Long-Term Secure Block Device

D2.2

Project reference no. 653884

August 2016

European Commission

Horizon 2020
European Union funding
for Research & Innovation

Document information

Scheduled delivery	16.01.2017
Actual delivery	16.01.2017
Version	1.3
Responsible Partners	UniNE

Dissemination level

Public

Revision history

Date	Editor	Status	Version	Changes
27.05.2016	H. Mercier	Draft	0.1	Initial TOC
05.08.2016	H. Mercier	Draft	0.2	First draft completed
24.08.2016	D. Burihabwa	Draft	0.3	Internal review
25.08.2016	P. Felber	Draft	0.4	Internal review
26.08.2016	L. Yazdanov	Draft	0.5	Internal review
26.08.2016	D. Burihabwa	Draft	0.6	Internal review
29.08.2016	P. Felber	Draft	0.7	Comments from C&H, conclusion
29.08.2016	P. Felber	Final	1.0	Comments from INESC-ID, finalized
15.09.2016	H. Mercier	Final	1.1	Renaming of storage solutions for uniformity across deliverables
21.12.2016	P. Felber	Draft	1.2	Revision after first review
12.01.2017	H. Mercier	Final	1.3	Final check before resubmission

Contributors

H. Mercier, D. Burihabwa, V. Schiavoni, P. Felber (UniNE)

Internal reviewers

L. Yazdanov (C&H), M. Correia (INESC-ID)

Acknowledgements

This project is partially funded by the European Commission Horizon 2020 work programme under grant agreement no. 653884.

More information

Additional information and public deliverables of SafeCloud can be found at http://www.safecloud-project.eu

Table of contents

Executive summary

The deliverable presents the theoretical background of the long-term secure block device along with the description of a prototype implementation (SafeStore). Long term means that strong anti-tampering and data integrity can be provided once the data has been stored for a long period of time in the system.

We first discuss requirements and design considerations. We then describe the entanglement process and analyse its robustness to various types of attacks that try to corrupt documents in the data store. We then provide details on the architecture of SafeStore, an experimental testbed to evaluate the performance trade-offs of the security guarantees offered by the SafeCloud platform.

1 Introduction

The SafeCloud consortium provides three storage solutions: secure block storage (SS1), a secure data archive (SS2), and a secure file system (SS3). This deliverable describes how long-term data entanglement is achieved. More precisely, by long-term we mean that strong anti-tampering and data integrity can be provided once the data has been stored for a long period of time in the system. This is a fundamental part of SS2, but is also paramount to SS1 since our secure data archive is built on top of the secure block storage. This deliverable also describes SafeStore, the experimental testbed used to test all the storage solutions of SafeCloud, including SS3.

The remaining of this document is organized as follows. In Section 2, we discuss requirements and design considerations. In Section 3, we revisit how we use data entanglement in our secure data archiving system. The main operating principle behind long-term data entanglement and its application in the context of SafeCloud's secure block storage component are presented in Section 4. Finally, in Section 5, we describe our current implementation prototype and testbed. The implementation was first introduced in deliverable D2.1, and this current deliverable includes the progress made in the last six months.

1.1 Storage solutions new names

Please note that we changed the name of two of our three storage solutions for simplicity purposes and uniformity across all other deliverables. The old names, only used in D2.1, will no longer be used. We will use the new names, described as follows, until the end of the project:

Storage solution	Old name in D2.1	New name in all deliverables
Solution 1 (SS1)	Secure block device	Secure block device
Solution 2 (SS2)	Long term distributed encrypted document storage	Secure data archive
Solution 3 (SS3)	Distributed encrypted file system	Secure file system

2 Requirements and design considerations

The overall design of the SafeCloud architecture is guided by several requirements and practical considerations. We briefly discuss them in this section, with an emphasis on the thrustworthy storage layer.

The SafeCloud architecture design is driven by the need to provide security in two very different but complementary manners:

- Provide a secure data store spanning several data centres across distinct administrative domains in a way that data cannot be accessed without compromising all data sources;
- Provide a secure data store for long-term data archival that cannot be deleted or tampered with. In that sense, the data store will provide both integrity and censorship resistance.

The focus of SafeCloud is on providing innovative security mechanisms, and not on the practical aspects dealing with the implementation of proven technologies that are readily available. Hence our design assumes that the final SafeCloud architecture can be supported by state-of-the-art coding and storage techniques, which are key to producing a robust, efficient and scalable implementation. This approach was paramount in designing all the components of the storage layer. For instance, for the secure data archive, most proposed solutions so far either do not provide the privacy and security guarantees that we do, or has an unacceptable performance.

We thus have evaluated and benchmarked several existing components (backend data store and coding/encryption libraries) to identify those that could be best integrated as basic building blocks in the SafeCloud architecture. The results of our study are summarized Section 5.

One main requirements regarding the choice of the components is that they should be efficient, robust, and well maintained. Furthermore, they should integrate seamlessly with SafeCloud's own components, and hence use APIs that are interoperable with Java, C, and Python code—which are the main languages used in SafeCloud. While coding/encryption libraries should be linked directly with SafeCloud's code, other components can be accessed via a REST API and hence can use any programming language, as long as they can be packaged and executed in a standard container.

Regarding the backend data store, SafeCloud's mechanisms assume a simple key-value store with explicit placement of data blocks (i.e., values) by a given node. This is essential because the security and robustness of the algorithms depend on the placement of the blocks (in different data centres, or on different nodes/disks in the same data centre).

Regarding the coding libraries, SafeCloud's entanglement algorithms rely on classical Reed-Solomon codes. SafeCloud's security mechanisms also leverage classical cyphers (AES and RSA) and cryptographic algorithms (signatures, checksums). Our implementation choices, which are described in Section 5 and summarised in the table below, have been largely driven by these practical considerations and requirements.

Component	Requirements	Implementation language
Backend data store	Key-value API; explicit data placement	Any
Coding libraries	Reed-Solomon codes; classic cyphers	Java, C/C++, Python

3 SafeCloud archival using data entanglement (revisited)

As part of SafeCloud, we introduce STEP-archives, a storage system for archiving coded documents. Using data entanglement and erasure-correcting codes, we develop a data storage architecture where a stored document can only be deleted or modified by compromising the integrity of other documents in the system.

There are two main objectives behind this work. The first objective is data integrity. We want to provide guarantees to users that their data cannot be deleted or corrupted without compromising other data stored by themselves or other users. The second objective is to provide censorship resistance by forcing a censor who wants to tamper with data to do so noisily, i.e., being forced to corrupt a large number of other documents in the system. An ancillary result deriving from the two objectives is increased protection against failures, which can be seen as attacks from random or failure-specific censors.

