile$		
if $fcode = \perp$ break		
	$\xrightarrow{fcode}$	
		$I_{d_{fcode}} \leftarrow \texttt{ExtIntObj}(fcode, I_d)$
	$\xleftarrow{arch_{fcode}, I_{d_{fcode}}}$	
if $\neg\texttt{VfyIntObj}(I_v, I_{d_{fcode}}, k_{I_{curr}}, \Delta_{curr})$		
Abort		
for $i = n, \ldots, n-l+1$		
$arch_{fcode} \leftarrow$		
$\S_i.\texttt{Unlock}(k_{C_i}, \Pi_i, arch_{fcode})$		
endfor		
$fc \leftarrow arch_{fcode}$		
$\langle fc \rangle$		$\langle\rangle$

FIGURE 4.5: $ASLT - D1$ - Retrieve Protocol

Collector		**Provider**
$\langle ls_o, fn, fc \rangle$		$\langle ls_p, arch \rangle$
$(PolicyFile, MapFile, I_v) \leftarrow ls_o$		$I_d \leftarrow ls_p$
$((k_{C_1}, \Pi_1), ..., (k_{C_n}, \Pi_n)),$		
$(k_{I_{curr}}, \Delta_{curr}) \leftarrow PolicyFile$		
$fcode \leftarrow MapFile[fn]$		
if $fcode \neq \varnothing$ **then**		
	$\xrightarrow{fcode}$	
		$I_{d_{fcode}} \leftarrow \texttt{ExtIntObj}(fcode, I_d)$
	$\xleftarrow{I_{d_{fcode}}}$	
if $\neg\texttt{VfyIntObj}(I_v, I_{d_{fcode}}, k_{I_{curr}}, \Delta_{curr})$		
Abort		
endif		
		$arch \leftarrow arch \setminus arch_{fcode}$
		$I_d \leftarrow I_d - I_{d_{fcode}}$
else		
$fcode \leftarrow_{\$} \{0,1\}^{128}$		
endif		
$arch_{fcode} \leftarrow$		
$\S_{n-1}.\texttt{LockConf}(k_{n-1}, \Pi_{n-1}, fc)$		
$arch_{fcode} \leftarrow$		
$\S_n.\texttt{LockConf}(k_n, \Pi_n arch_{fcode})$		
$MapFile[fn] \leftarrow fcode$		
$tag \leftarrow \Delta_{curr}.\texttt{Sign}(k_{I_{curr}}, arch_{fcode})$		
$(I_v, I_{d_{fcode}}) \leftarrow \texttt{UpdIntObj}(I_{d_{fcode}},$		
$fcode, tag, k_{I_{curr}}, \Delta_{curr})$		
	$\xrightarrow{arch_{fcode}, I_{d_{fcode}}}$	
		$arch \leftarrow arch \cup arch_{fcode}$
		$I_d \leftarrow I_d + I_{d_{fcode}}$
$ls_o \leftarrow (PolicyFile, MapFile, I_v)$		$ls_p \leftarrow I_d$
$\langle ls_o \rangle$		$\langle ls_p, arch \rangle$

FIGURE 4.6: $ASLT - D1$ - Update Protocol

Collector		**Provider**
$\langle ls_o, fn \rangle$		$\langle ls_p, arch \rangle$
$(PolicyFile, MapFile, I_v) \leftarrow ls_o$		$I_d \leftarrow ls_p$
$fcode \leftarrow MapFile[fn]$		
	$\xrightarrow{fcode}$	
		$arch \leftarrow arch \setminus arch_{fcode}$
		$I_{d_{fcode}} \leftarrow \texttt{ExtIntObj}(fcode, I_d)$
		$I_d \leftarrow I_d - I_{d_{fcode}}$
	$\xleftarrow{I_{d_{fcode}}}$	
if $\neg\texttt{VfyIntObj}(I_v, I_{d_{fcode}}, k_{I_n}, \Delta_n)$		
Abort		
endif		
$(I_v, I_{d_{fcode}}) \leftarrow$		
$\texttt{UpdIntObj}(I_{d_{fcode}}, fcode, \emptyset, k_I, \Delta)$		
	$\xrightarrow{I_{d_{fcode}}}$	
		$I_d \leftarrow I_d + I_{d_{fcode}}$
		$ls_p \leftarrow I_d$
$ls_o \leftarrow (PolicyFile, MapFile, I_v)$		
$\langle ls_o \rangle$		$\langle ls_p, arch \rangle$

FIGURE 4.7: $ASLT - D1$ - Delete Protocol

Collector		**Provider**
$\langle ls_o, algs_{dep}, \S_{old}, algs_{cur}, \S_{new} \rangle$		$\langle ls_p, arch \rangle$
$(PolicyFile, MapFile, I_v) \leftarrow ls_o$		
$(k_{C_{new}}, k_{I_{new}}) \leftarrow \S_{new}.\texttt{KeyGen}()$		
$Deprecate(PolicyFile, \S_{old}.\Delta)$		
$PolicyFile \leftarrow PolicyFile \| (k_{I_{new}}, \S_{new}.\Delta)$		
for each *fn* in *MapFile*		
$fcode \leftarrow MapFile[fn]$		
	$\xrightarrow{fcode}$	
		$I_{d_{fcode}} \leftarrow \texttt{ExtIntObj}(fcode)$
		$I_d \leftarrow I_d - I_{d_{fcode}}$
	$\xleftarrow{arch_{fcode}, I_{d_{fcode}}}$	
if $\neg\texttt{VfyIntObj}(I_v, I_{d_{fcode}}, k_{I_{old}}, \Delta_{old})$		
Abort		
endif		
$(I_v, I_{d_{fcode}}) \leftarrow$		
$\S_{new}.\texttt{LockInt}(k_{I_{new}}, \Delta_{new}, arch_{fcode}, I_{d_{fcode}})$		
	$\xrightarrow{I_{d_{fcode}}}$	
endfor		
$ls_o \leftarrow (PolicyFile, MapFile, I_v)$		
		$I_d \leftarrow I_d + I_{d_{fcode}}$
		$ls_p \leftarrow I_d$
$\langle ls_o \rangle$		$\langle ls_p \rangle$

FIGURE 4.8: $ASLT - D1$ - Evolve Integrity Protocol

Collector		**Provider**
$\langle ls_o, algs_{dep}, \S_{old}, algs_{cur}, \S_{new} \rangle$		$\langle ls_p, arch \rangle$
$(PolicyFile, MapFile, I_v) \leftarrow ls_o$		
$(k_{C_{new}}, k_{I_{new}}) \leftarrow \S_{new}.\texttt{KeyGen}()$		
$Deprecate(PolicyFile, \S_{old}.\Pi)$		
$PolicyFile \leftarrow PolicyFile \| (k_{C_{new}}, \S_{new}.\Pi)$		
$((k_{C_1}, \Pi_1), ..., (k_{C_n}, \Pi_n)),$		
$(k_{I_{curr}}, \Delta_{curr}) \leftarrow PolicyFile$		
for each *fn* in *MapFile*		
	$\xrightarrow{fcode}$	
		$I_{d_{fcode}} \leftarrow \texttt{ExtIntObj}(fcode, I_d)$
		$I_d \leftarrow I_d - I_{d_{fcode}}$
	$\xleftarrow{arch_{fcode}, I_{d_{fcode}}}$	
if $\neg\texttt{VfyIntObj}(I_v, I_{d_{fcode}}, k_{I_{curr}}, \Delta_{curr})$		
Abort		
endif		
$arch_{fcode} \leftarrow \S_{new}.\texttt{LockConf}(k_{C_{new}},$		
$\Pi_{new}, arch_{fcode})$		
$tag \leftarrow \Delta_{curr}.\texttt{Sign}(k_{I_{curr}}, arch_{fcode})$		
$(I_v, I_{d_{fcode}}) \leftarrow \texttt{UpdIntObj}(I_{d_{fcode}},$		
$fcode, tag, k_{I_{curr}}, \Delta_{curr})$		
	$\xrightarrow{arch_{fcode}, I_{d_{fcode}}}$	
		$I_d \leftarrow I_d + I_{d_{fcode}}$
		$ls_p \leftarrow I_d$
endfor		
$ls_o \leftarrow (PolicyFile, MapFile, I_v)$		
$\langle ls_o \rangle$		$\langle ls_p, arch \rangle$

FIGURE 4.9: $ASLT - D1$ - Evolve Confidentiality Protocol

4.3.2 ASLT-D2

In this design, the data collector side is responsible for the data processing required for the initial securing of the archive and retrieving it. For evolution, the storage provider is responsible for the evolution process data processing while the data collector is still responsible for initiating the process, selecting the schemes and the keys' generation. In order for this design to be secure, we assume the provider will follow the defined protocols for the operations and will not try to learn any information related to the plaintext or the secret keys.

For the evolution process, the data collector selects the new scheme to be used and generates the new keys, then uses the storage provider's secure public-key cryptographic scheme to encrypt and send this information to the provider to use in the evolution process.

In this design, the data collector's implementation of the archive's local state ls_o consists of the list of cryptographic schemes used to secure the archive and their corresponding keys in addition to the integrity verification data object I_v. The storage provider's implementation of the archive's local state ls_p consists of the cryptographic scheme used to communicate with the data collector and its corresponding keys in addition to the integrity data object I_d. The ls_o structure on the data collector's side is { *PolicyFile*, *MapFile*, I_v}. The ls_p structure on the storage provider's side is $\{\S_p, k_{priv}, k_{pub}, I_d\}$. The Initialization, Update, Retrieve and Delete protocols in this design are similar to the ones in $ASLT - D1$. The implementations of the evolution protocols in $ASLT - D2$ are described in Figures 4.10 and 4.11.

Collector		**Provider**
$\langle ls_o, \S_{old}, \S_{new} \rangle$		$\langle ls_p, arch \rangle$
$(PolicyFile, MapFile, I_v) \leftarrow ls_o$		
$(k_{C_{new}}, k_{I_{new}}) \leftarrow \S_{new}.\texttt{KeyGen}()$		
$Deprecate(PolicyFile, \S_{old}.\Delta)$		
$PolicyFile \leftarrow PolicyFile \| (k_{I_{new}}, \S_{new}.\Delta)$		
	$\texttt{Lock}_p, k_{pub} \longleftarrow$	
$EvPck \leftarrow (k_{I_{new}}, \S_{new}.\Delta)$		
$EvPck \leftarrow \texttt{Lock}_p(k_{pub}, EvPck)$		
	$EvPck \longrightarrow$	
		$(k_{I_{new}}, \S_{new}.\Delta) \leftarrow$ $\texttt{Unlock}_p(k_{priv}, EvPck)$
for each file f in $arch$		
		$I_{d_f} \leftarrow \texttt{ExtIntObj}(f)$
		$I_d \leftarrow I_d - I_{d_f}$
		$(I_v, I_{d_f}) \leftarrow \S_{new}.\texttt{LockInt}($ $k_{I_{new}}, \Delta_{new}, arch_f, I_{d_f})$
		$I_d \leftarrow I_d + I_{d_f}$
		$ls_p \leftarrow I_d$
endfor		
	$I_v \longleftarrow$	
$PolicyFile \leftarrow PolicyFile \| (\S_{new}.\Delta, k_{I_{new}})$		
$ls_o \leftarrow (PolicyFile, MapFile, I_v)$		
$\langle ls_o \rangle$		$\langle ls_p \rangle$

FIGURE 4.10: $ASLT - D2$ - Evolve Integrity Protocol

Collector		**Provider**
$\langle ls_o, \S_{old}, \S_{new} \rangle$		$\langle ls_p, arch \rangle$
$(PolicyFile, MapFile, I_v) \leftarrow ls_o$		
$(k_{C_{new}}, k_{I_{new}}) \leftarrow \S_{new}.\texttt{KeyGen}()$		
$Deprecate(PolicyFile, \S_{old}.\Pi)$		
$PolicyFile \leftarrow PolicyFile \Vert (k_{C_{new}}, \S_{new}.\Pi)$		
$((k_{C_1}, \Pi_1), ..., (k_{C_n}, \Pi_n)),$ $(k_{I_{curr}}, \Delta_{curr}) \leftarrow PolicyFile$		
	$\xleftarrow{\texttt{Lock}_p, k_{pub}}$	
$EvPck \leftarrow (k_{C_{new}}, \S_{new}.\Pi, k_{I_{curr}}, \Delta_{curr})$		
$EvPck \leftarrow \texttt{Lock}_p(k_{pub}, EvPck)$		
	$\xrightarrow{EvPck}$	
		$(k_{C_{new}}, \S_{new}.\Pi, k_{I_{curr}}, \Delta_{curr}) \leftarrow$ $\texttt{Unlock}_p(k_{priv}, EvPck)$
for each file f in $arch$		
		$I_{d_f} \leftarrow \texttt{ExtIntObj}(f)$
		$I_d \leftarrow I_d - I_{d_f}$
		$arch_f \leftarrow \S_{new}\texttt{LockConf}($ $k_{C_{new}}, \Pi_{new}, arch_f)$
		$tag \leftarrow \Delta_{curr}.\texttt{Sign}(k_{I_{curr}}, arch_f)$
		$(I_v, I_{d_f}) \leftarrow \texttt{UpdIntObj}($ $I_{d_f}, f, tag, k_{I_{curr}}, \Delta_{curr})$
		$I_d \leftarrow I_d + I_{d_f}$
		$ls_p \leftarrow I_d$
endfor		
	$\xleftarrow{I_v}$	
$PolicyFile \leftarrow PolicyFile \Vert$ $(\S_{new}.\Pi, k_{C_{new}})$		
$ls_o \leftarrow (PolicyFile, MapFile, I_v)$		
$\langle ls_o \rangle$		$\langle ls_p, arch \rangle$

FIGURE 4.11: $ASLT - D2$ - Evolve Confidentiality Protocol

4.3.3 Security Analysis

Due to the framework having multiple protocols and in order not to miss any scenarios or execution paths, we use an automatic prover to prove the security of the confidentiality and integrity properties of the ArchiveSafe LT framework against the threat model described in Section 4.2.3. Using an automatic prover covers all possible adversarial scenarios and reduces the risk of human errors in the process.

