Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

(Secure?) Cloud Backup Solutions: A Survey

Bachelor Thesis

L. Schenck

August 28, 2023

Advisors: Prof. Dr. Kenny Paterson, Matteo Scarlata, Kien Tuong Truong

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

In this thesis, we analyze six popular cloud backup solutions. Cloud backup solutions are applications which aim to create secure backups of local data onto a remote server. Because many cloud backup solutions are open-source and claim strong security, they are attractive choices for many individuals and groups of all sizes looking to back up their data securely.

There exists literature on designing a secure cloud backup solution, but, to the best of our knowledge, all literature so far assumes a much weaker threat model than what we consider fitting. We define two threat models for cloud backup solutions, conduct a deep analysis of three cloud backup solutions Borg, EteSync and Tarsnap, and provide an overview of three more cloud backup solutions Kopia, Bupstash and Restic. Further, we describe deduplication, content-based chunking and its effect on side channel attacks, and define fingerprinting in the context of cloud backup solutions. Finally, we present multiple attacks on Borg, Kopia and Restic. More specifically, we present attacks that allow an attacker to confirm if a user has backed up a file or not, and we show how it is possible to extract a secret key in Borg.

Overall, we show that while the backup solutions successfully achieve data confidentiality (even with sub-optimal cryptographic designs), they often present a privacy risk due to the scarce resistance to fingerprinting attacks.

We conclude by discussing the current state of cloud backup solutions and recommend an algorithm which mitigates our attacks. We also comment on potential future research areas, and in particular advocate for conducting the same analysis as we have done for other cloud backup solutions and diving deeper into specific, complex content-defined chunking implementations.

Contents

Contents **iii**

1 Introduction **1**
1.1 Cloud Backups vs. Cloud Storage 1
1.2 Client-Server Model 2
1.3 Threat Model 2
1.4 Deduplication 3
1.5 Fingerprint Resistance 3
1.5.1 Fingerprinting 3
1.5.2 Fingerprint Resistance with Deduplication 4
1.5.3 Literature 4
1.6 Backup Solutions 4
1.6.1 Borg 5
1.6.2 EteSync 5
1.6.3 Tarsnap 6
1.6.4 Kopia 6
1.6.5 Bupstash 6
1.6.6 Restic 6
1.7 Contributions 6

2 Background **9**
2.1 Abbreviations 9
2.2 Notation 9
2.3 Cryptographic Zoo 10
2.3.1 Salsa20 and ChaCha20 10
2.3.2 AES-CTR 10
2.3.3 Poly1305 11
2.3.4 HMAC-SHA256 11
2.3.5 BLAKE2b 11
2.3.6 Ed25519 11

2.3.7 RSA-OAEP . . . 11
2.3.8 ChaCha20-Poly1305 . . . 12
2.3.9 AES-CTR & HMAC-SHA256 . . . 12
2.3.10 Diffie-Hellman . . . 12
2.3.11 PBKDF2 . . . 12
2.3.12 Argon2 . . . 12
2.3.13 HKDF-HMAC-SHA512 . . . 13
2.3.14 Scrypt . . . 13
2.4 SSH . . . 13
2.5 TLS . . . 13

3 Deduplication **15**
3.1 Chunking . . . 15
3.1.1 Content-defined Chunking . . . 16

4 Backup Systems Studies **19**
4.1 Borg . . . 19
4.1.1 Threat Model . . . 20
4.1.2 Key hierarchy . . . 20
4.1.3 Data Model . . . 22
4.1.4 Deduplication . . . 22
4.1.5 File Backup Process . . . 23
4.1.6 Own implementations . . . 23
4.2 EteSync . . . 23
4.2.1 Threat Model . . . 23
4.2.2 Key hierarchy . . . 24
4.2.3 Data Model . . . 24
4.3 Tarsnap . . . 25
4.3.1 Threat Model . . . 25
4.3.2 Key hierarchy . . . 25
4.3.3 File encryption . . . 25
4.3.4 File decryption . . . 27
4.3.5 Data Model . . . 27
4.3.6 Deduplication . . . 27
4.3.7 Analysis . . . 27
4.3.8 Countermeasures . . . 29
4.4 Kopia . . . 29
4.4.1 Threat Model . . . 29
4.4.2 Data Model . . . 29
4.4.3 Deduplication . . . 29
4.5 Bupstash . . . 29
4.5.1 Threat Model . . . 30
4.5.2 Data Model . . . 30
4.5.3 Deduplication . . . 30

4.6 Restic 30
4.6.1 Threat Model 31
4.6.2 Data Model 31
4.6.3 Deduplication 31

5 Fingerprinting Attacks on Chunking **33**
5.1 Buzhash 33
5.2 Rabin Fingerprint 35
5.3 Borg 36
5.3.1 Borg's Buzhash Implementation 36
5.3.2 Threat Model 36
5.3.3 Fixed Chunker 37
5.3.4 Extracting the Secret Seed 37
5.3.5 Proof of Concept Attack 40
5.3.6 Borg's Documentation 40
5.4 Tarsnap 41
5.5 Kopia 41
5.6 Bupstash 41
5.7 Restic 42
5.8 Mitigations 43

6 Conclusion **45**
6.1 Discussion 45
6.2 Future Work 46

A Appendix **49**
A.1 Borg Chunk Size Probability Analysis 49

Bibliography **51**

Chapter 1

Introduction

Data security is an ever-present topic for institutions and individuals. Many companies, organizations, and persons need backups of their data as a safety measure in case their data is lost or stolen. Saving backups on private servers, so-called "self-hosting", is an attractive solution for individuals and groups of smaller sizes. However, with increasing size and number of backups the cost of server hosting inflates exponentially, leading to many companies and organizations using public cloud hosting services such as Amazon Web Services and Cloudflare, known as "remote hosting". As the hosting services control the servers themselves, a malicious employee or a nation-state actor has the potential to gain access to the data hosted on the servers. The Snowden revelations showed that in the past governments have abused this potential to surveil what is otherwise considered private data [28]. To protect oneself from these threats several groups have published open-source cloud backup solutions which claim to provide security properties such as data confidentiality, data integrity, and resistance to fingerprinting attacks, even against an adversary with full access to the server. A user can use such a cloud backup solution in conjunction with a remote-hosted server to achieve "secure" cloud backups. Intuitively the security claims above are all minimal goals a self-proclaimed "secure" cloud backup system should aim to achieve. However, what does "secure" even mean? First, we must define what "security" means in the context of cloud backup solutions.

1.1 Cloud Backups vs. Cloud Storage

While it may seem that cloud storage and cloud backups are very similar, cloud backups are fundamentally a much simpler problem to solve. Cloud storage aims to offer a data storage service that is easily accessible and provides additional functionality such as collaboration with other users and

interactivity with the data, such as editing a text file. On the other hand, cloud backups prioritize simplicity and security. Backups are stored once and rarely accessed, and because data is seldom accessed there is no need for interactivity or collaboration. This simpler functional design implies a simpler cryptographic design, reducing the attack surface of a cloud backup system and thus improving its security.

1.2 Client-Server Model

We define a model for cloud backup solutions which consists of users and servers. Users have clients installed on their devices that can connect remotely to servers that hold backups of the users' data. The users have their clients periodically create backups, and in rare cases, retrieve a backup. The server can be self-hosted or remote-hosted with the user having full access to it, or the cloud backup solution may provide a server to which the user does not have full access. The server may contain code for handling requests from clients, or the client may connect to the server remotely and handle incoming requests itself.

1.3 Threat Model

So far no definition of threat models for cloud backup solutions has been standardized. There exists some consensus that in any threat model for cloud backup solutions, an adversary must have at least full access to the server for the reasons mentioned above. However, we consider this definition too weak in practice. In our opinion, this threat model is too passive in the sense that it assumes an adversary that can see the backup repository but not do or know anything else. The problem is that additional capabilities may arise in reality, that is: the ability to know some plaintext or the ability to inject plaintext.

We propose two definitions based on our analysis and findings. We differentiate between a *"weak adversary"* with full access to the server and the ability to know the contents of a specific file in the backup and a *"strong adversary"* with full access to the server and the ability to inject arbitrary files into the backup. This will allow us to show different security properties for the cloud backup systems that we will study.

More specifically, both adversaries not only have access to the data, but can also observe the content of all user requests, measure their timing, and arbitrarily modify the server's responses. The weak adversary additionally knows of a set of files contained in the victim's repository. This captures, for instance, the setting where the adversary and the victim share a git repository, an unencrypted cloud storage directory, or where the adversary knows

the operating system the victim is running, and the victim backs up operating system-specific files.

The strong adversary can additionally choose to include arbitrary files in the victim's repository. This captures the setting where the adversary can trick the victim into downloading files that will be included in the backup.

1.4 Deduplication

When a client creates multiple backups of a system the difference between a new backup and the previous backup is often very small. Almost always the amount of new data to be backed up is much less than the full size of the directory. In this case, one would like to only have to upload the new data, i.e. the difference between the old backup and the new backup. Also, one may have redundant files that are not exactly equal but share the majority of their content, for example, an image and its cropped version or multiple versions of a text file. To prevent already backed-up data from being wastefully backed up again cloud backup systems commonly implement a technique called deduplication. A local database contains information regarding what data has already been backed up, and upon uploading a new string of data the database is checked for potential duplicates. If a duplicate is found, the data is ignored and not uploaded. Otherwise, if a duplicate is not found, the file is processed and uploaded to the server.

Deduplication boosts the speed and efficiency of backup systems immensely. Instead of having to upload all the data in a directory again, the backup system can simply check which pieces of data are new and upload those. Some backup systems implement file-level deduplication while others use finer-grained "chunk"-level deduplication, where chunks are smaller parts of a file. File-level deduplication is easier to implement, but chunk-level deduplication is generally more efficient.

1.5 Fingerprint Resistance

While data confidentiality, data authenticity, and metadata authenticity are necessary for a secure cloud backup system they are by no means sufficient. Even if the contents of a backup are encrypted, an adversary may still discover a lot about the contents of a backup by analyzing metadata, ciphertext length, and information leaked by additional functionalities. Specifically, they may gain knowledge about what data was backed up.

1.5.1 Fingerprinting

We informally define fingerprinting in the context of cloud backup systems as "given a backup, and two messages of the same length, the ability to dif-

ferentiate between a previously backed up message and a random message with non-negligible probability". This definition attempts to capture any information obtainable by an adversary in the cloud backup system setting.

As a practical example, an adversary might be a nation-state seeking information about individuals possessing censored books. If the book is shared among censorship violators digitally, and the nation-state can successfully launch a fingerprinting attack on a cloud backup solution, the nation-state can confirm whether any given user of the cloud backup solution has backed up, and thus possesses the censored book.

1.5.2 Fingerprint Resistance with Deduplication

Deduplication by definition breaks fingerprint resistance under a strong adversary, as the adversary can inject an arbitrary file into the backup and check the server to confirm whether new data was added to the backup repository or not. If new data was added equal to the length of the file the adversary can confirm that the victim's backup repository does not contain any of the injected data. However, if the length of the new data that was added is less than the injected data's length, the adversary can conclude that some of the injected data was already present in the victim's backup.

1.5.3 Literature

Much research has been done on secure cloud backups and secure deduplication [2, 13, 19, 23, 29, 35]. Today's standards include ClouDedup [29] which uses the Convergent Encryption [13] encryption scheme based on a cryptographic primitive called Message-Locked Encryption [2], and DupLESS [19] which also uses Message-Locked Encryption and secures it against brute-force attacks.

One of the main desired security properties is that assuming independent users Alice and Bob, for any file F if Alice uploads F and Bob also uploads F after Alice, he cannot distinguish if a user has previously uploaded F or not. At the same time, efficient deduplication must be possible to reduce the required storage space for the cloud service provider.

However, in all of this literature, the threat model assumes a trusted server. There exists no literature for our threat model, which is surprising because cloud backup solutions have existed since the emergence of cloud services.

1.6 Backup Solutions

Here we introduce the backup solutions we evaluated, giving a brief overview of how they work and what functionalities they provide. In Chapter 4 we

go deeper and provide a deep analysis of Borg, EteSync, and Tarsnap, and analyze Kopia, Bupstash, and Restic superficially.

