2016 IEEE 35th Symposium on Reliable Distributed Systems

# On the Cost of Safe Storage for Public Clouds: an Experimental Evaluation

(Practical Experience Report)

Dorian Burihabwa†, Rogério Pontes*, Pascal Felber†, Francisco Maia*,
Hugues Mercier†, Rui Oliveira*, João Paulo*, Valerio Schiavoni†
*HASLab - High-Assurance Software Lab, INESC TEC & U. Minho, Portugal.
†University of Neuchâtel, Switzerland.

***Abstract*—Cloud-based storage services such as Dropbox, Google Drive and OneDrive are increasingly popular for storing enterprise data, and they have already become the de facto choice for cloud-based backup of hundreds of millions of regular users. Drawn by the wide range of services they provide, no upfront costs and 24/7 availability across all personal devices, customers are well-aware of the benefits that these solutions can bring. However, most users tend to forget—or worse ignore—some of the main drawbacks of such cloud-based services, namely in terms of privacy. Data entrusted to these providers can be leaked by hackers, disclosed upon request from a governmental agency's subpoena, or even accessed directly by the storage providers (e.g., for commercial benefits). While there exist solutions to prevent or alleviate these problems, they typically require direct intervention from the clients, like encrypting their data before storing it, and reduce the benefits provided such as easily sharing data between users.**

**This *practical experience report* studies a wide range of security mechanisms that can be used atop standard cloud-based storage services. We present the details of our evaluation testbed and discuss the design choices that have driven its implementation. We evaluate several state-of-the-art techniques with varying security guarantees responding to user-assigned security and privacy criteria. Our results reveal the various trade-offs of the different techniques by means of representative workloads on top of industry-grade storage services.**

## I. Introduction

Public online cloud-based storage services such as Dropbox, Google Drive or Microsoft OneDrive are nowadays the de facto standard for users to store their photos, music and other types of documents online. The extremely low economic barrier of these services (which typically offer free basic accounts), their ubiquitous availability, as well as their ease of use with transparent client integration contribute to making them an attractive solution for both individuals and organizations [1].

Cloud-based storage services are also largely exploited by application developers. They typically expose cross-platform REST-based APIs that can be seamlessly plugged into existing systems. Developers therefore use these services to add online storage backends to their applications without having to face the costs and burdens of managing their own storage infrastructure. Most online applications developed nowadays follow this architectural pattern (e.g., online word processors, mobile applications, etc.).

Nevertheless, as soon as the data enters the cloud provider's service perimeter, the client essentially surrenders control over it [2], which is highly undesirable. In fact, the control over personal data is among the major concerns of individuals and organizations. A recent report [3] carried with European citizens shows that 67% of the population is concerned by the information they disclosed online (voluntarily or not), and only 15% think to be in control of their own data. As a consequence, concerns over the disclosure of private information by malicious insiders [4] and data breaches [5] have motivated a new class of secure and safe cloud-based storage applications and services. This trend is further amplified by the lack of security expertise from software developers [6].

To protect the privacy of the users and their data, researchers proposed several systems [7, 8, 6] that encrypt data at the client side before sending it to the cloud providers. These systems offer various security guarantees to the end-users (e.g., integrity, authorization, privacy) and typically follow two different deployment strategies: single- or multi-cloud modes. The former stores data on a single storage provider, while the latter spreads it across multiple providers, possibly operating under distinct (non-colluding) administrative domains. Partitioning data across multiple storage providers ensures that, even if one of them is compromised, the attacker cannot access the complete original information. In fact, depending on the multi-cloud partitioning algorithm, it is possible to guarantee that no information from the original data is leaked as long as one of the storage providers remains secure [9].

Current systems suffer from a major drawback: they either provide very specific yet incomplete security mechanisms (e.g., some only provide data integrity, others provide only data privacy [10]), or they integrate general-purpose security measures that cannot be tailored for a given application (e.g., some systems bundle confidentiality, integrity, and access-control in a single package [7]). Neither approach allows further customization based on the user's security requirements, e.g. choosing among multiple security features with different guarantees concerning data confidentiality, anti-censorship and fault tolerance.

We strongly believe that it is essential to understand the impact of each security measure adopted by cloud storage systems in terms of resource consumption (e.g., computing

1060-9857/16 $31.00 © 2016 IEEE
DOI 10.1109/SRDS.2016.26

power, storage space, network throughput), economic impact, and overall performance (latency, ease of use, services offered). For example, increasing the size of an asymmetric encryption key to provide stronger security has non-negligible impact on the system's energy requirements, a crucial metric in today's mobile application market. Strong security measures can also render services unacceptably slow, and even disable them. The ability to take an informed decision on these design compromises is of paramount importance for the deployment of storage systems on public clouds [11].