An automatic prover is a verification tool used for formal verification of cryptographic protocols through model checking. It takes as input a security protocol model, the actions taken by the entities running the protocol, the adversary specifications and specifications of the protocol's desired security properties. It either finds an attack or it automatically constructs a proof that takes into consideration the possibility of having arbitrarily many instances of the protocol's roles working in parallel, together with the actions of the adversary. The protocol is then tested to fulfill the specified security properties.

In our proof, we use Tamarin prover [31]. Tamarin utilizes the symbolic approach where cryptographic values are represented as terms and cryptographic primitives as functions. Functions are applied to terms to produce other terms.

Utilizing an automatic prover definitely provides the advantages mentioned above, but it comes with a risk: The validity of the proof depends on the accuracy of the system model developed by the user. To mitigate this risk, we modeled all the system's components, their states and transitions, in order to have an accurate model of the system.

We modeled our system as follows:

Functions are declared as `KeyGen`, `Lock` and `Unlock`: `KeyGen/2, Lock/3, Unlock/3`. The function is defined by a name and the number of parameters it accepts.

Correctness is enforced through the equation:

`Unlock(scheme, KeyGen(scheme, secretkey), Lock(scheme, KeyGen(scheme, secretkey), data)) = data`.

which states that if the system unlocks a previously locked data the output must be identical to the original data.

Oracles OUpdate, ORetrieve, ODelete, OGetArchive, OCorruptKey, OEvolveConf and OEvolveInt are modeled as Tamarin rules.

Security Experiments. are modeled as rules modeling the adversary's guessing processes and lemmas modeling the challenges. For confidentiality, we modeled a challenge `OUpdateChallenge` shown in Figure 4.12 to check if the adversary can break the archive confidentiality by being able to retrieve the contents of the archive. For integrity, we modeled a challenge `ForgeAnswer` shown in Figure 4.13 to check if the adversary can change the contents of the archive while the integrity data object stays valid. We finally define the security properties through lemmas. Lemma `confidentiality` shown in Figure 4.14 represents our confidentiality experiment in Figure 4.1 and lemma `integrity` shown in Figure 4.15 represents our integrity experiment in Figure 4.3.

We have developed two security theory files, one for checking the confidentiality property and the other is for checking the integrity property. The two security theories represents the $\mathrm{Exp}^{\mathsf{ind}}_{n,\S_1,\ldots,\S_n}(\mathcal{A})$ and $\mathrm{Exp}^{\mathsf{forge}}_{n,\S_1,\ldots,\S_n}(\mathcal{A})$ experiments from Section 4.2.5 for confidentiality and integrity challenges, respectively. The theory files are publicly available[1].

[1] `https://github.com/moesabry/ArchiveSafeLT_Prover`

We use $ASLT - D1$ scheme in the theories. We modeled the `Lock` and `Unlock` as a symmetric key authenticated encryption scheme.

OUpdateChallenge

```
1 :   let
2 :     lso = <ko1, ko2, ko3>
3 :     // Lock using scheme 1 then scheme 2
4 :     ctxt1 = Lock('1', ko1,  fcontents)
5 :     ctxt2 = Lock('2', ko2, ctxt1)
6 :   in
7 :   [
8 :     In(fname), // File name that adversary wants to store
9 :     Fr( fcontents),
10 :    !StateOwner(lso)
11 :  ]–[
12 :    ChallengeStored(fname,  fcontents) // Record what we stored
13 :  ]->[
14 :    !Archive(fname, '2', ctxt2) // Save the ciphertext in the archive
15 :  ]
```

FIGURE 4.12: ASLT-D1 Confidentiality challenge

Tamarin verified the security properties of the design based on the model presented above. All execution paths were tested and successfully verified.

There are some limitations in our Tamarin analysis that we believe do not affect the validity of the results. The first limitation is that our confidentiality experiment is modeled using indistinguishability while our Tamarin rule is modeled using message recovery. This is due to Tamarin's use of the Dolev–Yao symbolic model. The second limitation, since Tamarin does not handle unbounded protocols well, we modeled only three layers of locking which represents one evolution process not arbitrarily many as in our experiment. We believe this limitation does not affect the reliability of the results because the model still gives the adversary the opportunity to initiate an evolve

ForgeAnswer

```
rule OForgeAnswer2:
  let
    lso = <ko1, ko2, ko3>
  in
  [
    !StateOwner(lso),
    In(<fname, Lock('2', ko2, Lock('1', ko1, fcontents))>)
  ]–[
    ForgeAnswer(fname, '1', '2', fcontents)
  ]->[
  ]
rule OForgeAnswer3:
  let
    lso = <ko1, ko2, ko3>
  in
  [
    !StateOwner(lso),
    In(<fname, Lock('3', ko3, Lock('2', ko2, Lock('1', ko1, fcontents)))>)
  ]–[
    ForgeAnswer(fname, '2', '3', fcontents)
  ]->[
  ]
```

FIGURE 4.13: ASLT-D1 Integrity challenge

process. We also modeled one design, ASLT-D1. We believe ASLT-D2 should follow the same proof construction and produce similar results.

4.4 Evaluation

We evaluate ArchiveSafe LT by measuring its performance in real-life imitated scenarios. We compare its performance in terms of providing long-term confidentiality and integrity against other systems, such as LINCOS [8], PROPYLA [23], ELSA [35] and

Confidentiality Lemma

```
1 :  lemma confidentiality:
2 :    All fname fcontents #tchallenge
3 :    .
4 :    ChallengeStored(fname, fcontents)  #tchallenge
5 :    & not(Ex #tr . RetrievedContents(fname, fcontents)  #tr)
6 :    & not(
7 :    (Ex #tga #tc1 #tc2 . GotArchive(fname, '2')  #tga & Corrupted('1')
8 :                                                    #tc1 & Corrupted('2')  #tc2)
9 :    | (Ex #tga #tc2 #tc3 . GotArchive(fname, '3')  #tga & Corrupted('2')
10 :                                                   #tc2 & Corrupted('3')  #tc3)
11 :   )
12 :   ==>
13 :   not(Ex #tk . K(fcontents)  #tk)
14 :
```

FIGURE 4.14: ASLT-D1 Confidentiality lemma

Integrity Lemma

```
1 :  lemma integrity:
2 :    All fname layer1 layer2 fcontents #tforgeanswer
3 :    .
4 :    ForgeAnswer(fname, layer1, layer2, fcontents)  #tforgeanswer
5 :    ==>
6 :    ((Ex fname2 #tstored . Stored(fname2, fcontents)  #tstored)
7 :    | (Ex #tc1 #tc2 . Corrupted(layer1)  #tc1 & Corrupted(layer2)  #tc2)
8 :
```

FIGURE 4.15: ASLT-D1 Integrity lemma

SAFE [10]. The experiment's goal is to capture the system's performance metrics related to two areas: 1) how does the system perform compared to other systems, and 2) how does the system perform in a common real-life setting. The experiment measures the system's performance in working with different archives and measures the time the system requires to perform the main archiving processes: Update, Evolve

Confidentiality, Evolve Integrity and Retrieve. We did not measure the Initialize process times due to its negligible values.

4.4.1 Experiment Implementation

The implementation used for the experiment is based on $ASLT - D1$. In this implementation, I_d is the Merkle tree and I_v is the Merkle tree root. Next, we present the basic definitions of the integrity schemes used.

We present next how the `LockConf`, `LockInt` and `Unlock`() APIs are implemented in this experiment.

- `LockConf` takes the confidentiality key k_C, the encryption scheme Π and the contents of the file to be secured *fc* as input. It uses the k_C and Π to encrypt *fc* into $arch_{fn}$. The algorithm is shown in Figure 4.16.
- `LockInt` takes as input the integrity key k_I, the MAC scheme Δ, the data to be secured *fc* and the list of internal nodes $I_{d_{fcode}}$ connecting the root to the leaf corresponding to the file to be secured. It uses the integrity k_I and Δ to generate the new integrity tags for $arch_{fn}$. Next, it updates the internal nodes and the roots based on the newly generated tags and outputs the updated nodes $I_{d_{fcode}}$ and roots I_v for both trees. The algorithm is shown in Figure 4.17.
- `Unlock` takes the confidentiality key k_C, the encryption scheme Π and the secured archive data *arch* as input. It uses k_C and Π to decrypt the secured file $arch_{fn}$ into *fc*. It outputs the file contents *fc*. The algorithm is shown in Figure 4.18.

$\texttt{LockConf}((k_C, \Pi, fc)$

1 : $arch_{fn} \leftarrow \Pi.\text{Enc}(k_C, fc)$
2 : **return** $arch_{fn}$

FIGURE 4.16: The `LockConf` API

$\texttt{LockInt}(k_I, \Delta, fc, I_{d_{fcode}})$

1 : $(k_{I_a}, k_{I_b}) \leftarrow k_I$
2 : $(I_{v_a}, I_{v_b}) \leftarrow I_v$
3 : $(I_{d_a}, I_{d_b}) \leftarrow I_{d_{fcode}}$
4 : $(\Delta_a, \Delta_b) \leftarrow \Delta$
5 : $tag_a \leftarrow \Delta_a.\texttt{Sign}(k_{I_a}, fc)$
6 : $(I_{v_a}, I_{d_a}) \leftarrow \texttt{UpdIntObj}(I_{d_a}, fc, tag_a, k_{I_a}, \Delta_a)$
7 : $tag_a \leftarrow \Delta_a.\texttt{Sign}(k_{I_a}, fc)$
8 : $(I_{v_b}, I_{d_b}) \leftarrow \texttt{UpdIntObj}(I_{d_b}, fc, tag_b, k_{I_b}, \Delta_b)$
9 : $I_v \leftarrow (I_{v_a}, I_{v_b})$
10 : $I_{d_{fcode}} \leftarrow (I_{d_a}, I_{d_b})$
11 : **return** $(I_{d_{fcode}}, I_v)$

FIGURE 4.17: The `LockInt` API

$\texttt{Unlock}(k_C, \Pi, arch_{fn})$

1 : $fc \leftarrow \Pi.\text{Dec}(k_C, arch_{fn})$
2 : **return** fc

FIGURE 4.18: The `Unlock` API

4.4.2 Experimental Setup

To measure the system's performance in a real-life mimicking scenario, we designed the experiment to mimic the evolution of an archive with common properties such as the size, structure and the schemes used in the archive evolution process. Our scenario assumes the archive was initially built in 1992 and is still being kept secure till 2022

through an ArchiveSafe LT system. Based on the following release timelines for the available encryption schemes and hashing functions:

- 1992: Archive was created using DES and 3DES for confidentiality and MD2 and MD5 for integrity.
- 2001: Archive was evolved by combining another pair of schemes AES-128 and SHA-256 for confidentiality and integrity respectively.
- 2004: Archive was evolved by combining another pair of schemes AES-192 and SHA-384 for confidentiality and integrity respectively.
- 2015: Archive was evolved by combining another pair of schemes AES-256 and SHA3-512 for confidentiality and integrity respectively.

Measurements. We measure the time used by the system to perform the archiving processes on three archive sizes 1 MB, 1 GB and 10 GB to show how the system performs with common files sizes. We also measure the system performance on a 158 GB archive to compare its performance with LINCOS [8] and ELSA [35] since this is the only file size measured by these system experiments. Each archive consists of 1000 equal-sized files. File sizes are 1 KB for the 1 MB archive, 1 MB for the 1 GB archive, and 10 MB for the 1 GB and 158 GB archives. For the archive creation process, we measure the time used to read the data files, generate new encryption and integrity keys, double encrypt the files, double sign them and generate the Merkle tree from the integrity tags, then write the files to the disk. For Evolve Confidentiality process, we measure the time used to read the encrypted files, generate new encryption and integrity keys, apply an extra layer of encryption and generate two integrity tags for each file, write the files to the disk, update the two Merkle trees with the new leaves and update all internal nodes. For the Evolve Integrity process, we read the

encrypted files, generate new hashes using the new secure hash function, build a new Merkle tree using these new hashes as leaves then replace the compromised tree with the new one. For the archive retrieval process, we measure the time used to retrieve archives with a different number of evolution processes applied to them. We measure the time for retrieving the archive while having two layers of encryption, meaning its confidentiality was never evolved, up to five layers of encryption corresponding to three confidentiality evolution processes. All processing is performed locally and not over a network.

Test Environment. The experiment was performed on a single-user Linux machine with no other processes running. The computer was an HP Z420 running Ubuntu Linux 20.04.3 LTS with an 8-core Intel Xeon CPU E5-1620 processor with a frequency of 3.6 GHz. The computer had 32 GiB of RAM. The hard drive was a 1 TB solid-state drive with 512-byte logical sectors and 512-byte physical sectors. The experiment program was written in Python and used the Cryptodome library for cryptographic functions.

Execution. We performed 100 repetitions of the following tasks. Sample files were created with randomly generated alphanumeric characters using a non-cryptographic random number generator to form the archives. The files generation times are not measured. Update Archive, Evolve Archive and Retrieve Archive processes were performed on the archives and time was measured as detailed above. We use Merkle trees formed from the files' integrity tags as the integrity data objects.

4.4.3 Results

Table 4.1 shows the time used by the system to perform the initial creation of the archives. The processes include the keys generation, the double encryption and the creation of Merkle trees. We compared our system performance versus LINCOS and SAFE. The algorithms used in each stage are described in Section 4.4.2.

Archive Size	1 MB	10 MB	1 GB	10 GB	158 GB
ArchiveSafe LT	0.53s	3.3s	3.18m	31.54m	7.7h
LINCOS	N.A.	N.A.	N.A.	N.A.	2.3d
SAFE	1s	10s	N.A.	N.A.	N.A.