3.1 Practical considerations

We emphasize that one of the objectives of SafeCloud is to achieve both data integrity and censorship resistance in a way implementable in practical systems. Thus, we use practical constraints that keep an actual implementation realistic (for instance avoiding reads and writes requiring a non-constant part of the total system size) while being simple enough to allow analysis. All our underlying assumptions and design choices are implementable using state-of-the-art coding and storage techniques, which is paramount for a large-scale implementation.

3.2 Entanglement architecture using erasures codes

Definition 1. A (s,t,e,p)-archive is a storage system where each archived document consists of a codeword with s source blocks, t tangled blocks, p parity blocks and that can correct $e = p - s$ block erasures.

When a document is archived, it is split into $s \geq 1$ source blocks. Using the s source blocks with t distinct old blocks already archived, a systematic maximum distance separable (MDS) code [LC04] is used to create $p \geq s$ parity blocks which are then archived on the system.

An archived document can be recovered from $s + t$ or more of its blocks. The code can correct p block erasures per document codeword, but since the source blocks are not archived and are considered as erased, at most $e = p - s$ block erasures per document on the storage medium can be corrected. Note that increasing t does not increase storage overhead or error-correcting capability, but does increase coding and decoding complexity.

An attacker can censor a document d_k by erasing more than e of its blocks. However, by entangling new documents with documents already archived, it might be possible for the system to recover the deleted blocks by decoding other documents that use them. As an example, consider the (1,3,2,3)-archive presented in Figure 1. Each document codeword consists of one source block, three pointers to old blocks, and three parity blocks. Only the parity blocks are stored when a new document is archived; the source block is not stored and the pointer blocks were previously stored as parity blocks of older documents. Block 0 is a known anchor that cannot be corrupted. If an MDS code is used, any four of the six stored blocks belonging to a document are sufficient to recover it (i.e., $e = 2$). In Figure 1, an attacker wants to censor document d_5 by erasing its blocks {2,7,11,13,14,15} from the archive. However, although d_5 cannot be recovered directly,

all the blocks are recoverable: Block 2 can be recovered by decoding d_1 or d_2, Block 7 can be recovered by decoding d_3, d_4 or d_8, Block 11 can be recovered by decoding d_4, Block 14 can be recovered by decoding d_7, Block 15 can be recovered by decoding d_8, and in the last step Block 13 can be recovered by decoding d_5. Having been unable to erase d_5, the attacker continues his attack more cleverly and further erases Blocks 20, 21, 22 and 24, as illustrated in Figure 2. Document $d_5$is now destroyed irrecoverably, as are also $d_7$and d_8 (the irrecoverable blocks and documents are shown in red). Blocks 2, 7 and 11 are still recoverable, which means that the attacker could have irrecoverably destroyed d_5 without destroying them.

The challenging part of our approach is thus to choose the pointers to entangled blocks in a way that is practical. Since we target practically implementable data integrity and anti-tampering, we focus on archives with t constant and small. Studying how to assign these pointers to maximise the tamper-proofing and protection against censorship is an important task of the project.

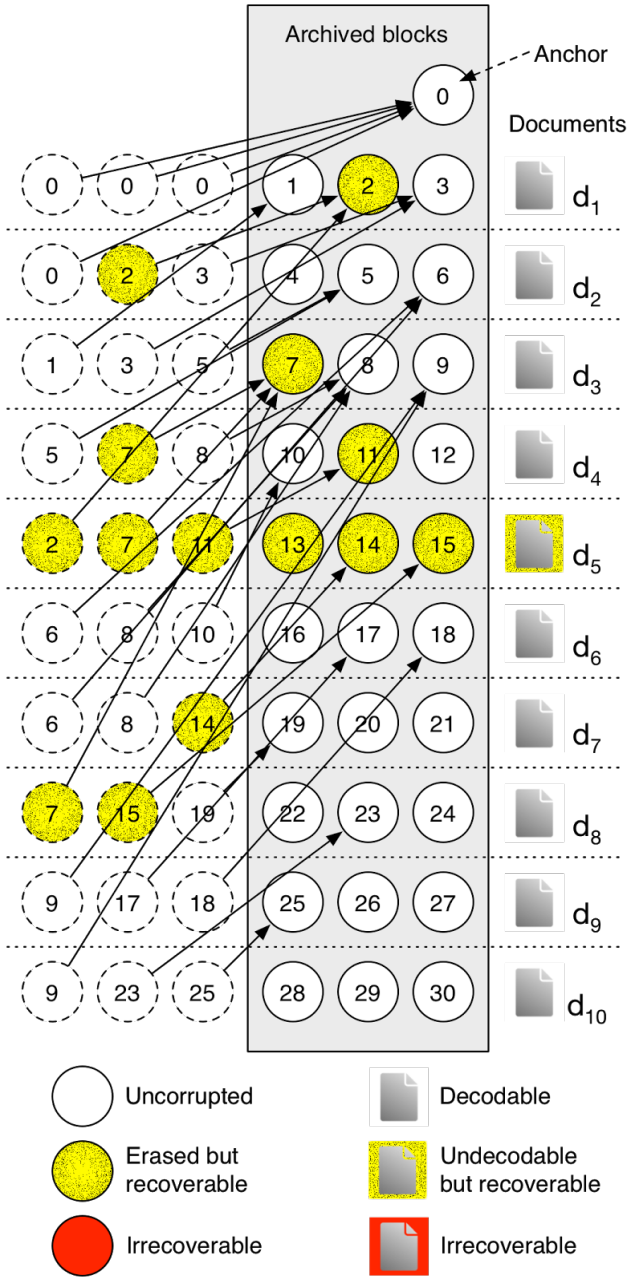


Figure 1: (1,3,2,3)-archive. 4 out of 6 blocks are required to recover a document. This example illustrates a recoverable attack.

