Ref. Ares(2017)3838631 - 31/07/2017

prisma cloud

Advanced architecture for distributed storage in dynamic environments (SECOSTOR Tool)

Deliverable D5.3

Editor Name	T. Loruenser (AIT)
Type	Report
Dissem. Level	Public
Release Date	July 29, 2017
Version	1.0

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644962.

More information available at https://prismacloud.eu.

Copyright Statement

The work described in this document has been conducted within the PRISMACLOUD project. This document reflects only the PRISMACLOUD Consortium view and the European Union is not responsible for any use that may be made of the information it contains. This document and its content are the property of the PRISMACLOUD Consortium. All rights relevant to this document are determined by the applicable laws. Access to this document does not grant any right or license on the document or its contents. This document or its contents are not to be used or treated in any manner inconsistent with the rights or interests of the PRISMACLOUD Consortium or the Partners detriment and are not to be disclosed externally without prior written consent from the PRISMACLOUD Partners.

Each PRISMACLOUD Partner may use this document in conformity with the PRISMACLOUD Consortium Grant Agreement provisions.

Document information

Project Context

Work Package	WP5 Basic Building Blocks for Secure Services
Task	T5.1 Secure Cloud Storage Solutions
Dependencies	D2.3, D4.1, D4.2, D4.3, D5.1, D5.3, D7.3, D7.4, D7.5

Author List

Organization	Name	E-mail
AIT	Thomas Lorünser	Thomas.Loruenser@ait.ac.at
AIT	Andreas Happe	Andreas.Happe@ait.ac.at
AIT	Benjamin Rainer	Benjamin.Rainer@ait.ac.at
AIT	Florian Wohner	Florian.Wohner.fl@ait.ac.at
AIT	Christoph Striecks	Christoph.Striecks@ait.ac.at
TUDA	Denise Demirel	ddemirel@cdc.informatik.tu-darmstadt.de
TUDA	Giulia Traverso	gtraverso@cdc.informatik.tu-darmstadt.de

Reviewer List

Organization	Name	E-mail
AIT	Stephan Krenn	stephan.krenn@ait.ac.at
ATOS	Angel Palomares	angel.palomares@atos.net
LISPA	Marco Decandia Brocca	marcodec.lispa@gmail.com

Version History

Version	Date	Reason/Change	Editor
0.1	2017-05-11	Basic structure and content from D5.2	Thomas Lorünser, Andreas Happe, Florian Wohner and Benjamin Rainer
0.2	2017-06-11	Extension of structure and fist content for full version	Thomas Lorünser
0.3	2017-06-08	Added methods for mixing with modern asymetric cryptography	Christoph Striecks
0.4	2017-06-21	Update on implementation in section 5	Thomas Lorünser and Bejamin Rainer
0.5	2017-06-29	Add section about additional topics and availability model	Thomas Lorünser, Guilia Traverso, Denise Demirel
0.6	2017-07-03	Final updates on section 9	Bejamin Rainer
0.7	2017-07-21	Final remarks and fixes	Christoph Striecks
1.0	2017-07-29	Addressed reviewers comments. Final version ready.	Thomas Lorünser

Executive Summary

In this report we present the specification of the secure object storage tool (SECOSTOR), as it has been developed in the project. The tool comes with a clean architecture and easy to use modules and interfaces. All components are specified in detail for implementation and usage. All together this report specifies core functionality to build secure distributed storage systems on the basis of secret sharing with many additional features not found in comparable projects and solutions. With its rich feature set it supports many possibilities for integration in cloud environments or similar infrastructures, hence, it is a useful tool for service and application developers who want to leverage the technology.

The document contains a description of the overall SECOSTOR tool architecture as well as of the three core software modules. One module is a comprehensive secret sharing library written in Java and comprising various encoding algorithms. The second module also implements secret sharing, but in JavaScript to target client side browser integration.The third one provides a robust concurrency layer for distributed transaction management. It has been developed in JavaScript (Node.js) and works seamlessly together with the provided secret sharing library. Using state of the art web technologies for our tool enables many options for integration and deployment, especially if public cloud providers are part of the configuration.

The architecture and protocols defined are based on the cryptographic work done in WP4, i.e. D4.1, D4.2 and D4.3. It also builds on the previous results of WP5, i.e., D5.1 and D5.2. More information about the software implementation can be found in WP6 reports and full documentation of the cloud services developed on the basis of the SECOSTOR in reports of WP7. The capabilities of the tool will be demonstrated in two services (SAaaS and DSaaS) which are going to be developed on the basis of the SECOSTOR tool and which will be piloted in two use cases during the last phase of the project.

Table of Contents

Executive Summary 1

1 Introduction 4
- 1.1 Scope of the document . . . 4
- 1.2 Relation to other project work . . . 4
- 1.3 Structure of the document . . . 6

2 The SECOSTOR Tool 7
- 2.1 Overview . . . 7
- 2.2 Tool architecture . . . 9
- 2.3 Component Model . . . 10
- 2.4 Software Implementation . . . 11
- 2.5 Services Based on SECOSTOR . . . 12

3 Terms and Definitions 14
- 3.1 Secret Sharing Basics . . . 14
- 3.2 Remote Data Checking . . . 15
- 3.3 Adversary models . . . 15
  - 3.3.1 Computational Power . . . 15
  - 3.3.2 Network Model . . . 16
  - 3.3.3 Protocol Level . . . 16
  - 3.3.4 Consistency Models . . . 17

4 Secret Sharing Library 19
- 4.1 Overview . . . 19
- 4.2 Parameters and Properties . . . 20
- 4.3 Algorithms and Protocols . . . 21
- 4.4 Operational Aspects . . . 25
- 4.5 Architecture and Design of Software Library . . . 26
  - 4.5.1 Java Library . . . 26
  - 4.5.2 JavaScript Library . . . 27

5 The Byzantine Fault Tolerance Library 29
- 5.1 Overview . . . 29
- 5.2 Algorithms and Protocols . . . 29
- 5.3 Architecture and Design of Software Library . . . 31

6 Additional Functionalities **34**
6.1 Remote Data Checking 34
6.2 Protection from Malicious Clients 38
6.3 Reverting to a Consistent State 40
6.4 Private Information Retrieval 40

7 Access Control and Data Sharing **44**
7.1 Beyond plain access control 44
7.2 Attribute-based encryption for access control and data sharing 45
7.3 A note on lightweight data sharing and revocation 47

8 Availability Model **48**
8.1 Theoretical Secret-Sharing Performance 48
8.2 Availability Model 49

9 Applications based on the tools **52**
9.1 Archistar Backup Proxy 52
9.1.1 Architecture 52
9.1.2 Implementation and Evaluation 54
9.2 Archistar Web Based Data Sharing 58
9.2.1 Preliminary Performance Evluation 58

10 Summary and Outlook **60**

List of Figures **62**

List of Tables **62**

Bibliography **67**

1 Introduction

This section is intended to give an overview of the document, introduce the expected content, explain the project context and give the reader some guidance for quick access to the most relevant information.

1.1 Scope of the document

In this document we present the results on the design of PRISMACLOUD's secure object storage tool *SECOSTOR*. The tool enables developers to build a more reliable and secure storage environment for unstructured data compared to existing solutions by leveraging a distributed (multi-cloud) architecture. The tool achieves this goal by fragmentation of plaintext data and dispersal of the generated fragments over different storage locations called servers. The tool concept follows the PRISMACLOUD architecture shown in Figure 1 which has been introduced and explained in D7.5.

The *SECOSTOR* tool specification gives guidance for the design and provides components to aid development of a storage service. It encapsulates cryptographic tasks as well as data manipulation operations within relatively simple software modules that can be used to build new or integrate existing storage services. This document makes the research results in cryptography from WP4 accessible for software developers, such that they can easily integrate them into their own software products. The different software modules can be used and combined in various ways and optional features can be used to augment the capabilities. This flexibility allows for the realization of many use cases in the context of data storage with many possible deployment and trust models underneath.

This document delivers a final specification of the PRISMACLOUD SECOSTOR tool and its core components, as it is going to be used in the service development and piloting of the project. The tool will be further extended in the last phase of the project, but only on the conceptual level. However, partner AIT is maintaining the software components and is planning to further extend its functionality even after the project's end.

1.2 Relation to other project work

This report specifies the *SECOSTOR* tool, its overall tool architecture and the core components developed in PRISMACLOUD. It integrates major results from work done in $WP4$ with respect to secure storage, namely *D4.1 Secret sharing protocols for various adversary models* and *D4.2 Progress report on efficient sharing based storage protocols* as well as *D4.3 Efficient sharing based protocols for mixed adversaries.*

Furthermore, it builds on the state-of-the art analysis and first conceptual work done in *D5.1 Design of distributed storage systems without single-point-of-failure.* A preliminary version of the SECOSTOR tool was already delivered in *D5.2 Progress report on architectures for distributed storage in dynamic environments.* This was done to test the structure of the tool documentation and to provide a first version of the tool for service developers

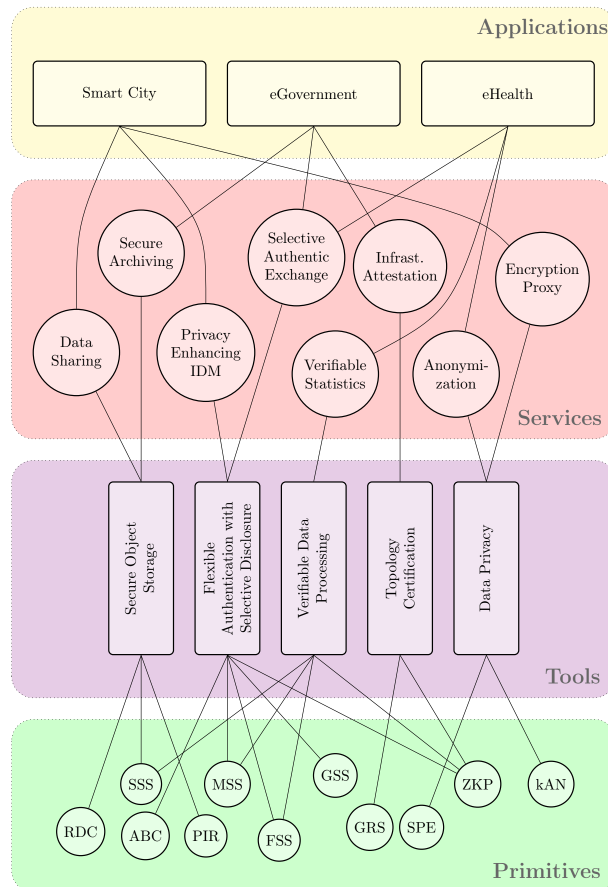


Figure 1: PRISMACLOUD architecture.

to start their work and it is now superseded by this report.

The SECOSTOR tool is one of the five tools developed in the project comprising the PRISMACLOUD toolkit or toolbox. Therefore the full toolkit specification is comprised by the following documents:

- D5.3 — SECOSTOR tool (this report, full title: Advanced architecture for distributed storage in dynamic environments)
- D5.4 — DATAPRIV tool (full title: Tools for encryption and tokenization techniques)
- D5.7 — TOPOCERT tool (full title: Final report on privacy and anonymization techniques)
- D5.9 — FLEXAUTH tool (full title: Analysis of Malleable signatures for defining allowed modification and providing verifiable means of conformant processing)
- D5.10 — VERIDAP tool (full title: Privately and publicly verifiable computing techniques providing privacy, integrity and authenticity)

Finally, there is also a close relation to WP6, where parts of the software implementation happens. In particular, the final software implementation of the SECOSTOR tool is deliv-

ered as part of the PRISMACLOUD toolkit in *D6.6 Final Release Efficient Implementation of Selected Components*.

1.3 Structure of the document

In section 2, a high-level overview of the *SECOSTOR* tool and its main idea and architecture is given. Before going into detail, in section 3 the most basic terms are defined in an ontology.

In the following sections the core software modules are specified and explained. In particular the *SECOSTOR* tool consists of two main software components, one being a secret sharing component and the other a robust and fault tolerant concurrency module. Furthermore, for the secret sharing component we support two different software implementations. One is in Java and the other one in JavaScript. For the concurrency layer we present a JavaScript implementation which fully integrated with the secret sharing component and based on modern web technologies (Node.js).

After introducing the software libraries, in section 6 we discuss protocols which can be built and added to storage system, which go beyond the basic storage functionalities. We present solutions which — in our opinion — fit best into the SECOSTOR concept and introduce least overhead, computation and/or communication wise.

In section 7 we discuss and address the importance of access control for secret sharing based systems which base its security on the on-collusion assumption. We also present cryptographic options and most suitable technologies from asymmetric cryptography which can be used to improve collusion resistance and further strengthen the security of the system, but without limiting data sharing capabilities in dynamic groups and without introducing substantial computational burden on the client or the server.

In section 8 we discuss the availability benefits we get and the storage overhead accompanying the different configurations. We also discuss, how virtual multi-cloud storage service enable customers in the design of their own dedicated availability based on given standard services from the cloud.

In the last technical section 9 we give an overview of the services and applications we are building in the project based on the SECOSTOR tool. The sample applications should give the reader a good understanding of the different trust models and security levels which can be achieved and what they can expect from the tool.

Finally in section 10 we summarize the status of the development, draw some first conclusions after developing the tool and give an outlook of future work to be addressed in the piloting phase and even after the project.

2 The SECOSTOR Tool

In this section we present the basic idea and concept of the SECOSTOR tool developed. We show the abstract concept and architecture of the tool as developed in [LSLP16]. We introduce the software libraries developed and how they are used in the project to build secure services, i.e., in particular we summarize the features which can be achieved to protect data at rest in multi-cloud applications.

2.1 Overview

The secure object storage tool *SECOSTOR* is dedicated to building secure cloud storage systems which protect data from provider related threats.

To achieve the desired security properties we are leveraging the concept of secure information dispersal – data is fragmented and distributed over different servers to build a secure and reliable virtual service on top of multiple less reliable services. Due to the use of secret sharing the individual fragments contain no information about the plaintext data. Our goal is to design a storage layer which is more than the sum of its parts. No single cloud provider shall have access to plaintext data or be able to tamper with them by modifying local fragments nor are all fragments necessary to recover the plaintext. In summary, the overall availability achieved by the Secure Object Storage Tool is better than the one of individual nodes and the tool also prevents from most of provider related threats in cloud usage.

In PRISMACLOUD [LRD+15] we apply secure variants of data fragmentation called secret sharing. With secret sharing, data can be encoded into multiple fragments such that no single fragment contains any information about the original content and a predefined threshold of k fragments is required to reconstruct the data. If these fragments (shares) are distributed to different storage servers, the data stays secure w.r.t. confidentiality protection as long as less than k servers are colluding to reconstruct the information and the data stays secure w.r.t. availability as long as at least k honest servers are reachable. Additionally, it must be noted that the confidentiality and availability guarantees of the PRISMACLOUD *Secure Object Storage Tool* are achieved via a fully keyless architecture, which further facilitates sharing of data between different users in the system.

From a functional point-of-view, the tool provides a way for outsourcing data storage to potentially less trustworthy and reliable storage providers with both increased data availability and confidentiality [LAS15]. Furthermore, we are targeting robustness against strong adversary models like active attackers or adversaries with unlimited computational power. Additionally the tool is supporting concurrent multi-user access by potentially malicious clients and provides other interesting features discussed below.

In Figure 2, the basic structure of a storage system based on a full scale *SECOSTOR* deployment is shown.

The main **advantages** from such a system are the following:

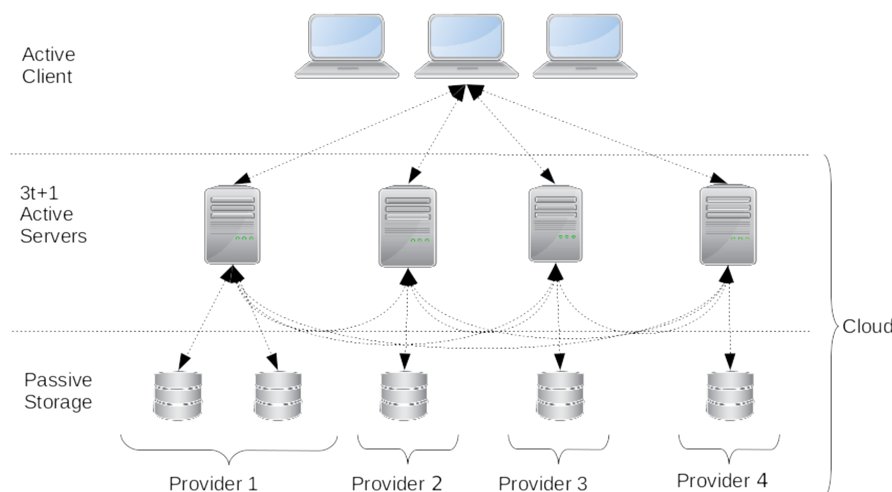


Figure 2: *SECOSTOR* overview.

- Increased data confidentiality and availability: The usage of threshold secret sharing for data fragmentation improves both data confidentiality and availability. This is due to parties in the system only holding a single data fragment are not able to read or tamper plaintext data. Availability is given by the fact that only a subset of generated fragments is needed to reconstruct the original data.
- Reduced storage overhead: if computational-secure secret sharing is used, the overall amount of data capacity required to reach a certain level of availability can be substantially reduced in comparison to simple mirroring strategies or perfectly secure secret sharing. In Figure 3 the overall amount of data used is compared to a simple 2 replicas strategy. The advantage is substantial for higher number of nodes, i.e., if more than 4 nodes are available to store fragments.
- Prevent from provider lock-in: by its very nature, the system builds on standard technologies and enable interoperability between different storage providers. The tool itself is not dependent on any particular storage offering and used data formats are fully open and documented. Using open formats as well as the usage of published and well documented algorithms is essential to the tool. In particular, this report (and its successor) serves as the main specification of the *SECOSTOR* tool, detailing all aspects of the implemented features and mechanisms.
- Keyless operation: using encoding instead of standard encryption techniques gives the advantage of keyless operation. Security is governed by non-collusion of parties holding the fragments and not by any private key. Without a key, that there is no complex key management required which is very favorable in many situations, although credentials are usually still needed. Nevertheless, the practical relevance of security based on non-collusion assumption is still a controversial topic, secret sharing is the only known information theoretical secure method to protect data at rest, i.e., it is the best candidate to start with when building long-term secure systems.