Table 4.1: ArchiveSafe LT archive creation time using DES + 3DES

Table 4.2 shows the time measured in seconds used by the system to retrieve the archive to its plaintext state. The results show the time used to retrieve the same archive from 2-layer encryption up to 5-layer encryption. These layers correspond to no confidentiality evolution processes applied to the archive up to three confidentiality evolution processes applied to the archive.

Archive Size	1 MB	1 GB	10 GB
Post Initial Creation	0.32	185.14	1838.4
Post 1st Evolution	0.17	69.91	690.62
Post 2nd Evolution	0.17	70.58	704.53
Post 3rd Evolution	0.20	79.33	785.39

Table 4.2: ArchiveSafe LT archive retrieval time in seconds

Table 4.3 shows the time measured in seconds used by the system to perform a confidentiality evolution process on the archive. This includes the keys generation, the encryption, new integrity tags generation and the creation of Merkle trees. Table 4.4 shows the time measured in seconds used by the system to perform an integrity

evolution process on the archive. This includes the keys generation, new integrity tags generation and the Merkle tree creation.

Archive Size	1 MB	1 GB	10 GB	158 GB
1st Evolution	0.18s	11.53s	2.02m	32.59m
2nd Evolution	0.20s	14.03s	2.69m	43.23m
3rd Evolution	0.24s	20.26s	3.29m	49.19m
LINCOS (Any Evolution)	N.A.	N.A.	N.A.	4.6d

Table 4.3: Confidentiality evolution time

Archive Size	1 MB	1 GB	10 GB	158 GB
SHA256	0.15s	6.36s	1.52m	24.34m
SHA384	0.20s	7.51s	73.29s	16.18m
SHA3-512	0.20s	12.26s	2.07m	29.43m
LINCOS	N.A.	N.A.	N.A.	4.6d

Table 4.4: Integrity evolution time

For individual file processes such as delete and update, we measure the time needed to update the Merkle trees' leaves and all affected internal nodes against the number of files in the archive. The number of files in the archive affects the Merkle trees' sizes which in turn affects the time needed to update the affected nodes. Table 4.5 shows the time needed versus the number of files in the archive.

Number of Files	100	1,000	1,000,000
	0.41ms	0.54ms	0.98ms

Table 4.5: Merkle trees update times for one node change

4.4.4 Discussion

The results show consistent performance across the file sizes and evolution processes. The variation between retrieval times is due to the difference in speed of decryption between the different encryption algorithms.

In comparison to LINCOS [8], PROPYLA [23], and ELSA [35], a 158 GB archive requires 2.3 days to exchange keys initially and 4.6 days for every reshare by these systems. The main time consuming task performed by these systems is the key sharing through a 40 Kb/sec QKD network. ArchiveSafe LT requires 7.7 hours for the archive creation, including all operations and between 33 to 50 minutes for every evolution process. The performance superiority of ArchiveSafe LT comes at the cost of sacrificing information-theoretic security for computational assumptions, but we mitigate that by utilizing robust combiners and the novel evolution protocol.

Table 4.6 shows a comparison between the systems discussed above and ArchiveSafe LT in terms of key generation. Moreover, ArchiveSafe LT does not require any private channels for the system to operate. The comparison parameters are: scheme, key generation speed, storage space needed compared to the original data size x, number of private channels in terms of data collectors number (d), storage servers (n) and reconstructing threshold (t). We use the archive size of 158 GB in the comparison because it is the size of the sample archive used in the evaluation of the other systems.

System	Scheme	Speed (/158 GB)	Storage Size Requirements
LINCOS*	Secret Sharing	2.3 Days	> 3x
ArchiveSafe LT	Robust Combiners	7.7 Hours	1x

TABLE 4.6: ArchiveSafe LT comparative analysis showing confidentiality Scheme used, key generation speed and the size of the resulting archive compared to the original data size x

The retrieval process, for a 1 GB archive after 30 years of secure archiving and evolution, requires 79.33 seconds which is acceptable for an archive retrieval process compared to the currently available archiving systems. These results confirm the ArchiveSafe LT performance in a practical scenario.

Space Usage: Since our design utilizes standard symmetric encryption schemes, the encrypted archives' sizes are in the same order as the original files with minor possible increases for practical processes such as padding. In order for our design to handle individual files operations such as addition and deletion, it keeps a map file. The minor overhead per file is the map file record size plus the hash values stored in the Merkle trees. Based on our proposed map file structure in Section 4.3, the record size is 266 bytes for each file.

4.5 Summary

In this chapter, we presented ArchiveSafe LT, a framework for archiving systems providing long-term confidentiality and integrity. The framework utilizes robust combiners of standard cryptographic suites to achieve its goal instead of the secret-sharing techniques used by currently available systems. This approach makes the system feasible for industrial adoption due to its independence from any private channels or QKD systems. Requiring minutes instead of days to create secure archives and evolving them in the future makes the framework more feasible to be implemented. The performance superiority of ArchiveSafe LT comes at the cost of sacrificing information-theoretic security for computational assumptions, but we mitigate that by utilizing robust combiners and the novel evolution protocol.

In the next chapter, we are looking to utilize a more robust and independent solution for long-term integrity. A more robust and secure data structure is needed to hold the integrity information to eliminate the need for redundant Merkle trees.

Chapter 5

Hybrid Merkle Trees

In this chapter, we introduce the *Hybrid Merkle Tree*, an authenticated data structure based on the Merkle tree. In a Hybrid Merkle tree, not all the nodes are generated by the same hash function. Through its lifetime, the tree evolves to a secure hashing function if its current state becomes insecure, making it suitable for integrity schemes used by long-term secure archiving systems. A sample hybrid Merkle tree using two hash functions H_1 and H_2 is shown in Figure 5.1. In this figure, the hybrid Merkle tree represents four data items: d_1, d_2, d_3 and d_4. Similar to a standard Merkle tree, the leaves are the hash values of the data items the tree represents, and the internal nodes contain the hash value of the concatenation of its two children's hash values. The difference between the hybrid and the standard Merkle tree is shown through data item d_4. In a standard Merkle tree, all leaves and internal nodes are generated by one hash function, in this case it would be H_1, but in this hybrid Merkle tree, the leaf representing d_4 and all internal nodes connected to it including the root were generated using a second hash function H_2.

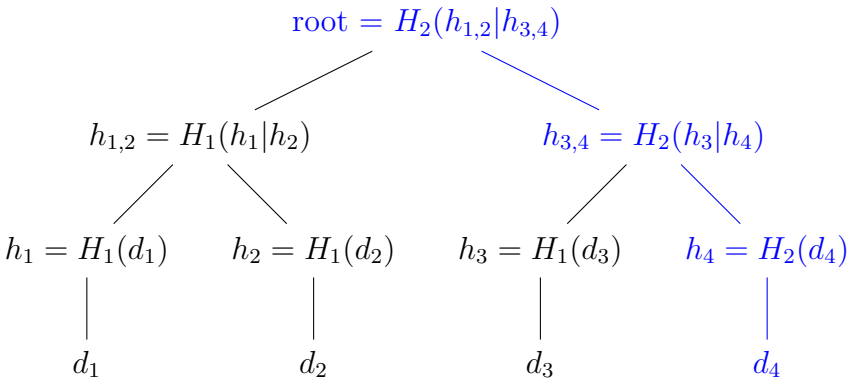

FIGURE 5.1: Hybrid Merkle tree example

Merkle trees are used for authentication in many areas. Certificate Transparency logs [25], public key signatures [4] and file systems [29][43] are some examples. These systems build and maintain significantly large trees, for example, Certificate Transparency logs have around 1.9 billion entries.

When a weakness is discovered in the underlying hash function used by the Merkle tree, it could affect the integrity of the whole tree and all the hashes must be recalculated using a secure hash function. The process of recalculating the hashes is costly in terms of needed computing resources and time. Also, during this process, the whole tree's integrity is questionable.

In ArchiveSafe LT, we overcame the problem of hash functions becoming insecure by utilizing more than one Merkle tree where each tree is built using a different hash function. Although this solution addresses the problem of the function becoming insecure overtime, it introduces other issues, such as multiplying the space required to store the integrity information and the processing needed to perform any updates on it. A better solution is to use a different data structure for the integrity information.

A structure that can be used to provide proofs for inclusion in an ordered list and its own integrity, while eliminating the need for excessive space and processing power.

In this chapter, we construct a succinct updatable proof structure. We start by presenting a formal definition of the structure. We present an instantiation of the structure as a Merkle tree and prove the security of this instantiation under existential unforgeability against a chosen message attack (EUF-CMA).

Next, we construct an evolving version of the structure which is suitable to address the aforementioned problem. It is similar to the first structure but has the ability to evolve its security by utilizing multiple compression functions simultaneously. When the currently used compression function is deemed insecure, the structure evolves by switching to use a secure compression function and deprecating the insecure one.

We present a formal definition of this structure and introduce the Hybrid Merkle tree as an instantiation of it. We prove the security of this instantiation under existential unforgeability against a chosen message attack (EUF-CMA).

5.1 *sStruct*: Succinct Updatable Proof Structure

In this section, we present a succinct updatable proof structure *sStruct*. The structure represents a finite ordered list of data values $\mathcal{D} = (d_1, d_2, d_3...)$. The structure consists of three types of data objects: 1) data digest objects DOs representing the stored data, 2) a verification object VO used to verify the inclusion of the data objects in the list, 3) intermediate objects IOs containing information connecting the data digests and the verification object, they are used in the verification process. The data values, data objects and intermediate objects are stored in ordered lists.

An *sStruct* data structure has the following algorithms:

- $(VO', IO'_i, DO[i]) \leftarrow \texttt{Add}(VO, IO_i, d)$: This algorithm adds to the structure a data object representing a data value d. It takes as input the verification object VO, the list of intermediate objects connecting the new node to the root IO_i, and the data value to be added d. The algorithm starts by verifying the list of the intermediate objects against the verification object, it exits with an error if the verification fails. If the verification passes, the algorithm updates the intermediate objects' list with the new objects added to accommodate the newly added data item and updates the rest of the intermediate objects accordingly in addition to the verification object. It outputs the updated verification object VO', the updated list of intermediate objects IO'_i, and the data object $DO[i]$ representing the data value d.
- $(VO', IO'_i, DO[i]') \leftarrow \texttt{Update}(VO, IO_i, d'_i, i)$: This algorithm updates the structure by assigning a new value d'_i to position i. It takes as input the verification object VO, the list of intermediate objects IO_i related to i, the new value d'_i to be assigned to data item i, and the index of the item to be updated i. The algorithm starts by verifying the list of the intermediate objects against the verification object, it exits with an error if the verification fails. If the verification passes, the algorithm updates the value of data object DO_i to reflect the new value d'_i, calculates the new values for all intermediate objects IO'_i related to item i and the new verification object value VO'. It outputs the updated verification object VO', the updated list of intermediate objects IO'_i related to i and the updated data object $DO[i]'$.
- $\{0, 1\} \leftarrow \texttt{Verify}(VO, IO_i, d_i, i)$: This algorithm verifies whether a data value d_i represents the correct value of item i. It takes as input the verification object

VO, the list of intermediate objects IO_i related to item i, and d_i, the data value for i. The algorithm uses the intermediate objects and the data value to verify against the verification object. It outputs 1 for verification success or 0 for failure.

- $(VO, IO, DO) \leftarrow \texttt{Build}(\mathcal{D})$: This algorithm builds a complete *sStruct* from an ordered list of data values $\mathcal{D}$. It takes the list of data values $\mathcal{D}$ as input and outputs the verification object VO, the intermediate objects list IO and the data objects list DO forming the *sStruct*. It uses a simple approach to build the tree by calling the `Add` algorithm for each data value in $\mathcal{D}$.

5.1.1 Security Property

For *sStruct* to be secure, it must not allow a data object $DO[i]$ to be successfully verified unless its corresponding data value d_i belongs to the honest ordered list $\mathcal{D}$. We call such a secure structure *unforgeable*. Proving that *sStruct* is *unforgeable* is not simple since we have to accommodate for the update functionality of the structure. We define the *unforgeable* security property through the experiments in Figure 5.2 and Figure 5.3. The security experiment $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$ lets the adversary operate in a real environment where the experiment does not keep track of all objects belonging to *sStruct* but rather uses VO for verification. $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$ on the other hand, lets the adversary operate in an ideal environment where the experiment keeps track of all objects belonging to *sStruct*. All internal objects are stored by the experiment in IO_E. Running these two experiments shows if and how the adversary behaves differently running in a real setup of our structure versus an ideal one where they cannot win. If the adversary does not behave differently, then the structure is secure. We define the *unforgeable* security property as follows:

unforgeable: $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$ is indistinguishable from $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$.

The experiment provides the adversary with the following oracles:

- $\mathsf{OAdd}(IO_i, d)$: Allows the adversary to add a new data item to the structure. It takes as input the list of intermediate objects connecting the new node to the root IO_i, and the data value to be added d. In $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$, the oracle uses the `Add` algorithm to add the data item to the structure with no additional verification. In $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$, the oracle verifies first if the provided internal nodes list IO_i matches what is stored in the oracle memory. It fails the experiment if they do not match. If the two lists match, the oracle continues in the same manner as $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$.

- $\mathsf{OUpdate}(IO_i, d'_i, i)$: Allows the adversary to update the value of a data item and update the structure accordingly. It takes as input the list of intermediate objects IO_i related to i, the new value d'_i to be assigned to data item i, and the index of the item to be updated i. In $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$, the oracle uses the `Update` algorithm to update the data item in the structure with no additional verification. In $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$, the oracle verifies first if the provided internal nodes list IO_i matches what is stored in the oracle memory. It fails the experiment if they do not match. If the two lists match, the oracle continues in the same manner as $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$.