1.6.1 Borg

Borg [4] is an open-source backup application that creates backups in a "repository" stored in a certain path. Borg organizes backups in an abstract hierarchy of objects, where the lowest levels are data chunks, which are grouped into objects representing files, which are grouped into objects representing a "snapshot" of a file system at a point in time, which are grouped into the repository. In the typical deployment, the repository is stored on a remote machine. In this case, Borg has a simple server component, and the communication between the client and the server is protected at the transport level by SSH. To communicate with the server over SSH the Borg client connects to the server through a separate process. The Borg client provides deduplication and authenticated encryption of files, and SSH provides a (confidential and authenticated) link between the client and the server.

For Borg, we assume a weak adversary as we introduced in Section 1.3. One of our goals is to analyze fingerprint resistance for the cloud backup solutions that we study, and deduplication trivially breaks fingerprint resistance under a strong adversary as we explained in Section 1.5.2. This makes the weak adversary the more logical option for our threat model.

1.6.2 EteSync

EteSync is a notes and contact sync application built upon Etebase [15], where Etebase is an open source end-to-end encrypted "backend": a set of client libraries and a server for building end-to-end encrypted applications. We study EteSync as an end-to-end encrypted cloud backup solution. Etebase promises "battle-tested strong encryption using modern cryptography" [15]. Instead of creating snapshots of a file system when a user requests a backup, EteSync processes each file separately, meaning on the server backed-up files are not organized under snapshots of a file system, but individually. EteSync provides file-level deduplication but not chunk-level deduplication as we discussed in Section 1.4. The EteSync server can be self-hosted (or remote-hosted), but EteSync also offers access to their instances as a service. EteSync supports the sharing of "collections" of data with different users.

Even though EteSync does not provide chunk-level deduplication we still assume a weak adversary as defined in Section 1.3, the same threat model as for Borg.

1.6.3 Tarsnap

Tarsnap [36] is an online end-to-end encrypted backup service where Tarsnap hosts the server. Self-hosting is not supported. Tarsnap advertises itself as a “backup service for the truly paranoid” [36]. While the client code is publicly available, the server implementation is not publicly released. The Tarsnap client provides data deduplication.

We assume a weak adversary for Tarsnap as described in Section 1.3.

1.6.4 Kopia

Kopia [21] is an open-source end-to-end backup solution. Backups are snapshots of user-chosen files and directories which are uploaded to a self-hosted or remote-hosted server. The Kopia client provides encryption, compression, and deduplication, and claims to provide “end-to-end zero-knowledge encryption” [20].

For Kopia we assume a weak adversary as defined in Section 1.3.

1.6.5 Bupstash

Bupstash [11] is a simple snapshot-based open-source backup solution where files and directories are backed up to a self-hosted or remote-hosted server. Encryption is end-to-end and deduplication and compression are supported. The Bupstash client is currently in open beta and only recommended for redundant backups [10].

We assume a weak adversary as defined in Section 1.3 for Bupstash.

1.6.6 Restic

Restic [34] is a snapshot-based end-to-end backup solution. The user self-hosts or remote-hosts the server and the Restic client provides encryption and data deduplication. Restic claims to be “fast, efficient and secure” [33].

For Restic we assume a weak adversary as defined in Section 1.3.

1.7 Contributions

We first introduce and discuss deduplication and content-defined chunking in Chapter 3. In Chapter 4 we conduct a deep analysis of Borg, EteSync, and Tarsnap, and provide a short analysis of Kopia, Bupstash, and Restic. We present multiple attacks on Borg, Kopia, and Restic in Chapter 5 that allow an adversary to confirm if a user has backed up a file or not, and in the case of Borg, allow a secret key to be extracted.

Overall, we show that the cloud backup solutions that we study successfully provide data confidentiality (even with sub-optimal cryptographic designs), but also that they contain vulnerabilities to fingerprinting attacks which present a privacy risk to users.

Finally, we discuss mitigations for the vulnerabilities we found and advocate for further research in the development of secure content-defined chunking algorithms.

Chapter 2

Background

Before we evaluate the security of the cloud backup solutions we want to study we must introduce the notation we will use for the rest of this thesis. The definitions below for cryptographic primitives, encryption, and authentication schemes provide the base we need for our analysis.

2.1 Abbreviations

We use the following abbreviations in this thesis:

MitM	Man-in-the-middle
XOR	Exclusive or
Nonce	Number only used once
MAC	Message authentication code
HMAC	Hash-based MAC
AEAD	Authenticated encryption with associated data
EtM	Encrypt-then-Mac
E&M	Encrypt-and-Mac
MtE	Mac-then-Encrypt
IKM	Input key material
IV	Initialization vector
IND-CPA	Indistinguishability under chosen-plaintext attack
IND-CCA	Indistinguishability under adaptive chosen-ciphertext attacks
PRF	Pseudo-random function family
PRP	Pseudo-random permutation

2.2 Notation

We use standard notation in literature for equations and pseudocode in this thesis:

- $a||b$: the result of the concatenation of the strings a and b
- $a \oplus b$: the result of the XOR operation on bit strings a and b
- $s[i]$: assuming s is an array or a string, the i-th element of s
- $s[i:j]$: the substring of s from i (inclusive) to j (exclusive)
- $s[i:]$: the substring of s starting from i (inclusive)
- $s[:j]$: the substring of s from 0 to j (exclusive)

2.3 Cryptographic Zoo

Here we introduce and describe the cryptographic primitives used by the cloud backup systems that we study.

2.3.1 Salsa20 and ChaCha20

Salsa20 and ChaCha20 are very similar stream ciphers that take a nonce and a key as their input and output a random bit stream. This bit stream is often used for encrypting data by XOR'ing the plaintext with the bit stream. When used for encryption, since the output of stream ciphers is deterministic, a new nonce must be utilized every time a cipher is called to prevent two identical plaintexts resulting in identical ciphertexts. This would allow an adversary to draw relations between messages and, if the adversary knows a plaintext-ciphertext pair, to decrypt future messages. The usage of a secret key is essential for confidentiality, as an adversary with knowledge regarding the key can easily decrypt any message.

XSalsa20 and XChaCha20 are variants of Salsa20 and ChaCha20 respectively with extended nonces.

2.3.2 AES-CTR

AES-CTR is an encryption algorithm that uses the block cipher AES in CTR (counter) mode. AES-CTR takes a message, a key, and an initialization vector (IV) to which a counter is concatenated as an input, and outputs a random bit stream. This bit stream is then XORed with the message.

If the same key is used for two separate messages, the IV must be different for each message as re-using the IV would lead to two identical messages resulting in two identical ciphertexts. This would allow an adversary to learn the result of the XOR operation of the plaintexts, breaking IND-CPA security.

2.3.3 Poly1305

Poly1305 is a MAC scheme that takes a message and a key and outputs a keyed hash of the message. Anybody with knowledge of the message and the key can run Poly1305 on the above inputs and confirm the keyed hash. This allows a user to confirm the integrity of messages, as the probability of an adversary successfully forging a valid tag for a message is negligible.

2.3.4 HMAC-SHA256

HMAC-SHA256 is a HMAC [22] scheme which uses SHA256 as its hash function. SHA256 is a very popular hash function and many security protocols and applications use it today. HMAC-SHA256 hashes the message together with two separate keys that are both generated from the original key to generate secure authentication codes. These authentication codes can be used by anybody with knowledge of the message and the key to confirm the integrity of the message.

2.3.5 BLAKE2b

Blake2b [1] is a very popular cryptographic hash function based on ChaCha20 which takes a message, an optional key, and the desired hash length in bytes. Used with a key, BLAKE2b acts as an HMAC scheme and can be used for message authentication.

2.3.6 Ed25519

Ed25519 is a digital signature algorithm that generates a public key and a private key, takes a message, and generates a signature for that message using the generated private key. The public key can be shared (hence "public") and anybody with knowledge of the public key and the message can confirm that the signature was created by the holder of the corresponding private key.

2.3.7 RSA-OAEP

RSA-OAEP [18] is a combination of the public-key cryptography encryption algorithm RSA and the padding scheme OAEP. RSA by itself has many problems, such as the fact that it is a deterministic encryption algorithm that, for example, allows an eavesdropping adversary with access to an encryption oracle to test arbitrary plaintexts against a known ciphertext. OAEP fixes these problems, and RSA-OAEP has been proven to be IND-CCA secure.

2.3.8 ChaCha20-Poly1305

ChaCha20-Poly1305 is an AEAD scheme that uses ChaCha20 for encryption and Poly1305 for authentication of messages. The ciphertext is used in the computation of the authentication tag which makes ChaCha20-Poly1305 an encrypt-then-mac (EtM) scheme. ChaCha20-Poly1305 is one of the worldwide standards [26] for AEAD and many applications use it today.

2.3.9 AES-CTR & HMAC-SHA256

AES-CTR & HMAC-SHA256 is an E&M authenticated encryption scheme which uses AES-CTR for encryption and HMAC-SHA256 for authentication of messages. AES-CTR & HMAC-SHA256 being an E&M scheme means the ciphertext computed by AES-CTR is not used in the computation of the authentication tag, but the plaintext is used instead.

2.3.10 Diffie-Hellman

The Diffie-Hellman key exchange is a way of securely agreeing on a secret common value over a public (but authentic) communication channel for two entities. Both entities compute their personal public and private keys, and by exploiting the difficulty of the discrete logarithm problem they can both compute a secret without any adversary being able to learn of it, even if the adversary can eavesdrop on the entire communication.

2.3.11 PBKDF2

PBKDF2 is a key derivation function that takes a pseudorandom function, a password, a salt, the desired number of iterations, and the desired key bit length. It outputs a secret key which can be used for further cryptography. The number of iterations affects how easy it is to brute-force a key in the sense that higher numbers of iterations increase the amount of time required for each brute-force attempt. However, a password that is common and/or easy to guess is still easy to brute-force as it is still vulnerable to dictionary attacks.

2.3.12 Argon2

Argon2 is a key derivation function that takes a password, a salt (random data that makes brute-force attacks more difficult because it prevents precomputation and usage of rainbow tables), the desired degree of parallelism, the desired key bit length, the amount of computer memory to use, and the desired number of iterations. Argon2 requires a significant amount of computer memory to evaluate and access computer memory in an order dependent on the password. Argon2 is thus a “memory-hard function”,

making it resistant to FPGA and GPU-based cracking attacks because they have limited and expensive memory. However, the fact that Argon2 accesses computer memory in an order dependent on the password has resulted in side-channel attacks. Argon2i and Argon2id aim to mitigate these attacks, and thus Argon2id is the recommended variant.

Similarly to PBKDF2, a higher number of iterations increases the amount of computation required for brute-force attempts.

2.3.13 HKDF-HMAC-SHA512

HKDF-HMAC-SHA512 is a key derivation function that takes input key material (also called IKM, for example a password), a salt, an info string (some public information regarding the input), and the desired key bit length. The info string is useful when one, for example, wants to derive multiple keys from the same password with the property that each key is independent of the others.

2.3.14 Scrypt

Scrypt is a key derivation function originally developed for the Tarsnap cloud backup system. Scrypt first generates a salt using an expensive computation and then derives the key from PBKDF2 with HMAC-SHA256 with that salt. The design goal is to increase the amount of computation and thus time required to successfully execute a brute-force attack.

2.4 SSH

SSH [24] is a widely used network protocol that lets users log in to and execute commands on remote machines securely. Communication between the client (the user) and the server (the remote machine) is encrypted by the transport layer and user authentication is provided by the server. SSH also provides secure file transfers, which is especially useful for cloud services.

2.5 TLS

TLS [31] is a security protocol that provides data confidentiality, data integrity, and user authentication. It is used for email, messaging, voice over IP (VoIP), and most commonly in the encryption between end user and website. HTTPS is the main web protocol used today and is a combination of HTTP and TLS, where HTTP provides the communication channel between the end user and the website and TLS provides the security properties above.

Chapter 3

Deduplication

Users create backups of their systems at different points in time, and often there will be overlaps between the files in different backups. Additionally, files can be very similar to each other. This implies that there are likely to be duplicates among the data that is uploaded. Avoiding unnecessary duplicates in backups results in much faster, more space-efficient, and thus less expensive backups. Almost every cloud backup solution that we study works on a snapshot basis, which means that every time we create a backup, we upload *all* the files in the snapshot to the server. This implies a lot of potentially wasteful re-uploading of already backed-up data. For this reason, all snapshot-based cloud backup solutions we study provide deduplication in one form or another. However, implementing deduplication is not a simple task and requires careful consideration of multiple trade-offs. A naive approach may be to simply hash every backed-up file on the server and keep a list of all file hashes on the server. This, however, would be inefficient as every file still needs to be sent to the server, even if the file has already been backed up. The solution to this problem is to do the same hashing as before but keep the list of file hashes locally on the client. But what if we only change a single bit in a file? Do we need to back up the whole file again?