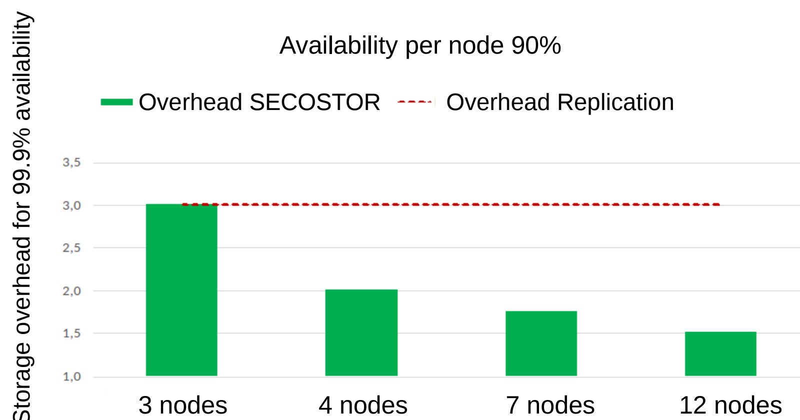


Figure 3: Required redundancy to achieve high-availability.

2.2 Tool architecture

The tool provides essential functionalities to build systems as shown in Figure 2. The tool's architecture makes usage as simple as possible and encapsulates cryptographic protocols as good as possible (cf. Figure 4). The tool comprises two main modules:

archsitar-smc is a comprehensive secret sharing library containing the most important algorithms relevant for the storage setting. It supports efficient encoding and decoding for different adversary scenarios, i.e. the different secret sharing algorithms selected.

archsitar-bft is *SECOSTOR*'s second core library. It contains all components necessary to easily support concurrency and robustness in a distributed system. It provides robustness against active Byzantine attackers or failures and therefore provides the strongest possible notion of security for our storage network. This module is essential for scenarios where multiple clients or users have to be supported.

While they can be run alone, they have been also been designed to be used in conjuction with each other. In addition, higher level protocols will be built from the basic tools by configuring them in a specific manner or applying them for specific purposes, e.g, like for remote data checking or secure deletion.

Both modules are part of the *Archistar* software framework. *Archistar* is intended to provide a full *SECOSTOR* implementation as well as sample service implementations to simplify adoption of PRISMACLOUD's results. As such the Archistar framework will be a comprehensive environment for commercialization of project results in the area of secure object storage.

In addition to the two main software libraries the *SECOSTOR* tool will provide additional guidance in the usage and application of the provided software and present some recommendations for integration in services and applications.

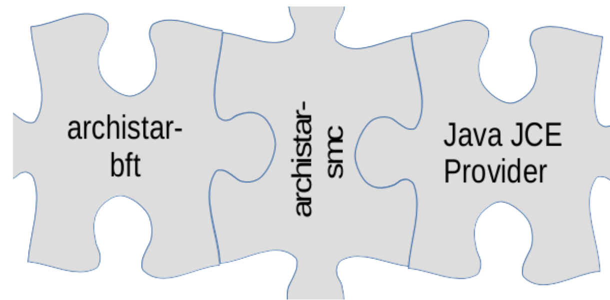


Figure 4: Core libraries of the *SECOSTOR* tool.

2.3 Component Model

This tool is dedicated to build cloud storage systems with strong security guarantees, w.r.t., confidentiality and availability. It builds upon the idea of a federated or multi-cloud application [SH12] and the different components of the tool are shown in Figure 5. A *dealer* component generates data fragments and sends them to storage nodes called *servers*. The servers can be considered storage nodes which are also comprising active components with the ability to execute protocol logic in addition to the basic read and write operations supported by passive cloud storage interfaces. The *reader* component is reconstructing the data for read operations and the *verifier* is able to audit the storage system, i.e., it can remotely check the state of the data stored in the system.

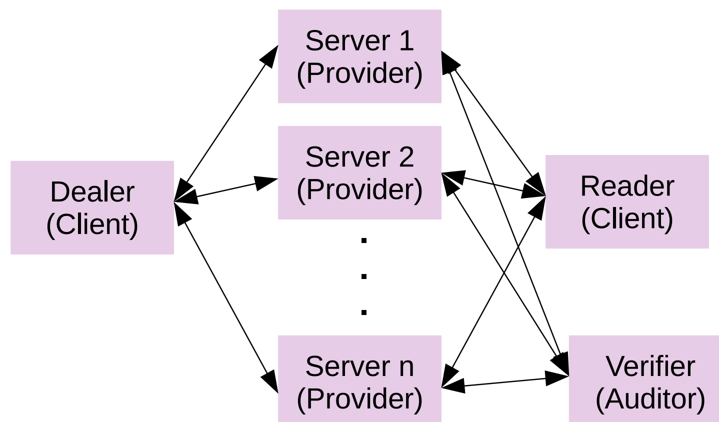


Figure 5: Secure Object Storage Tool (SECSTOR).

In the following we are briefly reviewing the components and their function and interfaces.

Dealer Component. The main function of the dealer component (Dealer) is to encode the data and generate the different fragments used in the distributed storage network. The dealer works on unstructured data, and at its core, it implements secret sharing protocols. However, in order to support different application scenarios, adversary models

and network models, the component must support various encoding schemes and protocols. The common functionality behind the different protocols is that of threshold secret sharing. The choice of threshold sharing used ranges from plain information theoretical solutions (PSS) to more efficient but only computational secure variants (CSS) and also verifiable versions if the dealer component is running on untrusted platforms. On the connectivity level, the dealer requires an authentic and private communication channel to each server in the system.

Server Component. The server component (Server) represents the storage backend used to compose the cloud-of-clouds storage layer. Multiple servers are required and each of them communicates with the Dealer over a secure channel (authentic and private). The different servers are holding the data and represent storage nodes in different trust zones of the system. The trust zones are used to model the non-collusion assumption. In practice, storage options range from fully-federated dedicated cloud providers to storage nodes under different administrative domains within a cloud provider's data center. The server components are communicating with the dealer over secure channels and for some of the features provided by the tool, the servers also have secure channels between them.

Reader Component. The reader component (Reader) is responsible for reconstruction of data stored in the system. Basically it encapsulates the reconstruction procedure of the used secret sharing method plus the interaction protocol to get sufficient shares to recover the plaintext information desired. We assume secure channels between a reader and all involved servers in the system.

Verifier Component. This component is responsible for the verification of data stored in the system. Together with the servers it conducts a protocol to obtain a proof about the retrievability of stored data, i.e., it remotely checks if the servers are still storing consistent fragments. The verifier is not considered to be trusted, therefore, the auditing protocols executed must be privacy preserving, i.e., it must not leak any information about the data stored. Based upon this, the verifier must have an authentic channel to each of the servers, but those channels don't need to be confidentiality protected. The Verifier is intended to model a third-party auditing service which can check the data consistency remotely with strong cryptographic properties but without learning anything about the data.

2.4 Software Implementation

The modules of the *SECOSTOR* tool are currently being implemented. The secret sharing library *archistar-smc* is implemented in Java and provides a clean interface to access various kinds of secret sharing algorithms. In addition, there is a JavaScript implementation under development to fully support client side encoding within browsers.

The BFT module (*archistar-bft*) will encapsulate the concurrency layer of the distributed storage system and is also under development. It is implemented in JavaScript on the

basis of modern web technologies (Node.js). The current version supports the storage of plaintext data in redundant forms; integration of secret sharing protocols is ongoing.

2.5 Services Based on SECOSTOR

In PRISMACLOUD two services are based on the *SECOSTOR* tool. The services are aiming at different application scenarios and require different features of the tool. In Figure 6 the basic idea for deployment of different components is shown for the two flavor of services. In the following we briefly review the services, a detailed description of the services can be found in *D*7.5.

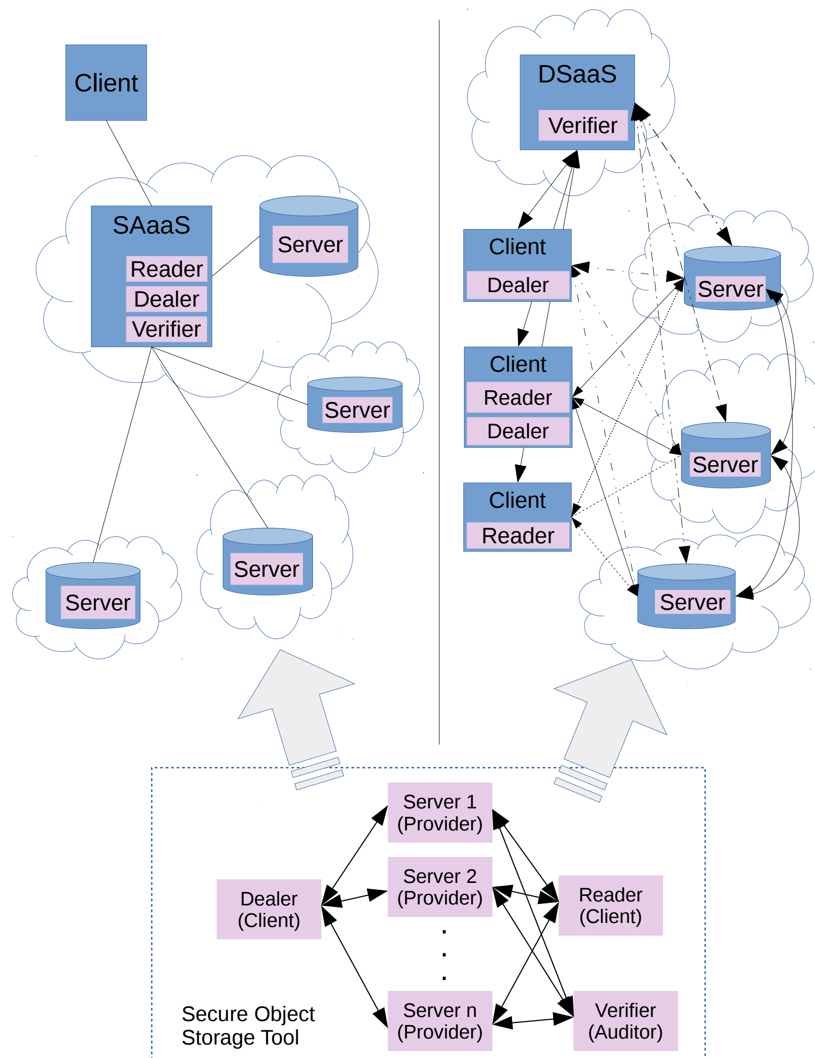


Figure 6: Services based on SECOSTOR.

Secure Archiving Service (SA or SAaaS). This PRISMACLOUD service is intended as a replacement of existing storage services. The service provides a standard object storage interface to end-users and performs all data fragmentation and dispersal transparently w.r.t. the back-end. It serves as a gateway hiding the full complexity from the user of the cloud storage solution. However, because the gateway does everything in one place it has to be fully trusted by the user and run in a trustworthy environment. Due to this trust-model and the provided interface it is ideally suited in backup and archiving scenarios, like also demonstrated in the e-Government pilot of the project. In summary, the service can easily be integrated into cloud based backup solutions and provides a demonstratable higher level of data privacy and availability than the usage of standard cloud storage services. The service model for this service is IaaS.

Data Sharing Service (DS or DSaaS). This PRISMACLOUD service targets scenarios with multiple users who like to share data between them. The service allows multiple parties to securely store data in a cloud-of-clouds network, such that no single storage node learns plaintext data, while still enabling the owner to share the data with other users of the system, i.e., the data sharing service supports secure collaboration without the need to trust one single storage provider. The service model of this service is IaaS and the service is intended to be deployed in a private cloud but enables the user to build hybrid cloud storage systems.

3 Terms and Definitions

In this section we introduce an ontology for the *SECOSTOR* tools. We summarize and define most important terms in order to establish an unambiguous vocabulary for the project and beyond.

3.1 Secret Sharing Basics

Secret sharing is a core functionality of the *SECOSTOR* tool and at the heart of its encoding engine. In the following we are going to define most important terms for the specification of secret sharing algorithms.

For best compatibility with standards, the secret sharing part of the ontology is heavily based on ISO/IEC 19592-1:2016, which has just recently been released and is exactly intended as reference and basis to specify secret sharing algorithms. In fact, the first standard on secret sharing algorithms is currently in preparation and will be published as part 2 under ISO/IEC 19592-2:2017 very soon. All this standards activities of secret sharing have great importance for us and come in the right time to support our exploitation activities. It is also interesting to observe, that although secret sharing is a rather old concept, standardization of it in ISO was just started recently. Nevertheless, the upcoming standard is very basic in nature and not containing all relevant algorithms relevant for efficient storage scenarios nor advanced protocols with verifiability properties.

The following values and algorithms are used:

Message. The message is the secret which is going to be encoded for confidentiality protection.

Treshold. The minimal number of shares needed for the share vector to reconstruct the message. In the following, this value is denoted by k.

Message sharing algorithm. The message sharing algorithms is the *Share* method from the secret sharing algorithm. It takes a message as input and outputs the shares.

Share. Shares are the fragments generated by encoding the input message in sharing algorithm *Share* which are then stored at the server by the dealer.

Share vector. A vector of shares output by the sharing algorithm *Share*.

Message reconstruction algorithms. The message reconstruction algorithm is the *Rec* procedure from the sharing algorithm. It takes k the shares as input and outputs the recovered message.

Secret sharing scheme. A secret sharing scheme consists of a set of two algorithms, *Share* and *Rec*. *Share* is used to encode input message into n shares which are then distributed to different servers. A more formal definition is given in Section 4.3.

We distinguish between the following types of parties:

Dealer. The dealer is responsible for encoding of the data and he does so by calling the *Share* algorithm of the secret sharing scheme. The dealer is equivalent to the dealer component presented in Section 2.3.

Reader. The reader, also often called receiver or reconstructor, is receiving the shares and decoding the message by running the reconstruction algorithm *Rec*.

Server. A party holding shares distributed by the dealer.

3.2 Remote Data Checking

Remote data checking enables a party to check the retrievability of remotely stored data. One characteristic of this idea is, that it has to be more efficient than downloading all data and checking it locally. From the requirements gathering for the SECOSTOR tool we learned that this is an extremely favorable property for cloud based data backup and archival, therefore we made it an integral part of the SECOSTOR architecture.

Verifier. In the context of remote data checking, the verifier is the component triggering the protocol for checking the consistency and retrievability of shares stored on servers.

Remote data checking. A protocol run between a verifier and one or more provers to assure the possession or retrievability of data (shares) hold by the prover(s).

Prover. The entity holding the data (shares) and the counterpart of the verifier in the remote data checking interactive protocol. In the SECOSTOR model the servers are also the provers.

3.3 Adversary models

There are different ways to categorize adversaries based on the computational power they have, the network model they are bound to, and protocol level activities they can conduct. In the following we define the most relevant adversary models for our purpose. In a distributed system these modeling decision and assumption can have a bit impact on the design of the protocols.

3.3.1 Computational Power

Bounded vs. unbounded. We distinguish computationally bounded and unbounded adversaries. For the former, we assume that the number of computations the adversary can perform is bounded above by some polynomial in the security parameter. Security is then proved under some computational assumption; that is, breaking the scheme would require

the adversary to solve some computational problem which is believed to become super-polynomially more complex when increasing the security parameter. This approach is also taken for most other cryptographic primitives, such as encryption or signing. Protocols secure against bounded adversaries are also said to provide *conditional* security.

For unbounded adversaries, we do not pose any restrictions whatsoever on the computational power; schemes secure against unbounded adversaries are also called *information theoretically* or *unconditionally secure*, and are of key importance when designing long-term private cryptographic systems. This is because security is guaranteed independent of potential future developments in the computational power of the adversary.

3.3.2 Network Model

Synchronous vs. asynchronous. In synchronous systems, it is assumed that protocols proceed in rounds: all messages sent in one round are available to their receivers at the beginning of the next round. This assumption often allows for elegant protocols and relatively simple proofs. However, such a network model is only realistic if all nodes run at roughly the same speed, the network is very stable, etc. In contrast, in an asynchronous model, also network latency or simply different computation speeds on different nodes where one does not want to always wait for the slowest node are modeled. Assuming an asynchronous network is often far closer to reality, in particular for protocols that are to be carried out over the Internet and not only, e.g., in a company-internal data center. However, allowing asynchronicity often causes a considerable overhead in the computational costs of a protocol, and also results in much more intricate security proofs.

3.3.3 Protocol Level

Passive vs active. This is the most basic categorization of adversaries on a protocol level. A passive adversary is only allowed to corrupt a party in the sense that he may see the party's internal state and all messages it receives or sends, but it is not allowed to change the behavior of the machine. Such adversaries are also called *honest-but-curious*. In contrast, an active or *Byzantine* adversary may in general force corrupted nodes to deviate arbitrarily from the protocol specification.

Adaptive vs non-adaptive. Generally speaking, a non-adaptive adversary has to decide which parties to corrupt right away when the protocol starts. On the contrary, an adaptive adversary is allowed to corrupt parties depending on what he has already seen in the protocol so far. That is, an adaptive adversary is allowed to corrupt the "most promising" parties at a given stage of the protocol. One can think of a non-adaptive adversary as a malicious party which already compromises a system at manufacturing time, but which cannot compromise additional devices once they have been shipped, which would still be able for an adaptive adversary. From a practical point of view, adaptive security clearly becomes more important with an increasing lifetime of the system.

The precise meaning of adaptiveness depends on the concrete protocol, and will be specified in detail for each protocol.

Static vs mobile. Assuming a static adversary means that once a party got corrupted, it will stay corrupted until the end of the system. This may be reasonable for short living systems. However, in the case of long-term storage scenarios, a single node might be reset to an honest state once the corruption is detected. We call such an adversary mobile, as he is able to corrupt different nodes at different points in time. Sometimes the respective corruptions are referred to as *transient*.

A special case of mobile adversaries are so-called *smash-and-grab* attacks. There, it is assumed that the adversary corrupts a node, copies its entire internal state, and immediately leaves the node again. For instance, this is meaningful when one wants to model data leaks where an insider copies information and makes it publicly available.

Rushing vs non-rushing. This differentiation is specific to robust secret sharing schemes, namely when an adversary sends maliciously altered shares back to the user, e.g., in order to make it impossible to reconstruct the shared secret. There, a non-rushing adversary is forced to decide how he wants to alter the shares of corrupted servers before the reconstruction phase starts. A rushing adversary has the additional possibility to first see the shares sent by the honest nodes, and then to decide how to alter the shares.

This adaptive behavior of the adversary must not be confused with the categorization of adaptive or non-adaptive adversaries, which lets the adversary decide which parties to corrupt; so, even an adaptive adversary may be non-rushing or vice versa.

Honest vs corrupted dealer. An honest dealer is always assumed to send consistent shares to all servers, whereas a corrupted dealer might send arbitrary and inconsistent shares to the servers. In particular, for a corrupted dealer it cannot be guaranteed that every qualified set of shares reconstructs to the same secret, or whether it reconstructs to a valid secret at all. While assuming a dishonest dealer might seem artificial at first glance, this might, e.g., happen due to corrupted devices sending altered messages on the network interface without the user noticing. Also, transmission errors on a network level could be detected if a scheme is resistant against dishonest dealers.

3.3.4 Consistency Models

In the following, a *trace* is defined as sequence of *Read* and $Write$ operations upon a shared resource (i.e. x). The operations can be executed by one or more processes.