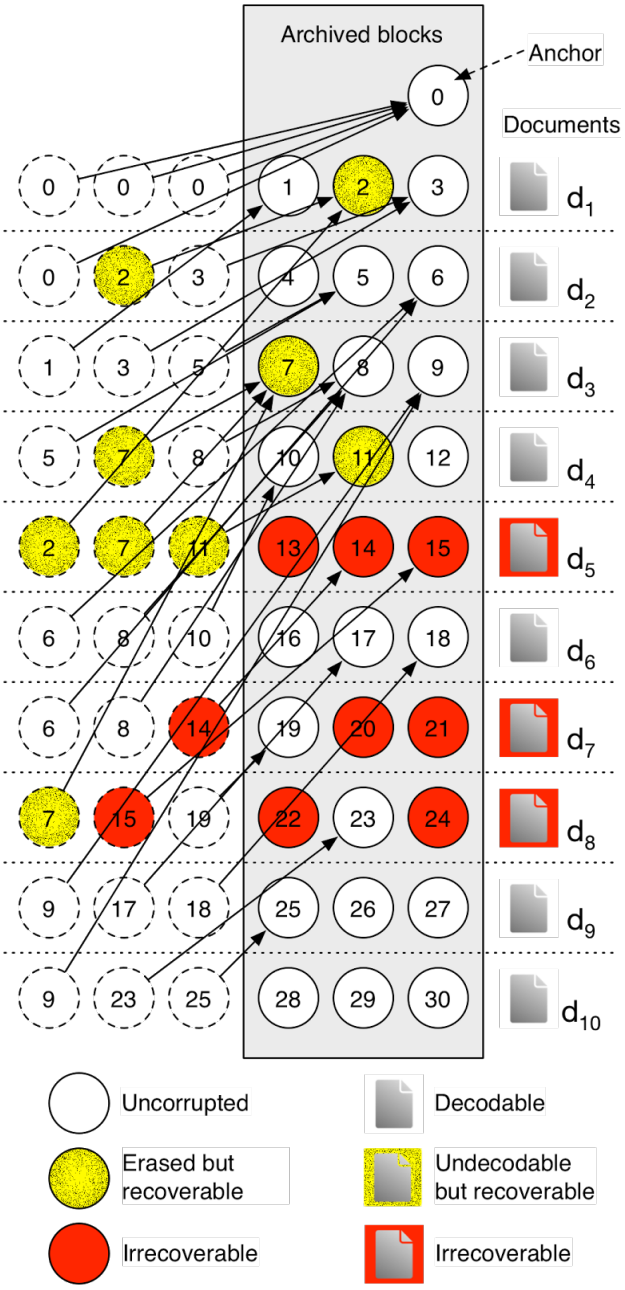


Figure 2: (1,3,2,3)-archive. 4 out of 6 blocks are required to recover a document. This example illustrates an irrecoverable attack.

4 Long-term data entanglement

In this section, we summarize the main theoretical concepts behind long-term data entanglement, which is achieved by selecting the pointers to old data blocks uniformly at random. We first describe the main principle of random entanglement. We then discuss various types of attacks to corrupt documents in the data store. Finally, we present simulation results of how random entanglement tolerates the considered attacks. The complete analysis, including all the mathematical proofs and an extended set of simulations, can be found in [MAL16].

4.1 Random entanglement

In practice, choosing entangled blocks uniformly at random offers three important advantages over highly structured entanglement. First, the problem is asymmetric between attackers and defenders: while a defender can efficiently recover from suboptimal attacks, an attacker must solve a NP-hard problem to find a perfect (irrecoverable) attack that minimizes collateral damage, or even just approximate this minimum within a reasonable ratio. The creation of randomness in the structure prevents the attacker from planning the attack in advance, for instance by using amortized cost expensive pre-computations tied to the system structure. Second, a deterministic structure is harder to implement and maintain in real-time in a large-scale distributed setting. Third, random entanglement can provide strong security guarantees once data has been archived long enough.

The main drawback of uniformly random entanglement is that as the archive gets bigger, it takes an increasingly longer time until new documents become properly protected. Providing quick protection after archival is one of the objectives of deliverable D2.5, thus this issue will be tackled later in the project.

4.2 Suboptimal attacks

In order to test random entanglement, and because we do not know any good polynomial-time algorithms to optimally attack our system (or even to find a good approximate solution), we turn to more specific techniques, taking the special structure of our archive into account. We therefore consider several linear-time heuristics.

All the heuristics formulate the attack as a search problem on a tree of partial solutions. A partial solution consists of a set of target documents we are currently committed to destroy, and a set of erased blocks. A solution is complete if the set of erased blocks is sufficient to make the target document set irrecoverable. A partial solution must be completed by deleting some blocks referenced by recoverable documents. To make sure the target document set is not recoverable, no destroyed blocks must be referenced by documents outside of the target set; every time we choose to destroy a new block, we must commit to destroy all documents referencing it. From a partial solution, every possible choice of new blocks to erase gives a new partial (or possibly complete) solution, forming a tree of solutions. For the initial solution, we take the set of documents to censor, along with a (yet) empty set of erased blocks. The simplest way to explore this lattice is with a greedy algorithm. We iteratively walk down the lattice of solutions along a single path, eventually reaching a complete solution.

3.2.1 Greedy Attacks

We defined, implemented and studied four greedy attacks. The **minimum attack** minimizes the set of corrupted target documents by always preferring blocks that are referenced by the least amount of documents not already in the target set. The **leaping**

attack is based on the intuition that it is easier to attack recent documents than older ones. Intuitively, we try to leap over documents by moving forward in time as fast as possible toward the end of the archive. The **creeping attack** intuitively tries to keep the set of corrupted documents as compact as possible in time. Finally, the **tailored attack** uses the expected number of times a parity block is used as a pointer by younger documents. This number can be calculated mathematically when the entangled blocks are chosen uniformly at random hence this attack is specifically tailored to uniformly random entanglement. This allows us to estimate, when we erase a block, the propagation of the attack to all the documents used by that block.

3.2.2 Depth-first search

The greedy algorithms are fast since their complexity is linear in the number of archived documents, but the quality of the solution they find is not always good. We therefore implemented a recursive depth-first search over the tree of partial solutions using our four heuristics. The first complete solution produced always matches the output of the greedy attack. The tree is then backtracked, looking for smaller integrity sets. Since the cost function is non-decreasing as we go down the tree, we can perform branch pruning as soon as the partial cost exceeds the cost of the currently best known solution.

This algorithm thus offers a trade-off between time and solution quality, at the expense of increased memory usage. Unfortunately, even with pruning, depth-first search is expensive and does not always progress to the optimal solution quickly. By searching depth-first, we spend a lot of effort trying to optimize the later stages of the attack, which may already be close to optimal, whereas the decisions with the most impact on the overall cost are the ones taken at the beginning of the attack.