In this context, our contributions are threefold. First, we define a set of basic security guarantees that can be combined and implemented by a client to securely store content in the cloud. Second, we design and implement a modular software architecture that can operate in single-cloud or multi-cloud mode, interfacing with well-known public storage clouds as well as on-premise private data stores. Third, we evaluate the different considered security features using a set of well-defined workloads. Our evaluation unveils the costs of each feature in terms of resource usages, storage space, latency, and associated financial cost.

The rest of the paper is organized as follows. Section II overviews related work on secure cloud-based storage systems. Section III defines the security guarantees considered in the paper in order to allow clients to understand and combine them according to specific needs. Section IV describes the deployment scenarios and trust models that our system supports. Section V discusses the design, architecture, implementation choices of our system. Section VI present extensive evaluation results, and Section VII concludes.

## II. Related Work

Several solutions were proposed during the last decade to address the challenges of secure cloud storage [12, 13, 7]. This section focuses on systems that fit into two main categories. We start by discussing approaches that rely on a single storage provider. Then, we discuss federated storage systems that split data across multiple providers. Table I presents a summary survey of these two types of systems.

### A. Secure Single-cloud

SUNDR (Secure Untrusted Data Repository) [10] proposes an architecture that leverages asymmetric encryption and cryptographic hash function to ensure the integrity and consistency of stored data (blocks). In particular it uses SHA-1 digests to index each data block and a protocol based on ESIGN [14] to detect unauthorized attempts of file modifications.

Depot [15] offers stronger liveness guarantees under node failure than SUNDR, however it lacks native support for data confidentiality or privacy. Data is stored in plain-text along with SHA-256 message digests to enable data integrity checking. Also, data is cryptographically signed using RSA with 1024-bit keys to ensure data authenticity. Both signatures and digests are verified upon each read request.

CloudProof further adds an encryption step: data blocks are protected with AES [13]. Once encrypted, blocks are signed with RSA to prevent unauthorized users to tamper with the data. Also, data integrity is assured by means of SHA-1 digests.

| System | Conf. | Self-integr. | Origin auth. | Anti-censor. | Sym. encr. | Asym. encr. | Hash func. |
|---|---|---|---|---|---|---|---|
| CloudProof [13] | √ | √ | √ | × | AES | RSA | SHA-1 |
| Kamara *et al.* [16] | √ | √ | √ | × | AES | × | × |
| Depot [15] | × | √ | √ | × | × | RSA | SHA-256 |
| SUNDR [10] | × | √ | √ | × | × | ESIGN | SHA-1 |
| BlueSky [17] | √ | √ | × | × | AES | × | SHA-256 |
| Hail [18] | × | √ | × | × | × | × | Univ. Hash |
| MetaSync [19] | √ | × | √ | × | AES | × | MD5 |
| DepSky [7] | √ | √ | √ | × | AES | RSA | SHA-1 |
| UniDrive [8] | √ | √ | √ | × | DES | RSA | SHA-1 |
| SafeSky[6] | √ | √ | × | × | CCM | × | × |

TABLE I: Security features offered by secure cloud solutions for single- (top) and multi-cloud (bottom) solutions: confidentiality, self-integrity, origin authentication, anti-censorshop, symmetric encryption, asymmetric encryption, hash functions.

Kamara *et al.* [16] discuss a high-level architecture for a storage service that can be implemented with different cryptographic primitives, thus offering different security features. Moreover, data privacy is ensured by using symmetric/asymmetric ciphers.

BlueSky [17] tackles the privacy and integrity problem of enterprises with a proxy server that handles the communication between the client and cloud provider. This proxy is installed in the enterprise network so clients do not require any modification. The proxy encrypts the clients data and checks the integrity of the files retrieved from the cloud provider.

The key differentiating factor of our contribution is that it describes a modular architecture that operates over both single- or multi-provider storage schemes. Furthermore, this modular architecture opens the possibility to offer novel security primitives, such as secret sharing and entanglement, that allow users to have a broader set of security measures for distinct requirements.

### B. Secure Multi-cloud

The previous solutions store all information in a single storage service. By doing so, users are locked to a specific storage service that also represent a single point of failure, both for data availability and security breaches. We can mitigate these drawbacks by resorting to multiple cloud providers.

In MetaSync, users' files are replicated to tolerate data loss and the unavailability of storage providers [19]. An additional plug-in can also be used to conceal data using AES encryption.

Hail [18] adopts a secure multi-cloud approach. While it does not provide native support for data confidentiality, it handles data integrity and recoverability from node failures by using a single trusted verifier. This verifier can be a client or a proxy that performs a periodic check of the files integrity on the providers and reconstructs corrupted blocks.