Strict consistency denotes that after a $Write$ operation $W(x)$ at time t all subsequent *Read* operations $R(x)$ on any node at time t' with $t < t'$ will always return the newly set value. As t' can be chosen arbitrarily small this implies that an operation's result is visible after execution on all nodes immediately.

Linearization or total ordering. Traces contain two orderings: internal and external. The internal order is given through the execution flow within a single process; the external order depends upon the ordering of execution time of all operations. Linearization is achieved if both orders are preserved, especially operations are executed in the same external order. In contrast to strict consistency this does not imply that an *write*-operation is immediately visible on all nodes at the same time.

Sequential consistency. Within this model all operations need the internal order but not the external order. All processes see all the write operations in the same order. Linearization is stronger than sequential consistency thus each linearized trace is also sequential consistent.

Causal consistency requires that processes see all writes that are causally related in the same order. The order of read operations must be consistent with the causal order. Write operations that are not causally related can be seen in any order by different processes. For example, given two non-related write operations $Write(x, 10, process1, t1)$ and $Write(x, 15, process2, t2)$, processes $process3$ and $process4$ are free to read the new values in any order i.e. $process3$ could first read $x = 10$ then $x = 15$ while $process4$ could do this in reverse. This example would not be sequentially consistent as all processes do not see all writes in the same order.

4 Secret Sharing Library

Now we are going to present the secret sharing library. Therefore, we start with a quick overview of supported algorithms and present major parameters relevant for its usage. We further specify the selected algorithms and discuss the current software implementation.

4.1 Overview

The basic idea of the *SECOSTOR* tool is to protect the confidentiality, integrity and availability of data by the application of secret sharing. By the use of secret sharing data is fragmented into different parts which are then stored at different server which are also different trust zones.

Trust zones are a way formalize the security given by the non-collusion assumption of secret sharing and they can be realized in different ways. If multiple cloud providers are used to store the fragments (shares) we are in a multi-cloud environment and the non-collusion is evident, because the different providers are not assumed to collude. However, the tool can also be used within an organizations boundary, if administrative boundaries exist between different parts of the infrastructure which can be used to implement trust zones. This is also a realistic assumptions, because larger infrastructures typically have different administrative zones to avoid larger breakdowns caused by configuration errors. Hence, also in a single operator situation can the *SECOSTOR* tool help to protect the confidentiality of customer data. Furthermore, if both concepts are combined in hybrid cloud scenarios the best of both worlds can be achieved.

In order to provide best flexibility and to cope with different adversary models we decided to support 4 main classes of algorithms. In particular we selected the following:

- Perfectly secure secret sharing (PSS): This mode provides security against unbounded attackers in the long term and is easy to use and implement. However, because its security is information theoretical, each share has the same length as the plaintext data. This mode is ideal for highest security requirements and small to medium size of data.
- Information dispersal (IDS): This mode does not support confidentiality at all, but efficiently fragments data to be protected against loss. Similarly to secret sharing data is split in parts whereby only a subset of the generated parts are required to reconstruct the message. However, the threshold functionality enables one to efficiently store data on multiple servers in a reliable way more efficiently that by simply replicating the information. This mode is ideal for high availability, if no confidentiality is needed.
- Computational secure secret sharing (CSS): This mode combines both properties of the above ones into a single secret sharing protocol. It provides efficient and secure way to fragment data, although its security can only be against computationally bounded adversaries. Furthermore, this method also does not have any homomorphic properties anymore and is therefore ideally suited for bulk storage of large amounts of data.

- Robust secret sharing (RSS): All of the above protocols support robust versions and are able to cope with actively malicious parties to some extent. However, if the robustness guarantees given by the different solutions are not good enough, specific protocols have been designed to even protect the integrity of data as long as the majority of parties are honest.

A detailed description of the implemented algorithm is discussed in Section 4.3 after we introduced the basic parameters and properties in the following.

4.2 Parameters and Properties

Following are the important *parameters* of the core secret sharing part of the *SECOSTOR* tool:

- Message space: finite field $\mathbb{F}_q$.
- Share space: same as the message space.
- Number of shares/parties: $n \geq 2$, $n < |\mathbb{F}_q|$
- Threshold: k, so that $n \geq k \geq 2$.
- Fixed field elements: $x_i \in \mathbb{Z}_q$ for $1 \geq i \geq n$.
- Number of actively corrupt parties: t
- Number of failstop parties (erasures): e
- i^{th} party/server: P_i
- Share given to P_i: σ_i

In our description we assume that each storage node is getting one share. When nodes with different trust levels are used it may be more efficient to vary the number of shares per server, i.e., more trustworthy nodes can hold more shares. In this case, the reader of this document can easily transfer the presented results to the respective scenario with individual servers holding multiple shares.

All secret sharing variants in the *SECOSTOR* tool only support a fixed message space of 256 (or 2^8). However, this does not impose any restriction on the algorithms, because compound data can always be processed in sequential form. Hence, arbitrary file length can be supported without loss of generality. In fact, the more efficient variants specifically leverage the possibility to encode more plaintext elements into one set of shares.

The main properties which will be analyzed for the different algorithms are

- Confidentiality: The level of security guarantees which will be provided.
- Integrity: The bounds for the protection of the identity.
- Information rate: Gives an indication of the share size which can be achieved.
- Homomorphic operations: Tells if the secret sharing scheme supports homomorphic operations.
- Complexity: Considerations regarding the computational complexity or the communication complexity of the algorithm.

4.3 Algorithms and Protocols

Different flavors of secret sharing exist in the literature and we selected the most relevant ones to be used in the *SECOSTOR* tool after assessment of the state-of-the-art in *D*4.1. For PRISMACLOUD we focus on secret sharing schemes with threshold access structures, because they are the most efficient and practical ones for storage systems. All variants share the same abstract interface which is defined in the following.

Abstract interface

A secret sharing scheme allows a dealer to distribute a secret s between n parties P_i for $i = 1, \dots, n$ such that the secret can only be reconstructed if a qualified subset of these parties collaborates, while no other subset can learn any information about the secret. The set of all qualified subsets forms the access structure of the system:

Definition 4.1. *A set* $\Gamma \subseteq 2^{\{1,\dots,n\}}$ *is called a* monotone access structure *on* $\{1, \dots, n\}$ *if it is a family of sets such that the following is satisfied: if* $\mathcal{A} \in \Gamma$ *and* $\mathcal{B} \supseteq \mathcal{A}$*, then* $\mathcal{B} \in \Gamma$*. A set* $\mathcal{G} \in \Gamma$ *is called* qualified*, a set* $\mathcal{N} \notin \Gamma$ *is called* non-qualified*.*

For the sake of simplicity we here identified the set $\mathcal{P} = \{P_1, \dots, P_n\}$ of parties with the set $\{1, \dots, n\}$. Here and in the following, we will mainly focus on threshold schemes, where a set is qualified if it consists of at least k parties for some fixed $k \leq n$. That is, for a threshold scheme, the access structure is given by $\Gamma = \{\mathcal{G} \subseteq \{1, \dots, n\} : |\mathcal{G}| \geq k\}$.

We can now formally define a secret sharing scheme for an access structure Γ.

Definition 4.2. *A* secret sharing scheme *for a set* $\mathcal{S}$ *of secrets and a monotone access structure* $\Gamma \subseteq 2^{\{1,\dots,n\}}$ *is a pair of PPT algorithms* share, reconstruct*:*

share: *On input* $s \in \mathcal{S}$*, this algorithm outputs shares* $\sigma_1, \dots, \sigma_n \in \{0, 1\}^*$*.*

reconstruct: *On input* $\{(i, \sigma_i) : i \in \mathcal{G}\}$ *this algorithm outputs* $s' \in \mathcal{S}$ *or* $\perp$*.*

Perfectly secure secret sharing (PSS)

In this category of algorithms we selected the most prominent representative for implementation, namely Shamir secret sharing as proposed in [Sha79]. This version has many desirable properties and is good suited for both, software and hardware implementation, especially if small message spaces can be used.

Algorithm: Perfectly secure secret sharing (PSS)

share: On input $s \in \mathcal{S} := \mathbb{F}_q$, this algorithm chooses $a_1, \dots, a_{k-1} \xleftarrow{\$} \mathbb{F}_q$ such that $a_{k-1} \neq 0$, and defines:

$$f(x) := a_{k-1}x^{k-1} + \dots + a_1 x + s\,.$$

The algorithm now outputs $\sigma_i := f(i)$ for $i = 1, \ldots, n$.

reconstruct: On input at least k inputs of the form (i, σ_i), this algorithm computes the unique interpolation polynomial $g(x)$ of degree $k-1$ in $\mathbb{F}_q[x]$, and outputs $s' = g(0)$.

The main *properties* which can be achieved are given in the following:

- Confidentiality: The secret sharing scheme is information-theoretically confidential, when the receiver has less than k shares available, i.e., perfect. Even unbounded adversaries are not able to recover the plaintext.
- Integrity: The completeness of secret sharing schemes guarantees the integrity of stored data and if more than k shares are available for reconstruction additional error checking can be performed.
- Information rate: The size of message and share are the same as the size of an element of the finite field $\mathbb{F}_q$. Information rate is 1, i.e., ideal for unconditional confidentiality.
- Homomorphic operations: A secret sharing scheme is $(+,+)$-homomorphic where addition on share vectors is performed as $[a + a']i = [a]i + [a']i$.
- Complexity: Sharing algorithm requires $(k-1)n$ multiplications and $(k-1)n$ additions. The message reconstruction algorithm requires k divisions, $2k^2 - 3k$ multiplications, and $k^2 - 1$ additions. In summary the algorithms can be considered as efficient.

Information dispersal (IDS)

The information dispersal algorithm is very related to the PSS algorithm presented above. It is also very similar to Reed-Solomon codes an provides a very flexible and efficient way to increase the robustness of fragmented data. It provides the same interface like secret sharing but does not protect the confidentiality of data. Thus, it is also similar to the Ramp version of Shamir secret sharing which the highest information rate.

The main *properties* which can be achieved are given in the following:

- Confidentiality: No security is achieved by the algorithm.
- Integrity: This property is analog to PSS.
- Information rate: The size of message is reduced. Each share has only size s/k, which is extremely efficient, i.e., it is optimal.
- Homomorphic operations: IDS is also $(+,+)$-homomorphic like PSS.
- Complexity: Sharing algorithm requires $(k-1)n$ multiplications and $(k-1)n$ additions for share generation, however, k input words are processed in one cycle which reduces the complexity by a factor of $1/k$. The same is true for reconstruction.

Computational Secret Sharing (CSS)

This mode of encoding is a combination of the first two versions and enables efficient and secure storage of data. The combination is known as computational secret sharing and was first proposed by Krawczyk [Kra93b]. The tool was selected, because it has been well studied and it is ideally compatible with IDS and PSS algorithms selected in *SECOSTOR* and also efficient to implement. On a high level, the idea is to first encrypt the data s using a symmetric encryption scheme, and then apply the secret sharing scheme to the used key and the information dispersal scheme to the resulting ciphertext.

More formally, the algorithms share and $\mathsf{reconstruct}$ are as follows, where $(\mathsf{Enc}, \mathsf{Dec})$ is a symmetric key encryption scheme, $(\mathsf{share}', \mathsf{reconstruct}')$ is a perfectly private secret sharing scheme, and $(\mathsf{share}'', \mathsf{reconstruct}'')$ is an information dispersal scheme:

Algorithm: Compuational secret sharing (CSS)

share: On input data s, this algorithm first draws a random encryption key K for the symmetric encryption scheme, and computes $e = \mathsf{Enc}_K(s)$. It then computes:

$$(\sigma_{K,1}, \ldots, \sigma_{K,n}) \overset{\$}{\leftarrow} \mathsf{share}'(K) \quad \text{and} \quad (\sigma_{e,1}, \ldots, \sigma_{e,n}) \overset{\$}{\leftarrow} \mathsf{share}''(e)\,.$$

The algorithm outputs $\sigma_i = (\sigma_{K,i}, \sigma_{e,i})$ for $i = 1, \ldots, n$.

$\mathsf{reconstruct}$: On input at least k shares of the form $(i, \sigma_i) = (i, (\sigma_{K,i}, \sigma_{e,i}))$, this algorithm first computes:

$$K' \overset{\$}{\leftarrow} \mathsf{reconstruct}'(\{(i, \sigma_{K,i}) : i \in \mathcal{I}\}) \quad \text{and} \quad e' \overset{\$}{\leftarrow} \mathsf{reconstruct}''(\{(i, \sigma_{e,i}) : i \in \mathcal{I}\})\,,$$

where $\mathcal{I}$ is the set of input indices. It then outputs $s' \overset{\$}{\leftarrow} \mathsf{Dec}_K(e')$, or $s' := \perp$ if one of K', e' was equal to $\perp$.

It is easy to see that for long secrets s, the size of the shares is dominated by that from the information dispersal scheme. Similar to Shamir's scheme, a simple and efficient scheme to disperse k elements $a_0, \ldots, a_{k-1} \in \mathbb{Z}_q$ is to define the polynomial $f(x) = a_{k-1}x^{k-1} + \cdots + a_1 x + a_0$, and to output shares $\sigma_i = f(i)$ for $i = 1, \ldots, n$. Now, if the encryption scheme maps into $\mathbb{F}_q$, this information dispersal scheme – and as a result also Krawzcyk's computationally secure secret sharing scheme – asymptotically achieves the optimal storage overhead of n/k.

For symmetric encryption and message authentication following algorithms have been selected and are currently available:

- Advanced encryption standard (AES) is available in CBC and CTR mode as symmetric encryption cipher. AES is considered secure by many standards and can be regarded as the safe and conservative choice in our secret sharing suites.

- Chacha20 is available as alternative. Chacha20 is a fast and modern stream cipher based on Salsa20, a recommended cipher by the eSTREAM project[1].

The main *properties* which can be achieved are given in the following:

- Confidentiality: Security against computationally bounded adversaries is given by this scheme.
- Integrity: This property is analog to PSS and IDS, because they are used as subalgorithms.
- Information rate: The size of message is reduced. Each share has only size s/k plus a constant size part where the encryption key is stored, i.e. 128/256 bit word container.
- Homomorphic operations: CSS does not allow for any homomorphic operation.
- Complexity: Same as IDS plus one encryption run for the plaintext data plus one PSS run for a 128/256 encryption key.

Robust Secret Sharing (RSS)

Basic secret sharing schemes as discussed previously assume that all parties are honest-but-curious. That is, they are secure as long as at most $k-1$ parties pool their shares, but it is assumed that they otherwise stick to the protocol specification. Now, in the case of an outsourced storage application, it might be possible that an adversary sends malicious responses at reconstruction time. Such malicious shares could prevent the reconstruction of the original secret s, or result in a reconstructed secret $s' \neq s$ without the dealer being able to detect this. Unfortunately, basic secret sharing schemes in general do not offer any protection against such misbehavior; schemes offering this level of security are referred to as robust.

Error decoding. A straight forward way to make our PSS scheme robust for every $t < n/3$ is to use an error decoding algorithm in the reconstruction. Error decoding is supported by the *SECOSTOR* tool and can be used under the given condition. In particular it is required that for reconstruction always $k+2t+1$ shares are available, i.e., for every corrupt share there is at least an additional valid one available. The very same decoding algorithm can also be used to decode IDA fragments, hence, we also have a $t < n/3$ IDA available. Finally, because CSS is also based on PSS and IDA as subalgorithms *SECOSTOR* also provides robust CSS. The modular design enables full set of robust algorithms with just one error decoding algorithm implemented. However, we would like to emphasize, that the complexity increases substantially and decoding speed can be significantly slower compared to erasure decoding.

Information checking. For a reasonable size of storage nodes the very basic information checking algorithm came out to be still the most versatile one with reasonable overhead

[1] http://www.ecrypt.eu.org/stream/

compared to more advanced and optimized schemes. We therefore specify an unconditionally private RSS schemes, which is based on the idea of Rabin and Ben-Or [RB89] often referenced as *information checking*. On a high level, in the scheme the dealer extends the share sent to a party by message authentication codes. That is, for every party P_j, the share of P_i is authenticated with a tag $\tau_{i,j}$, and the corresponding key is given to P_j. At reconstruction phase, the shares can now be checked for correctness by checking whether they are accepted by sufficiently many tags. To have a failure probability negligible in λ, MACs with length at least λ bits need to be used, and thus each player needs to additionally store $\Omega(\lambda n)$ bits.

Fingerprinting. If one is aiming for computational privacy only, it is possible to significantly reduce the overhead of the construction from [RB89]. Namely, instead of using a keyed MAC, one can simply compute a hash of each share, which is then distributed among the parties P_i. This distribution is done such that only $t+1$ honestly returned parts are necessary to reconstruct the hash value. Doing so, the reconstruction algorithm can then check which shares of the secret were returned honestly, and the secret can be recomputed. This scheme was originally proposed by Krawczyk [Kra93a] together with his CSS proposal. In a straight forward way, the hash values of the shares can be appended to the shares which then let accept correct hashes by voting, i.e., if it appears at least $t+1$ times.

In summary, with error decoding all presented schemes (PSS, IDS and CSS) can be made robust against $n > 3t+1$ errors. In this case the decoding speed decrease significantly and at least $2t+1$ valid input shares are required to start decoding. If information checking or fingerprinting is used respectively, immediate decoding is possible as soon as $t+1$ shares are attested by the additional checking information and also faster decoding is possible although additional hash functions have to be calculated. Furthermore, the share size increase for information checking can be substantial if amortized over larger batches of data.

4.4 Operational Aspects

Based on the available algorithms in the *SECOSTOR* tool we propose to consider following operational aspects for system implementation.

- All connections between clients (dealer, reader) need to be encrypted and authenticated. We recommend to use mature technology to implement secure channels like TLS or similar.
- Because of the keyless nature of secret sharing, strong authentication and good access control mechanisms are extremely important for a complete system.
- Each user should have different credentials on the different system.
- If long-term security is of major importance PSS should be selected as fragmentation algorithm, it is the only which supports the unconditional security plus the homomorphic properties to execute more advanced proactive secret sharing protocols.

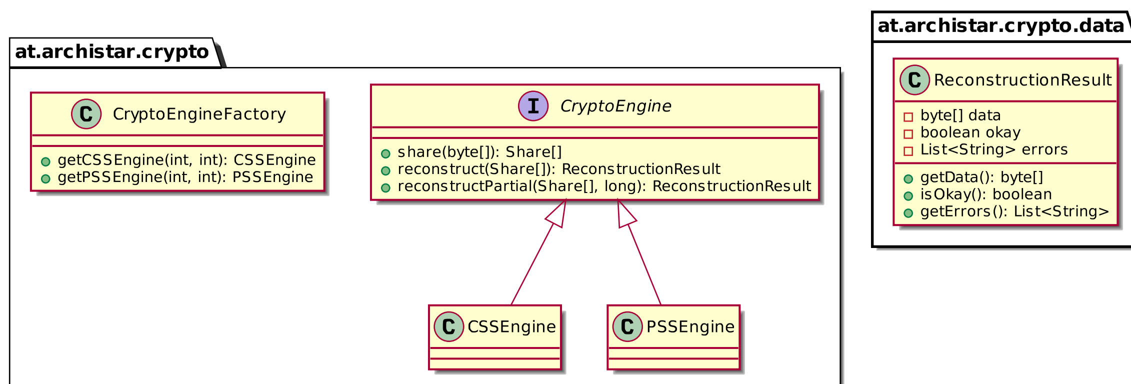


Figure 7: Overview crypto engine.