This is where chunking comes in.

3.1 Chunking

Chunking is a method that splits files into data smaller pieces of data called "chunks". These chunks can later easily be put together again to reconstruct a file, e.g. by the backup repository holding a list of pointers to chunks for every file. A chunking algorithm typically determines at what point a chunk is cut by scanning a file from left to right until a condition is met.

This allows us to perform deduplication on smaller pieces of data which

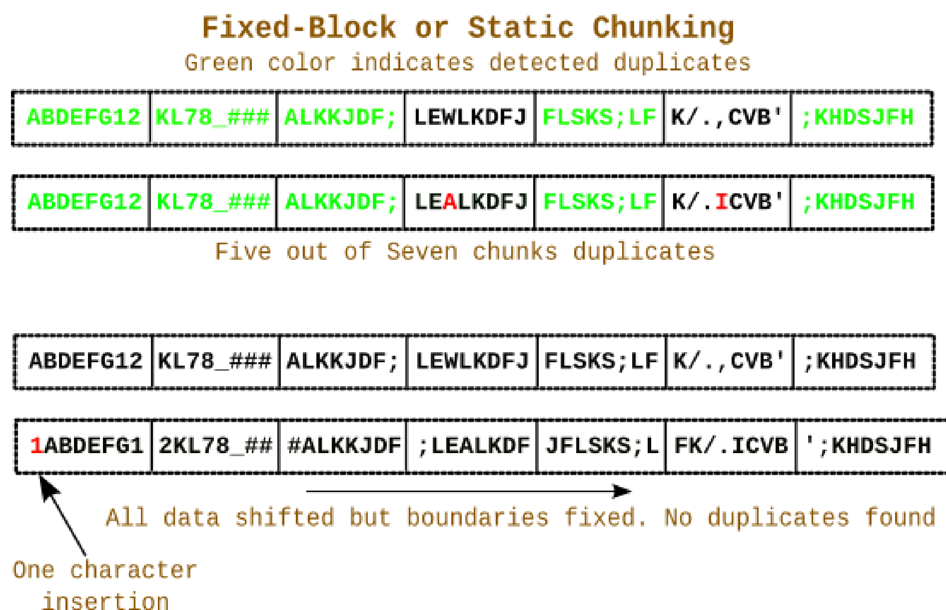


Figure 3.1: Fixed Chunking [25]

aids efficiency. For example, assume we have a file of size 2 GB which we already backed up once and our cloud backup solution allows a maximum chunk size of 2 MB. If we flip a single bit in this file and try to back it up again, we only have to send one chunk of a maximum of 2 MB to the server since all the other chunks are unchanged. Compared to sending the whole file again this is a reduction of 99.9%!

A chunking algorithm that simply chunks every k bytes works for us in this case, but if we *add* or *delete* a bit or series of bits anywhere in the file all subsequent chunks will be different, and we end up having to send many more chunks than necessary to the server again (see Figure 3.1). For this reason, many cloud backup solutions implement content-defined chunking.

3.1.1 Content-defined Chunking

Content-defined chunking is defined by chunking based on the actual data in the chunk. Content-defined chunking algorithms look for a *local* condition when deciding whether to perform a cut, ensuring that a change in previous bytes will not affect the decision to chunk in a specific position. In other words, a single byte addition does not cause *global* repercussions on the chunks (see Figure 3.2). This increases deduplication efficiency and is thus a very popular method. All snapshot-based cloud backup solutions that we study use chunking algorithms that are content-defined.

The most common way to implement content-defined chunking is to implement an algorithm that takes a string of data, defines a window with a fixed

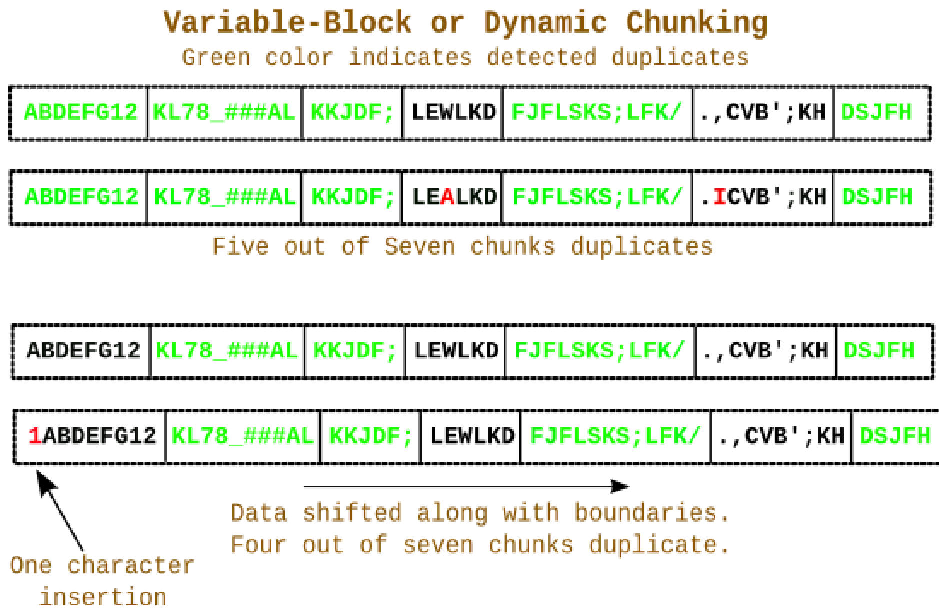


Figure 3.2: Content-Defined Chunking [25]

size (e.g. 64 bytes), and computes a hash over the bytes in that window. If the hash has a certain number of k trailing zeroes, usually configurable by the user, the chunk is cut. If it does not, the window "slides" one byte to the right, i.e. the first byte of the window is removed and the next byte in the data is added to the window, and the hash is recalculated. E.g. for a string of data b of size $l > 64$ bytes and a sliding window of size 64 bytes $b_0||b_1||...||b_{63}$, sliding the window one byte to the right results in a new sliding window $b_1||b_2||...||b_{64}$. This process is repeated until one of the hashes has k trailing zeroes or the specified maximum chunk size is reached, where the chunk is automatically cut. Because the output of the hash function is assumed to be distributed uniformly at random, this results in the average chunk size equalling 2^k bytes. Thus a user can choose his desired average chunk length by changing the value of k.

However, chunking over the actual data implies some relation between the point where the chunk is cut and the data in it. In other words, a chunk's length is directly related to its contents. This opens up potential side-channel attacks where an adversary with knowledge of how a (otherwise securely encrypted) file was chunked may be able to derive further information about the file itself. We will explore and analyze these attacks in Chapter 5.

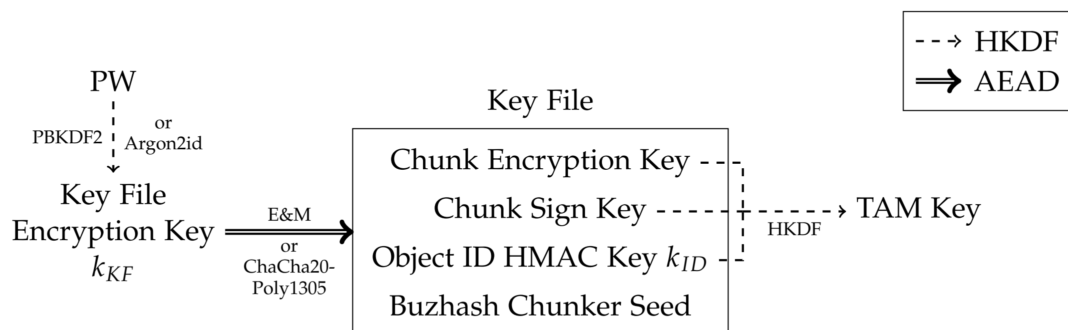
Chapter 4

Backup Systems Studies

In this chapter we take a deeper look at the backup solutions we study, focusing on threat models, key hierarchies, cryptographic primitives and data models.

4.1 Borg

We introduced Borg in Section 1.6.1. Here we delve deeper into its internals, starting with the threat model, and then exploring the key hierarchy and the data model.



PBKDF2:	SHA256, 100k iter., zero IV
Argon2id:	random salt
E&M:	AES-CTR with zero IV, HMAC-SHA256
ChaCha20-Poly1305:	zero IV
HKDF:	HMAC-SHA512

Figure 4.1: Borg Key Hierarchy

4.1.1 Threat Model

Borg introduces a typical threat model for cloud-based backup solutions where the client is trusted and an attacker has full access to the server and interactive capabilities in the form of man-in-the-middle (MitM) attacks [3]. Borg also assumes an attacker to possess a specific set of files which the victim also possesses and backs up. Under these assumptions, Borg claims the attacker is not able to:

1. modify the data of any archive without the client detecting the change
2. rename, remove, or add an archive without the client detecting the change
3. recover plaintext data
4. recover definite (heuristics based on access patterns are possible) structural information such as the object graph (which archives refer to what chunks)

In the case of multiple clients updating the same remote repository, Borg does not guarantee confidentiality. This is a result of AES-CTR being used for data encryption, which requires individual clients to synchronize their counter values to preserve confidentiality. A malicious secondary client could intentionally reuse counter values which would reveal the bitwise XOR of the plaintexts of the data which were encrypted with the same counter value.

We assumed a weak adversary for Borg in Section 1.6.1 to which Borg's threat model is identical. This will allow us to show that there are attacks that violate fingerprint resistance in Chapter 5.

4.1.2 Key hierarchy

When a repository is created Borg generates a key file that holds the randomly generated AEAD key ("Chunk Encryption Key" in Figure 4.1), the "Chunk Sign Key", the object ID generation key which is used to generate UIDs (unique identifiers) for each chunk with a keyed hash function, and a 4-byte string which is used as a secret key for the chunking algorithm (the *chunker seed*). The user provides a passphrase, from which a key file encryption key k_{KF} is derived using Argon2id by default, or PBKDF2 with 100′000 iterations if specified by the user. The client encrypts the key file with k_{KF} using the user-specified AEAD scheme: either an Encrypt-and-MAC scheme implemented by Borg using AES-CTR and HMAC-SHA256, or ChaCha20-Poly1305. The developers behind Borg note that while the use of Encrypt-and-MAC might seem problematic, AES-CTR prevents padding at-

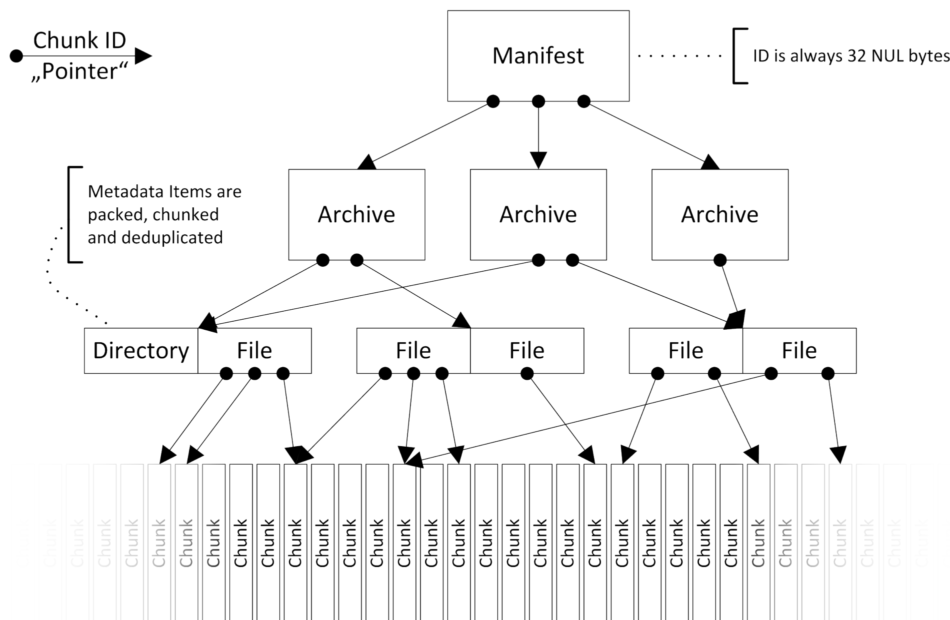


Figure 4.2: Borg's Data Model [9]

tacks, and thus their encryption method is secure [8]. The encrypted key file is either stored locally or remotely in the repository.