In DepSky [7] and Unidrive [8], data is balanced and replicated across multiple providers with MDS erasure coding [20]. A data object is split into $k$ blocks and coded to generate $n$ coded blocks which are then spread across the cloud providers. The user can reconstruct the original data object from any

subset of $k$ out of $n$ the blocks. This decreases the storage overhead compared to replication approaches like the one used in MetaSync, while achieving the same reliability level. Unidrive uses non-systematic Reed-Solomon codes so that data is not directly stored online. DepSky goes further by encrypting the data before encoding it. The secret key is then divided using secret sharing [21], and each server receives one code block and one share of the key. This insures that any malicious entity gaining access to less than $k$ blocks obtains no information whatsoever about the original data object. Unidrive encrypts the metadata with DES and replicates it on all clouds, whereas DepSky signs metadata files using AES.

SafeSky [6] provides a middleware layer at the operating system level that intercepts file system calls and redirects storage requests to cloud providers. The data object is first encrypted. The encrypted object, the secret key, and the cipher type are then divided using secret sharing.

Some systems offer censorship-resistant storage by creating dependencies, or entanglement, across stored data [22]. Such entanglement makes it difficult for unauthorized parties to censor or tamper with data, but usually require modifying the implementation of storage backends. In this paper we implement, deploy and evaluate entanglement techniques atop unmodified third-party public cloud storage providers.

In summary, the previous solutions leverage the storage capabilities of multiple storage providers to increase the availability and reliability of stored data. Each solution uses a set of known and well-established techniques to ensure privacy, integrity and anti-censorship. However, the system impact of choosing these specific techniques over other options is largely ignored. This means that users do not have the means or information to use this type of solutions and to choose the most appropriate techniques for the desired performance and security guarantees. This paper focus precisely on this challenge. To the best of our knowledge, this is the first attempt to evaluate experimentally the performance impacts of multiple security features in the context of a storage service running across multiple public clouds.

## III. Storage Security for End Users

The term *security* encompasses a rich set of different concepts, and the definition of security itself usually vary according to the context. In this section we clarify the security guarantees that a cloud storage system can offer. We consider four fundamental security guarantees: confidentiality, self-integrity, origin authentication, and anti-censorship.

**Confidentiality.** Confidentiality is a fundamental guarantee offered by storage systems. It ensures that stored data cannot be disclosed to third-party entities without the permission of its rightful owner. This guarantee is achieved by resorting to encryption schemes, as further discussed in Section IV. The security of cryptosystems used for data confidentiality can be formally quantified in different ways, such as information-theoretic and semantic security. Confidentiality is a primordial feature that we systematically include in the different security configurations explored in our evaluation in Section VI.

**Self-Integrity.** Self-integrity protects data against unauthorized or unintended undetectable data modifications caused by attackers or by data corruption. Self-integrity is typically achieved with secure hash functions: when fetching a requested stored block $B$, the system must either return the same block $B$ or an error; it cannot return another data block $B' \neq B$. In practice, we can achieve this by having the key computed as a cryptographically secure hash function of $B$, and have the client recompute the hash from the data and check the matching key after every read operation. A malicious server must be able to break the hash function in order to break this self-integrity. Self-integrity requires that metadata cannot be tampered. In practice, metadata is small enough to be massively replicated or kept at the client-side.

**Origin authentication.** Data origin authentication is a particular instance of message authentication that allows a storage provider receiving data to assess and verify the rightful owner of the data. Like message authentication in general, data origin authentication can be implemented using digital certificates like signatures and message authenticated codes. The server can verify that certified data comes from a party in possession of the corresponding private key.

**Anti-censorship and tamper resistance.** A censorship resistant system makes it difficult to selectively refuse to answer requests without denying service for other unrelated requests. The following three levels of censorship resistance (CR) were defined in [22]:

- ***Perfect CR***: Either all the read requests can be fulfilled, or none can.
- ***Strong CR***: If the system is unable to fulfill a read request, then a set of different read requests cannot be fulfilled (*collateral damage*).
- ***Weak (resource driven) CR***: The system must spend a large amount of hardware resources to censor a read request.

Anti-censorship is closely related to tamper resistance. Tamper-resistant data is usually achieved using "write-once, ready-many" (WORM) technology. Designing software approaches for anti-tampering is an active research area, and there is no practically implementable solution currently available.

## IV. Deployment Scenarios

We consider two deployment scenarios. In the first, users' data is stored in a single cloud storage provider, while in the second data is stored across multiple storage providers. These two distinct environments imply different trust models and security mechanisms. In this section, we revise their characteristics and discuss the security techniques that best fit each of them for obtaining different security guarantees. These deployment scenarios and mechanisms are then evaluated in Section VI.

### A. Single-cloud Deployment

The single-cloud scenario is representative of the way cloud storage services are typically used. Namely, users interact with a single cloud provider to store and fetch their data.