4.5 Architecture and Design of Software Library

In PRISMACLOUD we are developing two different implementations of the *SECOSTOR* tool.

4.5.1 Java Library

The archistar-smc cryptographic library allows applications to easily include secret-sharing functionality. We differentiate between high-level (intended for application use) and low-level (internal) interfaces.

Engines. The high-level application interface, named "CryptoEngine", is currently implemented by two engines, providing the CSS and PSS schemes described above. Engines therefore bundle secret-sharing algorithms with information checking (in the case of PSS) or fingerpriting (in the case of CSS) techniques. While all engines implement the same external interface, they differ feature-wise. This makes it technically possible to exchange engines with minimal user application changes – the developer has to verify that the new engine's features make it feasible for usage w.r.t. to error resilience, performance and privacy impact. An overview of the engines is given in Figure 7.

Algorithms. Each concrete algorithm implements a single theoretical algorithm. Engines combine one or more algorithms to provide a better application developer experience. Examples of implemented algorithms are Shamir's Secret-Sharing or Rabin Information Dispersal. In a sense, algorithms are pure logic as integration with the real environment, e.g., integrating a random number generator, happens within engines. A class diagram of the secret sharing implementation can be found in Figure 8.

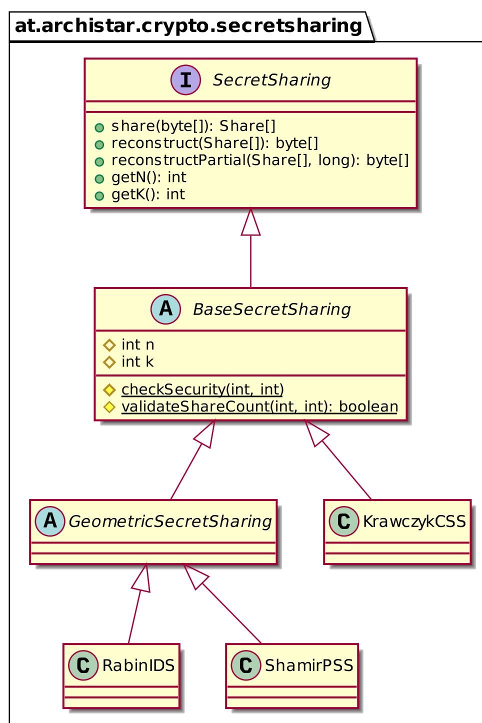


Figure 8: Class diagram secret sharing implementation.

Data Structures. Engines create shares from incoming user data and can be used to recreate the original data from a subset of those generated shares which are represented by the "Share" interface. For this purpose, it includes both the raw shared data and (dependent on the kind of algorithm used) the metadata necessary for reconstruction. The interface also encompasses methods and data necessary for checking the integrity of the data.

Every engine creates its own type of share, and to facilitate error handling, there is an additional "BrokenShare" so that methods can uniformly return (collections of) objects that conform to the interface.

A class diagram is given in Figure 9.

4.5.2 JavaScript Library

The JavaScript library is for the most part a straightforward translation of the Java version, though JavaScript's prototype-based object model and its good support for unsigned integer types made it possible to simplify the architecture and leave out a number of helper methods.

It is nevertheless considerably less advanced and contains only the bare minimum of functionality. There is therefore no distinction between Engines and Algorithms, and there

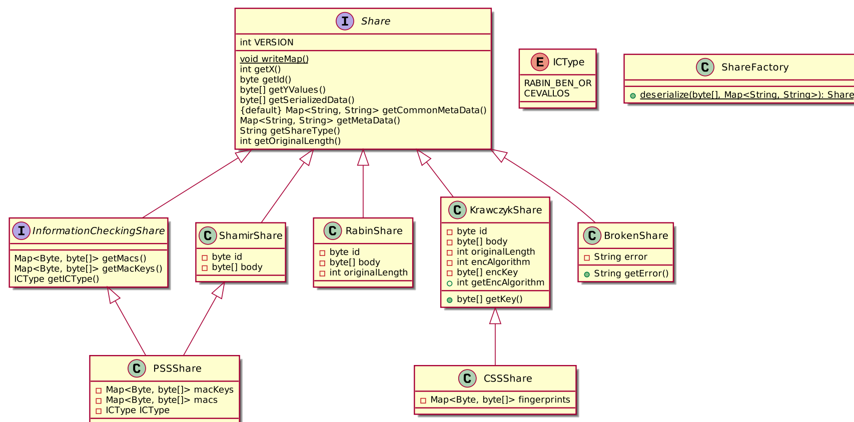


Figure 9: Class diagram of share/fragment data structure.

are no Shares. All three basic algorithms, however, are implemented: Shamir, Rabin, Krawczyk. Each is instantiated with n and k (and a random number generator, in the case of Shamir and Krawczyk). On calling the function "encode" on a fully instantiated algorithm, a JavaScript object containing the secret-shared data (and the secret-shared key in the case of Krawczyk) is produced.

5 The Byzantine Fault Tolerance Library

Besides the comprehensive secret sharing component the robust concurrency layer is the second main software component of the SECOSTOR tool. In the following we specify the basic algorithms and protocols as well as the software architecture of the software implementation in detail.

5.1 Overview

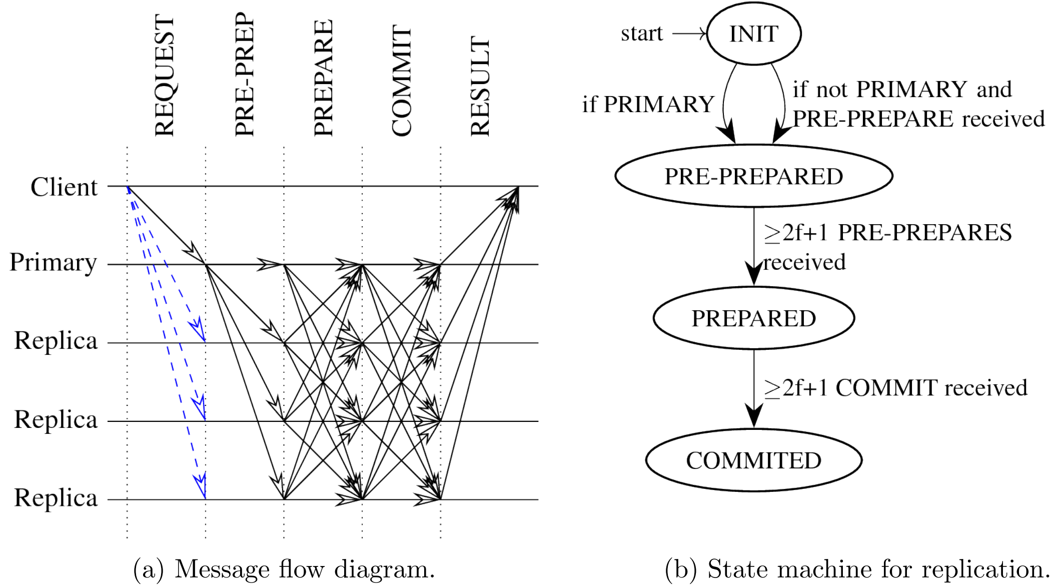
An important distinction between methods to manage concurrency in distributed systems is their underlying error model. In particular, what types of failures respective attackers are assumed. Within crash-fault systems, a defective part may either answer correctly or not answer at all. Depending on the detailed model, an once-defective part might not become online again. This closely resembles a server experiencing a power fault. Within the stronger Byzantine fault model a faulty part may fail in any conceivable way, i.e., this includes parts behaving maliciously. We have chosen the Byzantine model for our prototype because we want to have increased resiliency and build a system which would suit our mission-critical applications. Building a maximally robust system is also a very good exercise to test performance limits of such systems. Converting the results to more mild settings later on is easier and a straight forward exercise, once we fully understand the protocols for the Byzantine case.

The *SECOSTOR* data distribution mechanism must solve different problems: multiple clients must be able to access and alter data upon multiple data servers in parallel. Neither a defective client nor data server should be able to fully deny the overall system to other parties. More specific, in our use-case we need to distribute the created data shares upon multiple servers and strong consistency must be prevail, i.e., even if a small subset of malicious servers modify their shares, a non-malicious client that accesses the (larger) subset of non-malicious data servers should be able to access and modify stored data.

5.2 Algorithms and Protocols

The most prevalent BFT protocol for storage solutions is the *PBFT* protocol [CL02], which is the most efficient one in practical settings. It is based on a network model with only weak synchrony and has only three phases of overhead. It is the best suited solution for our more static scenario with a rather manageable amount of network nodes and good connectivity. Here we give a brief overview of the main protocols and underlying state machine. For clarity of presentation, we present the protocol in its vanilla form, and omit any possible performance optimization. *PBFT* assumes a system consisting of n *servers* (or *replicas*), which are connected through authenticated and private channels.

At most $f = \lfloor \frac{n-1}{3} \rfloor$ nodes may be faulty. Exactly one replica is designated as primary. The system moves through a sequence of *views*, where the primary changes with every view-change in order to cope with malicious primaries. The view-concept introduces the



(a) Message flow diagram.

(b) State machine for replication.

Figure 10: Overview *PBFT* protocol. Black arrows describe the message flow of the vanilla version. The dashed blue arrows show the extension needed for integration with secret-sharing.

concept of bounded synchronicity into an otherwise asynchronous system.

In *PBFT* a client sends the operation to the designated primary node. The primary associates an unique counter and broadcasts the operation and the counter to all other replicas as *PREPREPARE* message. Upon receiving this message all replicas broadcast a *PREPARE* message including a hash of the operation and its sequence number. If more than $2t+1$ prepare messages (including one's own) are received by a replica, it broadcasts a *COMMIT* message. After $2t+1$ matching *COMMIT* messages have been collected by a replica and all transactions with lower sequence numbers have been performed, the requested operation is executed and the result sent to the original client. After $t+1$ matching results the client knows the operation's result. The archetypal message flow diagram as well as state diagram describing a single operation/transaction can be seen in Figure 10.

As discussed, *PBFT* does provide robustness with respect to malicious clients or nodes but it does not allow for determining which of the entities did behave malicious. In order to detect which client or *PBFT* node behaves malicious we added log trails. These log trails allow for an audit which could be conducted in a distributed manner or at a central trusted instance, supporting the detection of malicious clients.

On top of *PBFT* we introduce the concept of multi-user operations on stored shares/files. Therefore, each *PBFT* node holds a version of the user database including a mapping of users and shares/files with specific access rights. On each access the user database is read from and written to the distributed system using the *PBFT* protocol in order to ensure that each *PBFT* node stores the same version of the database. Since operations on

shares/files are sent by external clients, we require those operations also to be authentic besides traditional security mechanisms [HKL16] (e.g., username and password).

We further support secure deletion of *old* (where old is subject to definition, e.g, TTL) objects by a majority vote protocol. It requires more than $2t+1$ deletion messages with the same list of objects in order to perform the requested deletion.

5.3 Architecture and Design of Software Library

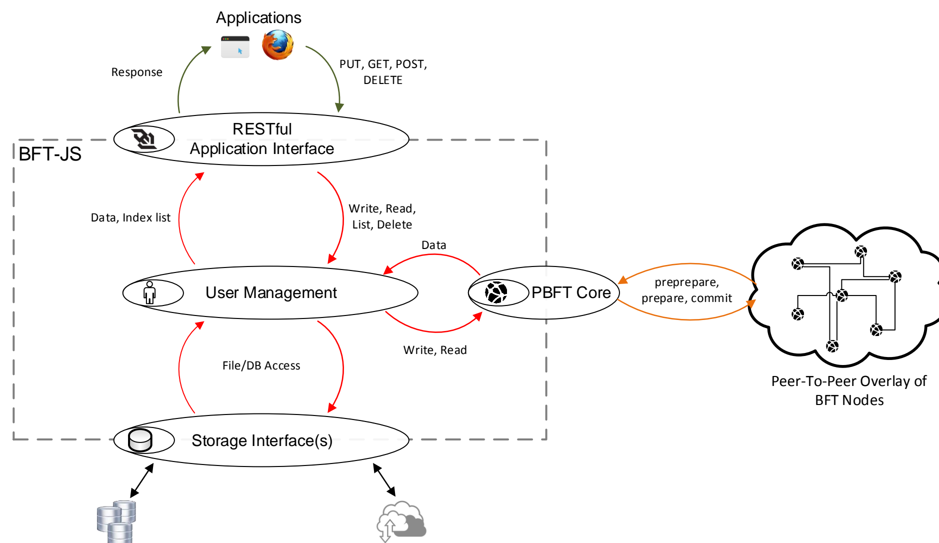


Figure 11: Architecure of the BFT Module.

Figure 11 depicts the architecture of the BFT module. The BFT module is implemented using the platform independent framework Node.js [Nod17]. Node.js provides an asynchronous event driven JavaScript runtime environment and, therefore, is very well suited for developing scalable network applications. The Application Interface provides a RESTful API for applications (cf. Figure 11), a separate interface definition is provided using swagger.io. The BFT module provides a secure and fault tolerant storage service (by means of [LHS15]) and can be easily accessed by any application using the provided RESTful API. If enabled, the BFT module support multi-user support and each of the operations on the distributed storage has to be authentic and accompanied by valid credentials (e.g., username and password). The User Management extends the RESTful API by means of user specific options. The interested reader is referred to the interface definition provided in the first open source release of this component in the Archistar software framework repository at *https://github.com/archistar*. The BFT core implements the *PBFT*(as described in [CL99]) protocol. It is decoupled from the actual storage and uses the underlying Storage Interface for reading and writing to the local copy of the distributed data.

Mediator Module

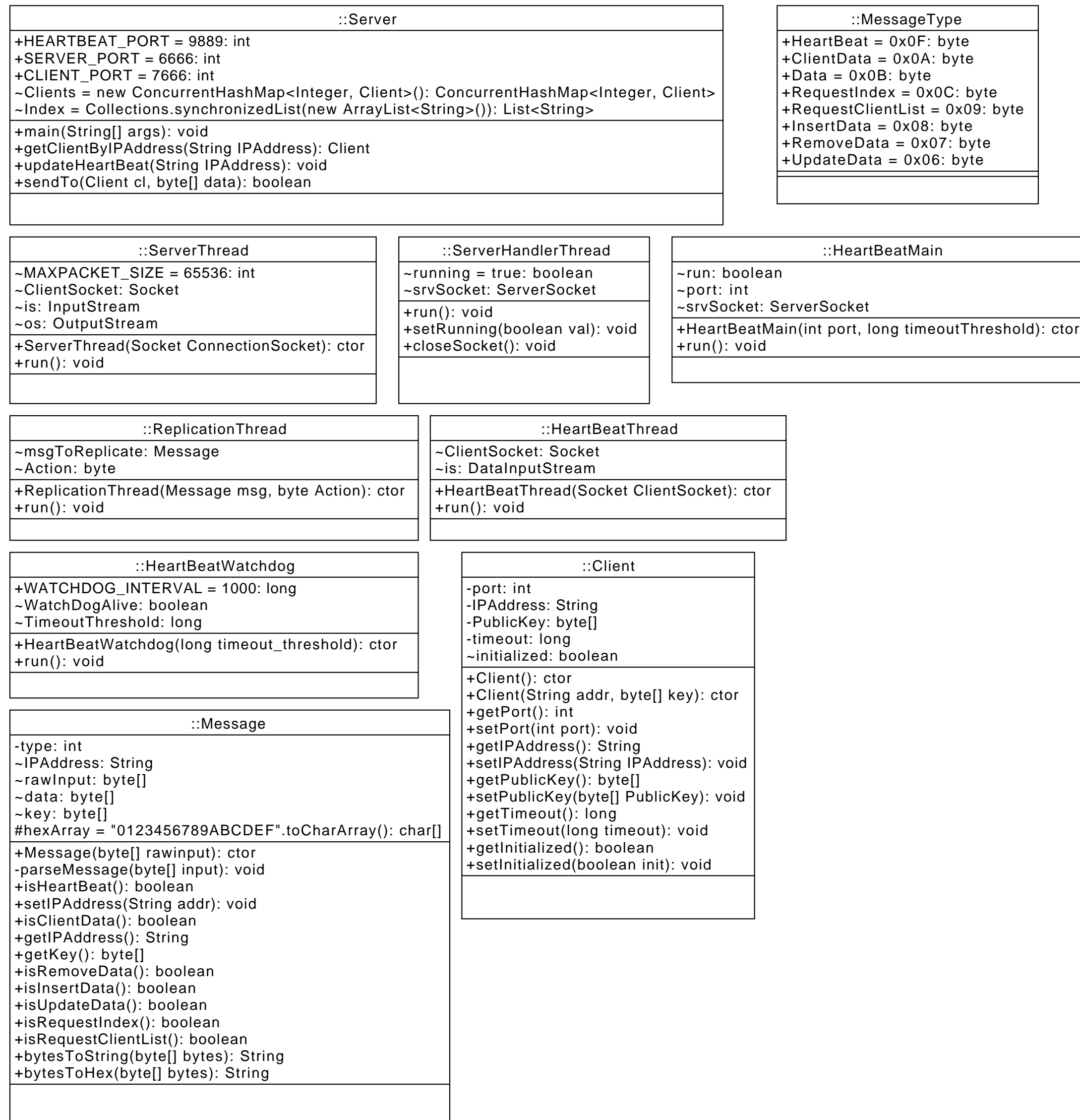


Figure 12: Classes of the Mediator module.

The mediator module is responsible for maintaining the Peer-to-Peer Overlay consisting of PBFT nodes and for replicating newly inserted data items to all available PBFT nodes. If enabled, the Mediator will generate secret shares of the provided data item depending on the number of PBFT nodes available and the replication factor. In order to maintain an up-to-date list of PBFT nodes, each PBFT node has to send a heartbeat within a certain time interval to the Mediator (otherwise the node will be considered as offline). Newly received data items will be split into n-shares according to Section 4 and will be distributed to all available PBFT nodes.