Data authentication is provided by the object graph stored on the server, an abstract data structure built upon key-value stores that resembles a tree and contains all data and metadata needed to fully reconstruct any previous backup. Each higher-level object contains a list of lower-level object UIDs, such that it guarantees the authenticity of the lower-level objects. The highest-level object, however, the "Manifest", cannot be authenticated by a higher-level object (as it is the highest). This requires an external authentication mechanism, which Borg implements by using a secret key called the "TAM" (tertiary authentication mechanism) key. The manifest is authenticated using the TAM key. The TAM key is derived from the object ID generation key *id_key*, the AEAD key *enc_key* and the otherwise redundant *auth_key* such that $tam_key = \text{HMAC-SHA512}(id_key||enc_key||auth_key, salt, context)$ for some random *salt* and an operation-dependent *context* string.

Using the data encryption key for two purposes, data encryption and deriving a new key (the TAM key), is questionable, as it violates the key separation principle and offers no security benefits.

4.1.3 Data Model

On a high level, Borg maintains a map of encrypted "objects", addressed by unique cryptographic IDs. Object o with byte representation o_{data} has the unique ID o_{ID}, generated by $o_{\text{ID}} = \mathsf{Blake2b}(k_{\text{ID}}, o_{\text{data}})$ or $o_{\text{ID}} = \mathsf{HMAC}(k_{\text{ID}}, o_{\text{data}})$ depending on a user-defined setting.

The structure of a Borg repository is encoded in a hierarchy of objects. The root object, called "Manifest", references a list of "Archive" objects. Each "Archive" represents a snapshot of a user's backup directories, and references a list of "Item" objects: the actual files and directories in a backup, with the relative metadata.

Finally, each "Item" contains a list of "Chunk" IDs, and the "Chunks" contain the actual file data split into chunks. We will refer to "Chunk" objects simply as chunks, where the ID c_{ID} of a chunk c is equal to its object ID. This hierarchy can be visualized in Figure 4.2.

On the file system of the server, a Borg backup repository is the user-specified directory that contains all data and metadata relevant to a user's backups. It consists of a data folder that contains "segments", files that contain chunk data or item metadata at certain offsets; an index file containing a dictionary of chunk ID to chunk location (segment and offset within the segment) key-value pairs; a hints file which contains the list of segments; and an integrity file which contains checksums and other metadata of the index and hints files.

Because of the nature of Borg's "transaction log" data structure [39] with which the repository is updated, Borg stores consecutively uploaded chunks next to each other in the segments, meaning an adversary with access to the repository can reconstruct which chunks likely belong to the same file, in their exact order.

4.1.4 Deduplication

Borg supports the deduplication of files to significantly reduce backup time and the amount of space used. For all previously backed-up files the item object representing it is saved locally in a hash map called the "cache" with an HMAC of the file's absolute path as the key, and a dictionary containing metadata and a list of its chunk IDs as the value. Whenever a file is to be backed up its representing item object is checked against the cache to make sure the file hasn't already been backed up before. If the key exists in the cache and the value is the same, the file is ignored. Otherwise, the file is chunked again and any new chunks are uploaded to the server.

4.1.5 File Backup Process

To upload a file Borg first chunks it into a set of chunks using the chunking algorithm. Borg then saves this set of chunks and metadata such as the file path and file inode number into an item object. The cache is then checked for potential deduplication as described in Section 4.1.4. Borg then performs authenticated encryption and compression, if specified, on the set of chunks locally with the chunk encryption key before sending the chunks to the server over SSH. Borg does not contain any networking code and simply runs SSH in a separate process, allowing for secure file transfers between the client and the server.

4.1.6 Own implementations

Borg does not code any cryptography themselves. All available methods of HMACs and authenticated encryption are implemented using wrappers around OpenSSL [27] functions.

4.2 EteSync

Following our introduction of EteSync in Section 1.6.2 we take a deeper look at EteSync's threat model, key hierarchy, and data model.

4.2.1 Threat Model

EteSync does not specify a threat model, but from our code analysis, we can deduce that they assume an adversary with at least full backup server access and MitM capabilities. EteSync does not give many promises regarding their software, but they claim to provide "strong encryption using modern cryptography" [15]. With this, we can assume they want to provide:

1. Data integrity
2. Plaintext confidentiality
3. Metadata confidentiality

for their users. EteSync supports multiple clients accessing the same repository using "invitations". The owner of a "collection", EteSync's name for a backup snapshot, may share their chunk encryption key with another user by sending an invitation together with an access level, allowing the receiver to read, write, and/or delete "items", i.e. files, in that collection.

The adversary in this threat model does not have a set of specific files that the victim also backs up, and the threat model is thus weaker than our assumed weak adversary for EteSync which we discussed in Section 1.6.2.

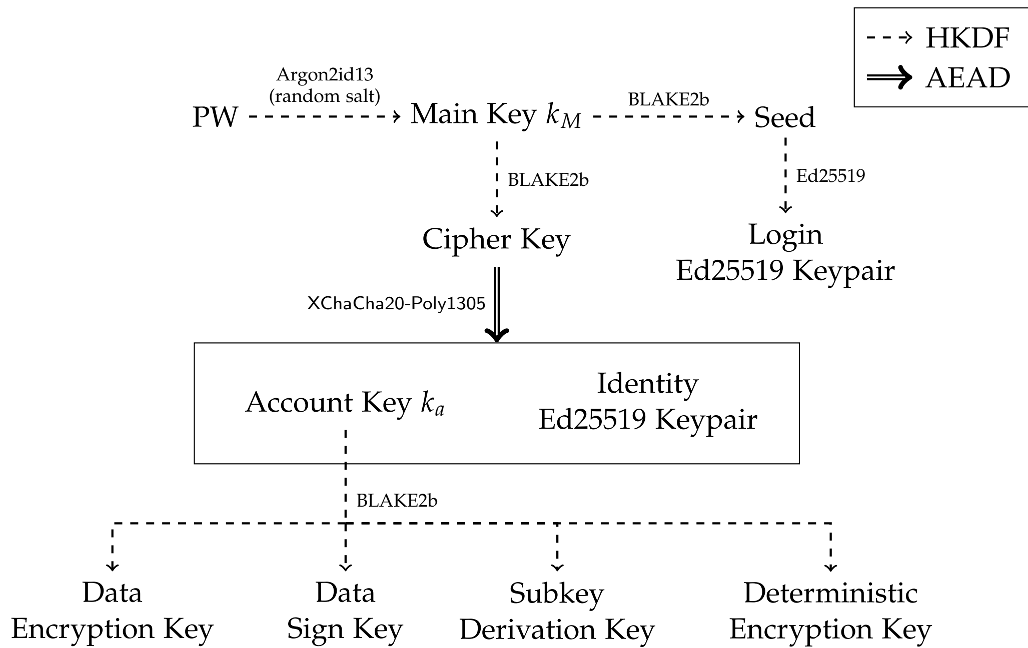


Figure 4.3: EteSync Key Hierarchy

4.2.2 Key hierarchy

EteSync queries the user for a passphrase from which the main key k_M is derived using Argon2id with a random salt. The salt is then sent to the server. The EteSync client derives a seed from k_M using Blake2b, with which an Ed25519 login keypair is generated. The EteSync client then randomly generates an account key k_a and an Ed25519 Identity Keypair. The EteSync client encrypts these two keys with AEAD with another key derived from k_M and sends them to the server. From k_a the client derives four keys using Blake2b: a data encryption key, a data sign authentication key, a subkey derivation key, and a deterministic encryption key.

4.2.3 Data Model

Data is stored in "Items," and organized by "Collections" which contain a list of items. The EteSync client uses deterministic encryption to derive a unique identifier (UID) for every item and every collection by hashing a part of their contents with Blake2b and using it as the nonce for the encryption. Using their UID as a salt, items in a collection are encrypted with a unique key derived from the subkey derivation key.

4.3 Tarsnap

We introduced Tarsnap in Section 1.6.3. In this section, we take a deeper look and inspect Tarsnap's threat model, key hierarchy, and data model.

4.3.1 Threat Model

Tarsnap's threat model assumes a trusted client and an adversary with MitM capabilities and full access to the server. Tarsnap does not explicitly state any security guarantees on its website, but from our code analysis, we can derive that they aim to achieve plaintext confidentiality, data integrity, and metadata confidentiality in the case of a single client. For multi-client support, Tarsnap users must share the key file with all machines, and the Tarsnap client must not run on multiple machines at the same time, or else Tarsnap does not guarantee any security properties.

In Tarsnap's threat model, the adversary does not have the ability to possess a set of specific files that the victim also backs up, and is thus weaker than our assumed weak adversary as we discussed in Section 1.6.3.

4.3.2 Key hierarchy

In the center of Tarsnap's cryptography lies the key file KF, where the file key encryption RSA keypair (pk_{ENCR}, sk_{ENCR}), the file authentication key k_t and more keys are held. In the same manner as in Borg, the user provides a passphrase from which a key file encryption key k_{KF} is derived using scrypt with PBKDF2-SHA256. k_{KF} is used to Encrypt-then-MAC the key file with AES-CTR with a zero IV and HMAC-SHA256.

Three further authentication keys $(k^r_{auth}, k^w_{auth}, k^d_{auth})$ are randomly sampled and shared with the server through a secure channel. The channel is set up by deriving a DH key pair (pk_{REG}, sk_{REG}) from the passphrase and computing a shared key k_M with the server's public key, which is included in the client code. k_M is used to HMAC-SHA256the three authentication keys $(k^r_{auth}, k^w_{auth}, k^d_{auth})$ together, and the keys are sent to the server over a secure channel which we describe in Subsection 4.3.3.

4.3.3 File encryption

To encrypt a file F the Tarsnap client first chunks F using its chunking algorithm based on the computation of polynomials with parameters derived from a secret key. The client then generates a random ephemeral file encryption key k_F and processes all chunks with AES-CTR with key k_F. To authenticate F an HMAC T_F is computed with the key k_t and appended. The file key k_F is encrypted with RSA-OAEP with the user's public key pk_{ENCR} and prepended with a nonce N_F. Finally, the client computes another HMAC T_E

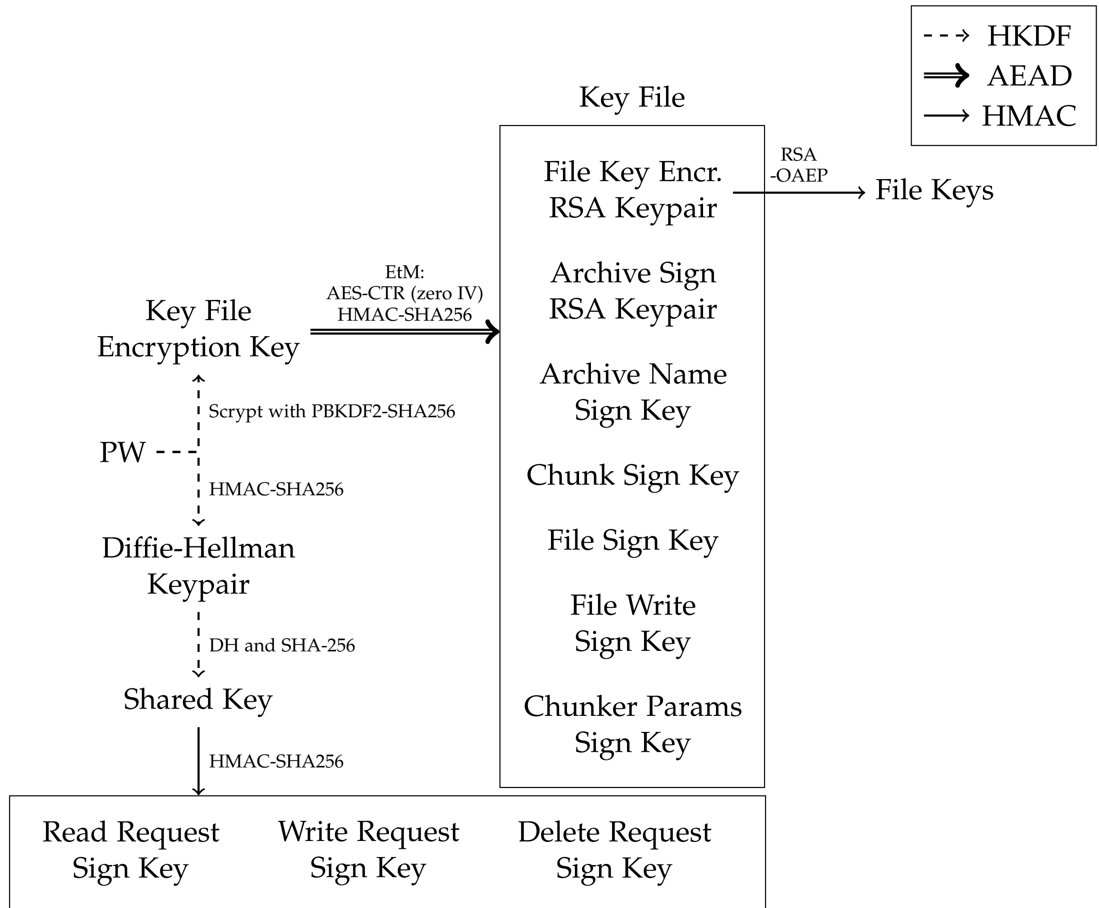


Figure 4.4: Tarsnap Key Hierarchy

with k^{w}_{auth} and appends it, which is used to authenticate the nonce and bind the nonce to the file.