**Trust model.** In the single-cloud deployment scenario, an adversary has access to users' data as soon as it has access to the provider's premises (e.g., it has hands-on access to the hardware hosting the persistent storage devices). The adversary can be internal to the cloud provider (e.g., the cloud provider itself for commercial benefits) or an external unauthorized user (e.g., hacker). The adversary can arbitrarily manipulate users' data. We consider the client to be trusted. The client-to-cloud network communications are protected and assured by secure protocols (e.g., HTTPS).

**Security mechanisms.** Security mechanisms must be applied at the client-side, as the cloud storage is untrusted. This entails encrypting data before uploading it to the cloud. In the single-cloud scenario, we only consider computationally-hard encryption techniques [9], such as symmetric and asymmetric encryption. Symmetric encryption guarantees data confidentiality. We resort to the AES block cipher with a 128-bit key size in CBC (Cypher Block Chaining) mode, which is a commonly used setup [23]. AES is nowadays considered secure and industrial providers are currently phasing out its predecessor DES [24].

Asymmetric encryption can also be used to achieve data confidentiality. It typically requires extra computational power, when compared with the symmetric approach, especially when dealing with large files [7]. For this reason, in the paper, we just use it to sign the users' data and to verify its authenticity. Multiple signature schemes are available. The most relevant ones are based on RSA or DSA, but we only consider RSA as it is faster on signature verification and typically users perform more verifications than signatures [25, 26].

Finally, cryptographic hash functions can be used to generate data digests and to support integrity guarantees. MD5 [27] and SHA [28] digests are commonly used cryptographic hash functions. In our evaluation, we use SHA-based message digests (in its most secure variant SHA-512), as it used in public software package repositories. SHA is also used with RSA to generate the digital signatures.

**Fault tolerance.** In a single-cloud scenario, as soon as the cloud-storage provider is unreachable, users lose access to their data. Moreover, if the service fails and data is lost permanently, the corresponding users' data cannot be recovered.

### B. Multi-cloud Deployment

In a multi-cloud deployment context, data is stored across several storage providers. This scenario brings several benefits: performance, storage capacity, data availability and security [6, 7, 8]. Under this scenario, we must guarantee that a single corrupted cloud provider does not lead to a full disclosure of the stored data.

**Trust model.** We assume that for $n$ providers, the adversary only has access to the data of $n-1$ storage providers and that it must be possible tolerate up to $n-1$ corrupted providers. Adversaries have the same computational powers as the single-cloud deployment scenario, and the client-to-cloud communications are secure.

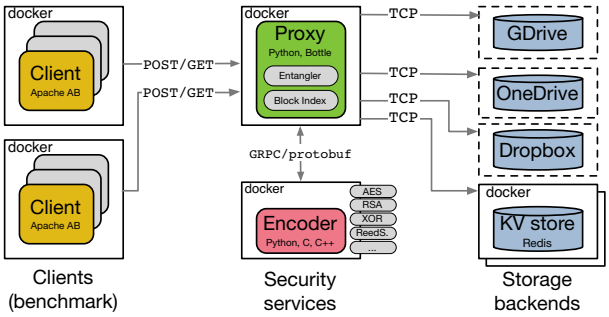


Fig. 1: Architecture of our experimental testbed.

**Security mechanisms.** Confidentiality in a multi-cloud deployment can be guaranteed by exploiting similar encryption mechanisms as the ones used in a single-cloud context. The multiple clouds can be leveraged to improve data availability and system performance. The usual workflow consists in encrypting the data using a symmetric cipher and splitting the encrypted data in several parts using erasure coding techniques. The parts are then dispatched to the data stores, e.g., the public clouds. Erasure coding allows maintaining data availability even when several cloud providers become suddenly unreachable, while consuming significantly less storage space than data replication. Moreover, erasure coding reduces by a fraction the costs of data migration from one cloud provider to another when compared to replication [6].

Our experimental testbed implements a similar workflow. For comparison purposes, we evaluate the performances of a one-time pad XOR [29] as alternative to erasure coding. XOR encryption splits the data in several blocks, but unlike erasure coding each part does not leak any information about the original data. On the other hand, all the parts must be available to successfully decode the original data. This way, in order to have data availability each part must be replicated, thus requiring additional storage space.

As in the single-cloud deployment, SHA-512 and RSA-based signatures are used for providing self-integrity and origin authentication in a multi-cloud scenario.

**Fault tolerance.** When multiple cloud providers are available different failure scenarios can be considered depending on the security measures being used. In particular, when deployed without any replication, the adoption of the one-time pad does not offer any fault-tolerance guarantees, i.e., if one of the providers becomes unavailable, it is impossible to recover the original data. Conversely, it is possible to support multiple storage faults using erasure coding techniques at the cost of increased storage overheads.

## V. The SafeStore System

This section details the architecture of SafeStore, the experimental testbed we have implemented to evaluate the performance trade-offs of the security guarantees discussed in Section III. In detail, we describe its components and the implementation details.