Figure 12 depicts the classes of the Mediator module. The Server class denotes the main

class responsible for setting up the listeners for the heartbeat server (depicted by the HeartBeatMain class) and the replication server (depicted by the ServerHandlerThread class). For each arriving client/peer ServerHandlerThread class spawns a new thread using the ServerThread class which processes the message sent by the client/peer. Clients/peers are represented by the Client class. BFT nodes that want to enter the system have to provide their public keys for verification purpose. The server holds a list of all BFT nodes available in the system including their public keys. Requests or Messages are pre-processed using the Message class which provides functions for identifying and parsing received messages according to their type (cf. MessageType class). If clients/peers want to write or update data in the system, the ServerThread spawns a thread using the ReplicationThread class which is responsible for creating secret shares (if enabled using a specified sharing method e.g., Shamir Secret Sharing) and/or distributing the data/shares to the available PBFT nodes. Furthermore, BFT nodes have to send a heartbeat for maintaining an up-to-date list of available BFT nodes. Each time a heartbeat is received a new thread is spawned using the HeartBeatThread class handling the notification from a specific BFT node. In order to detect out-dated/offline BFT nodes, a watchdog is running provided by the HeartBeatWatchdog class.

6 Additional Functionalities

In this section we describe how additional features can be achieved when building distributed cloud systems on the basis of the SECOSTOR tool. We explain and reference important protocols, which can be realized easily by using the tool libraries, especially the functionalities of the secret sharing module. In particular, remote data checking, the proposed batch verifiable secret sharing and private information retrieval are fully supported by the secret sharing library. However, implementation of the protocols require active components at the servers to execute the remote part of the protocol. This is no problem when the BFT engine is also used, but can be an issue for archiving use cases where typically only passive cloud storage services are used, e.g., as in the Archistar S3 gateway application shown in section 9. Nevertheless, because efficient means to check the consistency and retrievability of remotely stored data is of paramount interest a dedicated *verfier* component have been introduced in the basic SECOSTOR model of section 2 which is among other tasks responsible for triggering auditing procedures by challenging the servers holding the data, e.g., by running the verify algorithm of the presented remote data checking algorithm.

6.1 Remote Data Checking

The idea of remote data checking is very interesting, especially for IT outsourcing scenarios like cloud computing. It enables a party to check the retrievability of remotely stored data in an efficient way, i.e., without downloading all data. Additionally, if the auditor in such a system is not able to learn anything about the data he is checking, they system is also ideally suited for outsourcing the auditing to a third party, which is what we aimed at with our system.

Because efficient consistency checking for outsourced data if of paramount interest, remote data checking is and integral part of the SECTOR component. It is exactly the responsibility of the *Verifier* component from the model presented in section 2 to assure the retrievability of stored data and monitor the overall status of the servers. The servers on the other side are also the *Provers* in the audit protocols and are about to convince the auditor about the client data they are holding. For the SECOSTOR tool we developed a very simple auditing procedure which is exploiting the redundancy of the system [DKLT16]. It is extremely efficient and introduces only minimal computational burdens on the server as well as minimum network communication, independent of the size of the data to be checked. Furthermore, it only requires access to basic *Share* and *Rec* of the SECOSTOR secret sharing library. This scheme was developed and proofed secure in WP4, for the tool specification we are just presenting the algorithms to be implemented.

In fact, the scheme is based on arbitrary additively homomorphic threshold secret sharing scheme, which we will instantiate with Shamir's scheme [Sha79] for clarity of presentation. By exploiting the assumptions required for a meaningful secret sharing based storage system, a completely keyless system is achieved. Furthermore, the system does not require any pre-processing of messages. Therefore, our auditing procedure can be applied directly

to any data that has already been stored using Shamir's scheme, as no precomputation is required. The protocol is also information-theoretically hiding and allows for efficient batch audits across different dealers, does not require any storage overhead, supports proactivity, and is backward compatible with existing solutions. This is achieved by leveraging the non-collusion assumption and built in redundancy of the underlying distributed storage system to also prove privacy and extractability of the resulting scheme.

Audit Protocol for Single Messages. For better understanding of the protocol we first explain the basic version for single data words and extend it later to larger batches. The parties jointly compute a distributed random value, i.e., each party obtains a share of the randomness used to blind the shares of the message by simply adding it. If all those sums are consistent, the auditor accepts the audit, otherwise it rejects.

Because the system is fully key-less, the dealer does not need to store any information whatsoever, which relieves him from any complex key management issues when accessing the data from different devices. On a high level, our protocol actually proves that all shares held by the different servers have a valid (degree-t) interpolation polynomial, i.e., that the shares are consistent. Consistency with the originally shared message—and therefore integrity of the shares—then follows from the assumption that the majority of nodes is honest, and thus their shares are consistent with the original message.

$\mathsf{Setup}(1^\lambda, n)$: On input the security parameter λ and the number of servers n, this algorithm outputs the system parameters:

$$spar = (q, t, n, (PS_1, \ldots, PS_n)) .$$

More precisely, these are: a prime number $q > n$ defining the field $\mathbb{F}_q$, an integer $t \leq \frac{n-1}{2}$ defining the minimum amount of shares required to reconstruct the secret, and unique identifiers of n storage servers $PS_1, \ldots, PS_n$.

$\mathsf{KGen}(spar)$: On input the system parameters $spar$, this algorithm outputs a key pair (sk_D, pk_D) for the dealer. The key pair is computed as follows:

$$(sk_D, pk_D) = (\varepsilon, \varepsilon) .$$

$\mathsf{Store}(M, spar, sk_D)$: On input a message $M \in \mathbb{F}_q$, the system parameters $spar$, and the dealer's secret key sk_D, this algorithm outputs a share σ_i to be sent to server PS_i for $i = 1, \ldots, n$. The shares $\sigma_1, \ldots, \sigma_n$ are computed as follows:

$$(\sigma_1, \ldots, \sigma_n) \xleftarrow{\$} \mathsf{ShamirShare}(M, q, t, n) ,$$

where $\mathsf{ShamirShare}$ is Shamir's secret sharing algorithm. Note that each PS_i receives the corresponding σ_i over a secure channel.

$\mathsf{Reconstruct}(spar, sk_D, \{(i, \sigma_i)\}_{i \in \mathcal{I}})$: On input the system parameters $spar$, the dealer's secret key sk_D, and a set of shares $\{(i, \sigma_i)\}_{i \in \mathcal{I}})$, this algorithm outputs $\perp$ if $|\mathcal{I}| < t+1$.

Otherwise, it checks whether there exists a unique interpolation polynomial $f(x)$ of degree t such that $f(i) = \sigma_i$ for all $i \in \mathcal{I}$. If this is the case, it outputs $M' = f(0)$; otherwise it outputs $\perp$.

$\mathsf{Verify}(spar, M, sk_D, \{\sigma_i\}_{i=1}^n)$: On input the system parameters $spar$, a message $M \in \mathbb{F}_q$, the dealer's secret key sk_D, and a full set of shares $\{\sigma_i\}_{i=1}^n$, this algorithm outputs `accept`, if and only if:

$$M \stackrel{?}{=} \mathsf{Reconstruct}(spar, sk_D, \{(i, \sigma_i)\}_{i=1}^n)\,.$$

$\langle PA.\mathsf{Audit}(spar, pk_D); \{PS_i.\mathsf{Audit}(spar, pk_D, \sigma_i)\}_{i=1}^n\rangle$: This interactive protocol consists of the following steps.

1. The servers jointly compute a distributed uniformly random value in $\mathbb{F}_q$. That is, at the end of this interaction, every server has a share ρ_i of a Shamir-shared random value r with threshold t.
2. Next, the auditor broadcasts a challenge value $c \stackrel{\$}{\leftarrow} \mathbb{F}_q$ to the servers.
3. Each server computes $s_i = c\sigma_i + \rho_i$, which it returns to the auditor.
4. Finally, the auditor outputs `accept` if and only if the provided s_i are all consistent, i.e., if:
$$\mathsf{Reconstruct}(spar, sk_D, \{(i, s_i)\}_{i=1}^n) \neq \perp\,.$$

Batch-Auditing of Stored Messages. The main drawback of the basic protocol specified above is that it requires a fresh distributed random number to be computed for each message to be audited. We therefore show how to reduce these costs to a practically negligible amount if a large number of data blocks are audited at the same time. This is particularly interesting in our setting, where no pre-processing is necessary. Namely, it is even possible to simultaneously audit a large batch of message *across different dealers*. That is, the auditor can audit the storage solution of a company as a whole, and does not need to run this audit individually per employee, as different employees do not need to use different private keys in the Store phase. The only requirement is that all messages to be batch-audited have been shared for the same system parameters $spar$, i.e., using the same threshold t and the same n servers.

Assume now that each message $M_1, \dots, M_\ell$ has been distributed and stored according to Section 6.1, resulting in shares $\sigma_{1,i}, \dots, \sigma_{\ell,i}$ on each server PS_i, $i = 1, \dots, n$. We then have:

$\langle PA.\mathsf{Audit}(spar, pk_D); \{PS_i.\mathsf{Audit}(spar, pk_D, \{\sigma_{j,i}\}_{j=1}^{\ell})\}_{i=1}^n\rangle$: The protocol works just as in the basic case, except that in Step 3 each server computes its response as:

$$s_i = \sum_{j=1}^{\ell} c^j \sigma_{j,i} + \rho_i\,.$$

That is, the response is computed as a polynomial hash of the shares stored by the server. It can now be shown that for every constant ℓ, the resulting protocol is a t-extractable

system with retrievability error $(\ell - 1)/q$. The proof is similar to Σ_m-protocols [BKLP15]. First, rewinding allows one to extract sufficiently many transcripts and the second step then essentially corresponds to solving a linear system of equations.

The communication complexity is exactly the same as for the basic protocol, and in particular is independent of the batch size ℓ. The computational complexity in the batch setting consists of the joint computation of a single distributed random value, computing a sum for each server, and calling Verify once on the auditor's side.

Generation of a Random Value. In Step 1, we use a subroutine for jointly computing a Shamir shared, uniformly random value in $\mathbb{F}_q$. That is, by the end of the protocol, every PS_i should have a share of a uniformly random value. Besides completeness, we require the protocol to be private. That is, even if up to t servers behave maliciously, they must not be able to learn anything about the shared randomness. In particular, they must not be able to bias the resulting value in any sense.

This building block could trivially be instantiated by letting all servers draw a fresh random secret r_i which is shared using $\mathsf{ShamirShare}(r_i, q, t, n)$. The resulting shares are sent to the corresponding servers. At the end, each server adds all its shares. The resulting shared secret is clearly uniformly random as long as there is at least one honest server in the system. More advanced protocols have been proposed in the literature, and can be found, e.g., in [FMY98].

Alternatively, a pool of random values can also be added by clients, which is sometimes more practical than running a distributed protocol. Particularly in situations, where application designers want to avoid server side active components having to talk to each other using the trusted client to generate a pool of shared randomness seems very practical.

Increased Efficiency through Spot-Checking. The computational complexity on the server side can further be reduced by using spot checking techniques. For instance, the auditor could define a random sequence of shares that he wants to audit, and only these shares are used for computing the responses s_i. For instance, if for a $10'000$ blocks file the auditor wants to ensure that with 99% probability no server deleted more than 1% of its shares, spot-checking 460 blocks would be sufficient, cf. Ateniese et al. [ABC+07]. Applying an error-correcting code to the data before storing it on the servers could then be used to cope with this potential 1%-loss of data.

Identifying Malicious Servers Both, the basic and the batch version of our auditing protocol, so far required that $n \geq 2t+1$, i.e., that the majority of nodes behaves honestly. In this setting it is possible for the auditor to detect inconsistencies among the servers. However, it is not possible to identify which shares caused the inconsistency, and thus it is also not possible to blame malicious storage servers.

This feature can be introduced whenever $n \geq 3t+1$. Using standard results from coding theory, it is then possible to compute the correct reconstructed value in Step 4, and to identify the inconsistent shares, e.g., using the Berlekamp-Welch algorithm [WB83].

Additionally, when robust modes of secret sharing are used this feature can be even used up to $n \geq 2t+1$. In general, we recommend the use of robust secret sharing also for

computational efficiency, because error decoding is much slower than erasure decoding. This is not a problem for the auditing procedure, because only on block has to be recovered, but it is an issue for regular client access to data, which also has to apply error correction for robustness reasons.

Computationally Private Secret Sharing. The protocols presented so far are designed specifically for Shamir's secret sharing scheme. However, in practice, often its computational equivalent, proposed by Krawczyk [Kra93b], is used. While this scheme only gives computational security guarantees, its storage overhead is asymptotically optimal with respect to the failure safety of storage nodes: the multiplicative overhead for storing a single block is only $\frac{n}{n-t}$, if only $t+1$ servers are required for reconstruction.

The main idea there is the following: first, all message blocks are encrypted using a symmetric cipher under a fresh key k. Then, k is shared using Shamir's scheme. The ciphertexts are shared similar to Shamir's scheme: instead of only embedding one message block per polynomial into the constant term, all coefficients of the polynomial are ciphertexts. This yields a factor $t+1$ improvement compared to the information-theoretically secure version of Shamir. For details, we refer to [Kra93b].

We now note that our auditing protocol can be used directly to also audit systems that are based on Krawczyk's protocol. Namely, one simply runs two instances of the protocol: The first instance is used to guarantee that the secret key k is still stored consistently across the n storage nodes. The second instance is then executed on the data blocks. Interestingly, while Krawczyk's scheme is only computationally private towards the storage nodes, this auditing protocol still provides information-theoretic privacy against the third party auditor. This is because of the uniformly random blinding factors ρ_i that are added by the storage nodes before sending their responses to the auditor.

6.2 Protection from Malicious Clients

Additionally to the protection from malicious server, it is sometimes also interesting to protect from malicious clients. This can be especially true for mobile or web based clients which often run in untrusted environments. To solve this problem, we recommend an to use an efficient and unconditionally private instantiation of a batch verifiable secret sharing scheme — (n, m, t)-BVSS — which we developed particularly addressing dispersed storage applications [KLS17].

In the following, we let p be a prime or a prime power and define the message space of our BVSS scheme as $\mathcal{M} := \mathbb{F}_p$. Furthermore, we use an interactive randomness generation protocol $(\mathsf{RPar}, \mathsf{Rnd}_{\mathsf{BVSS}})$ as a building block. Informally, randomness generation protocol, executed among the n servers, has to guarantee that all honest servers and the dealer receive the same uniformly random output from $\mathbb{F}_p$, even if up to t of the servers behaved maliciously. This is similar to the auditing procedure from section 6.1 and the very same can also be used here.

Perfect (n, m, t)-BVSS for $n \geq 3t+1$. This instantiation works for all $n \geq 3t+1$ and is perfect in the sense that it only has only a constant additive storage overhead compared to plain Shamir secret sharing. (The scheme is a variant of the BVSS scheme presented in [BGR96].) The algorithms and protocols of our (n, m, t)-BVSS scheme are defined as follows:

Parameter generation. $\mathsf{SPar}(1^\lambda)$ obtains $pp' \leftarrow \mathsf{CPar}(1^\lambda)$ as well as $pp'' \leftarrow \mathsf{RPar}(1^\lambda, n, t, p)$, and outputs public parameters $pp := (pp', pp'')$.

Sharing phase. The sharing phase consists of the following two rounds plus the round(s) needed in the joint randomness generation protocol:

(a) The dealer D, on input pp and $(M_1, \ldots, M_m) \in \mathbb{Z}_p^m$, chooses $m+1$ degree-t polynomials

$$f_j(x) = a_{j,t}x^t + \cdots + a_{j,1}x + M_j$$
$$\text{and} \quad r(x) = b_t x^t + \cdots + b_0$$

with (uniform) coefficents $a_{j,v}, b_v, b_0 \leftarrow \mathbb{Z}_p$, for all $v \in [t]$ and all $j \in [m]$. Further, the dealer D sends shares $((f_j(i))_{j\in[m]}, r(i))$ to each server PS_i, for all $i \in [n]$.

(b) Once every server PS_i received $((\hat{f}_{j,i})_{j\in[m]}, \hat{r}_i)$ from the dealer D, the servers and the dealer engage in an execution of the joint randomness generation protocol. Let w be the output of $\mathsf{Rnd}_{\mathsf{BVSS}}(1^\lambda, n, t, p)$. (Note that after this step, by definition of the building block, it is guaranteed that all honest PS_i and D hold the same uniformly random value w.)

(c) Next, D computes $C_v := \sum_{j=1}^m a_{j,v}w^j + b_v$, for all $v \in [t] \cup \{0\}$, where $a_{j,0} := M_j$, and broadcasts $(C_v)_{v\in[t]\cup\{0\}}$. After the broadcast, D outputs ε.

(d) Upon having received $\hat{C}_0, \ldots, \hat{C}_t$, each server PS_i verifies the consistency of its shares by validating that

$$\hat{C}_0 + \hat{C}_1 i + \cdots + \hat{C}_t i^t \stackrel{?}{=} \sum_{j=1}^m \hat{f}_{j,i} w^j + \hat{r}_i \tag{1}$$

holds. (Note that each PS_i has received the values $((\hat{f}_{j,i})_{j\in[m]} = f_j(i)_{j\in[m]}, \hat{r}_i = r(i))$ as described in step (b).) If the Eq. (1) holds, then PS_i outputs $\vec{s}_i := (\hat{f}_{j,i})_{j\in[m]}$; otherwise, it terminates the protocol.

Reconstruction phase. The reconstruction procedure is only one-round. Concretely, each server first broadcasts its shares to all other servers. Now, on input $(\vec{s}_1, \ldots, \vec{s}_n)$, Rec first verifies each $\vec{s}_i$ by using a Berlekamp-Welch decoder [WB83]. If less than $2t+1$ values $(\vec{s}_i)_{i\in[n]}$ are valid decodings, Rec outputs $\perp$. Otherwise, for each $j \in [m]$, it takes the respective parts of the valid shares and computes the degree-t interpolation polynomial f'_j, and outputs the messages $M'_j := f'_j(0)$, for all $j' \in [m]$.

Looking at our protocol specification, it can be seen that the computational overhead on the server side is independent of the batch size, except for the computation of the sum in

Eq. (1). However, in practice, these costs are negligible, in particular compared to the evaluation of m equations of the same form in the direct generalization of existing VSS schemes.

6.3 Reverting to a Consistent State

Depending on the frequency how often a specific server fails in the audit mechanism, one might choose to either replace it, or to revert it to a consistent state again. In the following we recap a protocol based on the enrollment protocol by Nojoumian et al. in [NSG10], which allows one to recompute lost shares for a specific server.

Assume therefore that storage server PS_j has been identified to be inconsistent with the other servers, and the auditor has sent a message requesting a recomputation step for this server to the system. Receiving this message PS_j returns to the last state that has been accepted by the auditor and requests the missing shares from a set of storage servers that were verified positively.