To communicate with the server a secure channel is constructed roughly based on SSLv3's handshake. The client samples a random DH key pair, receives the server's public key and a nonce N from the server, and computes a shared key $k_M = \text{MGF1}(N||DH)$ with the server, where DH is the commonly computed DH value and MGF1 is a deterministic function that works similarly to a hash function. The message to be sent to the server is then hashed with HMAC-SHA256 in non-keyed mode and the hash is encrypted using AES-CTRinitialized with a key k_M^{enc} derived from k_M. Another HMAC T_S is computed of this encrypted hash with a key k_M^{auth} also derived from k_M and appended. Finally, everything except for the encrypted hash is encrypted with AES-CTR with the encryption keystream and uploaded together with the encrypted hash to the Tarsnap server.

4.3.4 File decryption

When a user requests to read a file from a backup the server sends the client a message containing the encrypted file. To decrypt a file F the client first decrypts the message using the keystream shared with the server, verifies the tags T_S, T_E, and T_F, and decrypts the file key k_F with the user's private key sk_{ENCR}. The client then uses k_F to decrypt the file itself.

4.3.5 Data Model

Tarsnap takes a list of files, chunks them based on the chunking algorithm, assigns every chunk an index, and uploads them to the server along with relevant metadata. The server manages an "Archive", a dictionary of chunk index to data key-value pairs and file index to chunk pointer lists.

4.3.6 Deduplication

Tarsnap holds a local list of all uploaded chunk hashes and only uploads new chunks if the new chunk's hash is not already in the local hash list.

4.3.7 Analysis

An observant reader might notice the unconventional usage of public-key cryptography for file encryption. Indeed, assuming an adversary with access to the server discovers the user's public key, file integrity is broken for all files of size $l \leq 16$ bytes.

Assume an adversary that knows the public key pk, meaning the adversary can compute RSA.Enc(pk, K) for an arbitrary K of his choice, and an encrypted file m. Let $c = \text{AES-CTR.Enc}(K, m, n)$ for a nonce n. Then the encrypted file on the server is $n||\text{RSA.Enc}(\text{pk}, K)||c||\mathsf{HMAC}(K_{\text{auth}}, c)$ where K_{auth} is the long-term MAC key.

Assume that m is 16 bytes, and we want to make sure that this file decrypts to a different file of size 16 bytes m'. Then we want that, from the same ciphertext c we obtain m' when decrypting instead of m. The reason why we want the same ciphertext c is that the MAC is done on c, so we cannot change it. However, AES-CTR encryption works by XOR-ing the message with the keystream, which is itself the encryption of the nonce.

$$c = m \oplus \mathsf{AES.Enc}(K, n) \tag{4.1}$$

Then, we want a new (K', n') such that:

$$c = m' \oplus \mathsf{AES.Enc}(K', n') \tag{4.2}$$

This can be done by setting K' to an arbitrary value and then:

$$n' = \mathsf{AES.Dec}(K', c \oplus m') \tag{4.3}$$

However, the randomization of messages provided by OAEP padding prevents us from algebraically reconstructing the public key from known ciphertext-plaintext pairs.

We require the length of the file to be of size $l \leq 16$ bytes for our attack because creating (K', n') pairs for ciphertexts that decrypt to meaningful plaintexts becomes hard when $l > 16$ bytes. The reasoning is that, assuming $l = 32$, Equation 4.1 becomes:

$$\begin{aligned} c[:16] &= m[:16] \oplus \mathsf{AES.Enc}(K, n||00000000) \\ c[16:32] &= m[16:32] \oplus \mathsf{AES.Enc}(K, n||00000001) \end{aligned} \tag{4.4}$$

This makes the properties we want our new (K', n') pair to fulfill the following:

$$\begin{aligned} c[:16] &= m'[:16] \oplus \mathsf{AES.Enc}(K', n'||00000000) \\ c[16:32] &= m'[16:32] \oplus \mathsf{AES.Enc}(K', n'||00000001) \end{aligned} \tag{4.5}$$

Applying our attack (Equation 4.3) means we try to find K' such that, for arbitrary n':

$$\begin{aligned} n'||00000000 &= \mathsf{AES.Dec}(K', c[:16] \oplus m'[:16]) \\ n'||00000001 &= \mathsf{AES.Dec}(K', c[16:32] \oplus m'[16:32]) \end{aligned} \tag{4.6}$$

However, finding such a K' can only be done by a brute-force attack, and this attack requires exponentially increasing amounts of computation for linear growth in the file size l because the number of equations we need to solve in Equation 4.6 also grows linearly.

One might also observe that a file F is encrypted and authenticated with separate keys, k_F and k_t respectively. Since k_F is randomly generated for every new file F, leakage of one file key does not compromise the confidentiality of other files. However, since k_t is used to compute T_F for every file F, if k_t is exposed it could lead to the loss of integrity on all files.

4.3.8 Countermeasures

Authenticating the nonce with a secret key means an adversary cannot compute a n' such n' is a valid nonce and $n' = \mathsf{AES.Dec}(K', c \oplus m')$ for an arbitrary K' without knowing the secret key, mitigating our attack.

Implementing a "Key Committing AEAD" scheme [17] that binds a key to a nonce, ciphertext, and tag by computing an additional value K_C also prevents an adversary from successfully computing a pair (K', n') that is valid under the Key Committing AEAD scheme, as long as K_C is authenticated with a secret key. This countermeasure also mitigates our attack.

4.4 Kopia

In Section 1.6.4 we introduced Kopia. Here we describe a brief overview of Kopia's threat model, data model, and deduplication implementation.

4.4.1 Threat Model

Kopia does not specify a threat model, but from their documentation and client code we can deduce a threat model which assumes an untrusted server and an adversary with full server access. This is a weaker model than our weak adversary as defined in Section 1.3 as it does not include the adversary's ability to know the plaintext of a specific file in the backup repository.

4.4.2 Data Model

The Kopia client creates backups on a snapshot basis and files are split into chunks based on a chunking algorithm called Buzhash which we will introduce in Section 5.1. Chunks are associated with file IDs to bind chunks to their respective files.

4.4.3 Deduplication

Kopia implements chunk-level deduplication by locally hosting a cache that contains a dictionary of chunk ID to file ID key-value pairs. Whenever the client creates a new backup each file is chunked and the client checks the cache for potential duplicate chunks. All duplicates are ignored and all non-duplicates are uploaded to the server.

4.5 Bupstash

We introduced Bupstash in Section 1.6.5. Here we give a brief overview of Bupstash's threat model, data model, and deduplication implementation.

4.5.1 Threat Model

Bupstash does not specify a specific threat model but does claim that "strong privacy", "secure remote access control" and "safety against malicious attacks" are part of their design goals [10]. It appears that they aim to reduce the attack surface on the server but do not explicitly mention that they assume an adversary with full access to the server. Assuming Bupstash's threat model contains an adversary with full access to the server, this threat model is still weaker than our weak adversary as we defined in Section 1.3 because the adversary does not have the ability to know the plaintext of a specific file in the backup repository.

4.5.2 Data Model

The Bupstash client creates backups on a snapshots basis and splits files, also called "items" once processed, into chunks based on a simple chunking algorithm which we will introduce in Section 5.6.

On the file level, the repository on the server consists of a "data" folder which contains encrypted data chunks, an "item" folder which contains metadata for each item, a "meta" folder which contains metadata regarding the repository and garbage collection, and some files which aid in atomicity and durability of the repository.

4.5.3 Deduplication

The Bupstash client implements chunk-level deduplication by tracking all uploads to the server in a local cache which they call the "send log". The send log contains a list of chunk IDs (keyed Blake3 tags) for all chunks that were uploaded in the previous backup and a dictionary of file paths to chunk ID mappings. This allows the Bupstash client to skip over chunks and files that were previously backed up, which for example happens often when periodic snapshots of the same directory are taken. Because the send log only contains chunk IDs for chunks uploaded in the previous backup, a user can manually save send logs and specify which send log he wants to use when requesting a backup to optimize deduplication.

4.6 Restic

In Section 1.6.6 we introduced Restic. Here we dive deeper and give a brief overview of its assumed threat model, its data model, and deduplication implementation.

4.6.1 Threat Model

Restic assumes a trusted client, an untrusted server, and an attacker with full access to the server and the ability to successfully execute MitM attacks. Under this threat model Restic guarantees "confidentiality and integrity of your data" [33].

Restic does not assume an attacker to know a specific file in the backup and thus has a weaker threat model than our assumed weak adversary as described in Section 1.6.6.

4.6.2 Data Model

The Restic client creates backups on a snapshot basis and splits files into chunks using a chunking algorithm based on Rabin fingerprints, which we will introduce in Section 5.2.

On the file level, the repository on the server consists of:

- a "data" folder which contains data chunks encrypted and authenticated with an EtM AEAD scheme using AES-CTRand Poly1305
- an "index" folder which contains "indexes", files that provide lists of chunk information including chunk ID, chunk length, and chunk location in the repository
- a "keys" folder which contains encrypted key files
- a "snapshots" folder which consists of a file for every backup snapshotted, containing the paths of the client directories or files which were snapshotted and metadata such as the time of the snapshot and the hostname
- some repository metadata files and a configuration file

For all files in the repository (apart from the configuration file), the name of the file is the SHA-256 hash of the file's contents. This makes it easy to verify that there were no accidental modifications to the file, such as in the case of disk read errors, and is used to uniquely identify the file. Files in the repository always contain enough data and/or metadata to prevent brute-force attacks.

4.6.3 Deduplication

The Restic client implements chunk-level deduplication by locally hosting the "master index" which consists of a list of "pack" IDs, where packs are objects that contain chunks. Whenever the client creates a new backup each file is chunked and each chunk's ID is checked against the master index to confirm if it is a duplicate or not. If it is a duplicate it is ignored and not

uploaded to the server. If it is not a duplicate the client marks it as pending and, once all chunks have been checked for deduplication and marked as pending, processes all pending chunks and uploads them to the server.

Chapter 5

Fingerprinting Attacks on Chunking

We saw in Chapter 3 that content-defined chunking is a popular method to efficiently implement deduplication. We also discussed that this content-defined chunking is done over the actual chunk data, which implies some correlation between the chunk length and the chunk data. If no secret is introduced in the chunking algorithm the length of a chunk leaks a lot of information about the plaintext. Section 5.1 and 5.2 introduce two of the most common rolling hash functions that chunking algorithms are built upon, called Buzhash and Rabin fingerprints. In the following sections, for each cloud backup solution that we study, we analyze their chunking algorithm implementations in detail. From these analyses, we will derive critical fingerprinting vulnerabilities and show a practical attack for Borg in 5.3.5, which allows a malicious server provider to launch fingerprinting attacks for arbitrary files. This attack extends to Kopia and Restic and possibly other cloud backup systems that use content-defined chunking as well. Finally, in Section 5.8 we will discuss possible countermeasures and mitigations for our discovered vulnerabilities.