### A. 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). The architecture is depicted in Figure 1.

The proxy component 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.[1] 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 preestablished 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 and is placed within the proxy to leverage locality. More details on the entangler component are presented in Section V-B.

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. Security is ensured if both the proxy and encoder are deployed on a trusted domain. Typically this domain can correspond to the client premises since the computational resources required are expected to be manageable even by handheld devices.

[1]Several proxy instances can co-exist if consistency is guaranteed across the various local mappings. However, we consider this extension out of the scope of the present paper.

### B. 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 [30]. 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 analyzed in [31]: 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.

### C. 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 [32] framework (v0.12.9). The proxy handles POST and GET requests via the WSGI [33] 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 [34], 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 [35].

We use our own implementation for the one-time pad XOR encoding driver that resorts to the numpy [36] 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 [37] library (v1.2).

For the client side, we built a suite of micro- and macro-benchmarks, leveraging Apache Bench [38] (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 [39].

Finally, we have implemented drivers for four storage backends. First, we deployed a set of on-premises storage nodes using Redis [40] (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 [41], Google Drive [42], and Microsoft OneDrive [43]. 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.

## VI. Evaluation

This section presents our evaluation study of the different security guarantees. First we describe the evaluation settings and related contextual information. Then, we organize the reminder in two sets of experiments. In the micro-benchmarks of Section VI-B we evaluate specific architecture components in isolation. The macro-benchmarks stress the complete system as a whole along different axes and workloads: the throughput (Sections VI-C), the resource usages (Section VI-D), the entanglement overhead (Section VI-E), and the storage requirements in Section VI-F.

The experiments discussed next resort to a representative set of encryption techniques, block size configurations and backend storage providers used in previous relevant work, which is discussed in Section II. Additionally, our experiments evaluate the resource consumption of different cryptographic techniques, which is often ignored in previous work.

### A. 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 hypervisor, 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` [44] 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 an Docker container running in a separate host. We use regular accounts for the selected cloud providers (Dropbox, GDrive, and OneDrive).

### B. Micro-benchmark — Throughput

In evaluate the system impact of the security measures considered in our study, we begin by a set of micro-benchmarks, intended to reveal the trade-offs. In detail, we consider 4 different encoder combinations, for distinct security measures. *Conf.* provides data confidentiality (`aes`, `cauchy`, `xor`). *Conf.+Int.* also includes self-integrity (`aes_sh_512`, `cauchy_sha_512`, `xor_sha_512`). *Conf.+Sign.* offers data confidentiality and origin authentication (`aes_rsa`, `cauchy_rsa`, `xor_rsa`). Finally *Conf.+Int.+Sign.* provides data confidentiality, self-integrity, and origin authentication (`aes_rsa_sha_512`, `cauchy_rsa_sha_512`, and `xor_rsa_sha_512`). The properties of each driver configuration, including the security features that each guarantee, are summarized in Table II.

| Deployment | Drivers | Guarantee | | |
|---|---|---|---|---|
| | | Conf. | Int. | Sign. |
| Single-Cloud | aes | √ | × | × |
| | aes_sha_512 | √ | √ | × |
| | aes_rsa | √ | × | √ |
| | aes_rsa_sha_512 | √ | √ | √ |
| Multi-Cloud | cauchy | √ | × | × |
| | cauchy_sha_512 | √ | √ | × |
| | cauchy_rsa | √ | × | √ |
| | cauchy_rsa_sha_512 | √ | √ | √ |
| | xor | √ | × | × |
| | xor_sha_512 | √ | √ | × |
| | xor_rsa | √ | × | √ |
| | xor_rsa_sha_512 | √ | √ | √ |

TABLE II: Deployment targets, drivers and security guarantees.

We encode and decode randomly generated binary blocks of increasing size (4MB, 16MB, and 64MB). Encoding and decoding throughputs are presented in Figure 2 (top) and Figure 2 (bottom) respectively. For each configuration, we present the average results observed for encoding/decoding 50 blocks. As expected, the best encoding throughputs are achieved with symmetric encryption (i.e., `aes` driver with 102.5 MB/s) since it avoids manipulating multiple data blocks. In fact, the encoding mechanism of the `xor` driver requires generating new random data blocks, a time-consuming operation leading to higher overhead. On the other hand, the decoding operation is reduced to the `xor` operation itself, which is very efficient. Consequently, the `xor` is significantly more efficient in decoding, and it consistently achieves the best decoding performance across the full block size range, and up to 195.7 MB/s for the 16 MB case.

As expected, combining several security options impacts negatively over the throughput. The most secure combination (*Conf.+Int.+Sign.*) consistently performs poorly compared to the other combinations, with throughput slowdown in the order of $2\times$ for encoding and $5\times$ for decoding.