The basic idea is to recompute a missing share σ_j for storage server PS_j by re-computing $f(j)$ in distributed fashion. Since the Lagrange interpolation is used for this computation only k storage servers are needed. The auditor can be in charge of selecting them. Assume it selected subset $\{PS_i\}_{i\in\mathcal{I}}$. Then, the following steps are performed:

1. Each storage server PS_i, for $i \in \mathcal{I}$, computes its Lagrange interpolation constant $\gamma_i = \prod_{1\leq l\leq k, i\neq l} \frac{j-l}{i-l}$. Then, it multiplies γ_i by its share σ_i and, randomly, splits the result into k portions, such that $\gamma_i \cdot \sigma_i = \sigma_{1,i} + \ldots + \sigma_{k,i}$. Finally, it sends value $\sigma_{l,i}$ to storage server PS_l using a private channel.
2. Each storage server PS_i, for $i \in \mathcal{I}$, collects all values $\sigma_{i,l}$ received from the storage servers PS_l, where $l \in \mathcal{I}$, and computes $\sigma_i = \sum_{l\in\mathcal{I}} \sigma_{i,l}$. Then, it sends σ_i to storage server PS_j through a private channel.
3. Storage server PS_j computes its share σ_j by adding all received values, i.e $\sigma_j = \sum_{i\in\mathcal{I}} \sigma_i$.

6.4 Private Information Retrieval

In this section we discuss a technique that go beyond data privacy (i.e., the confidentiality of outsourced data). When outsourcing data to the cloud, besides the content of the outsourced data, there are other privacy relevant information that we subsume under the term *access privacy*. Protecting access privacy is thereby comparable to measures that are taken to protect communication networks from various types of traffic analysis [Ray01]. Namely, in communication networks, even if content data is encrypted, without additional measures[2] other sensitive and potentially privacy-invasive information may be revealed to an adversary. For instance, the identities of sender and receiver, the message size as well as time and frequency of communication. It is clear that in several scenarios, revealing any of these information may already be problematic, even if the content data is unknown.

[2] Such as using anonymous communication networks as Tor [DMS04].

Such side channels do, however, also exist in storage systems. For instance, a cloud storage provider can learn *who* accesses data or who has access to which data, e.g., by inspecting authentication information as well as access control lists. Moreover, the cloud provider can learn *which* data is accessed, *how often*, and *what* action a client is taking or can potentially take, e.g., read and/or write access to the data. Finally, cloud providers also learn *how much* data is stored and from *where* a client is accessing data. In the cloud computing setting such information can be confidential business information [CPK10]. Cloud providers may use these side channels to learn the access history, which reveals user's habits and privileges. The fact that there exist reads to the same file from different users may indicate common interest or a collaborative relationship and the access frequency of files may also reveal individual and enterprise related interests respectively.

Subsequently, we will focus mainly on the question of hiding *which* data is accessed, *how often*, and *what* action a client is performing. In part we will also deal with the question *who* accesses data or who has access to which data.

Private information retrieval (PIR) [CGKS95] is an approach, which allows clients to query data items from a server without revealing to the server which item is retrieved. For now, let x be a bitstring of length n held by some server(s). A user wants to query the i-th bit x_i, for $1 \leq i \leq n$, from the server(s), without disclosing any information about i to the server(s). In this traditional setting, the database is modelled as a string of bits, but the generalization to a more abstract database is quite straightforward. Augmenting queries to larger strings, for instance, can be trivially obtained by running many PIR instance in parallel. This, however, will rapidly become too expensive. Thus, for a practical setting it is interesting to design protocols which allow to efficiently query larger strings. In a setting where one wants to obliviously retrieve (blocks of) files in a block oriented storage setting, such a generalized approach seems more suitable.

PIR approaches can be generally classified according to the privacy guarantees for the clients:

Computational PIR (cPIR): These schemes provide computational privacy guarantees to the users, i.e., privacy against computationally bounded server(s) and depends on some intractability assumption(s).

Information-theoretic PIR (itPIR): These schemes provide information-theoretic privacy guarantees to the users, i.e., privacy even against computationally unbounded server(s).

Furthermore, one can classify PIR schemes according to whether the data is hosted by a single server or multiple non-communicating servers, each of which hosts an identical copy of the entire data.

Single-Server PIR: The entire database x is hosted by a single server and thus all queries are issued to this single server.

Multi-Server PIR: The entire database x is replicated to $\ell > 1$ servers and every server holds an entire copy of x. The queries are issued to $k \leq \ell$ of the servers simultaneously.

It is not hard to verify that single-server PIR with information-theoretic privacy and sublinear communication cannot exist. It can also be shown that server side computation is always $\Omega(n)$, because each response to a query must necessarily involve all bits of the database of size n. Otherwise, the server(s) would learn that the remaining bits are not of interest to the user.

In general, multi-server PIR are of more practical relevance and also fit the idea of the SECOSTOR tool of a distributed storage system. Nevertheless, an important aspect in PRISMACLOUD is to protect from provider side data leakage and requires the confidentiality of data protected for insider attacks at cloud providers. Luckily, as we will see later, the most efficient and robust methods found in literature not only support PIR for replicated data but also for secret shared data. The protocols recommended in the following are compatible with both types of secret shared data, i.e., perfectly secure and computationally secure ones, and can therefore be easily added on top of the proposed storage solution.

For SECOSTOR we recommend to use the scheme proposed by Goldberg [Gol07] yielding a t-private v-Byzantine-robust k-out-of-ℓ PIR. In the following we are quickly reviewing the scheme and discuss some implications for usage in the SECOSTOR framework.

t-private v-Byzantine-robust k-out-of-ℓ PIR. In these PIR schemes, in addition to the above properties the client should be able to reconstruct the correct query result even if v out of the k servers who are required to respond to a query are byzantine, i.e., send malformed or incorrect answers.

Goldberg [Gol07] presents an approach that yields a t-private v-Byzantine-robust k-out-of-ℓ PIR. This means that the scheme involves ℓ replicated versions of the database, provides query privacy if up to t servers collude and for reconstructing the query result it is sufficient that at least k servers respond, from which at most v are allowed to respond byzantine.

As mentioned, Goldberg's PIR scheme employs Shamir's secret sharing to add robustness. The database x is treated as a $r \times s$ matrix $\vec{X}$ holding r data items of size b and the database is fully replicated to all ℓ servers. Every data item is divided into words of size w and every data item consists of $s = b/w$ words, whereas every word x_{ij} represents an element of a suitably large finite field $\mathbb{F}$. Consequently, the database can be considered to be the matrix

$$\vec{X} = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1s} \\ x_{21} & x_{22} & \dots & x_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ x_{r1} & x_{r2} & \dots & x_{rs} \end{pmatrix}.$$

Assume a client wants to obliviously query the q'th data item in $\vec{X}$, i.e., the q'th row vector $\vec{x}_q = (x_{q1}, \dots, x_{qs})$. Therefore, he takes the respective standard basis vector $\vec{e}_q$. To conceal

the desired index q to the servers, $\vec{e}_q$ is split into k vectors of shares $s_1, \ldots, s_k$ (for the ease of presentation, let us assume that we send the query to servers with index $1, \ldots, k$ and we do not send it to a larger number of servers as it would be the case when we assume that servers might fail responding). Each of these k share-vectors contains r elements. In order to compute these share-vectors, the client chooses r random polynomials $p_1, \ldots, p_r$ over $\mathbb{F}$ of degree t, whereas the constant term of p_i is the Kronecker delta δ_{iq}, i.e., is 1 if $i = q$ and 0 otherwise. Then the shares are computed as $\vec{s}_j = (p_1(j), \ldots, p_r(j))$ and the query for server j is $\vec{s}_j$. Every server computes the response vector containing s elements as the matrix-vector product $\vec{r}_j = \vec{s}_j \cdot \vec{X}$. Finally, the client computes the desired data item x_q from all his response vectors by using s instances of Lagrange interpolation on inputs $(r_{1,i}, \ldots, r_{k,i})$ for $1 \leq i \leq s$ (note that only $t+1$ values are necessary per invocation of the Lagrange interpolation. However, for the ease of presentation we take all k values).

The v-Byzantine-robust feature of this PIR scheme is achieved via list-decoding (which we will not discuss here) and can simply be dropped if assuming non-byzantine servers. The byzantine robustness for Goldberg's PIR scheme was recently further improved in [DGH12] and ℓ-computationally privacy can be added by using homomorphic encryption [Gol07].

In the theoretical setting PIR considers the entire data to be an n-bit string or as above as a $r \times s$ matrix $\vec{M}$. However, this does not really match requirements in a practical application. One could split files into adequate words and use padding to provide such a "matrix interface". Nevertheless, this may require multiple instances of PIR to retrieve a single file. Furthermore, in order to not reveal information about the file size from the number of PIR instances used, one needs to issue the maximum number of queries required for the largest file for every file (using dummy queries for all remaining blocks). Nevertheless, there is recent work which adopts Goldberg's PIR to allow efficiently retrieve multiple blocks simultaneously [HHG13].

7 Access Control and Data Sharing

In this section, we elaborate on possible extensions of the SECOSTOR architecture. Although this extensions are not in focus of the PRISMACLOUD project, we believe that those useful techniques might lead to richer product features in the future. We follow a modular approach, which means that the extensions can directly be applied to the current SECOSTOR framework. Furthermore, for the below mentioned techniques (particularly, for attribute-based encryption described in Section 7.2), implementation libraries are available. Only for the recently developed PRISMACLOUD research result described in Section 7.3, no implementation is available yet.

More concretely, we give different access-control and data-sharing mechanisms on top of secret sharing. In particular, for access control, we describe simple mechanisms based on plain policy enforcement (where the nodes have to be fully trusted who to give access to) and based on public-key encryption (where we can significantly relax the trust of the nodes by relying on computational assumptions). For the setting of data sharing, we sketch light-weight solutions also in the public-key setting (in addition to cryptographically enforced access control) and deal with the anonymity of data to be shared.

We describe a relaxed security model in comparison to our established secret-sharing model to enable efficient non-policy-based access control and light-weight data sharing. Despite the properties of our secret-sharing model (e.g., rely on the secure channels, no more that t-out-of-n nodes collude, and unconditional security), we rely on computationally bounded adversaries for access control and light-weight data sharing which allows us to remove the restriction to rely on secure channels and colluding nodes. Furthermore, we even can resist collusion attacks from users pooling their secret keys. We note that the secretly shared data is still unconditionally private, i.e., long-term secure, solely under the non-colluding assumption; however, cryptographically enforced access control and secure data sharing hold against computationally bounded adversaries (which is still a strong security guarantee).

7.1 Beyond plain access control

Because security of secret-sharing-based storage is based on the non-collusion assumption (i.e., no more that t-out-of-n unauthorized entities collude and are able to combine the shares), an explicit access-control mechanism has to be deployed at the storage nodes' end. Usually, this is done by the node itself in the following way. A user who wants to access the share has to show some user-specific credential identifying him/her towards the node. After successful evaluation of the user's credential, the user is authorized and access to the share is granted by the node. Obviously, this approach is sufficient when relying on the non-collusion assumption and, in particular, rely on the assumption that the node does not give shares to unauthorized users. However, such approach lacks at least in one aspect. Since the (plain) access control is enforced on the node itself, the user (that shares some sensitive data to storage nodes) has to fully rely that the access-control mechanism on the node is implemented correctly, e.g., in software with role-based techniques such as role-

based access control (RBAC) or policy-attribute-based techniques such as attribute-based access control (ABAC).

By cryptographically enforcing access control at the nodes' end, we are able to "shift" software-based access control towards the access control mechanism of the cryptographic algorithms. As a consequence, we are able to remove the assumption that the nodes are properly and correctly implementing the access control to the user data since this is inherent on the mathematical level. As shown in the next Section, we leverage a concrete and well-established cryptographic primitive, dubbed Attribute-Based Encryption (ABE) [SW04, GPSW06], to enforce access control at the nodes by the means of cryptography.

Interestingly, this allows to remove the non-colluding assumption for the storage nodes with the trade-off that we only can guarantee secure access control against computationally bounded. Nevertheless, this adversary model is still strong and very reasonable. Essentially, when using public-key cryptography, up to date, this is the strongest model we can achieve. However, we stress that in case of facing computationally unbounded adversary (that can break public-key cryptography and, hence, the secure access control), we still fall back to the non-collusion assumption to guarantee the privacy of secret-shared data in PRISMACLOUD.

7.2 Attribute-based encryption for access control and data sharing

Attribute-Based Encryption (ABE) [SW04, GPSW06] is cryptographic public-key primitive which allows for data confidentiality and attributed-based access control to that encrypted data via specific policies. Interestingly, encryption and access control are both done on the cryptographic layer which means that no software-based access control is needed anymore which is a significant improvement over prior publish solutions. Before we start of describing ABE, we briefly recap (asymmetric) public-key encryption (PKE).

In contrast to the widely known (symmetric) secret-key encryption (SKE) paradigm, in PKE, the key material consists of a public and secret part. The idea is to solve the key distribution problem which is inherent in SKE (and which limits SKE useful only in pre-shared secret-key scenarios). In PKE, one does not need to share secret keys in advance. Instead, one distributes the public part of your key material typically to an widely accessible bulletin board (e.g., key server) while the secret part of the key material is kept secret. Those public key can be used to encrypt plaintext data to yield a ciphertext while the secret key can reverse the operation to obtain the plaintext data back.

ABE is an enhanced form of PKE and comes in two forms: key-policy and ciphertext-policy ABE. In Key-policy ABE (KP-ABE) [GPSW06], secret keys exhibit a policy while the ciphertext is equipped with attributes. Policies can be "(ait AND scientist) OR ceo" with attributes "ait", "scientist", and "ceo" which losely speaking means that any AIT employee that is a scientist can decrypt or solely the CEO is capable to receiving the plaintext. A KP-ABE scheme consists of four randomized algorithms $\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec}$:

System setup. The setup algorithm Setup outputs a master secret and public key pair

(msk, mpk).

Encryption. The encryption algorithm $\mathsf{Enc}(A, mpk, M)$ takes as input the attributes set A, for which the message Mshould be encrypted. The output is a ciphertext C_A which is equipped with the attributes in A.

Key generation. The user-key generation $\mathsf{KeyGen}(P, msk)$ takes as input the policy and outputs a secret key sk_P that is tailored towards that policy.

Decryption. Decryption $\mathsf{Dec}(sk_P, C_A)$ yields the plaintext of the encryption C_A using the secret key sk_P if the attributes A and the policy P match.

The second ABE form is called Ciphertext-Policy ABE (CP-ABE) [BSW07]. A CP-ABE scheme consists of four randomized algorithms $\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec}$:

System setup. The setup algorithm Setup outputs a master secret and public key pair (msk, mpk).

Encryption. The encryption algorithm $\mathsf{Enc}(P, mpk, M)$ takes as input the policy P, for which the message Mshould be encrypted. The output is a ciphertext C_P associated with the policy.

Key generation. The user-key generation $\mathsf{KeyGen}(A, msk)$ takes as input the attribute list and outputs a secret key sk_A that is tailored towards the attributes in A.

Decryption. Decryption $\mathsf{Dec}(sk_A, C_P)$ yields the plaintext of the encryption C_P using the secret key sk_A if the attributes A and the policy P match.

The ABE schemes even allow for a full collusion of secret keys which means that not even coalition of secret-key holder can pool their keys to decrypt the ciphertext if no single secret key is already capable of doing so. Note that this is the key distinguishing feature of ABE which enable secure data sharing.

Access control is straightforward with CP-ABE. One encrypts sensitive data under some policy and distributes those ciphertext. Any secret-key holder where the attributes from her/his secret key match the policy of the ciphertext is able of decrypting those ciphertext while nobody else can do so. KP-ABE is also possible in the sense that the ciphertexts are associated with some attributes while the secret keys are limited to some policy.

In terms of SECOSTOR, the scenario would be the following. On top of the secret-shared data, we would deploy CP-ABE in the sense that the secretly shared data is encrypted under some policy. Hence, we do not need to rely on the access control of the storage nodes anymore since the data owner can decide to grant access to by giving appropriate secret keys under some attributes to delegatees. Thus, any secret-key holder (who is associated to some attributes) is able to receive SECOSTOR shares if the ciphertext policy and those attributes match. After receiving all the shares, the usual SECOSTOR procedure (combining shares via Shamir secret sharing and retrieve the secret) can be performed.

We suggest to implement CP-ABE from [BSW07] for access control mechanisms in SECOSTOR.

To conclude, we strive for CP-ABE to improve the access-control and data-sharing mechanisms in SECOSTOR. Without using enhanced mechanisms, current access control in SECOSTOR is in the hands of the nodes (solely under the non-collusion assumption and the assumption that the nodes do not give the shares to non-authorized users). With CP-ABE, SECOSTOR shares can be encrypted and stored on the nodes without relying on the access-policy mechanisms of the nodes. The privacy of the shares stays unconditionally secure while the access-control security holds against powerful but computationally bounded adversaries. The encrypted data can be shared by delivering a tailored secret key (associated with certain attributes) to potential delegatees.

7.3 A note on lightweight data sharing and revocation

Alongside data sharing, access revocation is a hard to achieve and heavy feature. To have the ability to revoke access to certain sensitive data — even after having once granted someone access to it — is very desirable; preferably, in a lightweight way.

As discussed above, revoking access to the shares can be enforced by the use of ABE. While ABE allows for very expressive policies and, hence, yields a very powerful data-sharing (and access-control) mechanism for SECOSTOR, we want to give an alternative lightweight solution if one is willing to restrict expressiveness of the policies. This solution has advantages over ABE in a data-sharing context allowing for dynamic revocation although it is not that expressive. While in ABE, at the time of creating a ciphertext one usually has to know either the policy or the attributes to include, the new solution offers dynamic addition and revocation of delegatees which means that the number of delegatees does not have to be known in advance. This allows for very versatile usage within the setting of SECOSTOR in order to dynamically add and revoke user. Furthermore, once an ABE decryption key was given out and at some point revoked, the key holder is still able to decrypt previously unseen but decryptable ciphertexts.

As a solution to that problem, in deliverable D4.3 [SST17], we propose Evolution-Based Proxy Re-Encryption (e-PRE) and we refer the reader to that deliverable for more details on the technical side.

To conclude, e-PRE is a lightweigth and dynamic-revocation alternative for ABE and can be of versatile use in SECOSTOR. A technical report of a e-PRE scheme is available in [DKL+17].

Table 1: Comparison of minimal server amount and storage overhead between different storage schemes. The baseline for the overhead is given as a factor based upon the original data amount.

Faulty Servers	Server Amount Replication	Secret-Sharing	Storage Overhead Traditional Replication	Computational Secret-Sharing	Perfect (ITS) Secret-Sharing
$f = 1$	$n = k = 2$	$k = 2, n = 3$	2	1.5	3
$f = 2$	$n = k = 3$	$k = 3, n = 5$	3	1.66	5
$f = 3$	$n = k = 4$	$k = 4, n = 7$	3	1.75	7

8 Availability Model

8.1 Theoretical Secret-Sharing Performance

The performance overhead of the Archistar Proxy system can be measured through latency, minimal number of needed servers, storage overhead and throughput. Given a storage system that must be able to cope with f faulty storage locations, traditional cloud storage systems that provide redundant fault-tolerant storage must distribute the original data upon (at least) $k = f + 1$ servers. The storage overhead is thus $f + 1$.