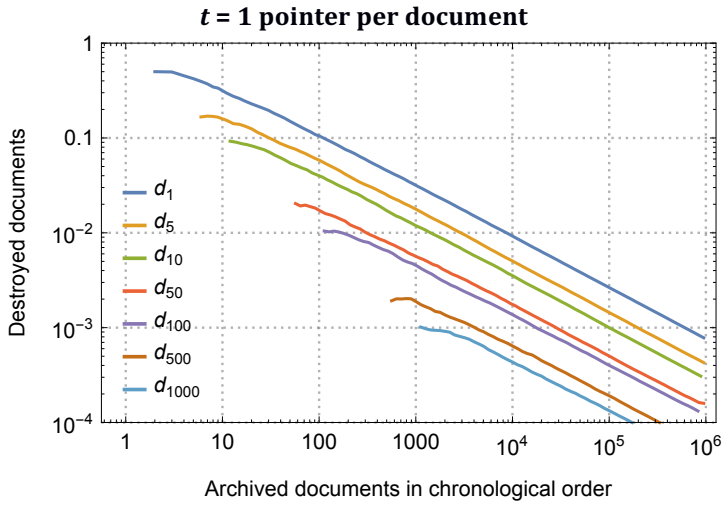
3.2.3 Bounded breadth-first search

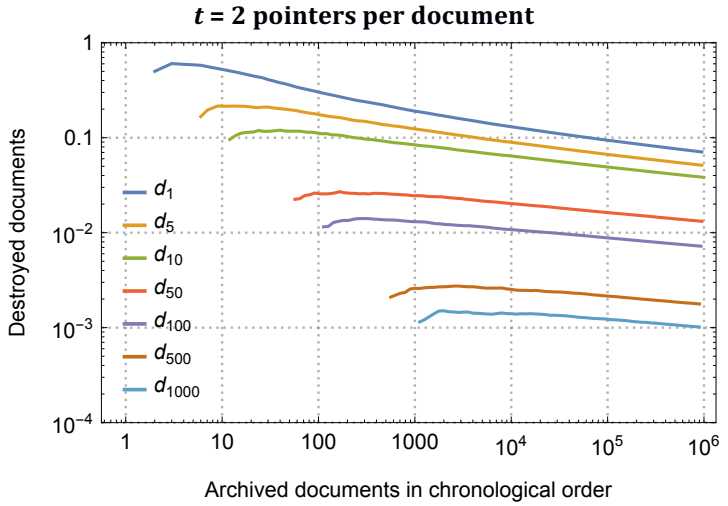
The inefficiency of depth-first search motivates the investigation of bounded breadth-first search algorithms. For large systems, it is impossible to traverse the entire solution tree, and bounded breadth-first search algorithms converge much faster than depth-first search protocols. We thus keep a collection of partial attack states, ranked according to some of our heuristics, and expand the most promising partial solutions first. We expand all the partial solutions into their child states at once, then only retain the best ones, up to the selected buffer size. We thus deal with a series of sets of solutions, for which all solutions in the same set are located at the same depth in the tree. This simplifies the analysis of the behaviour: we can enforce a constant maximum width, for all depths, on the subtree we are exploring. We cannot apply the same pruning strategy as in the depth-first search, because no complete solution is known before the end of a run, but we can control how much time we spend in the most critical part of the search tree.

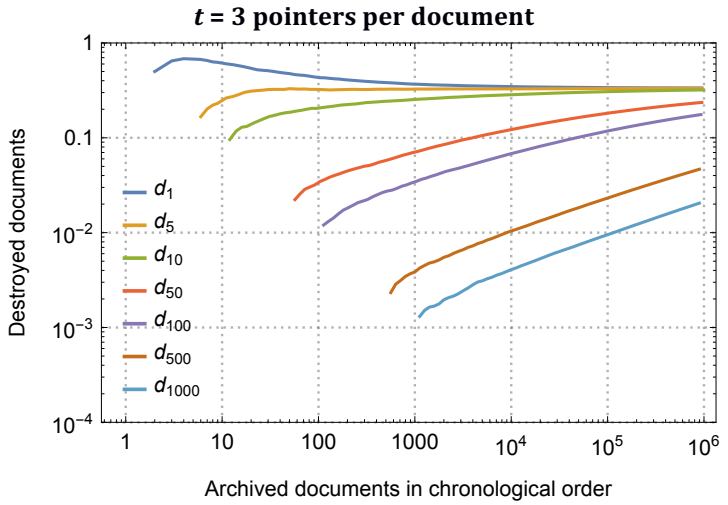
4.3 Simulation results

We simulated the damage caused by the leaping attack on (1, t, 2, 3)-archives of size 1,000,000 with pointers chosen uniformly at random. The results are shown in Figure 3. The six subfigures correspond to $t \in \{1,2,3,4,5,10\}$ pointers per document, respectively. On each subfigure, the seven curves respectively represent target document $\{d_1, d_5, d_{10}, d_{50}, d_{100}, d_{500}, d_{1000}\}$. Each curve is the average over 100 simulations. The phase transition as the number of pointers increases is obvious from the graphs. When reaching the threshold, the asymptotic cost of the leaping attack no longer depends on the target document: an attacker who wants to irrecoverably destroy a document must destroy a constant fraction of all documents archived after it. Increasing t further

accelerates the convergence and increases the fraction of documents that must be destroyed.