- $\mathsf{OVerify}(IO_i, d_i, i)$: Allows the adversary to verify a data item. In $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$, the item is verified using the internal objects IO_i provided by the adversary and the verification object VO provided by the experiment. In $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$,

the item is verified using the internal objects IO_{E_i} stored by the experiment and the verification object VO provided by the experiment.

The experiments and the oracles available to the adversary are shown in Figure 5.2 and Figure 5.3.

$\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$

1 : $VO \leftarrow \perp$
2 : $b \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{OAdd},\mathsf{OUpdate},\mathsf{OVerify}}()$
3 : **return** b

$\mathsf{OAdd}(IO_i, d)$

1 : $(VO, IO_i, DO[i]) \leftarrow\!\!\$\ \texttt{Add}(VO, IO_i, d, \mathit{PolicyFile})$
2 : **return** $(IO_i, DO[i])$

$\mathsf{OUpdate}(IO_i, d'_i, i)$

1 : $(VO, IO_i, DO[i]) \leftarrow\!\!\$\ \texttt{Update}(VO, IO_i, d'_i, i)$
2 : **return** $(IO_i, DO[i])$

$\mathsf{OVerify}(IO_i, d_i, i)$

1 : **return** $\texttt{Verify}(VO, IO_i, d_i, i)$

FIGURE 5.2: *sStruct unforgeable* Real Security experiment and oracles

$\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$

1 : $D \leftarrow []$
2 : $IO \leftarrow \perp$
3 : $b \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{OAdd},\mathsf{OUpdate},\mathsf{OVerify}}()$
4 : **return** b

$\mathsf{OAdd}(IO_i, d)$

1 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO_E, |D|)$
2 : // Check if the nodes provided by $\mathcal{A}$ matches the ones kept by the experiment
3 : **if** $IO_i \neq IO_{E_i}$ **then return** $\perp$
4 : $(VO, IO_i, DO[i]) \leftarrow\!\!\$\ \texttt{Add}(VO, IO_i, d, \mathit{PolicyFile})$
5 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO, i)$ // Store the new values in the experiment
6 : $D[|D|] \leftarrow d_i$
7 : **return** $(IO_i, DO[i])$

$\mathsf{OUpdate}(IO_i, d'_i, i)$

1 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO_E, i)$
2 : // Check if the nodes provided by $\mathcal{A}$ matches the ones kept by the experiment
3 : **if** $IO_i \neq IO_{E_i}$ **then return** $\perp$
4 : $(VO, IO_i, DO[i]) \leftarrow\!\!\$\ \texttt{Update}(VO, IO_i, d'_i, i)$
5 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO, i)$ // Store the new values in the experiment
6 : $D[i] \leftarrow d$
7 : **return** $(IO_i, DO[i])$

$\mathsf{OVerify}(IO_i, d_i, i)$

1 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO_E, i)$
2 : // Check if the nodes provided by $\mathcal{A}$ matches the ones kept by the experiment
3 : **if** $IO_i \neq IO_{E_i}$ **then return** $\perp$
4 : **return** $D[i] = d_i$

FIGURE 5.3: *sStruct unforgeable* Ideal security experiment and oracles

5.1.2 Merkle Tree as an *sStruct*

We present now $MT[H]$, an *sStruct* instantiation as a Merkle tree with a hash function H. H is used to generate the hash values of all the tree's nodes and root. $MT[H]$ is

a Merkle tree system where VO is the Merkle tree root r_T, the intermediate objects IOs are the tree's internal nodes represented by the ordered list *inodes* and the data objects DOs are the tree leaves represented by the ordered list *leaves*.

Our Merkle tree implementation uses a balanced weighted binary tree where the weight of any node is the number of leaves under its subtree. An example is shown in Figure 5.4.

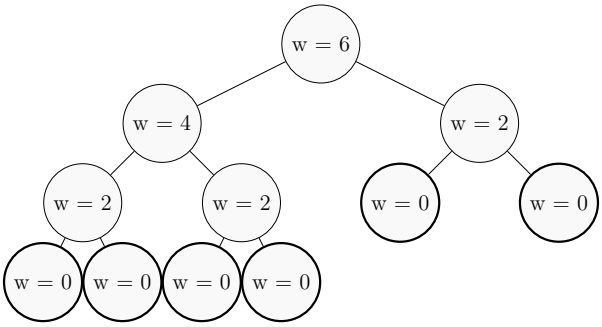

FIGURE 5.4: Weighted Merkle tree example

To add a new leaf, the algorithm follows the path with the lesser weight until it reaches the point of insertion. An example is shown in Figure 5.5. Since we only insert leaves and not internal nodes, we do not do rotations to balance the tree so the leaves do not get mixed with the internal nodes.

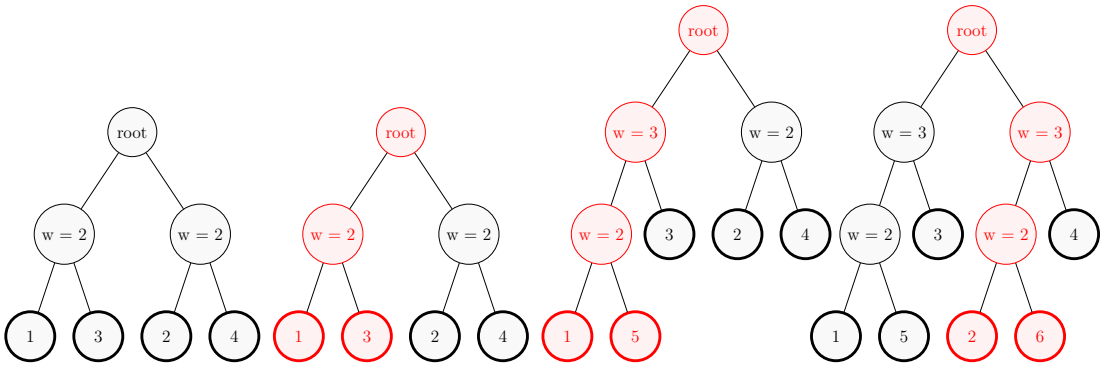

FIGURE 5.5: Weighted Merkle tree insertion example. Leaf 5 is inserted first followed by leaf 6.

Balancing the tree only through weights and not rotations leads to the tree losing the node ordering property. Node ordering allows searching for a node to be performed in time $O(log(n))$. To overcome this problem, we utilize a supplementary AVL balanced tree structure for *leaves* to hold the values and locations of the original tree T leaves. In an update operation, the function `Find` would query the values of the field *dataindex* in the *leaves* of an AVL tree to get the location of the leaf in T in $O(log(n))$ then update the original tree in $O(log(n))$ time. In an `Add` operation, the new leaf is added to the Merkle tree then a node is inserted in the AVL tree containing the hash value of the leaf along with its index from the Merkle tree. In an update operation, the algorithm searches for the hash value in the AVL tree in $O(log(n))$ time then uses the corresponding index value to locate the leaf in the Merkle tree in $O(n)$ time.

In our tree, each node stores the hash value of the concatenation of its two children nodes' hashes in $node.hash$, two pointers to each of its children $node.left$ and $node.right$, a pointer to its parent $node.parent$, $node.weight$, which is the number of all leaves in its subtree, and finally $node.dataindex$, which is null for all internal nodes but has the value i for the leaves. To avoid possible collisions between leaves and subtrees of internal nodes, we separate their domains by adding a prefix of 0 to each leaf and 1 to each internal node when we are calculating their hashing values.

The algorithms for $MT[H]$ are as follows:

- $(r'_T, inodes'_i) \leftarrow \texttt{Add}(r_T, inodes_i, d)$: This algorithm adds a new leaf to the tree corresponding to a data item d. It takes as input the tree root r_T, the internal tree nodes $inodes_i$, and the data value d. The algorithm starts by verifying the list of the internal nodes against the root, it exits with an error if the verification

fails. If the verification passes, the algorithm calculates the hash value of the new data item, then adds it as a new leaf to the tree after setting its weight to 0 to ensure it ends as a leaf in the tree. Next, the algorithm updates all internal nodes values connecting the new leaf to the root to reflect the newly added leaf through the `UpdateIO` function. The algorithm outputs the newly calculated tree root r'_T and the updated internal tree nodes $inodes'_i$. An illustration of the leaf addition is shown in Figure 5.5. The algorithm is shown in Figure 5.6.

- $(r'_T, inodes'_i, leaves[i]) \leftarrow \texttt{Update}(r_T, nodes_i, d'_i, i)$: This algorithm updates the value of an existing leaf to reflect a change in the data value of a data item i from d_i to d'_i. It takes as input the tree root r_T, the internal tree nodes $inodes_i$ connecting the root to the leaf i, the new data value d'_i, and the position i. The algorithm starts by verifying the list of the internal nodes against the root, it exits with an error if the verification fails. If the verification passes, the algorithm calculates the hash value of the new data value d'_i, then updates all internal nodes values connecting this leaf to the root to reflect the new value. It outputs the updated root r'_T and the updated subset of nodes $inodes'_i$ connecting the leaf to the root. The algorithm is shown in Figure 5.8.
- $\{0, 1\} \leftarrow \texttt{Verify}(r_T, inodes_i, d_i, i)$: This algorithm verifies whether a data value d_i is the correct value in leaf i. It takes as input the tree root r_T, the list of internal nodes connecting the i's leaf to the root of the tree $inodes_i$, the data value to be verified d_i and the position i. First, it locates the position of the item in the tree by using the standard AVL tree `Find` then calculates the value of the tree root r'_T using the provided values of the internal nodes and value d_i. The algorithm compares the calculated root to the provided root r_T and outputs 1 if the two roots match, that is, the verification succeeded and 0 otherwise. The algorithm is shown in Figure 5.10.

`Add`$(r_T, inodes_i, d)$

```
1 :  if ¬(VerifyIO(r_T, inodes_i)) then Abort
2 :  k = (|inodes_i|+1)/2
3 :  h = H(0|d), j = 0
4 :  for level = 1 ... k
5 :    if inodes_i[j].weight = 0
6 :      newdataindex = inodes_i[j].dataindex + 1
7 :      inodes_i, NewLeaf ← InsertNode(j, h, inodes_i, newdataindex)
8 :    else
9 :      if inodes_i[inodes_i[j].left].weight ≤ inodes_i[inodes_i[i].right].weight
10 :       j = inodes_i[j].left
11 :     else
12 :       j = inodes_i[j].right
13 :     endif
14 :   endif
15 : endfor
16 : r_T, inodes_i ← UpdateIO(inodes_i, j)
17 : return (r_T, inodes_i, NewLeaf)
```

FIGURE 5.6: The *sStruct* `Add` algorithm

$(r_T, nodes) \leftarrow$ `Build`$(\mathcal{D})$: This algorithm builds the complete tree from a set of data values. It takes an ordered list of data values $\mathcal{D}$ as input, and outputs the root of the tree r_T and an ordered list of internal nodes *inodes* forming the tree. It builds the tree by calling `Add` for each data value in the set $\mathcal{D}$.

5.1.3 Security Analysis

For the Merkle tree, we define the *unforgeable* security property as the ability of the tree to prevent a data object $DO[i]$ to be successfully verified unless its corresponding data value d_i is part of the honest list $\mathcal{D}$, that is, $d_i = \mathcal{D}[i]$.

Lemma 1: If H is collision resistant, then $MT[H]$ is *unforgeable*.

`InsertNode`$(j, h, inodes_i, newdataindex)$

1 : $NewNode.parent = inodes_i[j].parent$
2 : $NewNode.left = j$
3 : $NewNode.right = j + 2$
4 : $NewNode.weight = 2$
5 : $NewNode.index = j + 1$
6 : $NewNode.dataindex = null$
7 : $NewNode.hash = H(1|inodes[j].hash|h)$
8 : $inodes_i$.`Append`$(NewNode)$
9 : $NewLeaf.index = j + 2$
10 : $NewLeaf.dataindex = newdataindex$
11 : $NewLeaf.parent = NewNode.index$
12 : $NewLeaf.hash = h$
13 : **return** $(inodes_i, NewLeaf)$

FIGURE 5.7: The *sStruct* `InsertNode` algorithm

`Update`$(r_T, inodes_i, d'_i, i)$

1 : **if** $\neg$(`VerifyIO`$(r_T, inodes_i)$) **then** Abort
2 : $h \leftarrow H(0|d'_i)$
3 : $j \leftarrow$ `Find`(i)
4 : $inodes_i \leftarrow$ `UpdateIO`$(inodes_i, j, r_T)$
5 : **return** $(r_T, inodes_i, h)$

FIGURE 5.8: The *sStruct* `Update` algorithm

Proof of Lemma 1: If the adversary $\mathcal{A}$ can distinguish between $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$ and $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$, then they were able to find a collision by finding a data value that passes $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$ and fails $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$. The algorithm $\mathcal{B}^{\mathcal{A}}$ shown in Figure 5.12 finds the collision in H by identifying two lists $\mathcal{D}_1$ and $\mathcal{D}_2$ where $\mathcal{D}_1 \neq \mathcal{D}_2$, but $MT[H](\mathcal{D}_1) = MT[H](\mathcal{D}_2)$. $\mathcal{B}^{\mathcal{A}}$ oracles identify the collision event when $\mathcal{A}$ behaves differently during the two experiments. At this time, $\mathcal{B}^{\mathcal{A}}$ passes the data lists causing the collision to $\mathcal{F}$, which loops through the subtrees forming $\mathcal{D}_1$ and $\mathcal{D}_2$ bottom to top until it finds the collision. □

`UpdateIO`$(inodes_i, j, r_T)$

1 : **while** $inodes_i[j].parent \neq 0$
2 : $l = inodes_i[inodes_i[j].parent].left$
3 : $r = inodes_i[inodes_i[j].parent].right$
4 : **if** $l \neq \emptyset \wedge r \neq \emptyset$
5 : $p = inodes_i[l].hash|inodes_i[r].hash$
6 : **if** $l \neq \emptyset \wedge r = \emptyset : p = inodes_i[l].hash$
7 : **if** $l = \emptyset \wedge r \neq \emptyset : p = inodes_i[r].hash$
8 : $inodes_i[inodes_i[j].parent].hash = H(1|p)$
9 : $j = inodes_i[j].parent$
10 : **endwhile**
11 : $r_T.hash = H(1|inodes_i[r_T.left].hash|inodes_i[r_T.right].hash)$
12 : **return** $(r_T, inodes_i)$

FIGURE 5.9: The *sStruct* `UpdateIO` algorithm

The OAdd, OUpdate and OVerify oracles for $\mathcal{B}^{\mathcal{A}}$ are shown in figures 5.13, 5.14 and 5.15, respectively. The code added to the basic oracles specifically for $\mathcal{B}^{\mathcal{A}}$ is color coded in blue.