5.1 Buzhash

Buzhash [37] is a cyclic polynomial rolling hash. Buzhash takes as its input a byte window and a hash function that takes a single byte as an input and outputs a bit string of 32 bits, and produces a (non-cryptographic) hash of the sliding window by evaluating the polynomial in Figure 5.1. Sliding the input window one byte to the right does not require recomputing the entire hash as seen in Figure 5.2, allowing Buzhash to be used efficiently in a stream fashion (hence the ”rolling hash”). To do this Buzhash maintains a state, which we name bh. To update the Buzhash state one can simply call the Buzhash update function with the current state (bh), the byte to be removed ($b1$), the byte to be added ($b2$), and the length of the input window (n).

Buzhash(k: Byte Array, h: Hash Function)

```
1 :  bh := 0
2 :  n := length(k)
3 :  for i = 0 to n :
4 :      bh := BROTL(bh, 1) ⊕ h(k[i])
5 :  return bh
```

Figure 5.1: The Buzhash Algorithm
BROTL(k,i) is the bitwise rotate-left function, applied on the binary representation of k, i times

Buzhash-Update(bh: Integer, $b1$: Byte, $b2$: Byte, h: Hash Function, n: Integer)

```
1 :  nmod := n mod 32
2 :  bh := BROTL(bh, 1) ⊕ BROTL(h(b1), nmod) ⊕ h(b2)
3 :  return bh
```

Figure 5.2: The Buzhash Update Algorithm
BROTL(k,i) is the bitwise rotate-left function, applied on the binary representation of k, i times

A new chunk is cut when the k least significant bits of the state bh are all 0. We refer to k as the *chunking threshold*. To set minimum sizes for the chunks, some implementations skip the first S_m bytes of input, and only start evaluating Buzhash after that.

Rolling hashes like Buzhash are commonly used for separating large files into chunks that can be deduplicated efficiently. This is because they provide content-defined chunk cuts. For example, two files with different headers but overlapping content are likely to produce at least some identical chunks. Compared to a simple fixed chunking algorithm, this allows the beginning of a file to be changed without all the resulting chunks being different (see Figure 3.1 and Figure 3.2).

While advantageous for deduplication, content-defined chunking may leak some information about the file being chunked. This is crucial when the chunks are being encrypted: encryption does not traditionally hide the length of the plaintext being encrypted, and in this case, the length of a chunk itself leaks some information about the plaintext, since the hash is calculated over the plaintext.

Implementing a chunking algorithm that just uses Buzhash as is without any secrets (we also call this “plain Buzhash”) would allow an adversary with access to a target user’s backup server to execute a trivial fingerprinting attack by running the chunking algorithm on an arbitrary file F and comparing the resulting chunk lengths with the lengths of the chunks contained in the tar-

get user's repository. If the target user's repository contains a list of chunks for which their respective lengths equal the chunk lengths resulting from chunking F, an adversary can conclude with very high probability that the target user's repository contains F and thus the target user has backed up F. We call this the "plain Buzhash attack".

This implies that, under the threat models defined by the cloud backup solutions we study where the adversary has full access to the server, implementing a chunking algorithm which just uses Buzhash as is without any secrets would trivially break fingerprint resistance. Under the *weak adversary* threat model we defined in Section 1.3 plain Buzhash also breaks fingerprint resistance, as the adversary also has full access to the server.

5.2 Rabin Fingerprint

A Rabin fingerprint [30] of a bit string m with length n is the remainder of the division of two polynomials over the finite field GF(2). More specifically, a polynomial $f(x) = m_0 + m_1 x + ... + m_{n-1} x^{n-1}$, where m_i is the i-th bit of m, is constructed and divided by an irreducible polynomial $p(x)$ of degree q chosen at random. The remainder $r(x)$ of this division is a polynomial of degree $q - 1$ and can be interpreted as a q-bit number.

The Rabin fingerprint scheme is an efficient rolling hash function, as the computation of $r(x)$ for two different bit strings with overlapping regions has overlapping summands. For example, let bit string $a = m_0, m_1, ..., m_{63}$, bit string $b = m_1, m_2, ..., m_{64}$ and RF(a) be the Rabin fingerprint of a. If we know RF(a) we can compute RF(b) much more efficiently compared to having to compute $f(x)$ again.

For an arbitrary file F, content-defined chunking algorithms using Rabin fingerprints start chunking at the specified minimum chunk size and calculate Rabin fingerprints over a sliding window of a specific size l. The algorithm cuts a chunk whenever $\text{RF}(F[i], F[i+1], ..., F[i+l-1])$ has k trailing zero bits, where 2^k is the specified (desired) average chunk size value in bytes.

As we discussed in Section 5.1, implementing a content-defined chunking algorithm based on a rolling hash function without any secrets breaks fingerprint resistance under an adversary with full access to the server, and thus also under a weak adversary. This also applies to chunking algorithms based on Rabin fingerprints.

5.3 Borg

5.3.1 Borg's Buzhash Implementation

To combat the trivial Buzhash attack Borg introduces a 32-bit secret Buzhash chunker seed. As a substitute for a hash function, Borg uses a fixed hash table that maps single bytes to 32-bit values and XORs every element of the hash table with the secret chunker seed. Borg claims that this makes chunk size-based fingerprinting attacks difficult, but we will show that this is not true by easily extracting the secret Buzhash chunker seed.

The secret chunker seed is held encrypted locally in the key file and is only shared when specifically requested by the user.

In Borg, the chunking threshold k is by default set to 21, but it can be altered by users using the HASH_MASK_BITS configuration parameter. The minimum and maximum chunk sizes have a default value of 2^{19} bytes and 2^{23} bytes respectively and can also be changed by users. Borg achieves the minimum and maximum chunk sizes by, for the minimum size, starting to chunk only after 2^{19} bytes; for the maximum size forcing a cut after 2^{23} bytes; and for the average size having a chunk cut if the last 21 bits of the Buzhash equal zero. The expected chunk size is $1'568'770$ bytes and the probability of cutting a chunk after 2 MiB is ≈ 0.527. We provide a short probability analysis in the Appendix A.1.

5.3.2 Threat Model

In their threat model, Borg assumes an adversary who has full server access and possesses a set of files that the victim also backs up.

Borg's documentation recognizes that fingerprint attacks are problematic [6]:

> A Borg repository does not hide the size of the chunks it stores (size information is needed to operate the repository).

The documentation suggests some workarounds to make size-based fingerprinting difficult [6]:

- using the Buzhash chunker with a secret, random per-repo chunker seed
- optional obfuscate pseudo compressor with different choices of algorithm and parameters
- using the fixed chunker with a user-selectable compression algorithm

Borg's documentation also claims that the seed prevents chunk size-based fingerprinting attacks in [5]. This, together with the fact that the Buzhash chun-

ker is selected by default, means that many Borg users use Buzhash and rely on the secret chunker seed for fingerprint resistance.

5.3.3 Fixed Chunker

Instead of a Buzhash chunker, a user can specify that he wants to use a "fixed" chunker with a fixed chunk size p so that every file is chunked into chunks of size p except for the last chunk. This drastically reduces deduplication efficiency as we discussed in Chapter 3. Additionally, the fixed chunker does not protect a user against fingerprinting attacks. Assume a weak adversary that wants to execute a fingerprinting attack for a file FA of length l and let n be the number of files in the victim's repository. The adversary can calculate $lmod = l \bmod p$ and search the repository for chunks of length $lmod$. The reasoning is that if the victim backed up FA, it would have been chunked into chunks of length p except for the last chunk, where the last chunk would have length $lmod$. If the adversary finds a chunk of length $lmod$, he can deduce with probability at least $P = (1 - \frac{1}{p})^{n-1}$ that the victim's repository contains FA. If the adversary does not find a chunk of length $lmod$, he can safely deduce that the victim's repository does not contain FA.

5.3.4 Extracting the Secret Seed

A trivial way to obtain the secret seed would be to brute-force all the possible values of the seed. In our weak threat model, we assume the adversary knows a set of files contained in the repository. It is therefore possible for the adversary to check the chunk sizes of the known file to verify which seed guesses were correct.

A deeper mathematical inspection of how Borg's secret chunker seed interacts with Buzhash reveals that there exists an even easier method to recover the seed. Let b_i be the i-th byte of a chunk c, $i \in \mathbb{N}$, and T_j be the j-th Buzhash table entry for all $j \in \{0, ..., 255\}$. Let $s^m(b)$ be the bitwise rotate-left function, applied on the binary representation of byte b, m times. Assuming a default window size of 4095 bytes, the Buzhash of chunk c under seed σ is:

$$bh_\sigma(c) = s^{30}(h(b_0)) \oplus s^{29}(h(b_1)) \oplus ... \oplus s^0(h(b_{31})) \oplus s^{30}(h(b_{32})) \oplus ... \oplus s^0(h(b_{4095})) \tag{5.1}$$

where $h(b_i) = T_{i \bmod 256} \oplus \sigma$ for all $i \in \mathbb{N}$. Due to the linearity of $s()$ and the

associativity of $\oplus$ we derive:

$$\begin{aligned}
bh_\sigma(c) &= s^{30}(h(b_0)) \oplus s^{29}(h(b_1)) \oplus ... \oplus s^0(h(b_{31})) \oplus s^{30}(h(b_{32})) \oplus ... \oplus s^0(h(b_{4095})) \\
&= s^{30}(T_0 \oplus \sigma) \oplus s^{29}(T_1 \oplus \sigma) \oplus ... \oplus s^0(T_{31} \oplus \sigma) \\
&\quad \oplus s^{30}(T_{32} \oplus \sigma) \oplus ... \oplus s^0(T_{255} \oplus \sigma) \\
&= (s^{30}(T_0) \oplus s^{30}(\sigma)) \oplus (s^{29}(T_1) \oplus s^{29}(\sigma)) \oplus ... \oplus (s^0(T_{31}) \oplus s^0(\sigma)) \\
&\quad \oplus (s^{30}(T_{32}) \oplus s^{30}(\sigma)) \oplus ... \oplus (s^0(T_{255}) \oplus s^0(\sigma)) \\
&= (s^{30}(\sigma) \oplus s^{29}(\sigma) \oplus ... \oplus s^0(\sigma) \oplus s^{31}(\sigma) \oplus ... s^0(\sigma)) \\
&\quad \oplus (s^{30}(T_0) \oplus s^{29}(T_1) \oplus ... \oplus s^0(T_{31}) \oplus s^{31}(T_{32}) \oplus ... \oplus s^0(T_{255})) \\
&= s^{31}(\sigma) \oplus (s^{30}(T_0) \oplus s^{29}(T_1) \oplus ... \oplus s^0(T_{31}) \oplus s^{31}(T_{32}) \oplus ... \oplus s^0(T_{255}))
\end{aligned} \tag{5.2}$$

From equation 5.2 we can see that the only difference between a Buzhash sum computed without a seed and Borg's computed Buzhash sum is a summand of σ bit rotated left by 31. Let $C_i, i \in \mathbb{N}$ be arbitrary summands which are independent of the seed. Whenever we update the Buzhash sum we get the following:

$$\begin{aligned}
prevsum &= s^{31}(\sigma) \oplus C_1 \\
sum &= s(prevsum) \oplus s^{31}(\sigma \oplus C_2) \oplus (\sigma \oplus C_3) \\
&= s(s^{31}(\sigma) \oplus C_1) \oplus s^{31}(\sigma) \oplus \sigma \oplus s^{31}(C_2) \oplus C_3 \\
&= \sigma \oplus s(C_1) \oplus s^{31}(\sigma) \oplus \sigma \oplus s^{31}(C_2) \oplus C_3 \\
&= s^{31}(\sigma) \oplus C_4
\end{aligned} \tag{5.3}$$

Thus an update of the Buzhash sum does not alter the influence the seed has on the sum. Let $bh_{zero}(c)$ be the Buzhash of a chunk c computed with a zero seed, and $bh_\sigma(c)$ the Buzhash of chunk c computed with seed σ. From the conclusions above we derive:

$$\begin{aligned}
bh_{zero} &= s^{31}(0) \oplus (s^{30}(T_0) \oplus s^{29}(T_1) \oplus ... \oplus s^0(T_{31}) \oplus s^{31}(T_{32}) \oplus ... \oplus s^0(T_{255})) \\
&= s^{30}(T_0) \oplus s^{29}(T_1) \oplus ... \oplus s^0(T_{31}) \oplus s^{31}(T_{32}) \oplus ... \oplus s^0(T_{255}) \\
bh_\sigma &= s^{31}(\sigma) \oplus (s^{30}(T_0) \oplus s^{29}(T_1) \oplus ... \oplus s^0(T_{31}) \oplus s^{31}(T_{32}) \oplus ... \oplus s^0(T_{255})) \\
bh_\sigma &= s^{31}(\sigma) \oplus bh_{zero} \\
\sigma &= s(bh_{zero} \oplus bh_\sigma)
\end{aligned} \tag{5.4}$$

Assuming our chunk was cut because bh_σ had k trailing zero bits we con-

clude

$$\begin{aligned}
\sigma[(32-k):] &= s(bh_{zero}[(32-k):] \oplus bh_{\sigma}[(32-k):]) \\
\sigma[(32-k):] &= s(bh_{zero}[(32-k):] \oplus 0) \\
\sigma[(32-k):] &= s(bh_{zero}[(32-k):]) \\
\sigma[(32-k):] &= bh_{zero}[(32-k-1):32]
\end{aligned} \tag{5.5}$$

Thus we can derive k bits of the seed by calculating the Buzhash of a data input under the zero seed for which we know the Buzhash under the actual seed has k trailing zero bits. A natural question might be to ask how we can extract the remaining $32-k$ bits of the seed. As an observant reader may already have noticed, only those k bits of the seed ever have any influence, and the remaining $32-k$ bits might as well not exist.