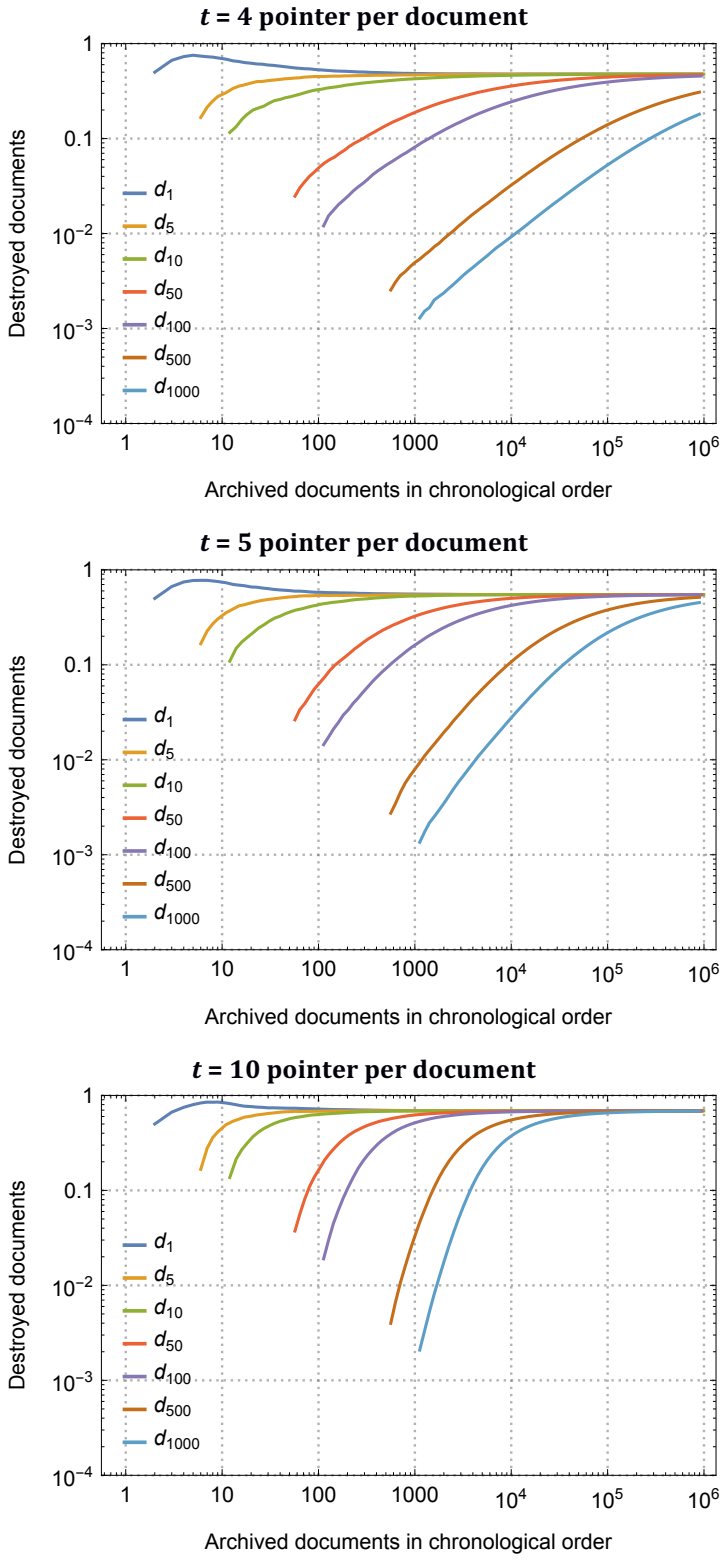
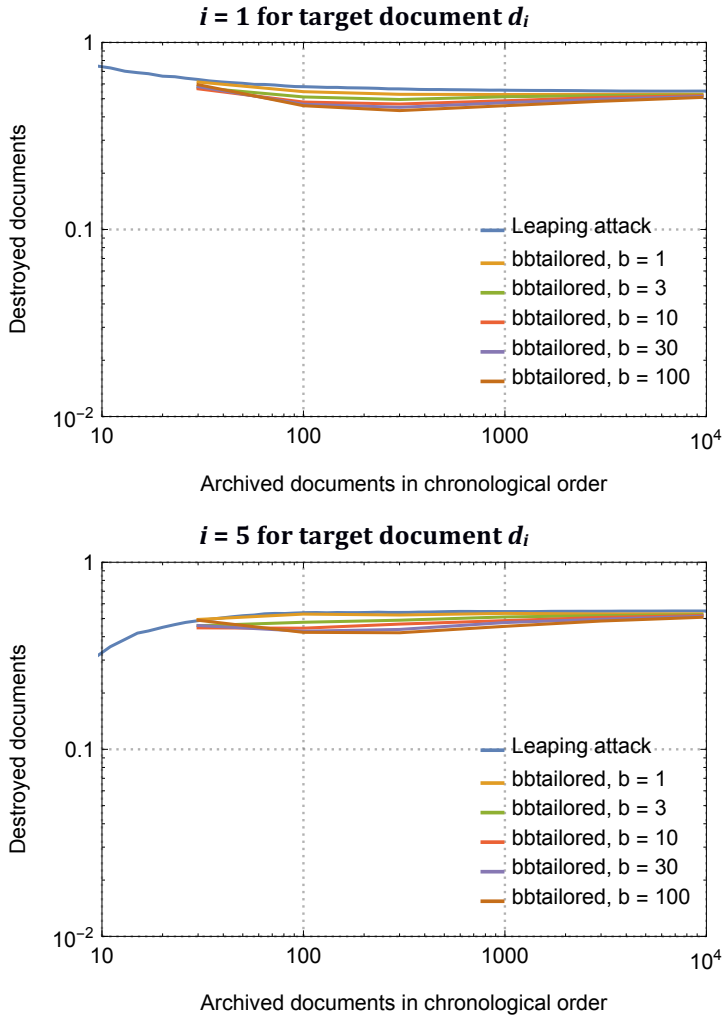


Figure 3: Damage caused by the leaping attack on (1, t, 2, 3)-archives of size 1,000,000 with pointers chosen uniformly at random. The six subfigures correspond to different configurations with $t \in \{1, 2, 3, 4, 5, 10\}$ pointers per document, respectively. On each subfigure the curves represent target document $\{d_1, d_5, d_{10}, d_{50}, d_{100}, d_{500}, d_{1000}\}$, respectively. Each curve is the average over 100 simulations.

We conjectured, with entanglement chosen uniformly at random, that there is a constant number of pointers threshold after which even the optimal attack will require the erasure of a constant fraction of all documents archived after an old enough target. Since simulating the optimal attack is computationally intractable and we do not have good enough theoretical lower bounds, to support this conjecture we simulate the bounded breadth-first search attack described in the previous section. By bounding the size of the buffer, we can control the number of nodes traversed at each level of the solution tree.

Figure 4 shows the damage caused by the tailored bounded breadth-first search attack on (1,5,2,3)-archives of size 10000 with pointers chosen uniformly at random and target document d_i for $i \in \{1,5,10,50,100\}$. On each subfigure, the curves represent different tree widths (buffer sizes). The leaping attack is also shown for comparison. Each curve is the average over 100 simulations. The figures provide numerical evidence supporting our conjecture and show the efficiency of the greedy leaping attack.