5.2 *esStruct*: Evolving Updatable Succinct Proof Structure

In this section, we present an evolving succinct updatable proof structure *esStruct*. This structure is similar to *sStruct* but with the additional capability of evolving when the compression function it uses becomes insecure. When the compression function is deemed insecure, the leaves and internal nodes created using it are deemed insecure. Initially, all the data object digests in the structure are generated using a single function F_a, whether it is for a new object or to update an existing one. If during the structure's lifetime a more secure function F_b emerges, *esStruct* stops

`Verify`$(r_T, inodes_i, d_i, i)$

1 : $j = nodes_i$.`Find`(i)
2 : $currnode \leftarrow inodes_i[j].parent$
3 : **if** $(i \mod 2 = 0)$ **then**
4 : $inodes_i[currnode].hash \leftarrow H(0|inodes_i[left].hash|d_i)$
5 : **else**
6 : $inodes_i[currnode].hash \leftarrow H(0|d_i|inodes_i[right].hash)$
7 : $currnode \leftarrow currnode.parent$
8 : **while** $inodes_i[currnode].parent \neq \emptyset$
9 : $left \leftarrow inodes_i[currnode].left$
10 : $right \leftarrow inodes_i[currnode].right$
11 : $nodes_i[currnode].hash \leftarrow H(1|inodes_i[left].hash|inodes_i[right].hash)$
12 : $currnode \leftarrow currnode.parent$
13 : **endwhile**
14 : $r'_T \leftarrow currnode$
15 : **return** $(r_T = r'_T)$

FIGURE 5.10: The *sStruct* `Verify` algorithm

using F_a, and starts using F_b to generate the digests for new objects and updates for existing ones. The rest of the existing unchanged objects keep their values generated by F_a. The evolution process is continuous, the object evolves every time a function is deemed insecure. Similar to *sStruct*, the structure represents a finite ordered list of data values $\mathcal{D} = (d_1, d_2, d_3...)$. The structure consists of three types of data objects: 1) Data digest objects DOs representing the stored data, 2) a verification object VO used to verify the authenticity of the data objects, 3) intermediate objects IOs containing information about the data digests and the verification object to be used in the verification process.

To implement the evolution process, the structure must have a set of compression functions $\mathcal{F}$ and a *PolicyFile* file containing the functions used in the structure and which ones are deemed secure to use. Due to the structure evolution, at any time,

`VerifyIO`$(r_T, inodes_i)$

1 : $j = nodes_i$.`FindLeaf`()
2 : $currnode \leftarrow inodes_i[j].parent$
3 : **while** $inodes_i[currnode].parent \neq \emptyset$
4 : $left \leftarrow inodes_i[currnode].left$
5 : $right \leftarrow inodes_i[currnode].right$
6 : $nodes_i[currnode].hash \leftarrow H(1|inodes_i[left].hash|inodes_i[right].hash)$
7 : $currnode \leftarrow currnode.parent$
8 : **endwhile**
9 : $r'_T \leftarrow currnode$
10 : **return** $(r_T = r'_T)$

FIGURE 5.11: The *sStruct* `VerifyIO` algorithm

$\mathcal{B}()$:

1 : $IO_E \leftarrow []$
2 : $IO \leftarrow []$
3 : $D \leftarrow []$
4 : $b \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{OAdd},\mathsf{OUpdate},\mathsf{OVerify}}()$

FIGURE 5.12: The $\mathcal{B}$ algorithm for *sStruct*

the structure could have objects generated using several functions. To identify which function is used to generate a certain object, we store the identifier of the function used to generate the node value in the node as an additional field.

A *esStruct* consists of the following components:

- $\mathcal{F} = \{F_1, F_2, F_3, ...\}$: A set of compression functions, where at any point in time at least one of them is deemed secure.
- *PolicyFile*: A file containing all the functions used in the structure and the currently active ones. The file structure is $\{(F_1$,Obsolete), $(F_2$,Obsolete), $(F_3$,Secure),...$\}$.

$\mathsf{OAdd}(IO_i, d)$

1 : $IO_i \leftarrow \mathtt{GetIO}(IO, |D|)$
2 : $IO_{E_i} \leftarrow \mathtt{GetIO}(IO_E, |D|)$
3 : **if** $IO_i \neq IO_{E_i} \wedge \mathtt{Verify}(VO, IO_i, \emptyset, |DO|)$ **then**
4 : Exit to $\mathcal{B}$ $(\mathcal{F}(IO_{E_i}, IO_i))$
5 : $(VO, IO, DO) \leftarrow\!\!\$\ \mathtt{Add}(VO, IO, DO, d, PolicyFile)$
6 : $IO_{E_i} \leftarrow \mathtt{GetIO}(IO, i)$
7 : $D[|D|] \leftarrow d_i$
8 : **return** $(IO_i, DO[i])$

FIGURE 5.13: The $\mathcal{B}$ OAdd oracle for *sStruct*

$\mathsf{OUpdate}(IO_i, d'_i, i)$

1 : **if** $H(d_i) = H(d'_i) \wedge d_i \neq d'_i$ **then**
2 : Exit to $\mathcal{B}$ (d_i, d'_i)
3 : $IO_{E_i} \leftarrow \mathtt{GetIO}(IO_E, i)$
4 : **if** $IO_i \neq IO_{E_i} \wedge \mathtt{Verify}(VO, IO_i, d'_i, i)$ **then**
5 : Exit to $\mathcal{B}$ $(\mathcal{F}(IO_{E_i}, IO_i))$
6 : $(VO, IO_i) \leftarrow\!\!\$\ \mathtt{Update}(VO, IO_i, d'_i, i)$
7 : $IO_{E_i} \leftarrow \mathtt{GetIO}(IO, i)$
8 : $D[i] \leftarrow d$
9 : **return** $(IO_i, DO[i])$

FIGURE 5.14: The $\mathcal{B}$ OUpdate oracle for *sStruct*

An *esStruct* has the following algorithms:

- $(VO', IO'_i, DO[i]) \leftarrow \mathtt{Add}(VO, IO_i, d, PolicyFile)$: This algorithm adds to the structure a data object representing a data value d. It takes as input the verification object VO, the list of intermediate objects connecting the new node to the root IO_i, the data value to be added d, and the policy file *PolicyFile* to extract the current secure compression function. The algorithm starts by verifying the list of the intermediate objects against the verification object, it exits with an error if the verification fails. If the verification passes, the algorithm

$\mathsf{OVerify}(IO_i, d_i, i)$

1 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO_E, i)$
2 : **if** $IO_i \neq IO_{E_i} \wedge \texttt{Verify}(VO, IO_i, d_i, i)$ **then**
3 : Exit to $\mathcal{B}$ $(\mathcal{F}(IO_{E_i}, IO_i))$
4 : **return** $D[i] = d_i$

FIGURE 5.15: The $\mathcal{B}$ OVerify oracle for *sStruct*

$\mathcal{F}(\mathcal{D}_1, \mathcal{D}_2)$:

1 : $n_1 = |\mathcal{D}_1|, n_2 = |\mathcal{D}_2|$
2 : $k_1 = 2^{\lceil \log(n_1)/2 \rceil}, k_2 = 2^{\lceil \log(n_2)/2 \rceil}$
3 : **if** $n_1 = 1 \wedge n_2 = 1$ **then return** $(\mathcal{D}_1[0], \mathcal{D}_2[0])$
4 : **elseif** $n_1 = 1 \wedge n_2 > 1$ **then**
5 : **return** $(\mathcal{D}_1[0], MT[H].\texttt{Build}(\mathcal{D}_2[0 : k_2]) || MT[H].\texttt{Build}(\mathcal{D}_2[k_2 : n_2]))$
6 : **elseif** $n_1 > 1 \wedge n_2 = 1$ **then**
7 : **return** $(MT[H].\texttt{Build}(\mathcal{D}_1[0 : k_1]) || MT[H].\texttt{Build}(\mathcal{D}_1[k_1 : n_1]), \mathcal{D}_2[0])$
8 : **elseif** $n_1 > 1 \wedge n_2 > 1$ **then**
9 : **if** $(MT[H].\texttt{Build}(\mathcal{D}_1[0 : k_1]) \neq MT[H].\texttt{Build}(\mathcal{D}_2[0 : k_2])) \vee$
10 : $((MT[H].\texttt{Build}(\mathcal{D}_1[k_1 : n_1]) \neq MT[H].\texttt{Build}(\mathcal{D}_2[k_2 : n_2])))$ **then**
11 : **return** $(MT[H].\texttt{Build}(\mathcal{D}_1[0 : k_1]) || MT[H].\texttt{Build}(\mathcal{D}_1[k_1 : n_1]),$
12 : $MT[H].\texttt{Build}(\mathcal{D}_2[0 : k_2]) || MT[H].\texttt{Build}(\mathcal{D}_2[k_2 : n_2]))$
13 : **elseif** $\mathcal{D}_1[0 : k_1] \neq \mathcal{D}_2[0 : k_2]$
14 : **return** $\mathcal{F}(\mathcal{D}_1[0 : k_1], \mathcal{D}_2[0 : k_2])$
15 : **elseif** $\mathcal{D}_1[k_1 : n_1] \neq \mathcal{D}_2[k_2 : n_2]$
16 : **return** $\mathcal{F}(\mathcal{D}_1[k_1 : n_1], \mathcal{D}_2[k_2 : n_2])$

FIGURE 5.16: The $\mathcal{F}$ algorithm

updates the internal nodes' list with the new nodes added to accommodate the newly added data item and updates the rest of the internal nodes accordingly in addition to the verification object. It outputs the updated verification object VO', the updated list of internal nodes IO'_i, and the data object $DO[i]$ representing the data value d.

- $(VO', IO'_i, DO[i]') \leftarrow \texttt{Update}(VO, IO_i, d'_i, i, PolicyFile)$: This algorithm updates the structure by assigning a new value d'_i to position i. It takes as input the

verification object VO, the list of intermediate objects IO_i related to i, the new value d'_i to be assigned to position i, the index of the item to be updated i, and the policy file *PolicyFile*. The algorithm starts by verifying the list of the intermediate objects against the verification object, it exits with an error if the verification fails. If the verification passes, the algorithm extracts the current secure compression function from *PolicyFile*, updates the value of position i with the new data value d'_i, calculates the new values for all intermediate objects IO'_i and the new verification object value VO' using the current secure function. It outputs the updated verification object VO', the updated list of intermediate objects IO'_i related to i and the updated data object $DO[i]'$.

- $\{0,1\} \leftarrow \texttt{Verify}(VO, IO_i, d_i, i, PolicyFile)$: This algorithm verifies whether a data value d_i represents the correct value of item i. It takes as input the verification object VO, the list of intermediate objects IO_i related to item i, and d_i, the data value for i, and the *PolicyFile* file. It outputs 1 for verification success or 0 for failure.
- $(VO, IO, DO) \leftarrow \texttt{Build}(\mathcal{D}, PolicyFile)$: This algorithm builds a complete *esStruct* from an ordered list of data values using the current secure function extracted from *PolicyFile*. It takes the list of data values $\mathcal{D}$ and the *PolicyFile* file as input and outputs the VO, IO and DO forming the *esStruct*. It uses a simple approach to build the tree by calling the `Add` algorithm for each data value in $\mathcal{D}$.
- $(PolicyFile', VO') \leftarrow \texttt{Evolve}(PolicyFile, F_{new}, VO_l, VO_r)$: This algorithm evolves the structure by setting the current compression function to F_{new} and recalculating the VO using F_{new}. It takes as input the *PolicyFile*, the new secure function F_{new} and the root's two children VO_l and VO_r. It outputs the updated *PolicyFile* file and the updated VO'.

5.2.1 Threat Model

Our threat model considers the adversary to be active, computationally bound and is able to obtain a copy of the Hybrid Merkle tree. The goal of the adversary is to alter the contents of the archive protected by a Hybrid Merkle tree without the data collector detecting it through integrity checks. The model is chosen to address a common real-life attack scenario on archiving systems and their adversarial goal of altering its data.

5.2.2 Limitations

The conditions and timing for rebuilding the parts of the tree that were built using insecure hash functions are not covered in this work.

5.2.3 Security Property

For *esStruct* to be secure, it must not allow a data object $DO[i]$ generated by a currently deemed secure compression function to be successfully verified unless its corresponding data value d_i belongs to the ordered list $\mathcal{D}$. Proving that *sStruct* is *unforgeable* is not simple since we have to accommodate for the update functionality of the structure and the fact that it is built using multiple compression functions. Similar to what we did with *sStruct* in Section 5.1.1, we define the *unforgeable* security property for *esStruct* through two experiments, $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$ runs in a real setup and $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$ runs in an ideal setup where it tracks all the correct objects in its memory. The experiments for the *esStruct* are different from the *sStruct* ones due to the fact that *esStruct* utilizes multiple compression functions at the same time, some of them might be insecure at a given time. We accommodate this case by challenging the adversary to forge one of the functions from the secure set. We keep

track of which data objects have been generated by a secure compression function in D_{sec} and their related internal objects in IO_E, and we challenge the adversary to forge a data object belonging to this secure objects' list.