Remark 5.1 *In Borg, the top $32-k$ seed bits never impact the chunking algorithm. This is because Borg only uses the secret* Buzhash *chunker seed to calculate the* Buzhash *sum, which is only used to check if its k trailing bits are all zeros, and the fact that at any point only bits $32-k$ to 31 of the seed have an effect on the k trailing bits of the seed, as seen in Equation 5.5. It follows that the seed would be even easier to brute-force.*

Theorem 5.2 (Seed Recovery Theorem) *Let bh_{zero} be the* Buzhash *under the zero seed of a chunk c in a Borg repository R with length l greater than the minimum chunk size and smaller than the maximum chunk size, and σ be the secret chunker seed of R. By the definition of* Buzhash *chunking, we know that when c was chunked, the chunking algorithm state bh_{σ} had k trailing zero bits. Let us assume that we know the content of c. Then we can recover the least significant bits of the secret seed σ by computing the zero-seed* Buzhash *state bh_{zero} over c: $\sigma[(32-k):] = bh_{zero}[(32-k-1):32]$.*

To apply Equation 5.5 and extract our victim's seed in practice we need a string of data for which we know that our victim backed it up and that a chunk was cut at the last bit because bh_{σ} had k trailing zero bits. By the specification of the *weak adversary* threat model, we assume that the adversary knows a specific file F in the backup. If F is big enough the probability of all chunks bh_{σ} never having k trailing zero bits becomes negligible. In practice, for a default average chunk size value of $k = 21$, F being 8 MiB or larger is sufficient. Because chunks are stored sequentially in the segment files, the adversary can search through the repository's segment files to check for chunk sequences that add up to the length of F. Once we have a chunk of size n for which we deduced that it was chunked because bh_{σ} had k trailing bits, we can take the corresponding n bits of F concatenated with the following 4095 bits. This gives us a data string for which the Buzhash sum bh_{σ} at the n-th bit has 21 trailing zero bits. Calculating the Buzhash sum under the zero seed for this data string gives us all we need as seen in Equation 5.5.

In conclusion, we only require knowledge of a file that is sufficiently large to extract the entire secret Buzhash chunker seed.

5.3.5 Proof of Concept Attack

We describe an attack that allows a weak adversary to extract the secret chunker seed and successfully execute a fingerprinting attack on an arbitrary file using the plain Buzhash attack we described in Section 5.1. We wrote a Python script to validate this attack and were able to consistently extract the secret chunker seed if the file we knew was sufficiently large (> 8 MiB). Once we had the secret chunker seed, executing the plain Buzhash attack was trivial and we were able to launch fingerprinting attacks for arbitrary files without fail.

Assume we have knowledge of a file F of sufficiently large size lf in the victim's repository and we want to execute a fingerprinting attack for a file FA. We access the repository's index file which contains the location of each data chunk in the repository. Going through each segment file we check for a list of subsequent chunks in the same segment file for which the data lengths of the chunks sum up to lf. Having found the chunks corresponding to our known file F, we note the length of the first chunk lc. We take the first $lc + 4095$ bytes of file F and run Buzhash on it, saving the result as bh_{zero}. Using Theorem 5.2 with bh_{zero} and the default $k = 21$ we can fully recover the secret chunker seed σ. With σ we can calculate bh_σ for file FA and chunk it the same way the victim's repository would chunk it. This gives us a list of n chunk lengths $la_1, la_2, ..., la_n$. Finally, we check the repository for data chunks of lengths $la_1, la_2, ..., la_n$. If we find a matching chunk for every la, we can deduce with a very high probability that the victim's repository contains FA. If we do not find a match, we can safely deduce that the victim's repository does not contain FA.

5.3.6 Borg's Documentation

We discovered this attack independently, but we later noticed that the same attack was first discovered in Borg's Github issue 3687 [16]. At the time, Borg's documentation mentioned this attack including the fact that [38]:

> Within our attack model, an attacker possessing a specific set of files that he assumes that the victim also possesses (and backups into the repository) could try a brute force fingerprinting attack based on the chunk sizes in the repository to prove his assumption.

Borg's documentation still mentions this problem today [7]. However, it is contained in the wrong section (the "Fixed Chunker" section) and shortly thereafter Borg's documentation mentions that, for the Buzhash chunker, the

secret chunker seed makes fingerprinting more difficult, which we showed is not true.

We commend Borg for warning its users of this vulnerability at the time, but we can see that since then this warning has been eroded from Borg's documentation and users are unknowingly exposing themselves to fingerprinting vulnerabilities.

5.4 Tarsnap

Tarsnap implements its own chunking algorithm based on the computation of some polynomials with parameters derived from a secret key. Tarsnap's minimum chunk size is 4 KiB, the maximum chunk size is 261′120 bytes and the average chunk size is 65′536 bytes.

As Tarsnap's chunking algorithm is constructed in a much more complex way compared to other cloud backup solutions that we study, an in-depth analysis of Tarsnap's chunking algorithm is out of scope for this thesis.

5.5 Kopia

Kopia uses Buzhash and implements it by using imported functions from a Buzhash implementation written by Duquesne [14]. This implementation uses a fixed hash table as a hash function and Kopia specifies a 64 byte rolling window size, 8 MiB maximum chunk size, 2 MiB minimum chunk size, and 4 MiB average chunk size. Further, Kopia does not use the standard method of calculating an initial hash over the initial rolling window. Instead, Kopia generates an empty rolling window and starts rolling from the first byte after the minimum chunk size.

Kopia does not introduce any secrets in the chunking process. This leaves Kopia vulnerable to the plain Buzhash attack shown in 5.1 and breaks fingerprint resistance under its threat model.

5.6 Bupstash

Bupstash's chunking algorithm uses a rolling hash function, called "rollsum", which is a very simple, modified version of Buzhash where instead of using the bit rotate left (BROTL) function to rotate values by 1, Bupstash's rollsum update function simply shifts the state left by 1. Bupstash's rollsum does not use a sliding window. Instead, the initial state rs is 0, and for every new byte Bupstash's rollsum left-shifts rs by 1 and XORs it with the new byte. Bupstash's rollsum uses a hash function h which hashes individual bytes to 4 byte byte string. This means at any point only the most recently

Rollsum-Update(rs: Integer, b: Byte, h: Hash Function)

1 : $rs := (rs \ll 1) \oplus h(b)$
2 : **return** rs

Figure 5.3: The Rollsum Update Algorithm

added 32 bytes influence rs, as a newly added byte b is left-shifted 32 times after 32 iterations, removing it from the equation. We describe the pseudocode of the rollsum update function in Figure 5.3. The minimum chunk size is 8 KiB, the maximum chunk size is 20 MiB and the average chunk size is 2 MiB, as a chunk is cut whenever rs has 21 trailing zero bits.

For the hash function h in Bupstash's rollsum implementation Bupstash's chunking algorithm uses a 256-element hash table derived from a deterministic ChaCha20call with a secret 32 byte key. Every element of the hash table is a 4 byte byte string. This prevents the plain Buzhash attack applied to Bupstash's chunking algorithm as long as the secret key or the hash table cannot be extracted.

However, due to the nature of how Bupstash's rollsum works, rs leaks information about the individual hashes of the last 32 bytes of a chunk. Let $b_0, b_1, ..., b_{31}$ be the last 32 bytes of the plaintext of a chunk that the adversary knows. We know that $rs = (h(b_0) \ll 31) \oplus (h(b_1) \ll 30) \oplus ... \oplus (h(b_{30}) \ll 1) \oplus h(b_{31})$. Because rs must have 21 trailing zero bits, the adversary can deduce that $h(b_{31})[31] = 0$, $h(b_{31})[30] \oplus h(b_{30})[31] = 0$, and so on.

Under our weak adversary threat model this is not enough information to conclusively extract the hash table, and thus we did not find a practical attack for Bupstash. We conjecture that a strong adversary, however, could easily extract the hash table by injecting many files and applying linear algebra, and with the hash table successfully execute a plain Buzhash attack adapted to Bupstash's chunking algorithm.

5.7 Restic

Restic's chunking algorithm uses Rabin fingerprints over a 64 byte sliding window with a minimum chunk size of 512 KiB, a maximum chunk size of 8 MiB and an average chunk size of 1 MiB [32].

Restic does not introduce any secrets in the chunking process. This makes Restic vulnerable to the plain Rabin fingerprints attack we described in Section 5.2 and breaks fingerprint resistance under Restic's threat model.

5.8 Mitigations

In this chapter, we showed that many popular cloud backup solutions are vulnerable to the plain Buzhash and Rabin fingerprint attacks which break fingerprint resistance under the corresponding cloud backup solution's threat model. In Section 5.3.1 we explained that Borg introduces a secret chunker seed to prevent the plain Buzhash attack, but their countermeasure ultimately fails as it is possible to fully recover the secret chunker seed as we showed in Section 5.3.4.

To prevent our chunk size-based fingerprinting attacks we must target the root problem, which is the fact that the result of the polynomial being evaluated for chunking is a non-keyed function of the plaintext and both the result of the polynomial and the function are available to an adversary under a standard threat model. Ideally, we want a PRF that works as a secure rolling hash function.

We recommend a secure rolling hash function suggested by user D.W. on the Cryptography Stack Exchange Q&A website [12]. D.W. proposes the rolling hash function construction $F_{k1,k2}(x) = E_{k1}(R_{k2}(x))$ where $R_{k2}(\cdot)$ is a non-cryptographic rolling hash function (e.g. Buzhash or Rabin fingerprint), E_{k1} is a cryptographically secure block cipher such as AES, and $k1, k2$ are secret keys. For R_{k2} the parameters of the rolling hash function should be based on $k2$, e.g. the average chunk size.

This is still a rolling hash function, as for any input x and corresponding output $y = F_{k1,k2}(x)$ we can compute $z = E_{k1}^{-1}(y)$, update z with the corresponding update function of R to get z', and encrypt z' for which we get $y' = E_{k1}(z')$. Updating the input x to x' would give us $u' = R_{k2}(x') = z'$ and thus $E_{k1}(u') = E_{k1}(R_{k2}(x')) = E_{k1}(z') = y'$.

D.W. argues that F is a secure PRF because:

> (1) if R is universal and E is a PRF, then $E \circ R$ is a PRF; (2) if E is a PRP with large domain, then E is a PRF. Of course, the definition of security for a block cipher is that it should be a PRP. Consequently, if E is a secure block cipher, then it is a secure PRF, and so too will F be. A PRF (pseudorandom function) provides exactly the security property you want; it is the right way to formalize what you mean by "cryptographic security" for a keyed hash.

This secure rolling hash function construction prevents our attack on Borg and the plain Buzhash and Rabin fingerprint attacks, as our Borg attack requires knowledge of the secret chunker seed and the plain Buzhash and Rabin attacks require the rolling hash function to not be keyed. However, we cannot formally prove that this rolling hash function prevents finger-

printing attacks in general, as a formal definition of fingerprinting has yet to be defined.