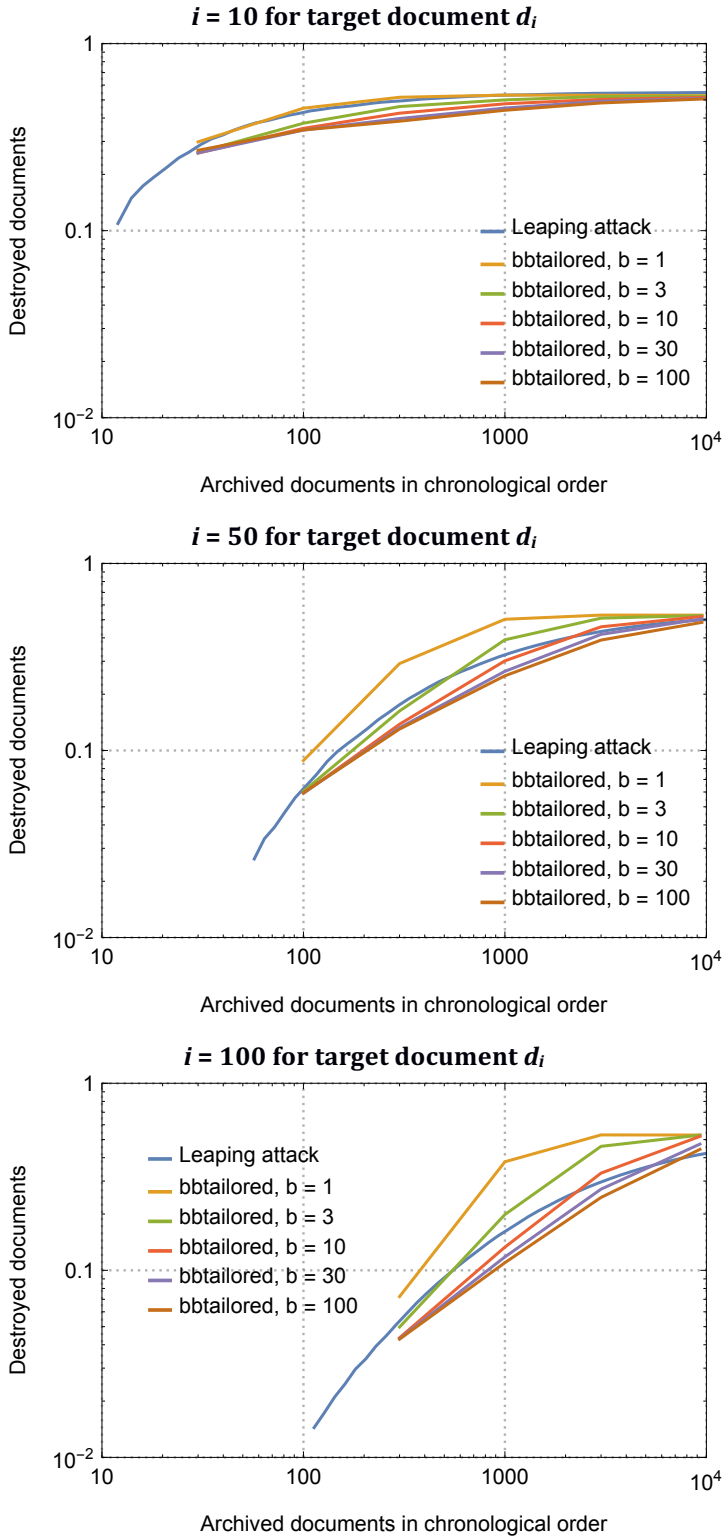


Figure 4: Damage caused by the tailored bounded breadth-first search attack on (1,5,2,3)-archives of size 10000 with pointers chosen uniformly at random. The five subfigures correspond to target document d_i for $i \in \{1, 5, 10, 50, 100\}$. On each subfigure, the curves represent different tree widths (buffer sizes), from 1 to 100. The leaping attack is also shown for comparison. Each curve is the average over 100 simulations

4.4 Discussion

The simulations are paramount to set the appropriate number of pointers to entangled blocks. For instance, using a code rate of 1/3, which corresponds to the storage overhead of triple replication, Figure 3 illustrates that using $t = 5$ pointers is a good empirical choice. Having less than five pointers might not lead to good long-term protection, especially if the attacker uses sophisticated attacks that we are unable to simulate. Conversely, using more than five pointers increases the speed at which new documents are protected, but at a performance cost since encoding, decoding and system maintenance will be more complex.

5 Implementation

This section details the architecture of SafeStore, the experimental testbed we have implemented to evaluate the performance trade-offs of the security guarantees offered by the SafeCloud platform. We first describe its components and its implementation, followed the evaluation of long-term entanglement over multiple clouds. A more detailed description of SafeStore as well as its evaluation in a large number of other scenarios can be found in [BPF+16]. The source code of SafeStore itself is publicly available online.

5.1 Architecture

The SafeStore architecture comprises the following components: a storage server ("proxy") that mediates interactions between clients and the SafeStore system, an encoder component, and a set of backend storage clouds (public clouds or private servers deployed on-premises). Figure 5 presents an instance of SafeStore connected to a set of clients and various cloud storage providers.

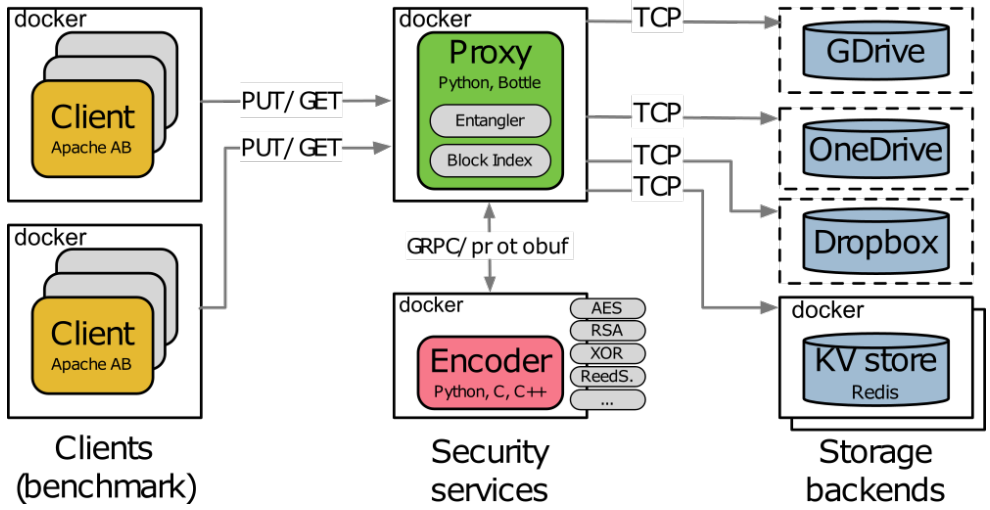


Figure 5: Architecture of our experimental testbed

The proxy component[1] acts as the SafeStore's front-end and is responsible for keeping a mapping between client's files and the actual storage backends where these are stored. Clients, which run in independent nodes, contact the proxy component to write or read data through a simple REST interface that mimics the operating principles of well-established services like Amazon S3. The interactions between the proxy and the clients happen via synchronous HTTP messages over pre-established TCP channels.