Erasure-Coding has a similar amount of needed servers, but the storage-overhead per server is reduced by a factor of k.

With information-theoretical secure secret-sharing there is the additional condition that collaborating storage providers should not be able to reconstruct the original data: $k = f + 1$, $n - k > f$. If we transform the latter this leads to $n = 2f + 1$. Table 1 shows the overhead results for different values of f. It can be seen that while secret-sharing increases the amount of minimal servers, the overall storage overhead depends upon the chosen secret-sharing algorithm. When information-theoretical secret-sharing is used, the overhead is higher than a comparable redundant backup solution. When computational secure secret-sharing is used, it is lower.

Another important metric for backup systems is their throughput. The achieved (simplified) throughput is the minimum of the incoming (local) network throughput, the secret-sharing engine's throughput as well the achievable outgoing (public) network throughput. We assume the internal network to be of infinite capacity and that our baseline — a traditional backup solution that uses simple redundancy — saturates the available outgoing network bandwidth. If we assume a scenario of $f = 1$, i.e., the user wants at least minimal safety from redundancy, then the impact upon throughput should be the ratio of secret-sharing overhead divided by the traditional overhead. When using the estimated overhead from Table 1, it can be seen that information-theoretical secure secret-sharing would decrease the throughput, while computational secure secret-sharing would actually

improve it. The throughput of the used secret-sharing engine thus becomes the limiting factor. As a reference, Table 3 shows the measured performance of our secret-sharing library upon multiple platforms.

8.2 Availability Model

In reliability theory an n-component system that works if and only if at least k of the n components work is called a k-out-of-n:G system. The presented storage system is exactly implementing such a structure in a multi-cloud setting which lets us directly apply some results from reliability theory in our availability analysis.

In particular, the reliability $R(k,n)$ of a k-out-of-n:G system with i.i.d. components, i.e., components which are independent of each other, is equal to the probability that the number of working components is greater than or equal to k. In particular the reliability is calculated as shown in equation 2, whereby p means the probability of an individual component still working after a given amount of time. Furthermore, when we talk about availability, the more interesting notion for us, p means the probability finding an individual component working at a given point in time (including maintenance and repair). The value q is given by $q = 1 - p$.

$$R(k,n) = \sum_{i=k}^{n} \binom{n}{i} p^i q^{n-1} \tag{2}$$

The k-out-of-n is a generic model for adding fault tolerance to systems by increasing redundancy, which is exactly what we are doing with secret sharing in the Archistar system, if we leave the security aspects aside for this treatment.

If we compare the different settings presented in the previous section with the reliability model we get the following results. For the case of data replication we have $k = 1$, which leads to the analogous of a parallel system in the reliability model and $R(1,n) = 1 - \prod_{i=1}^{n}(1-p_i) = 1-(1-p)^n$. For the cases of perfectly secure (ITS) secret sharing and computational secret sharing the reliability parameters can flexibly be adjusted through encoding between $1 \leq k \leq n$, which leads to a non trivial k-out-of-n system if k is selected accordingly ($k > 1$ and $k < n$). However, if the redundancy is fully removed for security reasons ($k = n$), the systems becomes a simple series system with $R(n,n) = \prod_{i=1}^{n} p_i = p^n$. Thus, from a reliability standpoint, both secret sharing variants provide the same level of reliability, although providing different levels of security and storage overhead.

Now, if we use the previous treatment to model the availability of our solution, the basic characteristics can be directly applied. As we show here, we can also use this approach to design our system based on availability criteria we have to fulfill for the data stored. Enabling this SLA tailoring via multi-cloud configurations is very attractive, because the standard cloud storage market provides only limited flexibility in the configuration of service level agreements (SLA). In many cases main design criteria like availability goals are

Table 2: Number of leading nines for system reliability $R(k, n)$ for given n and k and a individual storage node reliability of $p = 0.98$, which is a typical value taken from cloud storage provider SLA.

n	k											
	1	2	3	4	5	6	7	8	9	10	11	12
3	5	3	1	0	-	-	-	-	-	-	-	-
4	7	5	3	1	0	-	-	-	-	-	-	-
5	8	6	4	2	1	0	-	-	-	-	-	-
6	10	8	6	4	2	1	0	-	-	-	-	-
7	12	9	7	5	4	2	1	0	-	-	-	-
8	14	11	9	7	5	3	2	1	0	-	-	-
9	15	13	10	8	6	5	3	2	1	0	-	-
10	17	14	12	10	8	6	5	3	2	1	0	-
11	19	16	14	11	9	8	6	4	3	2	1	0
12	20	18	15	13	11	9	7	6	4	3	2	1
13	22	19	17	15	12	11	9	7	5	4	3	2
14	24	21	18	16	14	12	10	8	7	5	4	3
15	25	23	20	18	16	14	12	10	8	7	5	4

not even clearly stated in provider SLA[3], nor is there any real compensation foreseen in case of violation, except for some minor service credits. In fact, serving standardized SLAs to customers is a major feature of cloud computing which helps to enable the elasticity and self-service capabilities the customers want to have. However, as a downside of this approach, it is very difficult for customers to find offerings which perfectly fit their particular needs and almost impossible to negotiate special conditions.

Archistar is trying to solve this problem in a different way, i.e., by letting the customer design a storage system according their needs and requirements as a fault tolerant composition of different cloud offerings. In particular, the k-out-of-n paradigm is used to design systems which can theoretically provide arbitrary high levels of availability. Availability classes are typically given as number of leading nines of the availability value, i.e., a "three nines" availability means 99.9% which corresponds to a downtime of $8.76h$ per year or $43.8min$ per month. We used this type of availability classes to demonstrate the theoretical values we can reach in our system with reasonable number of storage nodes.

In table 2 we show the calculated availability classes for different configurations of n and k, whereby an overall availability of 98% ($p = 0.98$) is assumed for the individual cloud storage offerings used to store the data fragments. It is easy to see, that for all typical requirements configuration parameters exist. The system is giving the cloud customer much more flexibility and enables him to design his own SLA for a virtual Archistar storage service with respect to availability, confidentiality and integrity on top of existing cloud offerings. This can also help to speed up and improve the cloud migration process in general. Furthermore, if the configurations would be matched against cloud services

[3]e.g. see Amazon S3 service: https://aws.amazon.com/s3/sla/ (accessed 6/17/2017)

databases, the best provider offerings can be selected to also get a price optimal solution. Nevertheless, a more detailed model also considering failure modes like network outages and data loss would be desirable and is left for future work.

9 Applications based on the tools

9.1 Archistar Backup Proxy

Storage systems can be classified according to the data access methods they are providing to their clients. Most storage interfaces fall into one of the following classes: remote file systems, block storage or key-value stores[4]. In addition systems can be classified due to the CAP theorem[GL02]—denoting Consistency, Availability and Partition Tolerance. The theorem states, that a system can fulfill a maximum of two out of three of those parameters.

Remote file systems provide hierarchical data storage with high semantics[Sat90]. In case of cluster file systems parallel write-operations by multiple clients while keeping strong consistency are commonly supported. This high feature level is bought by complexity.

Block-level storage offers a virtual block device, e.g., a distributed hard-drive. Data representation is not files, but fixed-size storage blocks. It is the client's responsibility to provide a file system with all inherent complexities on top of the virtual block device. This reduced feature allows for increased simplicity and thus low latency. In addition the block-level storage is well suited for data deduplication on the lowest layer. This improves storage efficiency but has the inherent drawback of reduced data safety due to reduced redundancy as well as a potential negative confidentiality impact[HPSP10].

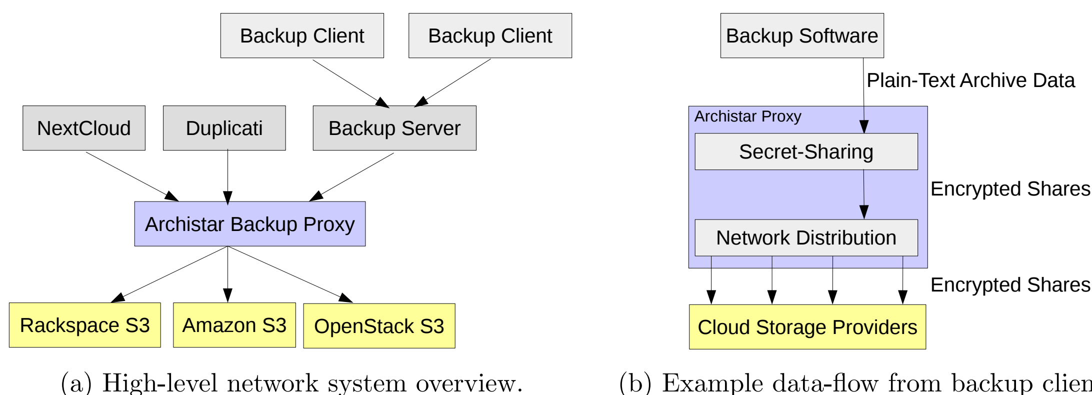
Key-Value stores provide functionality akin to non-hierarchical dictionaries[HHLD11]. Similar to block-based storage systems they utilize unique keys for identifying data, in contrast to block-storage they allow storage of arbitrarily sized values. They focus upon availability and horizontal scale-out, i.e., sharding, and have become prominent for cloud applications. The foremost known key-value network protocol is the industry-standard *Amazon S3* protocol which is served over HTTPS.

9.1.1 Architecture

The Archistar Backup Proxy uses secret-sharing to split up archive data—provided by an existing backup solution—into encrypted shares and then distributes those shares upon multiple potentially untrusted cloud storage providers. A simplified representation of our system architecture is shown in Figure 13a, a simplified version of the dataflow is shown in Figure 13b.

We target the enterprise backup sector, this leads to multiple simplifications for the resulting software architecture. Enterprise backup clients typically connect to a single backup server which used to store the clients' data upon tape drives but recently have begun to exchange those tape decks with cheap online cloud storage. Existing Enterprise Backup Solutions commonly support the Amazon S3 protocol for backend storage. By providing an emulation of this protocol, the Archistar Backup Proxy becomes a drop-in network

[4]Please note that we are focusing upon file-based storage systems and are not analyzing structured storage systems such as relational databases.



(a) High-level network system overview.

(b) Example data-flow from backup client.

Figure 13: High-Level network system overview and example data-flow from backup clients to the cloud storage servers.

component. While this improves the interoperability, the situation is not perfect: not all backup solutions utilize the same Amazon S3 subset.

The resulting system is inherently single-user: while multiple clients connect to the backup server, just a single backup server will connect to the Archistar Backup Proxy. While the backup proxy itself might provide multi-user support to its clients, the overall system is still a single-client system with the backup proxy being the single storage client. The single-user nature of the system allows to delegate large parts of the user authentication and authorization problem to the backup server software. As we target real-world deployments, we cannot assume active servers with code-execution capabilities. This reduces our server-side options as many storage systems mandate active servers with communication channels between them, e.g., PBFT-based solutions[GWGR04], or active servers with user-supplied code, e.g., fork-consistency based systems.

A high-level benefit of the backup storage sector is its focus upon bandwidth. In contrast, desktop-level interactive applications are forced to focus upon latency. As we can thus allow for higher latencies, this enables easy implementation of batching and caching, thus further improving the achieved bandwidth.

As mentioned, we have chosen the Amazon S3 protocol for client interactions due to its mass market adaption. We also support the Amazon S3 protocol for interaction with our backend storage providers but offer additional in-memory and local-storage providers for testing purposes. The concrete storage driver implementations are abstracted behind an interface, so supporting other cloud providers or modes of storage is a matter of implementing few very basic methods. Apart from utility functions — for connecting and disconnecting, returning status information — one only needs to provide methods for storing, retrieving, and deleting binary objects. As the proxy server keeps its own metadata within its index, no methods for listing directories or retrieving file stats are needed.

The index itself is a container for file data, i.e., a file's original name, size, SHA-256 and MD5[5] hashes, modification date, content type, used secret-sharing algorithm and

[5]The MD5 hash is needed for compatibility with the Amazon S3 protocol.

metadata. In addition, file data includes information on where its secret-shared parts are stored. This is needed, as we store each under a separate identifier to hamper data analysis. Apart from that, each index can — depending on the versioning configuration — contain a reference to its predecessor. As each index includes information about all available files and directories, this allows for implicit versioning of all data. To reduce storage overhead, the number of stored versions, reaching from zero to infinity, can be configured through the backup server's configuration.

Secret-sharing is utilized to distribute the index upon the available storage clouds. It is therefore only identified by its unique random identifier[6]; to identify the current index, a special file with a fixed name is used akin to a super-block in filesystem design. In order not to have to retrieve the current index on every operation and thus spare round-trips, it is cached locally. Storage operations that do not manipulate actual file data, are thus "cheap". If we assume single-proxy operation, we do not have to check if our locally cached index and the server-side index are identical. Listing all stored files, or retrieving the metadata of one specific file, does not entail any backend operations. This is paramount for performance, as some clients issue a surprisingly high number of such requests before and after each actual file operation. In addition this allows the backup server to better cope with the potential high-latency during accessing networked Amazon S3-compatible storage servers.

9.1.2 Implementation and Evaluation

This section describes findings that were discovered during implementation. In contrast to the "Architecture"-findings, these findings are more concerned with mechanics and protocol choices.

S3 Integration Challenges

During implementation of the proxy prototype, we discovered significant negative performance interactions between our initial data representation and the Amazon S3 protocol, leading to decreased performance and scalability.

As mentioned before, S3 is a key/value store with an assumed flat, i.e., non hierarchical, key space whose internal value structure is orthogonal to the protocol. Sadly, this is not what clients expect: when dealing with files, instead of a flat name space, one would probably like to have directories. For handling large files, incremental up- and download facilities would be useful. S3 provides for this: in the case of directories, this is done by treating directory names as "common prefixes". For incremental uploads of large files, there is a special "Multipart Upload" operation. As for downloading big files, S3 makes use of byte-wise access through HTTP Range headers[7]. As our security model assumes that we only serve (and integrity-check) whole files, this introduces a payoff between security and performance — either we retrieve, reconstruct, and serve just the requested part of

[6] Other than regulars files though, all secret-shared parts of the index have the same identifier upon all used storage providers.

[7] Specified in RFC 2616 §14.35, current version: RFC 7233.

the file but are then not able to verify its integrity, or we can retrieve the whole file, check it, and serve only the requested part. This is further exacerbated by the fact that client can — via the Range headers — specify arbitrary ranges to be retrieved. The size of the "chunks" in which the file is retrieved cannot a priori be determined and thus, no check-sum be pre-calculated. A 150MB file might be retrieved in 2 chunks, or in 16.

This does not only impact performance. Our current prototype implementation is based on Java byte arrays for incoming data, processing as well as for outgoing data. This has obvious negative implications for memory usage, but also less obvious implications for stability. For every l bytes of input, at least $n + 1$ byte arrays of length l have to be allocated by the Java virtual machine. In typical use, allocations do not fail, but in principle, the JVM can throw an OutOfMemoryError at any time — as we have found, frequently accessing files of moderate size such as 50 Megabyte are sometimes enough to trigger this condition. This happens exactly when a client sends multiple requests for different ranges of the same file.

Secret-Sharing Engine Performance

When analyzing the sequential single-thread performance of our data-processing engine, the typical culprits can be found: finite field multiplication and random number generation. Speeding up finite field multiplication is an intensively researched subject, but for our specific purposes of bit fields of width 8, we found the approach of using lookup tables more than adequate. A high random number generation rate is paramount, because when using Shamir's secret-sharing algorithm, for every byte of input, $k-1$ bytes of randomness must be generated. The availability of a high-performance hardware random number generator is thus highly recommended. The single-thread performance of our secret-engine can be seen in Table 3.

Our current prototype uses a sequential and single-threaded secret-sharing engine. To improve performance we are offering a first version of a parallelized erasure coding algorithm. We are investigating the potential of splitting up the workload upon multiple CPU cores, esp., to be better suited for operation on dedicated network hardware which often provides many low-performance CPU cores. Rabin's algorithm allows for easy optimization, as each of the n output shares can be generated on a different core because no synchronization between them is necessary. Shamir's algorithm depends upon $k - 1$ generated random bytes for every byte of input, so a parallel implementation would either have to pre-generate the extra coefficients, or synchronize their generation across multiple threads. Preliminary performance numbers for a parallelized Rabin can be seen in Table 4. Please note, that we are currently parallelizing up to n (maximum number of shares) threads — in our test-case, the embedded Intel Atom processor would offer more cores than our maximum number of shares, thus no full utilization of the processor cores is achieved. There is a slight performance hit when the multi-threaded code is run on a single-core processor: this is grounded within the initial setup overhead of the multi-threaded code. To get the best performance, two separate versions of the library (or instantiations within the library) for single- and multi-core usecases might be advantageous.

The speed-up for Rabin on a quad-core was around the expected factor of four. Krawczyk

Table 3: Single-Core throughput upon plaintext-data of different secret-sharing algorithms on different Intel CPUs with $n = 4, k = 3$, measurements in kilobyte/second. The Intel i5 is a good example of a middle-range Desktop CPU while the Atom C2758 is typical for an embedded high-range CPU.

CPU	Speed	Algorithm	Split-Up	Combine
i5-4690K	3.5GHz	Shamir	42533.7	54179.9
C2758	2.4Ghz	Shamir	9212.8	10790.3
Exynos A9	1.7GHz	Shamir	3400.6	3940.7
ARM A53	1.2Ghz	Shamir	1695.1	2034.9
i5-4690K	3.5GHz	Rabin	123746.2	129211.4
C2758	2.4Ghz	Rabin	28603.4	26056.0
Exynos A9	1.7GHz	Rabin	9915.3	10559.4
ARM A53	1.2Ghz	Rabin	5102.1	5346.6
i5-4690K	3.5GHz	Krawczyk	79534.0	80313.7
C2758	2.4GHz	Krawczyk	17754.7	16828.3
Exynos A9	1.7GHz	Krawczyk	6626.8	6813.0
ARM A53	1.2Ghz	Krawczyk	3591.4	3773.0

Table 4: Preliminary performance (Version $a8a344b$) with multi-core enabled Rabin Data Dispersal. As Rabin is also used by Krawczyk, its performance should also improve. Please note, that currently a maximum number of n threads is used. The tested Atom C2758 board offers 8 cores, so it is not utilized to the maximum of its abilities.

CPU	Speed	Algorithm	Split-Up	Combine
i5-4690K	3.5GHz	Shamir	53753.3	52378.0
C2758	2.4Ghz	Shamir	8192.0	7881.5
Exynos A9	1.7GHz	Shamir	6917.8	3932.0
ARM A53	1.2Ghz	Shamir	3502.4	2056.9
i5-4690K	3.5GHz	Rabin	694237.3	124498.5
C2758	2.4Ghz	Rabin	99417.5	26072.6
Exynos A9	1.7GHz	Rabin	67590.8	10529.6
ARM A53	1.2Ghz	Rabin	40634.9	4395.2
i5-4690K	3.5GHz	Krawczyk	167868.9	79844.1
C2758	2.4GHz	Krawczyk	32507.9	16862.9
Exynos A9	1.7GHz	Krawczyk	16430.0	6949.4
ARM A53	1.2Ghz	Krawczyk	9669.5	3765.1

Table 5: Impact of Processor Architecture and difference incoming data sizes upon Rabin throughput. Multi-threaded algorithm used, all Values in kByte per Second.