### C. Macro-benchmark — Single-cloud Latency

In the reminder of the evaluation, we present an extensive set of macro benchmarks, where the full stack of the system is under test. First, we present the observed latency performance of the system. We configure the testbed to operate in single-cloud mode using Dropbox, OneDrive, and GDrive as cloud backends. We include the same results executed against a local Redis storage backend deployed locally on our cluster. For each scenario, we store 250 blocks of 1 MB. We measure the insertion latency for each block. Figure 3 presents our results. We use a heatmap representation with shades of gray to show at once the observed latencies across the 48 distinct configurations. Each cell of the heatmap shows the average value of the measured latencies for the corresponding configuration.

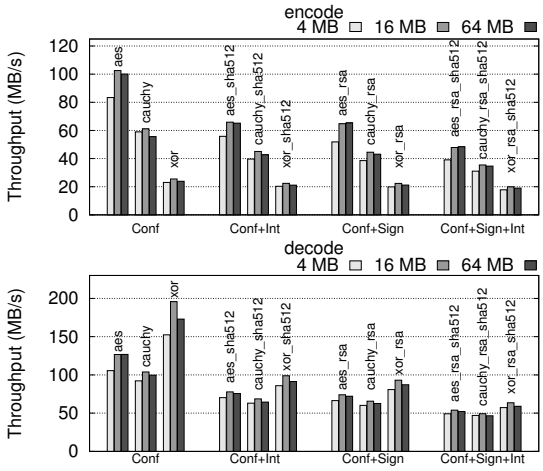


Fig. 2: Micro-benchmark: encoding and decoding throughput.

Three factors influence the reponse time of storage requests:

1) **security measure complexity:** raising the computing time as the number of security measures combined grows
2) **number of blocks generated:** raising the number of connections made to the cloud storage as the number of blocks generated by erasure coding or xoring grows
3) **proxy to cloud storage latency:** raising the overall response time when leaving the local network

Each configuration displayed in the heatmap is a variation on one or more of these factors.

Based on proxy-to-cloud latency, we can split the heatmap in two parts: the first column with Redis in the same cluster and the last three with the remote cloud storage configurations. This distinction highlights or rubs out the noticeable differences of performance between the various security measure combinations. Indeed the impact of computation heavy processing such as `xor` is more significant when running the experiments on the local redis database. In the first column, the response time of any combination using `xor` is systematically longer than the other combinations. But when run over remote cloud storage (the three right-most columns), the impact of the number of blocks to store using erasure coding (`cauchy`) dwarfs the longer computing time in the overall response time. For instance, in the Google Drive scenario, the average latency for the `cauchy_rsa` case is 30.2 s, whereas `aes_rsa` and `xor_rsa` achieve 1.3 s and 4 s respectively. While expected, the overhead of sending a larger number of blocks to components located out the cluster is not the only factor affecting the results. The large variance in response time to each of the providers, in particular Google drive, has previously been discussed [8].

### D. Macro-benchmark — Resource Usage

Next, we evaluate the resource requirements (CPU and live memory) for each of the security configurations. These results intend to unveil the hidden costs that clients need to face when using systems that offer such security guaranteed deployed in environments with constrained resources (embedded devices, smartphones, etc.).

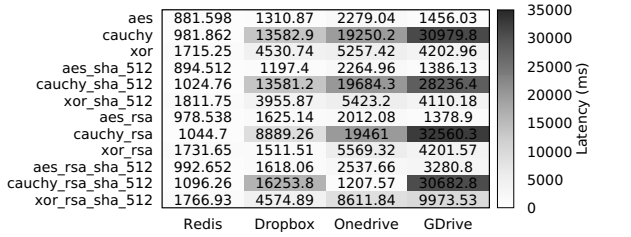


Fig. 3: Macro-benchmark: average latency per block for single-cloud deployments. Each cell's label indicates the average latency for the given pair of driver/cloud provider.

Table III presents a comprehensive survey of our experimentation. We evaluate single- and multi-cloud deployment scenarios, for each of the different combinations of security mechanisms.

For the single-cloud deployment we consider `aes` as the baseline configuration, and report the results for the other drivers as a ratio against it. Similarly, for the multi-cloud deployment, we consider `cauchy` and `xor` as the baseline measurements and present the other results by comparison.

CPU usage does not vary significantly between different approaches for the encoding process. However, different security techniques yield contrasting resource usage in the decoding process. The most CPU-demanding encoder component is the `xor_rsa_sha_512`, with an increase of 31.94% for 4 MB blocks during decoding operations.

We also measured the live memory consumption of each configuration. Once again, the `xor_rsa_sha_512` reveals to be the most memory-demanding configuration with an increase of 16.99% over the steady operational mode.