The SafeStore system is configurable and different security mechanisms can be put in place. According to such configuration, the proxy component coordinates the other components in the system and different workflows may arise. For instance, some configurations require a single cloud backend while others require two or more. Upon a write request, the proxy component asks the encoder component to encode data blocks according to the configured security mechanisms. The resulting block or blocks are then dispatched, by the proxy, to the storage backends. To this end, the proxy maintains a data block index to keep track of where data is stored at the backends. Additionally, and for the case where anti-censorship mechanisms are in place, the proxy also maintains an entangler component. This component requires access to the block index component

[1] Note that, in practice, one should deploy multiple such proxies for scalability and dependability reasons, or even co-locate them with the clients.

and is placed within the proxy to leverage locality. More details on the entangler component are presented in Section 5.3.

Upon a read request for a piece of data, the proxy checks the block index to figure out where the corresponding encoded blocks are stored. It fetches them from the backend storage and forwards them to the encoder that decodes the blocks before returning the data to the client. The encoder is co-localized within the same host as the proxy to maximize throughput and avoid bottlenecks induced by high pressure on the network stack. To increase the flexibility of our testbed, our encoder provides a plugin mechanism to dynamically load and swap different coding and cryptographic libraries and associated bindings. This mechanism relies on a platform-independent transport mechanism (using protocol buffers) and a stable interface between the proxy and the encoder.

5.2 The SafeStore API

Once started, a SafeStore instance can be interacted with using an HTTP client such as cURL [curl] by sending PUT and GET requests. The PUT command is used to insert documents into SafeStore while the GET command is used to retrieve those documents from the system.

A user willing to store a document can issue the following command to store a new document named `report.pdf` into the system.

```
curl -X PUT http://example.local/ -T report.pdf
```

Retrieving the newly inserted document can be done by running the following command.

```
curl -X GET http://example.local/report.pdf -o copy.pdf
```

These commands should run transparently for the user regardless of the internal configuration chosen by the administrator.

5.3 The SafeStore Entangler

In order to illustrate the modularity of our architecture, we implemented a simple exclusive-or-based entanglement approach to provide anti-censorship. The technique we implemented is similar to Dagster [SW01]. In Dagster, the size of documents and blocks is identical. When a new document D must be stored, Dagster randomly chooses c blocks already archived and XOR them with D. The resulting block is then stored. Dagster is analysed in [AFYZ07]: an attacker who wants to censor a document must erase one of its $c + 1$ blocks, and this will destroy on average $O(c)$ other documents in the system. Older documents are more protected than newer ones. Dagster thus provides a low level of Strong CR, low in the sense that the average amount of collateral damage is in $O(c)$. We use $c = 5$ in our implementation.

5.4 Implementation Details

Our implementation choices have been largely driven by performance and programming simplicity considerations, as well as by constraints from the storage backends interfaces.

The proxy component is implemented in Python (v2.7.10) and exploits the exporting facilities of the Bottle [bot] framework (v0.12.9). The proxy handles POST and GET requests via the WSGI [wsgi] Web framework.

The encoder, also written in Python, integrates with various encoding libraries. Each library is wrapped exposing the same API to the encoder allowing the system to be

expanded and to abstract SafeStore from the implementation details of each library. This allows SafeStore to support not only Python libraries but also native ones.

We leverage Cryptography [cryp], a python library that exposes a wide range of cryptographic primitives with an easy to use and well documented API. Namely, this library provides the AES and RSA cyphers by wrapping OpenSSL's cryptographic protocol implementations [ssl].

We use our own implementation for the one-time pad XOR encoding driver that resorts to the numpy [numpy] library to optimize vector computation. As the erasure coding driver, SafeStore supports Jerasure, an efficient Cauchy Reed-Solomon driver implemented in C/C++ that is exported by the PyEClib [pye] library (v1.2).

For the client side, we built a suite of micro- and macro-benchmarks, leveraging Apache Bench [apa] (v2.3), to measure the throughput and latency of client storage requests. The CPU and memory measurements presented in the evaluation are gathered with the dstat tool [dstat].

Finally, we have implemented drivers for four storage backends. First, we deployed a set of on-premises storage nodes using Redis [redis] (v3.0.7), a lightweight yet efficient in-memory key-value store. Redis tools provide easy-to-use probing mechanisms (e.g., the redis-cli command-line tool), which allowed us to measure the impact of the several security combinations used in our evaluation. Second, we have implemented drivers for the three most widely used cloud storage services: Dropbox [dbox], Google Drive [gdrive], and Microsoft OneDrive [odrive]. The drivers are implemented leveraging the official Python SDKs from each provider. Similarly to the approach taken with the encoding component, storage backends are wrapped to expose a common interface with the required set of operations, i.e., store, fetch and delete data, which allows to easily plug-in new storage backends in the future. Overall, our implementation consists of 2,723 lines of Python code, all components included.

5.5 Evaluation

This section presents our evaluation study of the different security guarantees.

4.4.1 Evaluation Settings

We deploy our experiments over a cluster of machines interconnected by a 1 Gb/s switched network. Each physical host features 8-Core Xeon CPUs and 8 GB of RAM. We deploy virtual machines (VM) on top of the hosts. The KVM hyper- visor, which controls the execution of the VM, is configured to expose the physical CPU to the guest VM and Docker containers by mean of the host-passthrough [kvm] option, to allow the encoders to exploit special CPU instructions. The VMs leverage the virtio module for better I/O performances.

We deploy Docker (v0.10) containers on each VM (1 container per VM) without any memory restriction to minimize interference due to co-location and maximize performance. In particular, the proxy, the encoder and the Redis storage nodes reside in isolated containers, each of them running in VMs executed by separated hosts. Similarly, the client that injects requests into the testbed runs in a Docker container running in a separate host. We use regular accounts for the selected cloud providers (Dropbox, GDrive, and OneDrive).