Data Size	Intel Core i5-4570		ARM A9	
	Split-Up	Combine	Split-Up	Combine
4 kB	117028.6	84553.6	39309.0	9992.7
64 kb	481882.4	121904.8	74744.5	10890.7
128 kb	460224.7	123373.5	76560.7	10922.7
512 kb	630153.8	124121.2	65958.1	10873.4
1024 kb	650158.7	123746.2	107789.5	10870.5
2048 kb	660645.2	124498.5	111304.3	10884.9
4096 kb	620606.1	124878.0	111404.3	10873.4

is a combination of Shamir, Rabin and symmetrical encryption. The secret-sharing key is shared using Shamir, the original data is encrypted using a traditional symmetric encryption and then shared with Rabin. If the existing symmetric encryption library (BouncyCastle in our case) scales as well as our new parallelized Rabin code, then Krawczyk should also improve around a factor of 4. As our empirical tests have shown, a factor of 2 is achieved. This indicates that the existing symmetric encryption engine also has become a bottleneck with our new code. Future research into alternative cryptographic engine as well as into Java 9 (which improves hardware support for cryptographic directives prevalent in modern processors) is planned.

Another possibility is to split the input across its length. In that case, the problem becomes the assembly and ordering of the generated output. Currently, this is feasible as both input and output buffers are byte arrays. Parallelization can be achieved by computing indices within those arrays. Alas, as mentioned before, byte arrays are not a scalable solution. Therefore, we will switch to a stream-based implementation, which renders this parallelization approach infeasible.

For further optimization, we performed pre-processing of constant factors for common operations. E.g., when sharing a secret, for every generated output share, all multiplications are performed with the same factor. We are currently investigating another example of preprocessing: when reconstructing a secret, a decoder matrix has to be generated — instead of using generic matrix multiplication we could decompose and simplify the matrix.

When testing multiple input file sizes, we found that a block size of 4 Kilobyte produced significantly reduced performance. With larger input sizes, the performance was generally constant on the Intel Core i5 CPU. On passively cooled embedded devices (ARM as well as Intel Atom processors) the power- and thermal management prevented the creation of reproducible performance benchmarks. Future research in to the power consumptions and thermal impact of differnet algorithms might be needed. An overview of the recorded throughput values can be seen in Table 5.

9.2 Archistar Web Based Data Sharing

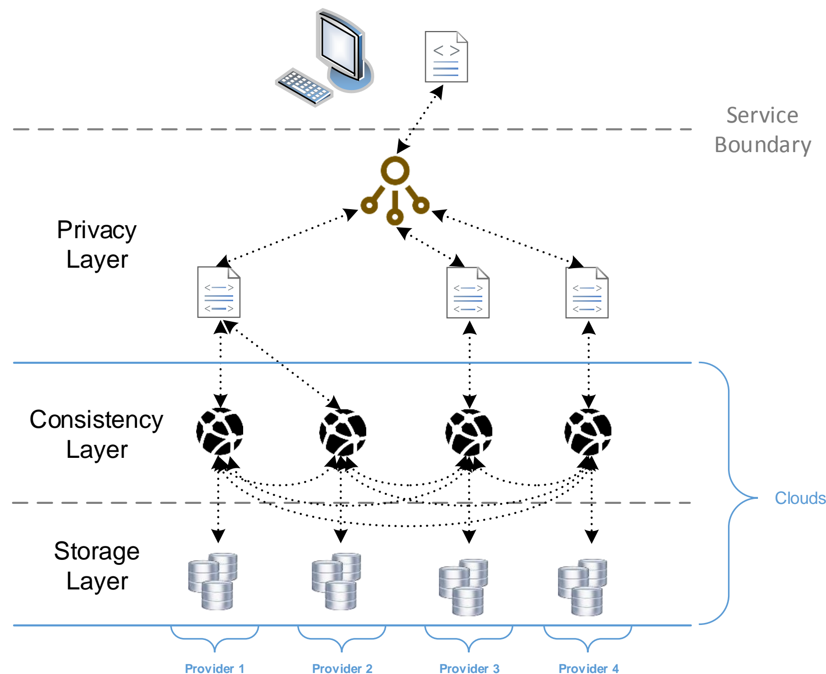


Figure 14: Archistar Web Based Data Sharing Service.

As depicted in Figure 14 our web-based data sharing service consists of two different sites. Thus, we provide a module that operates on the client or end-user site, providing privacy and security by generating secret shares which are distribued among the selected cloud providers (c.f. Figure 14, privacy layer). We further provide a module for the cloud providers which provides protocols for guaranteeing robustness, availability and consistency among the employed cloud providers (c.f. Figure 14, conistency layer). However, both modules should not be tightly coupled such that both can be independently used by third party software. Allowing third parties to provide their own solution for maintaining high availability or confidentiality.

The modules are implemented using node.js. For further information on the used modules we refer the interested reader to Section 5 and Sectiond 4. The modeuls provide RESTful APIs such that they can easily combined with third party software providing either part (privacy layer or consistency layer).

9.2.1 Preliminary Performance Evluation

In order to conduct a preliminary performance evaluation, we compare the performance, in terms of time in milliseconds until shares are stored in the cloud and time needed for

				With PBFT		Without PBFT	
		Upload		Donwload			
		Shares (ms)	Total (ms)	Reconstruct (ms)	Total (ms)	Reconstruct (ms)	Total (ms)
20 kiB	Upper CI	187.13	1146.92	367.93	397.62	355.64	366.09
	Average	144	945	303.8	341.9	280.3	291.7
	Lower CI	100.87	743.08	239.67	286.18	204.96	217.31
250 kiB	Upper CI	1014.00	1951.15	1914.38	1956.70	1836.50	1854.75
	Average	973	1793.1	1784.9	1815.6	1757.8	1777.9
	Lower CI	932	1635.05	1655.42	1674.50	1679.10	1701.05
5 MiB	Upper CI	17938.91	20343.63	36019.98	36048.38	31707.09	31724.95
	Average	17597.3	19933.8	33644.1	33669.4	30632.5	30647.1
	Lower CI	17255.69	19523.97	31268.22	31290.42	29557.91	29569.25
10 MiB	Upper CI	34937.95	40338.87	69009.32	69194.00	57705.20	57721.21
	Average	34077.6	38860	65588.9	65000.2	55702	55718.7
	Lower CI	33217.25	37381.13	62168.48	60806.40	53698.80	53716.19

Table 6: Preliminary performance results (95% confidence intervals) for different file sizes with our cloud storage solution and without.

fetching those stored shares again, of our cloud storage service solution to only using secret sharing at the end-user's site and without our BFT solution. Using real cloud services to deploy and evaluate the performance will introduce uncontrolable influence factors, e.g., bandwidth and latency fluctations caused by other network applications or the Internet Service Provider (ISP). Thus, we decided to use virtual machines on the same host device. Each virtual machine had the following specification: single core CPU with 2.7 Ghz, 1 GB RAM, 10 GB disc space, running Ubuntu Server 16.04. For the BFT test case, we instantiated five nodes, three nodes running our BFT solution only, one node for the mediator module, and one node representing the client. The single node for a client is important due to the nature of our BFT solution. Since we are using the BFT system to store secret shares, collecting them upon requrest at a single BFT node would allow to reveal the secret at this node. For the test case without BFT we used three storage nodes which represented three different cloud storage service provider. We selected 20 kiB, 250 kiB, 5 MiB, and 10 MiB test files which were subject to secret sharing prior uploading them to one of the cloud storage solutions. For each pairing of cloud storage solution and file size we made ten runs.

Table 6 depicts the results of the preliminary evaluation. For small file sizes, i.e., 20 kiB, there is only a small difference between the cases with PBFT and without PBFT. With an increase in file size also the difference in downloading increases. However, thinking of the overhead PBFT introduces even the higher retreival times for 10 MiB are acceptable compared to not having PBFT in place.

10 Summary and Outlook

In this report we present the design of the *SECOSTOR* tool. The tool architecture is introduced and most important terms are defined. A high level component model was used to communicate the overall architecture and building blocks as well as major features to expect from the tool.

After the introduction of the architecture, the two major modules of the tool are introduced, namely the secret sharing module and the BFT module. The secret sharing module contains a comprehensive set of encoding and decoding algorithms for secret sharing and information dispersal, all of which are relevant in the storage setting. The module is implemented as Java library and give developers easy access to most relevant schemes. An additional JavaScript implementation has also been done to increase the deployment options and improve exploitation possibilities.

The BFT module implements a robust consistency layer for managing concurrency and faults in a distributed storage network. The BFT module is able to cope with malicious behavior and currently resembles the protocols known as PBFT (practical Byzantine fault tolerance) for replicated plaintext data. The software implementation of the BFT module was done in JavaScript and leverages modern web technologies, Node.js among others for maximum portability and flexibility in deployment. The BFT module is also fully compatible with the secret sharing module to speedup service development.

Additionally to the basic functionality more protocols have been researched and developed. In particular the report recommends a set of very efficient and easy to implement add-on protocols for remote data checking, client verification, recovery, and private information retrieval, which can be easily integrated into applications based on the SECOSTOR tool libraries.

Another important topic discussed in this report is access control mechanisms. Because of the security model of secret sharing, i.e. the non-collusion assumption, access control is extremely important to prevent adversaries from getting enough shares. We therefore, also discussed favorable combinations with asymmetric cryptography to get best of both worlds in a mixed (hybrid) system. The proposed cryptographic solutions are currently not part of the prototype, but existing implementations from other projects can be used and used together with the SECOSTOR tool. However, future versions of the tool will also support reasonable combinations of secret sharing and modern asymmetric cryptography.

In the last section we quickly present the services built in PRISMACLOUD based on the SECOSTOR tool. We do this to demonstrate the usage and flexibility of the tool as well as to showcase two extreme deployment scenarios. On one hand, we show how a straightforward solution could look like in form of a trusted gateway. On the other hand, we demonstrate a complex multi-user multi-cloud deployment with no single point of trust or failure, which is the most secure and robust one in our current security model based on non-collusion. Additionally, in this section we also report performance values to give the reader a glimpse on the achievable throughput of such systems.

The tool can be considered feature complete and ready for integration into the pilot and

use cases specified during the first phase of the project. The rest of the software activities on the tool level will focus on support and bug fixing activities to assist in the piloting activity. Nevertheless, research on possible extensions is still ongoing in the project. For the last phase of PRISMACLOUD the SECOSTOR tool will be extended on the conceptual level with more verification protocols and quantum-safe algorithms for long-term security, i.e., by integrating research results in verifiable computing.

List of Figures

1 PRISMACLOUD architecture. 5

2 *SECOSTOR* overview. 8

3 Required redundancy to achieve high-availability. 9

4 Core libraries of the *SECOSTOR* tool. 10

5 Secure Object Storage Tool (SECSTOR). 10

6 Services based on SECOSTOR. 12

7 Overview crypto engine. 26

8 Class diagram secret sharing implementation. 27

9 Class diagram of share/fragment data structure. 28

10 Overview *PBFT* protocol. Black arrows describe the message flow of the vanilla version. The dashed blue arrows show the extension needed for integration with secret-sharing. 30

11 Architecure of the BFT Module. 31

12 Classes of the Mediator module. 32

13 High-Level network system overview and example data-flow from backup clients to the cloud storage servers. 53

14 Archistar Web Based Data Sharing Service. 58

List of Tables

1 Comparison of minimal server amount and storage overhead between different storage schemes. The baseline for the overhead is given as a factor based upon the original data amount. 48

2 Number of leading nines for system reliability $R(k, n)$ for given n and k and a individual storage node reliability of $p = 0.98$, which is a typical value taken from cloud storage provider SLA. 50

3 Single-Core throughput upon plaintext-data of different secret-sharing algorithms on different Intel CPUs with $n = 4, k = 3$, measurements in kilobyte/second. The Intel i5 is a good example of a middle-range Desktop CPU while the Atom C2758 is typical for an embedded high-range CPU. 56

4 Preliminary performance (Version $a8a344b$) with multi-core enabled Rabin Data Dispersal. As Rabin is also used by Krawczyk, its performance should also improve. Please note, that currently a maximum number of n threads is used. The tested Atom C2758 board offers 8 cores, so it is not utilized to the maximum of its abilities. 56

5 Impact of Processor Architecture and difference incoming data sizes upon Rabin throughput. Multi-threaded algorithm used, all Values in kByte per Second. 57

6 Preliminary performance results (95% confidence intervals) for different file sizes with our cloud storage solution and without. 59

References

[ABC+07] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Xiaodong Song. Provable Data Possession at Untrusted Stores. In *CCS '07*, pages 598–609. ACM, 2007.

[BGR96] Mihir Bellare, Juan A. Garay, and Tal Rabin. Distributed Pseudo-Random Bit Generators - A New Way to Speed-Up Shared Coin Tossing. In *PODC 1996*, pages 191–200. ACM, 1996.

[BKLP15] Fabrice Benhamouda, Stephan Krenn, Vadim Lyubashevsky, and Krzysztof Pietrzak. Efficient Zero-Knowledge Proofs for Commitments from Learning with Errors over Rings. In *ESORICS '15, Part I*, volume 9326 of *LNCS*, pages 305–325. Springer, 2015.

[BSW07] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, pages 321–334, 2007.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, pages 41–50, 1995.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[CPK10] Yanpei Chen, Vern Paxson, and Randy H. Katz. What's New About Cloud Computing Security? Technical Report UCB/EECS-2010-5, University of California, Berkeley, 2010.

[DGH12] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally Robust Private Information Retrieval . In *21st USENIX Security Symposium*, 2012.

[DKL+17] David Derler, Stephan Krenn, Thomas Lorünser, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. Forward-secure proxy re-encryption. Technical report, July 2017.

[DKLT16] Denise Demirel, Stephan Krenn, Thomas Lorünser, and Giulia Traverso. Efficient and Privacy Preserving Third Party Auditing for a Distributed Storage System. In *11th International Conference on Availability, Reliability and Security, ARES 2016, Salzburg, Austria, August 31 - September 2, 2016*, pages 88–97. IEEE Computer Society, 2016.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, pages 303–320, 2004.

[FMY98] Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust Efficient Distributed RSA-Key Generation. In *PODC '98*, page 320, 1998.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[Gol07] Ian Goldberg. Improving the robustness of private information retrieval. In *IEEE Symposium on Security and Privacy*, pages 131–148, 2007.

[GPSW06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 89–98, 2006.

[GWGR04] Garth R Goodson, Jay J Wylie, Gregory R Ganger, and Michael K Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Dependable Systems and Networks, 2004 International Conference on*, pages 135–144. IEEE, 2004.

[HHG13] Ryan Henry, Yizhou Huang, and Ian Goldberg. One (Block) Size Fits All: PIR and SPIR Over Arbitrary-Length Records via Multi-block PIR Queries . In *20th Network and Distributed System Security Symposium*, 2013.

[HHLD11] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[HKL16] Andreas Happe, Stephan Krenn, and Thomas Loruenser. Malicious clients in distributed secret sharing based storage networks. 2016.

[HPSP10] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[KLS17] Stephan Krenn, Thomas Loruenser, and Christoph Striecks. Batch-verifiable Secret Sharing with Unconditional Privacy. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*, pages 303–311. INSTICC, ScitePress, 2017.

[Kra93a] Hugo Krawczyk. Distributed Fingerprints and Secure Information Dispersal. In Jim Anderson and Sam Toueg, editors, *Principles of Distributed Computing - PODC 1993*, pages 207–218. ACM, 1993.

[Kra93b] Hugo Krawczyk. Secret Sharing Made Short. In Douglas R. Stinson, editor, *Advances in Cryptology - CRYPTO 1993*, volume 773 of *LNCS*, pages 136–146. Springer, 1993.

[LAS15] Thomas Lorünser, Andreas Happe, and Daniel Slamanig. ARCHISTAR: Towards Secure and Robust Cloud Based Data Sharing. In *IEEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015, Vancouver, November 30 - December 3*. IEEE, 2015.

[LHS15] T. Loruenser, A. Happe, and D. Slamanig. Archistar: Towards secure and robust cloud based data sharing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 371–378, Nov 2015.

[LRD+15] Thomas Lorünser, CharlesBastos Rodriguez, Denise Demirel, Simone Fischer-Hübner, Thomas Groß, Thomas Länger, Mathieu des Noes, HenrichC. Pöhls, Boris Rozenberg, and Daniel Slamanig. Towards a New Paradigm for Privacy and Security in Cloud Services. In Frances Cleary and Massimo Felici, editors, *Cyber Security and Privacy*, volume 530 of *Communications in Computer and Information Science*, pages 14–25. Springer International Publishing, 2015.

[LSLP16] Thomas Lorünser, Daniel Slamanig, Thomas Länger, and Henrich C Pöhls. PRISMACLOUD Tools: A Cryptographic Toolbox for Increasing Security in Cloud Services. In *11th International Conference on Availability, Reliability and Security, ARES 2016, Salzburg, Austria, August 31 - September 2, 2016*, pages 733–741. IEEE Computer Society, 2016.

[Nod17] Node.js. An asynchronous JavaScript runtime environment, 2017. Online: *https://nodejs.org/*.

[NSG10] Mehrdad Nojoumian, Douglas R. Stinson, and Morgan Grainger. Unconditionally secure social secret sharing scheme. *IET Information Security*, 4(4):202–211, 2010.

[Ray01] Jean-François Raymond. Traffic Analysis: Protocols, Attacks, Design Issues and Open Problems. In *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, pages 10–29. Springer-Verlag New York, Inc., 2001.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract). In David S. Johnson, editor, *ACM Symposium on Theory of Computing - STOC 1989*, pages 73–85. ACM, 1989.

[Sat90] Mahadev Satyanarayanan. A survey of distributed file systems. *Annual Review of Computer Science*, 4(1):73–104, 1990.

[SH12] Daniel Slamanig and Christian Hanser. On cloud storage and the cloud of clouds approach. In *ICITST 2012*, pages 649–655, 2012.

[Sha79] Adi Shamir. How to Share a Secret. *Commununications of the ACM*, 22(11):612–613, 1979.

[SST17] Daniel Slamanig, Christoph Striecks, and Giulia Traverso. Efficient sharing-based storage protocols for mixed adversaries. Technical Report D4.3, July 2017.

[SW04] Amit Sahai and Brent Waters. Fuzzy identity based encryption. *IACR Cryptology ePrint Archive*, 2004:86, 2004.

[WB83] Lloyd Welch and Elwyn Berlekamp. Error Correction of Algebraic Block Codes. US Patent #4,633,470, 1983.