### E. Macro-benchmark — Multi-cloud Entanglement

We evaluate the overhead of the entanglement (see Section V-B) 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`. 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 4 and Figure 5 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 50*th* percentile of the former is below 4 s whereas the latter is at 14.7 s. 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 95*th* percentile of

| Deployment | Drivers | Encoding | | | | | | Decoding | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | CPU | | | Memory Used | | | CPU | | | Memory Used | | |
| | | 4MB | 16MB | 64MB | 4MB | 16MB | 64MB | 4MB | 16MB | 64MB | 4MB | 16MB | 64MB |
| **Single-Cloud** | aes_sha_512 | 5.93% | 1.62% | 0.81% | 1.74% | 0.22% | -1.35% | 16.54% | 12.51% | 9.01% | 3.86% | 4.11% | 5.85% |
| | aes_rsa | 15.25% | 2.38% | 0.90% | 3.47% | 1.44% | -0.74% | 19.90% | 14.27% | 10.09% | 6.51% | 5.84% | 6.99% |
| | aes_rsa_sha_512 | 11.50% | 3.04% | 1.43% | 5.79% | 2.43% | -1.27% | 24.55% | 19.21% | 13.41% | 6.75% | 9.31% | 8.62% |
| **Multi-Cloud** | cauchy | 9.84% | 10.81% | 10.62% | 532 | 917 | 2413 | 8.17% | 9.57% | 9.01% | 439 | 497 | 699 |
| | cauchy_sha_512 | -1.32% | -1.94% | 1.51% | 0.38% | 0.00% | 0.12% | 12.48% | 9.30% | 9.32% | -0.23% | 0.80% | 1.57% |
| | cauchy_rsa | -0.20% | -1.11% | 1.98% | 4.70% | 1.64% | 0.87% | 14.32% | 10.24% | 10.21% | 1.59% | 3.02% | 2.86% |
| | cauchy_rsa_sha_512 | 1.12% | 0.00% | 2.82% | 3.76% | 1.09% | 0.62% | 19.58% | 14.63% | 14.32% | 2.51% | 4.63% | 4.01% |
| | xor | 10.49% | 11.20% | 10.91% | 521 | 887 | 2322 | 7.20% | 8.12% | 7.85% | 409 | 471 | 705 |
| | xor_sha_512 | -1.05% | 0.09% | 0.55% | 3.54% | 2.03% | 1.94% | 19.17% | 20.81% | 19.36% | 5.13% | 7.43% | 8.37% |
| | xor_rsa | 0.67% | 0.45% | 1.19% | 8.45% | 5.30% | 2.93% | 23.47% | 23.03% | 20.51% | 12.96% | 16.99% | 12.62% |
| | xor_rsa_sha_512 | 0.57% | 0.63% | 1.65% | 5.76% | 4.40% | 3.96% | 31.94% | 30.91% | 28.66% | 9.05% | 12.95% | 13.33% |

TABLE III: Resource consumption of the security mechanisms. We evaluate three distinct configurations: Rows 1-3 for single-cloud, Rows 4-7 for the `cauchy` driver on multi-clouds, and Rows 8-11 for the `xor` driver on multi-clouds. The first row of each configuration on the multi-cloud deployments defines the baseline, and the subsequent rows indicate the overhead over the baseline.

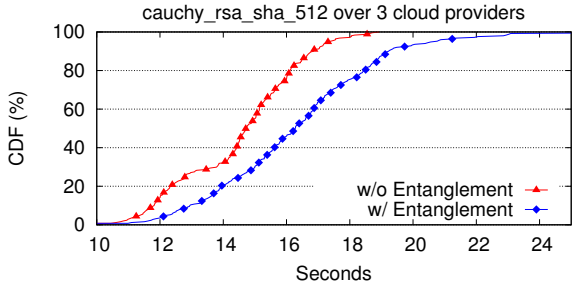


Fig. 4: Macro-benchmark: latency distribution (CDF) for the `cauchy_rsa_sha_512` driver with and without entanglement over 3 cloud providers.

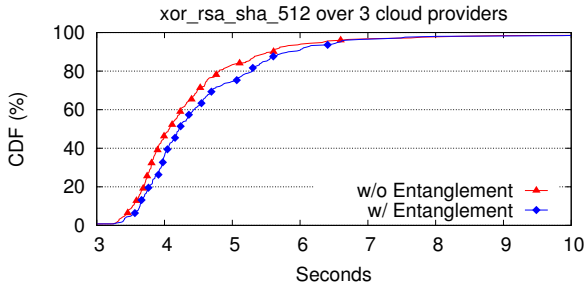


Fig. 5: Macro-benchmark: latency distribution (CDF) for the `xor_rsa_sha_512` driver with and without entanglement over 3 cloud providers.

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.