4.4.2 Macro-benchmark – Multi-cloud entanglement

We evaluate the overhead of the entanglement by configuring our testbed to use three distinct public cloud backends at the same time, namely Google Drive, Dropbox, and OneDrive. We choose the driver combinations that provide the higher degree of security in a multi-cloud deployment, in particular cauchy_rsa_sha_512 and xor_rsa_sha_512. Both combinations provide encryption, integrity checking, non-repudiation using. The only difference between cauchy_rsa_sha_512 and xor_rsa_sha_512 is that one uses Reed-Solomon codes while the over performs an exclusive-or operation. We compare the latency of inserting 250 blocks of 1 MB with and without entanglement for both drivers. Once blocks are entangled, the proxy dispatches them to the chosen provider in a round-robin fashion to spread the load among them. We present the cumulative distribution function (CDF) of the results for cauchy_rsa_sha_512 and xor_rsa_sha_512 in Figure 6 and Figure 7, respectively.

Our observations are twofold. First, the exclusive-or based driver xor_rsa_sha_512 is considerably faster than the erasure- coding driver cauchy_rsa_sha_512. For example, the 50th percentile of the former is below 4 s whereas the latter is at 14.7s. This results from the fact that the exclusive-or is a very computationally efficient operation. Second, the overhead induced by the entanglement phase is modest. In particular, in the case of cauchy_rsa_sha_512, the entanglement only adds a +18.1% latency overhead for the 95th percentile of the blocks. In the xor_rsa_sha_512 scenario, this overhead is lowered to +0.3%. These results prove that a multi-cloud entanglement scheme can be practically operated by clients with very low performance gaps when compared to the default, non-entangled operational mode.

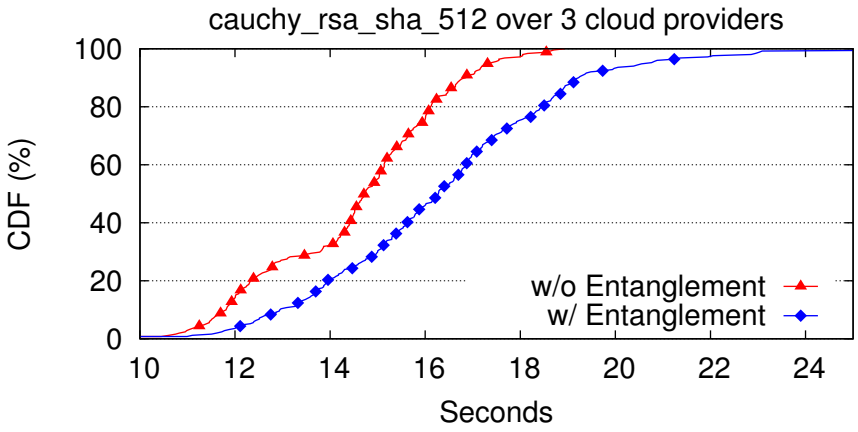


Figure 6: Macro-benchmark: latency distribution (CDF) for the cauchy_rsa_sha_512 driver with and without entanglement over 3 cloud providers.

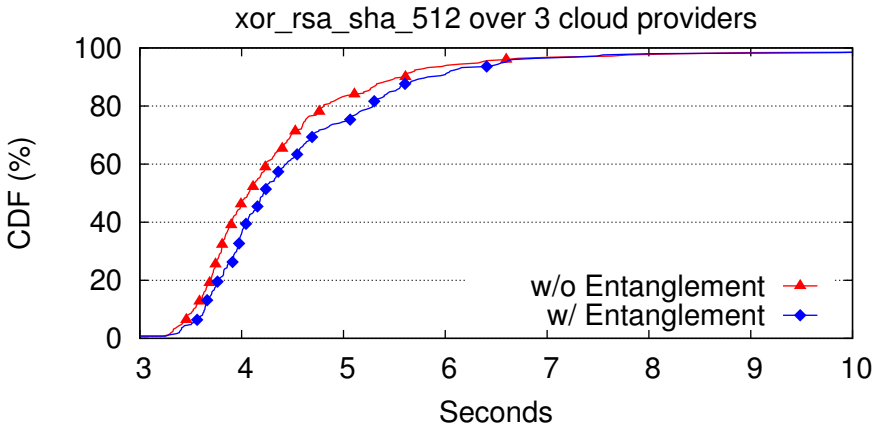


Figure 7: Macro-benchmark: latency distribution (CDF) for the xor_rsa_sha_512 driver with and without entanglement over 3 cloud providers.

6 Conclusion

This deliverable has addressed the design of the first two storage solutions of the SafeCloud project: the secure block storage (SS1) and the secure data archive (SS2). We have specifically addressed the theoretical background upon which long-term data entanglement is achieved, in order to achieve strong anti-tampering and data integrity. Furthermore, we have presented the first prototype implementation of the data store (SafeStore), which allows clients to store and retrieve documents via a REST-based key/value API. It supports various data coding libraries as well as a simple entangler component. SafeStore will be extended toward supporting the other features of the SafeCloud storage solutions: the secure data archive (SS2) and the secure file system (SS3).

7 References

[AFYZ07] J. Aspnes, J. Feigenbaum, A. Yampolskiy, and S. Zhong, Towards a theory of data entanglement, Theoretical Computer Science, vol. 389, no. 1–2, Dec. 2007.

[apa] https://httpd.apache.org/docs/2.4/programs/ab.html.

[bot] Bottle, http://www.bottlepy.org/.

[BPF+16] Dorian Burihabwa, Rogerio Pontes, Pascal Felber, Francisco Maia, Hugues Mercier, Rui Oliveira, Joao Paulo and Valerio Schiavoni, On the Cost of Safe Storage for Public Clouds: an Experimental Evaluation, 35th Symposium on Reliable Distributed Systems (SRDS), 2016.

[cryp] Cryptography, https://cryptography.io.

[dbox] Dropbox, https://www.dropbox.com.

[dstat] DStat, http://linux.die.net/man/1/dstat.

[gdrive] GDrive, https://www.google.com/drive.

[kvm] http://www.linux-kvm.org/page/Tuning_KVM.

[LC04] Shu Lin and Daniel J. Costello. Error Control Coding. Paerson Prentice Hall, second edition, 2004.

[MAL16] Hugues Mercier, Maxime Augier and Arjen K. Lenstra, STEP-archival: Storage Integrity and Tamper Resistance using Data Entanglement, Submitted to the IEEE Transactions on Information Theory, 2016. Available upon request.

[numpy] NumPy, https://numpy.org.

[odrive] OneDrive, https://onedrive.live.com.

[pye] PyECLib, https://pypi.python.org/pypi/PyECLib.

[redis] Redis, http://redis.io.

[SSL] OpenSSL, https://openssl.org.

[SW01] A. Stubblefield and D. S. Wallach, Dagster: Censorship resistant publishing without replication, Rice University, Technical Report TR01-380, July 2001.

[wsgi] WSGI, http://www.wsgi.org.