Chapter 6

Conclusion

We endeavored to analyze security properties and discover potential vulnerabilities of commonly used cloud backup solutions. To this end, we defined two threat models, a weak adversary, and a strong adversary, defined fingerprint resistance in the context of cloud backup solutions, and studied popular cloud backup solutions for achieved (or non-achieved) security properties under our threat models.

Although we did not find attacks against data confidentiality, data integrity, or metadata integrity thanks to their simple design and general understanding of AEAD usage, our analysis of six popular cloud backup solutions in Chapter 5 showed that many cloud backup solutions that provide chunk-level data deduplication are vulnerable to side-channel attacks which break fingerprinting resistance, even against our *weak adversary* as defined in Section 1.3.

6.1 Discussion

An adversary with the ability to successfully execute a fingerprinting attack can check for an arbitrary user if the user possesses any specific files. This adversary might take the form of, for example, a nation-state that aspires to censor certain books or find independent journalists who are acting against the will of the state. If some journalists communicate amongst themselves and share a file that the nation-state knows of, for example, a guidebook on how to successfully evade the nation-state's surveillance, the nation-state can test all users of a backup solutions' backups for that file and discover the identities of the journalists.

We showed that, as of the date of publication of this thesis, three of the cloud backup solutions we studied including Borg, Kopia, and Restic are vulnerable to fingerprinting attacks under the assumption of a weak adver-

sary with full access to the server and knowledge of a specific file. Tarsnap implements a complex chunking algorithm with parameters derived from secret keys, for which we leave its security analysis for further research, and Bupstash introduces a secret that prevents our attacks against content-defined chunking. EteSync does not provide deduplication and is thus not exploitable through our attacks.

Borg introduces a secret chunker seed to counteract the plain Buzhash attack outlined in Section 5.1. However, we rediscovered an attack on Borg's implementation and showed that it is trivial for a weak adversary to extract the secret chunker seed, which nullifies Borg's mitigation of the plain Buzhash attack and thus breaks fingerprinting resistance in Borg even under the weak adversary.

In Section 5.8 we recommended a method to implement secure content-defined chunking using a secure block cipher such as AES. We highly recommend all cloud backup solutions that support chunk-level deduplication using content-defined chunking to implement this scheme and secure themselves against fingerprinting attacks.

6.2 Future Work

We showed that content-defined chunking algorithms that use Buzhash or Rabin fingerprints as is without introducing any secrets do not provide fingerprint resistance even against a weak adversary. Our analysis focused on six individual cloud backup solutions and we found half of them to be vulnerable to fingerprinting attacks.

Name	Content-defined Chunking	Vulnerable to Fingerprinting
Borg	✓	✓
EteSync	✗	✗
Tarsnap	✓	?
Kopia	✓	✓
Bupstash	✓	?
Restic	✓	✓

Further, Tarsnap's implementation of content-defined chunking evaluates polynomials with parameters derived from secret keys. A deep analysis of the security of this scheme was out of scope for this thesis, but we assume that there exists some plaintext and parameter information is leaked. Bupstash's chunking algorithm implementation also reveals some information about the secret hash table.

Thus we propose that further research focus on the security of other cloud backup solutions that support chunk-level deduplication with content-defined

chunking, and specific implementations such as Tarsnap's and Bupstash's chunking algorithms.

Appendix A

Appendix

A.1 Borg Chunk Size Probability Analysis

Let X be a random variable defined as the amount of bytes of input data resulting in a cut by the last $k = 21$ bits of the Buzhash resulting in zero. Assuming a uniformly distributed hash table and a uniformly distributed hash input, the probability of the Buzhash resulting in a chunk after k bytes is:

$$
\begin{aligned}
Pr[X = k] &= 0 && \text{for } k < 2^{19} + 4095 \\
Pr[X = k] &= \frac{1}{2^{21}} && \text{for } k = 2^{19} + 4095 \\
Pr[X = k] &= (1 - \frac{1}{2^{21}})\frac{1}{2^{21}} && \text{for } k = 2^{19} + 4095 + 1 \\
Pr[X = k] &= (1 - \frac{1}{2^{21}})^2\frac{1}{2^{21}} && \text{for } k = 2^{19} + 4095 + 2 \\
&\dots \\
Pr[X = k] &= (1 - \frac{1}{2^{21}})^{2^{23}-2^{19}}\frac{1}{2^{21}} && \text{for } k = 2^{23} + 4095 \\
Pr[X = k] &= 0 && \text{for } k > 2^{23} + 4095
\end{aligned}
$$

i.e.

$$
Pr[X = k] = \begin{cases} (1 - \frac{1}{2^{21}})^{k-2^{19}-4095}\frac{1}{2^{21}} & \text{for } 2^{19} + 4095 \leq k \leq 2^{23} + 4095 \\ 0 & \text{else} \end{cases} \tag{A.1}
$$

thus X is geometrically distributed $X \sim Geo(\frac{1}{2^{21}})$ for $2^{19} + 4095 \leq X \leq 2^{23} + 4095$, from which we derive

$$Pr[X \leq k] = \begin{cases} 0 & \text{for } k < 2^{19} + 4095 \\ 1 - (1 - \frac{1}{2^{21}})^{2^{23} - 2^{19} + 1} & \text{for } k > 2^{23} + 4095 \\ 1 - (1 - \frac{1}{2^{21}})^{k - (2^{19} + 4095 - 1)} & \text{else} \end{cases} \tag{A.2}$$

We expect to cut a chunk after 1568770 bytes

$$\begin{aligned} \mathbb{E}[X] &= 2^{21} - (2^{19} + 4095 - 1) \\ &= 1568770 \end{aligned} \tag{A.3}$$

The probability of a chunk being cut after 2 MiB is

$$\begin{aligned} Pr[X \leq 2^{21}] &= 1 - (1 - \frac{1}{2^{21}})^{2^{21} - (2^{19} + 4095 - 1)} \\ &\approx 0.527 \end{aligned} \tag{A.4}$$

Bibliography

[1] Jean-Philippe Aumasson. *Blake2 Homepage*. URL: https://www.blake2.net/. (accessed: 24.08.2023).

[2] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. "Message-locked encryption and secure deduplication". In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2013, pp. 296–312.

[3] Borg. *Attack Model*. URL: https://borgbackup.readthedocs.io/en/stable/internals/security.html#attack-model. (accessed: 26.07.2023).

[4] Borg. *Borg Homepage*. URL: https://www.borgbackup.org/. (accessed: 26.08.2023).

[5] Borg. *Buzhash Chunker*. URL: https://borgbackup.readthedocs.io/en/stable/internals/data-structures.html#buzhash-chunker. (accessed: 18.07.2023).

[6] Borg. *Fingerprinting*. URL: https://borgbackup.readthedocs.io/en/stable/internals/security.html#fixed-chunker. (accessed: 18.07.2023).

[7] Borg. *Fixed Chunker*. URL: https://borgbackup.readthedocs.io/en/stable/internals/security.html#fixed-chunker. (accessed: 23.08.2023).

[8] Borg. *Offline Key Security*. URL: https://borgbackup.readthedocs.io/en/stable/internals/security.html#offline-key-security. (accessed: 25.08.2023).

[9] Borg. *The Object Graph*. URL: https://borgbackup.readthedocs.io/en/stable/internals/data-structures.html#the-object-graph. (accessed: 26.08.2023).

[10] Bupstash. *Bupstash Github*. URL: https://github.com/andrewchambers/bupstash. (accessed: 18.08.2023).

[11] Bupstash. *Bupstash Homepage*. URL: https://bupstash.io/. (accessed: 26.08.2023).

[12] D.W. *Cryptographically secure keyed rolling hash function*. URL: https://crypto.stackexchange.com/a/16087. (accessed: 25.08.2023).

[13] John R Douceur, Atul Adya, William J Bolosky, P Simon, and Marvin Theimer. "Reclaiming space from duplicate files in a serverless distributed file system". In: *Proceedings 22nd international conference on distributed computing systems*. IEEE. 2002, pp. 617–624.

[14] Christophe-Marie Duquesne. *Buzhash32*. URL: https://github.com/chmduquesne/rollinghash/blob/master/buzhash32/buzhash32.go. (accessed: 04.08.2023).

[15] Etebase. *Etebase Homepage*. URL: https://etebase.com. (accessed: 29.07.2023).

[16] Borg Github. *Issue 3687: Small file size and directory order information leakage*. URL: https://github.com/borgbackup/borg/issues/3687. (accessed: 14.08.2023).

[17] Shay Gueron. "Key Committing AEADs". In: *Cryptology ePrint Archive* (2020).

[18] Jakob Jonsson and Burt Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. RFC 3447. Feb. 2003. DOI: 10.17487/RFC3447. URL: https://www.rfc-editor.org/info/rfc3447.

[19] Sriram Keelveedhi, Mihir Bellare, and Thomas Ristenpart. "{DupLESS}:{Server-Aided} encryption for deduplicated storage". In: *22nd USENIX security symposium (USENIX security 13)*. 2013, pp. 179–194.

[20] Kopia. *Kopia Documentation*. URL: https://kopia.io/docs/. (accessed: 04.08.2023).

[21] Kopia. *Kopia Homepage*. URL: https://kopia.io/. (accessed: 26.08.2023).

[22] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. Feb. 1997. DOI: 10.17487/RFC2104. URL: https://www.rfc-editor.org/info/rfc2104.

[23] Jin Li, Xiaofeng Chen, Mingqiang Li, Jingwei Li, Patrick PC Lee, and Wenjing Lou. "Secure deduplication with efficient and reliable convergent key management". In: *IEEE transactions on parallel and distributed systems* 25.6 (2013), pp. 1615–1625.

[24] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Connection Protocol*. RFC 4254. Jan. 2006. DOI: 10.17487/RFC4254. URL: https://www.rfc-editor.org/info/rfc4254.

[25] Moinakg. *High Performance Content Defined Chunking*. URL: https://moinakg.wordpress.com/tag/rabin-fingerprint/. (accessed: 26.08.2023).

[26] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. June 2018. DOI: 10.17487/RFC8439. URL: https://www.rfc-editor.org/info/rfc8439.

[27] OpenSSL. *OpenSSL Homepage*. URL: https://www.openssl.org/. (accessed: 23.08.2023).

[28] The Washington Post. *NSA infiltrates links to Yahoo, Google data centers worldwide, Snowden documents say*. URL: https://www.washingtonpost.com/world/national-security/nsa-infiltrates-links-to-yahoo-google-data-centers-worldwide-snowden-documents-say/2013/10/30/e51d661e-4166-11e3-8b74-d89d714ca4dd_story.html. (accessed: 24.07.2023).

[29] Pasquale Puzio, Refik Molva, Melek Önen, and Sergio Loureiro. "ClouDedup: Secure deduplication with encrypted data for cloud storage". In: *2013 IEEE 5th international conference on cloud computing technology and science*. Vol. 1. IEEE. 2013, pp. 363–370.

[30] Michael O Rabin. "Fingerprinting by random polynomials". In: *Technical report* (1981).

[31] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[32] Restic. *Chunker Github*. URL: https://github.com/restic/chunker. (accessed: 25.08.2023).

[33] Restic. *Restic Github*. URL: https://github.com/restic/restic. (accessed: 18.08.2023).

[34] Restic. *Restic Homepage*. URL: https://restic.net/. (accessed: 26.08.2023).

[35] Haonan Su, Dong Zheng, and Yinghui Zhang. "An Efficient and Secure Deduplication Scheme Based on Rabin Fingerprinting in Cloud Storage". In: *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. Vol. 1. 2017, pp. 833–836. DOI: 10.1109/CSE-EUC.2017.166.

[36] Tarsnap. *Tarsnap Homepage*. URL: https://tarsnap.com. (accessed: 29.07.2023).

[37] Robert Uzgalis. *BUZ hash*. URL: http://www.serve.net/buz/Notes.1st.year/HTML/C6/rand.012.html. (accessed: 13.06.2023).

[38] Thomas Waldmann. *Borg Commit 8ac272f*. URL: https://github.com/ThomasWaldmann/borg/commit/8ac272f35f9c3ee51bbe42c32616b54f05911233. (accessed: 23.08.2023).

[39] Wikipedia contributors. *Transaction log — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-August-2023]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Transaction_log&oldid=1098930416.