Experiment $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$ and its oracles are shown in Figure 5.17 and experiment $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$ and its oracles are shown in Figures 5.18, 5.19, 5.20, 5.21 and 5.22. The security experiment $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$ lets the adversary operate in a real environment where the experiment does not keep track of all objects belonging to *esStruct* but rather uses VO for verification. $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$ on the other hand lets the adversary operate in an ideal environment where the experiment keeps track of all objects belonging to *esStruct*. Running these two experiments shows if and how the adversary behaves differently when running in a real setup of our structure versus an ideal one where they cannot win. If the adversary does not behave differently, then the structure is secure. If the adversary behaves differently, this means they were able to find a collision.

The security experiments $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$ and $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$ provide the adversary with the following oracles:

- $\mathsf{OAdd}(IO_i, d)$: Allows the adversary to add a new data item to the structure. It takes as input the list of intermediate objects connecting the new node to the root IO_i, and the data value to be added d. In $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$, the oracle uses the `Add` algorithm to add the data item to the structure with no additional verification. In $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$, the oracle verifies first if the provided list of internal nodes IO_i matches what is stored in the oracle memory. It fails the experiment if they do not match. If the two lists match, the oracle continues in the same manner as $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$.

- $\mathsf{OUpdate}(IO_i, d'_i, i)$: Allows the adversary to update the value of a data item and update the structure accordingly. It takes as input the list of intermediate objects IO_i related to i, the new value d'_i to be assigned to data item i, and the index of the item to be updated i. In $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$, the oracle uses the `Update` algorithm to update the data item in the structure with no additional verification. In $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$, the oracle verifies first if the provided internal nodes list IO_i matches what is stored in the oracle memory. It fails the experiment if they do not match. If the two lists match, the oracle continues in the same manner as $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$.

- $\mathsf{OVerify}(IO_i, d_i, i)$: Allows the adversary to verify a data item. In $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$, the item is verified using the internal objects IO_i, provided by the adversary, and the verification object VO provided by the experiment. In $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$, the item is verified using the internal objects IO_{E_i}, stored by the experiment, and the verification object VO provided by the experiment.
- $\mathsf{OEvolve}$ (): Allows the adversary to evolve *esStruct*.

In $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$, we keep track of the correct values of nodes and leaves in ordered lists. IO_E stores the internal nodes objects, IO_{sec} stores the internal nodes generated by secure compression functions, IO_{insec} stores the internal nodes generated by insecure compression functions, D_{sec} stores the leaves generated by secure compression functions, and D_{insec} stores the leaves generated by insecure compression functions.

$\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$

1 : $VO \leftarrow \perp$
2 : $b \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{OAdd},\mathsf{OUpdate},\mathsf{OVerify},\mathsf{OEvolve}}()$
3 : **return** b

$\mathsf{OAdd}(IO_i, d)$

1 : $(VO, IO_i, DO[i]) \leftarrow\!\!\$\ \texttt{Add}(VO, IO_i, d)$
2 : **return** $(IO_i, DO[i])$

$\mathsf{OUpdate}(IO_i, d'_i, i)$

1 : $(VO, IO_i) \leftarrow\!\!\$\ \texttt{Update}(VO, IO_i, d'_i, i)$
2 : **return** $(IO_i, DO[i])$

$\mathsf{OVerify}(IO_i, d_i, i)$

1 : **return** $\texttt{Verify}(VO, IO_i, d_i, i)$

$\mathsf{OEvolve}()$

1 : $(PolicyFile, VO) \leftarrow \texttt{Evolve}(PolicyFile, F_{new}, VO)$
2 : **return** $\perp$

FIGURE 5.17: *esStruct unforgeable* Real Security experiment and oracles

$\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$

1 : $D_{E_{sec}}, D_{insec}, IO_E, IO_{sec}, IO_{insec} \leftarrow []$
2 : $b \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{OAdd},\mathsf{OUpdate},\mathsf{OVerify},\mathsf{OEvolve}}()$
3 : **return** b

FIGURE 5.18: *esStruct unforgeable* Ideal Security experiment

5.2.4 Hybrid Merkle Tree as an *esStruct*

We present now $HMT[H]$, an *esStruct* instantiation using a Merkle tree and a set of hash functions $\mathcal{H}$. The hash functions are used to generate the data digests DOs from the data values belonging to $\mathcal{D}$. The $HMT[H]$ is a hybrid Merkle tree system where VO is the root of the tree r_T, the intermediate objects IOs are the tree internal nodes and the data objects DOs are the tree leaves.

$\mathsf{OAdd}(IO_i, d)$

1 : $IO_i \leftarrow \texttt{GetIO}(IO, |D|)$
2 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO_E, |D|)$
3 : $sec \leftarrow 0$
4 : **if** (**for** each $k = 0$ to $|IO_{E_i}|$: $IO_{E_i}[k] = IO_{sec}[k]$) **then**
5 : // Check if the nodes provided by $\mathcal{A}$ matches the ones kept by the experiment
6 : **if** $IO_i \neq IO_{E_i}$ **then return** $\perp$
7 : $sec \leftarrow 1$
8 : $(VO, IO, DO) \leftarrow \texttt{Add}(VO, IO, DO, d, PolicyFile)$
9 : $i = |D|, D[i] \leftarrow d$
10 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO, i)$ // Store the new values in the experiment
11 : **if** $sec \leftarrow 1$ **then**
12 : **for** each node k in IO_{E_i}
13 : $IO_{sec}[k.index] \leftarrow k$
14 : $D_{sec}[k.index] \leftarrow k$
15 : **endfor**
16 : **return** $(IO_i, DO[i])$

FIGURE 5.19: Ideal Security experiment OAdd oracle for *esStruct unforgeable*

Similar to our Merkle tree implementation in Section 5.1, our hybrid Merkle tree implementation uses a balanced weighted binary tree and utilizes a supplementary AVL balanced tree structure for *leaves* to hold the values and locations of the original tree T leaves.

Hybrid Merkle trees are similar to Merkle trees with one difference, the hash values of their leaves and internal nodes are not all generated using the same hash function. Initially, the tree is generated using a single hash function H_a. New or updated leaves and nodes continue to be generated using H_a until a more secure hash function H_b emerges. The tree will adopt H_b to be used for the generation of new nodes and processing any updates for existing ones. The rest of the existing unchanged nodes keep their hash values generated by H_a.

$\mathsf{OUpdate}(IO_i, d'_i, i)$

1 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO_E, i)$
2 : $sec \leftarrow 0$
3 : **if** (**for** each $k = 0$ to $|IO_{E_i}|$: $IO_{E_i}[k] = IO_{sec}[k]$) **then**
4 : // Check if the nodes provided by $\mathcal{A}$ matches the ones kept by the experiment
5 : **if** $IO_i \neq IO_{E_i}$ **then return** $\perp$
6 : $sec \leftarrow 1$
7 : $(VO, IO_i) \leftarrow \texttt{Update}(VO, IO_i, d'_i, i)$
8 : $D[i] \leftarrow d_i$
9 : $IO_{E_i} \leftarrow IO_i$ // Store the new values in the experiment
10 : **if** $sec \leftarrow 1$ **then**
11 : **for** each node k in IO_{E_i}
12 : $IO_{sec}[k.index] \leftarrow k$
13 : $D_{sec}[k.index] \leftarrow k$
14 : **endfor**
15 : **return** $(IO_i, DO[i])$

FIGURE 5.20: Ideal Security experiment OUpdate oracle for *esStruct unforgeable*

$\mathsf{OVerify}(IO_i, d_i, i)$

1 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO_E, i)$
2 : **if** (**for** each $k = 0$ to $|IO_{E_i}|$: $IO_{E_i}[k] = IO_{sec}[k]$) **then**
3 : // Check if the nodes provided by $\mathcal{A}$ matches the ones kept by the experiment
4 : **if** $IO_i \neq IO_{E_i}$ **then return** $\perp$
5 : **return** $D[i] \leftarrow d_i$

FIGURE 5.21: Ideal Security experiment OVerify oracle for *esStruct unforgeable*

In our tree, we store the hash function identifier used to generate the node's hash value in *node.h*, a pointer to its parent *node.parent*, in addition to two pointers to each of its children $node.left$ and $node.right$, and finally *node.weight* which is the number of all leaves in its subtree. To avoid a possible attack where the adversary changes the hash function identifier *node.h* and compromises the verification process,

OEvolve()

1 : $PolicyFile \leftarrow \texttt{Evolve}(PolicyFile, F_{new}, VO_l, VO_r)$
2 : $D_{insec}.\texttt{Append}(D_{sec})$
3 : $IO_{insec}.\texttt{Append}(IO_{sec})$
4 : **return** $\perp$

FIGURE 5.22: Ideal Security experiment OEvolve oracle for *esStruct unforgeable*

in *node.hash*, each node stores the hash value of the concatenation of its two children nodes' hashes and the identifiers of the hash functions used to generate its two children hash values. To avoid possible collisions between leaves and subtrees of internal nodes, we separate their domains by concatenating 0 to each leaf and 1 to each internal node when we are calculating their hash values.

The algorithms for $HMT[H]$ are as follows:

- $(r'_T, inodes'_i, leaves[i]) \leftarrow \texttt{Add}(r_T, inodes_i, d, PolicyFile)$: This algorithm adds a new leaf to the tree corresponding to a data item d_i. It takes as input the tree root r_T, the internal tree nodes $inodes_i$ connecting the leaf i to the root, and the data value d. The algorithm starts by verifying the list of the internal nodes against the root, it exits with an error if the verification fails. If the verification passes, the algorithm extracts the current secure hash function from the *PolicyFile* then calculates the hash value of the new data item. Next, the algorithm adds the hash value as a new leaf to the tree after setting its weight to 0 to ensure it ends as a leaf in the tree. Next, it updates all internal nodes values connecting the new leaf to the root to reflect the newly added leaf through the `UpdateIO` function. It outputs the newly calculated tree root r'_T, the updated internal tree nodes $inodes'_i$, and the new leaf $leaves[i]$. The algorithm is shown

in Figure 5.23.

- $(r'_T, inodes'_i, leaves[i]) \leftarrow$ `Update`$(r_T, inodes_i, d'_i, i, PolicyFile)$: This algorithm updates the value of an existing leaf to reflect a change in the data value of a data item i from d_i to d'_i. It takes as input the tree root r_T, the internal tree nodes $inodes_i$ connecting the root to the leaf i, the new data value d'_i, the position i, and the *PolicyFile*. The algorithm starts by verifying the list of the internal nodes against the root, it exits with an error if the verification fails. If the verification passes, the algorithm extracts the current secure hash function from the *PolicyFile* then calculates the hash value of the new data value d'_i, calculates the updated leaf $leaves[i]$ and then updates all internal nodes values connecting this leaf to the root to reflect the new value. It outputs the updated root r'_T, the updated subset of nodes $inodes'_i$ connecting the leaf to the root and the updated leaf $leaves[i]$. The algorithm is shown in Figure 5.25.
- $0, 1 \leftarrow$ `Verify`$(r_T, inodes_i, d_i, PolicyFile)$: This algorithm verifies whether a data value d_i is the correct value in leaf i. It takes as input the tree root r_T, the list of internal nodes connecting the leaf i to the root of the tree $inodes_i$, the position i and the policy file *PolicyFile*. First, it locates the position of the item in the tree by using the standard AVL tree `Find`, then calculates the value of the tree root r'_T using the provided values of the internal nodes, the hash functions extracted from *PolicyFile* and the value d_i. The algorithm compares the calculated root to the provided root r_T and outputs 1 if the two roots match, that is, the verification succeeded, and 0 otherwise. The algorithm is shown in Figure 5.28.
- $(r_T, nodes) \leftarrow$ `Build`$(\mathcal{D}, PolicyFile)$: This algorithm builds the complete tree from a set of data values. It takes the set of data values $\mathcal{D}$ as input and outputs the root of the tree r_T and the internal nodes forming the tree. It builds the

tree by calling `Add` for each data value in the set $\mathcal{D}$.

- $(PolicyFile, r_T) \leftarrow$ `Evolve`$(PolicyFile, H_{new}, lchild, rchild)$: This algorithm evolves the $HMT[H]$ by setting the current hash function to a H_{new}. It takes as input the *PolicyFile* file, the new secure function H_{new} and the root's two children nodes *lchild* and *rchild*. It outputs the updated *PolicyFile* file and the new root value calculated by H_{new}. The algorithm is shown in Figure 5.29.

`Add`$(r_T, inodes_i, d, PolicyFile)$

```
1 :  if ¬(VerifyIO(r_T, inodes_i)) then Abort
2 :  H ← GetCurrentFunction(PolicyFile)
3 :  h ← H(0|d)
4 :  k = (|inodes_i|+1)/2
5 :  j = 0
6 :  for level = 1 ... k
7 :    if inodes_i[j].weight = 0
8 :      newdataindex = inodes_i[j].dataindex + 1
9 :      inodes_i, NewLeaf ← InsertNode(H, j, h, inodes_i, newdataindex)
10 :   else
11 :     if inodes_i[inodes_i[j].left].weight ≤ inodes_i[inodes_i[i].right].weight
12 :       j = inodes_i[j].left
13 :     else
14 :       j = inodes_i[j].right
15 :     endif
16 :   endif
17 : endfor
18 : r_T, inodes_i ← UpdateIO(inodes_i, j)
19 : return (r_T, inodes_i, NewLeaf)
```

FIGURE 5.23: The *esStruct* `Add` algorithm

5.2.5 Security Analysis

For a hybrid Merkle tree, built using a set of hash functions $\mathcal{H}$, we define the *unforgeable* security property as the ability of the tree to prevent a data object $DO[i]$, generated

`InsertNode`$(H, j, h, inodes_i, newdataindex)$

1 : $NewNode.parent = inodes_i[j].parent$
2 : $NewNode.left = j$
3 : $NewNode.right = j + 2$
4 : $NewNode.weight = 2$
5 : $NewNode.index = j + 1$
6 : $NewNode.dataindex = null$
7 : $NewNode.hash = H(1|inodes_i[NewNode.left].h|$
8 : $\qquad inodes_i[NewNode.left].hash|inodes_i[NewNode.right].h|h)$
9 : $inodes_i$.`Append`$(NewNode)$
10 : $NewLeaf.index = j + 2$
11 : $NewLeaf.dataindex = newdataindex$
12 : $NewLeaf.parent = NewNode.index$
13 : $NewLeaf.hash = h$
14 : **return** $(inodes_i, NewLeaf)$

FIGURE 5.24: The *esStruct* `InsertNode` algorithm

`Update`$(r_T, inodes_i, d'_i, i)$

1 : **if** $\neg$(`VerifyIO`$(r_T, inodes_i)$) **then** Abort
2 : $H \leftarrow GetCurrentFunction(PolicyFile)$
3 : $h \leftarrow H(0|H|d'_i)$
4 : $j \leftarrow$ `Find`(i)
5 : $inodes_i \leftarrow$ `UpdateIO`$(inodes_i, j, r_T, H)$
6 : **return** $(r_T, inodes_i, h)$

FIGURE 5.25: The *esStruct* `Update` algorithm

by a collision-resistant hash function $H_m \in \mathcal{H}$, to be successfully verified unless its corresponding data value d_i is part of the honest list $\mathcal{D}$, that is, $d_i = \mathcal{D}[i]$. This holds true even if the adversary is able to compromise all other hash functions in $\mathcal{H}$.

Lemma 2: If $H_m \in \mathcal{H}$ is collision resistant and used in generating leaf i, then $HMT[H]_i$ is *unforgeable*. $HMT[H]_i$ is the subtree from $HMT[H]$ containing all nodes connecting leaf i to the root.

`UpdateIO`$(inodes_i, j, r_T, H)$

1 : **while** $inodes_i[j].parent \neq 0$
2 : $l = inodes_i[inodes_i[j].parent].left$
3 : $H_l = inodes_i[l].h$
4 : $r = inodes_i[inodes_i[j].parent].right$
5 : $H_r = inodes_i[r].h$
6 : **if** $l \neq \emptyset \wedge r \neq \emptyset$
7 : $p = H_l|inodes_i[l].hash|H_r|inodes_i[r].hash$
8 : **if** $l \neq \emptyset \wedge r = \emptyset : p = H_l|inodes_i[l].hash$
9 : **if** $l = \emptyset \wedge r \neq \emptyset : p = H_r|inodes_i[r].hash$
10 : $inodes_i[inodes_i[j].parent].hash = H(1|p)$
11 : $inodes_i[inodes_i[j].parent].h = H$
12 : $j = inodes_i[j].parent$
13 : **endwhile**
14 : $r_T.hash = H(1|inodes_i[r_T.left].h|inodes_i[r_T.left].hash|$
15 : $inodes_i[r_T.right].h|inodes_i[r_T.right].hash)$
16 : **return** $(r_T, inodes_i)$

FIGURE 5.26: The *esStruct* `UpdateIO` algorithm

Proof of Lemma 2: If the adversary $\mathcal{A}$ can distinguish between $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$ and $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$, then the adversary $\mathcal{A}$ was able to find a collision by finding a data value that passes $\text{Exp}^{\mathsf{verify\text{-}real}}(\mathcal{A})$ and fails $\text{Exp}^{\mathsf{verify\text{-}ideal}}(\mathcal{A})$. The algorithm $\mathcal{B}^{\mathcal{A}}$ shown in Figure 5.30 finds the collision in H by identifying two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$ where $\mathcal{D}_1 \neq \mathcal{D}_2$, but $HMT[H](\mathcal{D}_1) = HMT[H](\mathcal{D}_2)$. $\mathcal{B}^{\mathcal{A}}$ identifies the collision through its oracles presented to $\mathcal{A}$ and the recursive algorithm $\mathcal{F}$ shown in Figure 5.16. $\mathcal{B}^{\mathcal{A}}$ oracles identify the collision event when $\mathcal{A}$ behaves differently during the two experiments. At this time, $\mathcal{B}^{\mathcal{A}}$ passes the data lists causing the collision to $\mathcal{F}$, which loops through the subtrees forming $\mathcal{D}_1$ and $\mathcal{D}_2$, bottom to top, until it finds the collision. □

The OAdd, OEvolve, OVerify, and OUpdate oracles for $\mathcal{B}^{\mathcal{A}}$ are shown in Figures

`Verify`$(r_T, inodes_i, d_i, i)$

1 : $j = inodes_i$.`Find`(i)
2 : $currnode \leftarrow inodes_i[j]$
3 : $H_p \leftarrow currnode.h$
4 : $H_l \leftarrow inodes_i[currnode.left].h$
5 : $H_r \leftarrow inodes_i[currnode.right].h$
6 : **if** $(i \mod 2 = 0)$ **then**
7 : $\quad inodes_i[currnode.index].hash \leftarrow H_p(0|H_l|inodes_i[currnode.left].hash|H_r|H_r(d_i))$
8 : **else**
9 : $\quad inodes_i[currnode.index].hash \leftarrow H_p(0|H_l|H_l(d_i)|H_r|inodes_i[currnode.right].hash)$
10 : $currnode.index \leftarrow currnode.parent$
11 : **while** $inodes_i[currnode.index].parent \neq 0$
12 : $\quad l \leftarrow inodes_i[currnode.index].left$
13 : $\quad r \leftarrow inodes_i[currnode.index].right$
14 : $\quad H_p \leftarrow currnode.h$
15 : $\quad H_l \leftarrow inodes_i[currnode.left].h$
16 : $\quad H_r \leftarrow inodes_i[currnode.right].h$
17 : $\quad inodes_i[currnode.index].hash \leftarrow H_p(1|H_l|inodes_i[l].hash|H_r|inodes_i[r].hash)$
18 : $\quad currnode.index \leftarrow currnode.parent$
19 : **endwhile**
20 : $r'_T \leftarrow currnode$
21 : **return** $(r_T = r'_T)$

FIGURE 5.27: The *esStruct* `Verify` algorithm

5.31, 5.32, 5.33 and 5.34, respectively. The code added to the basic oracles specifically for $\mathcal{B}^{\mathcal{A}}$ is color coded in blue.

5.3 Discussion

In this section, we discuss the impact of using hybrid Merkle trees in some real-life applications, such as archiving systems and certificate transparency logs. Generically, using a hybrid Merkle tree decreases the time and processing needed to rebuild a Merkle tree, when its hash function becomes insecure. During the lifetime of a hybrid Merkle tree, more advanced and secure hash functions get utilized in building and

`VerifyIO`$(r_T, inodes_i)$

```
1 :  j = inodes_i.FindLeaf()
2 :  currnode ← inodes_i[j]
3 :  currnode.index ← currnode.parent
4 :  while inodes_i[currnode.index].parent ≠ 0
5 :     l ← inodes_i[currnode.index].left
6 :     r ← inodes_i[currnode.index].right
7 :     H_p ← currnode.h
8 :     H_l ← inodes_i[currnode.left].h
9 :     H_r ← inodes_i[currnode.right].h
10 :    inodes_i[currnode.index].hash ← H_p(1|H_l|inodes_i[l].hash|H_r|inodes_i[r].hash)
11 :    currnode.index ← currnode.parent
12 : endwhile
13 : r'_T ← currnode
14 : return (r_T = r'_T)
```

FIGURE 5.28: The *esStruct* `VerifyIO` algorithm

`Evolve`$(PolicyFile, H_{new}, rchild, rchild)$

```
1 :  PolicyFile.Append(H_new, Secure)
2 :  PolicyFile.SetCurrToObsolete()
3 :  r_T = H_new(1|lchild.h|lchild.hash|rchild.h|rchild.hash)
4 :  return (PolicyFile, r_T)
```

FIGURE 5.29: The *esStruct* `Evolve` algorithm

updating parts of the tree. When a hash function is deemed insecure, only the nodes generated by it need to be regenerated, not the whole tree. Any update of a leaf value uses the most recent secure hash function and subsequently secures $log(n)$ internal nodes in the tree, including the root. Each new leaf added or a leaf value updated, secures more tree nodes by regenerating them using the most recent secure hash function.

$\mathcal{B}()$:

1 : $D \leftarrow []$
2 : $IO \leftarrow []$
3 : $IO_E \leftarrow []$
4 : $b \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{OAdd},\mathsf{OUpdate},\mathsf{OVerify},\mathsf{OEvolve}}()$

FIGURE 5.30: The $\mathcal{B}$ algorithm for *esStruct*

5.3.1 CT Logs

The CT logs hold the records for all certificates generated. The logs are used to verify authentic sites versus malicious ones. The CT logs currently contain around 1.9 billion records[1] and they are stored in a Merkle tree system. The system is append-only and no updates are allowed.

In the current implementation of CT logs, if the hash function used for this system becomes insecure, the whole tree structure needs to be rebuilt which requires a long time and a sizeable amount of processing. Additionally, during this build process, the system is insecure. Utilizing a hybrid Merkle tree system will reduce the number of nodes to be regenerated in such cases to be only the ones generated by the compromised hash function and the system is only partly insecure during the nodes update.

The drawback of utilizing our system in the CT logs application is the need to maintain a *PolicyFile* and add the hash function identifier to the node structure.

5.3.2 Digital Archiving Systems

Digital archiving systems need a mechanism to ensure the authenticity and integrity of the archived files. One way of achieving this goal is to implement a Merkle tree where the leaves are the hash values of the archived files, as described in the previous

[1] `https://sslmate.com/labs/ct_growth/`

OAdd(IO_i, d)

```
1 :  IO_i ← GetIO(IO, |D|)
2 :  IO_Ei ← GetIO(IO_E, |D|)
3 :  sec ← 0
4 :  if (for each k = 0 to |IO_Ei|: IO_Ei[k] = IO_sec[k]) then
5 :     if IO_i ≠ IO_Ei ∧ Verify(VO, IO_i, ∅, |DO|) then
6 :        Exit to B (F(IO_Ei, IO_i))
7 :     endif
8 :     sec ← 1
9 :  endif
10 :  (VO, IO, DO[i]) ← Add(VO, IO_i, d, PolicyFile)
11 :  i ← |D|
12 :  D[i] ← d_i
13 :  IO_Ei ← GetIO(IO, i)
14 :  if sec ← 1 then
15 :     for each node k in IO_Ei
16 :        IO_sec[k.index] ← k
17 :     endfor
18 :  endif
19 :  return (IO_i, DO[i])
```

FIGURE 5.31: The $\mathcal{B}$ OAdd oracle for *esStruct*

chapter. When a user requests verification of a file, the system uses the hash value of the file to verify if it will generate the root value stored with the archive as the verification value.

In such systems, to ensure the integrity of the archived files in the long term, the utilized Merkle tree must be kept secure by ensuring that its hash function is secure. Due to advancements in computational power and cryptanalysis techniques, the hash function might become insecure. To overcome this problem, digital archiving systems utilize multiple Merkle trees built with different hash functions to ensure the integrity is never compromised. In case a hash function becomes insecure, the

OEvolve()

1 : $PolicyFile \leftarrow \texttt{Evolve}(PolicyFile, F_{new}, VO_l, VO_r)$
2 : $D_{insec}.\texttt{Append}(D_{sec})$
3 : $IO_{insec}.\texttt{Append}(IO_{sec})$
4 : $D_{sec} \leftarrow []$
5 : $IO_{sec} \leftarrow []$
6 : **return** $\perp$

FIGURE 5.32: The $\mathcal{B}$ OEvolve oracle for *esStruct*

OVerify(IO_i, d_i, i)

1 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO_E, i)$
2 : **if** (**for** each $k = 0$ to $|IO_{E_i}|$: $IO_{E_i}[k] = IO_{sec}[k]$) **then**
3 : **if** $(IO_i \neq IO_{E_i}) \wedge \texttt{Verify}(VO, IO_i, d_i, i)$ **then**
4 : Exit to $\mathcal{B}$ $(\mathcal{F}(IO_{E_i}, IO_i))$
5 : **endif**
6 : **endif**
7 : **return** $D[i] = d_i$

FIGURE 5.33: The $\mathcal{B}$ OVerify oracle for *esStruct*

corresponding tree is discarded and a new one is rebuilt using a currently deemed secure hash function. During this time, the integrity of the system is not fully secured.

Utilizing a hybrid Merkle tree eliminates the need to have multiple Merkle trees and decreases the time needed to update the nodes generated by an insecure function. When one of the hash functions utilized in the hybrid Merkle tree becomes insecure, only the nodes generated by this function need to be regenerated by the most recent secure hash function.

5.3.3 ArchiveSafe LT: Case Study

In this section, we study the effect of using a hybrid Merkle tree in the ArchiveSafe LT system presented in Chapter 4. ArchiveSafe LT uses two Merkle trees generated

$\mathsf{OUpdate}(IO_i, d_i, i)$

1 : **if** $H(d_i) = H(d'_i) \wedge d_i \neq d'_i$ **then**
2 : Exit to $\mathcal{B}$ (d_i, d'_i)
3 : **endif**
4 : $IO_{E_i} \leftarrow \texttt{GetIO}(IO_E, i)$
5 : $sec \leftarrow 0$
6 : **if** (**for** each $k = 0$ to $|IO_{E_i}|$: $IO_{E_i}[k] = IO_{sec}[k]$) **then**
7 : **if** $IO_i \neq IO_{E_i} \wedge \texttt{Verify}(VO, IO_i, d'_i, i)$ **then**
8 : Exit to $\mathcal{B}$ $(\mathcal{F}(IO_{E_i}, IO_i))$
9 : **endif**
10 : $sec \leftarrow 1$
11 : **endif**
12 : $(VO, IO_i) \leftarrow \texttt{Update}(VO, IO_i, d_i, i)$
13 : $D[i] \leftarrow d_i$
14 : $IO_{E_i} \leftarrow IO_i$
15 : **if** $sec \leftarrow 1$ **then**
16 : **for** each node k in IO_{E_i}
17 : $IO_{sec}[k.index] \leftarrow k$
18 : **endfor**
19 : **endif**
20 : **return** $(IO_i, DO[i])$

FIGURE 5.34: The $\mathcal{B}$ OUpdate oracle for *esStruct*

using two different hash functions to ensure the system is resilient to the failure of any of the hash functions. Utilizing a hybrid Merkle tree eliminates the need to maintain multiple Merkle trees, which simplifies some of the system's maintenance processes such as the integrity evolution, and eliminates the space and maintenance required for keeping the second tree. The integrity evolution process in the current implementation of ArchiveSafe LT involves rebuilding a new full Merkle tree with a currently deemed secure hash function, but by utilizing a hybrid Merkle tree, the integrity evolution process will be reduced to rebuilding only the nodes generated by the insecure function and switching the tree to use a newer more secure hash function going forward.

Utilizing a hybrid Merkle tree simplifies the structure of the integrity verification object I_v to be the root of one tree and the integrity data object I_d which will consist of the nodes of one tree instead of two. On the other hand, it adds a slight complexity of maintaining the *PolicyFile*. This change reduces the amount of data exchanged between the data collector and the storage provider during the update, evolve and delete processes, since the size of I_d will be cut in half. For the same reason, the time and processing needed to update I_d during the update, evolve and delete processes will also be cut in half.

We utilized a hybrid Merkle tree in ArchiveSafe LT and reran the file update experiment. The results showed improvement in performance as expected since we are updating one tree instead of two. The new results show 42% - 48% reduction in tree updates processing time compared to the original results. The reason the reduction was not 50% even though we cut the number of trees in half, is that in the Hybrid Merkle tree case, there is the slight overhead of maintaining the *PolicyFile*.

Number of Files	100	1,000	1,000,000
Hybrid Merkle Tree	0.23ms	0.30ms	0.51ms
Merkle Tree	0.41ms	0.54ms	0.98ms

TABLE 5.1: Tree update time comparison for one node change

Chapter 6

Conclusion

In this chapter, we summarize the thesis contributions and present future directions for research.

6.1 Summary of Contributions

This thesis is focused on addressing two main challenges facing digital archiving systems, mass data breaches and long-term security. We had two objectives: 1) finding a solution to the mass data breaches challenge without the need for managing keys or the risk of depending on a single master key, and 2) finding a solution to support long-term confidentiality and integrity of digital archiving systems, with better performance and lower cost and complexity compared to the current state-of-the-art systems.

6.1.1 Mass Data Leakage

Our proposed mass-leakage resistant system ArchiveSafe, achieves the goal of preventing mass data breaches without the need to store any keys. ArchiveSafe's performance

is acceptable according to conventional users' standards. The system adds a 140–520 ms overhead when writing a file, and a customizable overhead when reading a file, ranging from 510 ms to 110 seconds depending on the difficulty level chosen. The overhead is acceptable for honest users who are accessing one or few files at a time but it is strongly hindering an adversary aiming for mass data acquisition. Different difficulty levels could be set to different groups of files depending on how sensitive is the information they contain. Difficulty levels settings is configurable and left to the system administrators to adopt based on their needs.

6.1.2 Long-Term Security

Our proposed archiving system ArchiveSafe LT achieves the goal of providing long-term confidentiality and integrity while eliminating the high cost and complexity needed by the current state-of-the-art systems. It also outperforms these systems and significantly reduces the storage space needed for archiving. These advantages over state-of-the-art systems comes at the cost of sacrificing information-theoretic security for computational assumptions, but we mitigate that by utilizing robust combiners and the novel evolution protocol.

ArchiveSafe LT requires only 14% to 33% of the time needed by current systems to process the same archives' sizes and utilizes less than one-third of the storage space required by these systems. The system's performance and space utilization improves significantly with larger file sizes.

6.1.3 Succinct Updatable Proof Structure

Building on our work on the long-term secure archiving system, we introduced a new evolving succinct updatable proof data structure, *esStruct*, that can be used to hold

the integrity information for archiving systems. *esStruct* is updated in $O(log(n))$ time without the need to rebuild it from scratch. We instantiate it as a Hybrid Merkle tree and prove the security of this instantiation. The structure is capable of evolving to a secure scheme if the scheme it uses becomes insecure.

We presented a more efficient version of ArchiveSafe LT that utilizes a Hybrid Merkle tree for long-term integrity. We measured the performance compared to the original implementation and showed 42% - 48% reduction in tree updates processing time.

6.2 Future Work

There are multiple directions for future research based on the work done in this thesis for digital archiving systems.

For the mass data breach protection area, future research should investigate how to make ArchiveSafe adaptable to the user's behavior. ArchiveSafe should be able to increase the puzzles' difficulty if a user is acting suspiciously, for example, trying to access large number of files in a short time span or using an untrusted hosting service. The system should increase the puzzles' difficulty of the files once it detects a suspicious behavior. The increase in difficulty should be dynamic, it should increase proportionally with how suspicious the user's behavior is until it reaches a level of difficulty that is impossible for an attacker to obtain large number of files. Since this difficulty increase will affect the honest users, it should be temporary until the system shuts down the attacker. The research goals should be to:

- Identify and formulate the different types of suspicious users' behavior.

- Develop methods to detect such behaviors.
- Develop appropriate system responses to such behaviors.

For the long-term security area, one direction is to incorporate updatable encryption [7] [21] in ASLT-D2 to eliminate the risk of sharing a key with the storage provider. This leads to more secure storage outsourcing in the cloud. In updatable encryption schemes, an *update key* k_Δ could be generated using two keys k_a and k_b. k_Δ could be used with the scheme's `Update` function to change the encryption key for a ciphertext from k_a to k_b without the need to obtain either. Utilizing updatable encryption will allow the data collector in case of a compromised key, to send the storage provider only the k_Δ to perform the evolution. Throughout the archive life-cycle, there will never be a time where the storage provider possesses the full keys' set needed to completely decrypt the archive. The possible future research goals would be to:

- Investigate how updatable encryption could be best utilized in ArchiveSafe LT.
- Update the ArchiveSafe LT protocols to incorporate updatable encryption.
- Prove the security of the updated system.

For the Hybrid Merkle tree, two of the main remaining open questions are 1) when to update the nodes that were generated using insecure hash functions, and 2) how to efficiently update them. The possible future research goals would be to:

- Identify events or hash functions utilization ratios that should trigger an update for the nodes generated by insecure hash functions.
- Develop an efficient algorithm to update these nodes using the most recent secure hash function with minimal processing and storage overhead.

Bibliography

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. *ACM Transactions on Internet Technology* (2005), 299–327.

[2] R. Alleaume, F. Roueff, E. Diamanti, and N. Lütkenhaus. Topological optimization of quantum key distribution networks. *New Journal of Physics* (2009), 075002.

[3] T. Aura, P. Nikander, and J. Leiwo. DOS-resistant authentication with client puzzles. In: *Security Protocols: 8th International Workshop Cambridge*. 2000, 170–177.

[4] G. Becker. Merkle signature schemes, Merkle trees and their cryptanalysis. *Ruhr-University Bochum, Technical Report* (2008), 19.

[5] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In: *IEEE European Symposium on Security and Privacy*. 2016, 292–302.

[6] M. Blaze. A cryptographic file system for UNIX. In: *ACM Conference on Computer and Communications Security*. 1993, 9–16.

[7] D. Boneh, K. Lewi, H. Montgomery, and A. Raghunathan. Key homomorphic PRFs and their applications. In: *Advances in Cryptology–CRYPTO*. 2013, 410–428.

[8] J. Braun, J. Buchmann, D. Demirel, M. Geihs, M. Fujiwara, S. Moriai, M. Sasaki, and A. Waseda. LINCOS: A Storage System Providing Long-Term Integrity, Authenticity, and Confidentiality. In: *ACM on Asia Conference on Computer and Communications Security*. 2017, 461–468.

[9] J. Braun, J. Buchmann, C. Mullan, and A. Wiesmaier. Long term Confidentiality: A survey. *Designs, Codes and Cryptography* (2014), 459–478.

[10] J. Buchmann, G. Dessouky, T. Frassetto, Á. Kiss, A.-R. Sadeghi, T. Schneider, G. Traverso, and S. Zeitouni. SAFE: A Secure and Efficient Long-Term Distributed Storage System. In: *ACM International Workshop on Security in Blockchain and Cloud Computing*. 2020, 8–13.

[11] A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management using undeniable attestations. In: *ACM Conference on Computer and Communications Security*. 2000, 9–17.

[12] Cashapp.com. *Over 8 million Cash App users possibly affected by data breach from a former employee*. `https://www.usatoday.com/story/money/2022/04/06/cash-app-data-breach/9490327002`. 2022.

[13] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The design and implementation of a Transparent Cryptographic File System for UNIX. In: *USENIX Annual Technical Conference*. 2001, 10–3.

[14] Crypto.com. *Crypto.com Admits 35 Dollars Million Hack*. `https://www.forbes.com/sites/thomasbrewster/2022/01/20/cryptocom-admits-35-million-hack/?sh=69a360357513`. 2022.

[15] D. Dean and A. Stubblefield. Using Client Puzzles to Protect TLS. In: *USENIX Security Symposium*. 2001.

[16] Y. Dodis and J. Katz. Chosen-ciphertext security of multiple encryption. In: *Theory of Cryptography Conference–TCC*. 2005, 188–209.

[17] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In: *Advances in Cryptology-CRYPTO*. 2003, 426–444.

[18] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In: *Advances in Cryptology—CRYPTO*. 1993, 139–147.

[19] eHealth. *Retention Policy: Electronic Health Record*. `https://ehealthontario.on.ca/files/public/support/EHR_Retention_Policy_EN.pdf?v=20201023`. 2016.

[20] S. Even and O. Goldreich. On the power of cascade ciphers. *ACM Transactions on Computer Systems–TOCS* (1985), 108–116.

[21] A. Fabrega, U. Maurer, and M. Mularczyk. A fresh approach to updatable symmetric encryption. *Cryptology ePrint Archive* (2021).

[22] I. Gassko, P. S. Gemmell, and P. MacKenzie. Efficient and fresh certification. In: *International Workshop on Public Key Cryptography*. 2000, 342–353.

[23] M. Geihs, N. Karvelas, S. Katzenbeisser, and J. Buchmann. PROPYLA: Privacy Preserving Long-term Secure Storage. In: *International Workshop on Security in Cloud Computing*. 2018, 39–48.

[24] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. *Johns Hopkins Information Security Institute, Technical Report* (2000).

[25] Google.com. *Certificate Transparency*. `https://sites.google.com/site/certificatetransparency/log-proofs-work`. 2023.

[26] A. Herzberg. On tolerant cryptographic constructions. In: *Topics in Cryptology–RSA*. 2005, 172–190.

[27] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In: *IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security–CMS*. 1999, 258–272.

[28] A. Juels and J. Brainard. Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks. In: *Network and Distributed Systems Security Symposium*. 1999.

[29] J. Kan and K. S. Kim. MTFS: Merkle-tree-based file system. In: *IEEE International Conference on Blockchain and Cryptocurrency–ICBC*. 2019, 43–47.

[30] P. C. Kocher. On certificate revocation and validation. In: *International conference on financial cryptography*. 1998, 172–177.

[31] S. Meier, B. Schmidt, C. Cremers, and D. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In: *International Conference on Computer Aided Verification*. 2013, 696–701.

[32] R. C. Merkle. *Secrecy, authentication, and public key systems*. 1979.

[33] Microsoft. *Microsoft Data Breach*. `https://msrc.microsoft.com/blog/2022/10/investigation-regarding-misconfigured-microsoft-storage-location-2/`. 2022.

[34] F. Moghimifar. Securing Database Using Client Puzzles. Master's report. Queensland University of Technology, 2015.

[35] P. Muth, M. Geihs, T. Arul, J. Buchmann, and S. Katzenbeisser. ELSA: efficient long-term secure storage of large datasets. *The European Association for Signal Processing–EURASIP* (2020), 1–20.

[36] R. Poddar, T. Boelter, and R. A. Popa. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive* (2016).

[37] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In: *ACM Symposium on Operating Systems Principles*. 2011, 85–100.

[38] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. *MIT, Technical Report* (1996).

[39] M. Sabry and R. Samavi. ArchiveSafe LT: Secure Long-term Archiving System. In: *Annual Computer Security Applications Conference–ACSAC*. 2022, 936–948.

[40] M. Sabry, R. Samavi, and D. Stebila. ArchiveSafe: Mass-Leakage-Resistant Storage from Proof-of-Work. In: *Data Privacy Management Workshop–DPM*. 2020, 89–107.

[41] A. Shamir. How to share a secret. *Communications of the ACM* (1979), 612–613.

[42] C. E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal* (1949), 656–715.

[43] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: A scalable cloud file system with efficient integrity checks. In: *Annual Computer Security Applications Conference–ACSAC*. 2012, 229–238.

[44] L. Vargas, G. Hazarika, R. Culpepper, K. R. Butler, T. Shrimpton, D. Szajda, and P. Traynor. Mitigating risk while complying with data retention laws. In: *ACM Conference on Computer and Communications Security–SIGSAC*. 2018, 2011–2027.

[45] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for DoS resistance. In: *ACM conference on Computer and communications security*. 2004, 246–256.

[46] C. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In: *USENIX Annual Technical Conference*. 2003, 197–210.

[47] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A Stackable vnode Level Encryption File System. *Computer Science Deptartment, Columbia University, Technical Report* (1998).