### *F. Macro-benchmark — Storage*

Finally, we consider storage space requirements. In a multi-cloud scenario, the storage providers need to accommodate more than the source data's original size. The impact of the erasure coding techniques, as well as `xoring` data in terms of storage usage are presented in Table IV. The original block size

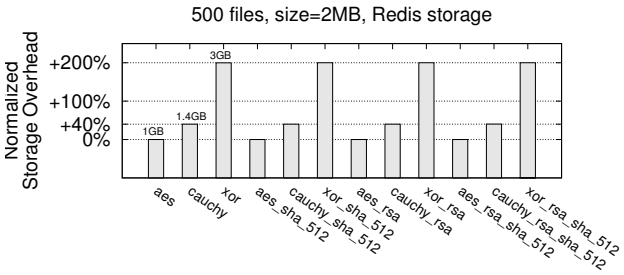


Fig. 6: Macro-benchmark: storage overhead.

| Driver | Growth | 4MB | 16 MB | 64MB |
|---|---|---|---|---|
| Erasure | $(0.10 * b) * 14$ | 6MB | 23MB | 94MB |
| XOR | $b * 3$ | 12MB | 50MB | 201MB |

TABLE IV: Disk space increase.

is $b$. In our implementation, the XOR technique requires $n$ times more space than the original size, with $n$ being the number of storage providers. The erasure-coding drivers will conversely require much less space. For instance, for 64 MB blocks, `cauchy_rsa_sha_512` spends an extra 45.88% of storage space. We confirmed these observations by measuring the effective storage overhead induced by all the drivers when storing 500 files of 2 MB each on a local Redis server. The results are presented in Figure 6.

Note that evaluating the storage usage impact of security techniques is of particular relevance since it typically implies additional costs. As a result, storage space is a key variable to take into account in the decision of which security features to add to a particular system.

### *G. Lessons Learned*

Considering the different evaluation experiments, there are some results that stand out. First, the `aes` encryption driver proved to be the most balanced solution with a virtually constant performance across all benchmarks. As seen in Figure 2, the encoding and decoding throughput is very similar across

benchmarks, only with an average difference of 20MB/s at most. Additionally, it is observable that the measured latency of block encryption with aes is very similar to the latency of uploading a block to a storage provider. This follows from the fact that aes encryption does not increase block size and that the latency of the computational part of the algorithm is negligible.

Secondly, the cauchy driver has a similar discrepancy between the encoding and decoding throughput (30MB/s on average) but, in absolute values, it always exhibits lower throughputs when compared with aes encryption. Moreover, the need to generate 14 blocks has a small impact on the throughout when considering a low-latency deployment such as Redis. Conversely, when using cloud providers, the impact is highly significant. This also has a significant negative impact when using entanglement on the proxy, where in the worst case it has a difference of 8s. We note however that the number of generated blocks can be configured and cauchy is the only driver capable of tolerating the failure of a cloud provider.

Thirdly, despite the fact that the xor driver has the biggest difference of throughput in the micro-benchmark, it has better performance than the cauchy driver on a real deployment. This is due to the fact that only three blocks are generated by the driver, which implies less uploaded data and, consequently, less communication latency. This happens with and without entanglement. Notably, while in the worst case the xor driver with entanglement exhibits a latency around 10s, the cauchy driver with entanglement only achieves comparable performance in the best case scenario.

In summary, the aes driver protects the users information with minimal overhead, but stores all the information in a single cloud. The cauchy driver ensures the privacy of the users data while supporting the failure of a single cloud provider, however this has a significant cost in processing and latency. For a middle ground approach, the xor driver protects the users information by dividing the information among multiple cloud providers with a smaller cost on latency but does not support the failure of a cloud. Finally, security measures such as integrity, origin authentication and anti-censorship have a relatively small impact on the latency when considering baseline encryption and provider latencies.

Regarding the latency across cloud providers, if the aes encryption on Redis is considered as a baseline, Dropbox exhibits the lowest latency on average, with an increase of 77%. Google Drive increases the latency by 108 %. Finally, in our experiments Onedrive had the worst latency, with a latency increase of 192%.

## VII. Conclusion

In this practical experience report, we have compared a wide range of security mechanisms that can be used to protect data stored in the cloud. Our experimental study sheds light on the performance and memory overheads incurred with increasing levels of security. Unlike previous studies, we consider the trade-offs of security mechanisms when used in isolation, as well as the security guarantees they provide, so that users can take informed decisions about which ones to use depending on their specific needs.

Our experiments were conducted using a testbed that we built and deployed across several standard cloud-based storage services. Unsurprisingly, we observe noticeable degradation of the throughput of block encoding with increasing layers of security. The impact of security guarantees is mainly visible in terms of CPU usage, which in turn yields increased latency, but we also observe some variations in terms of memory consumption. Furthermore and as expected, throughput is generally higher and more stable in single-cloud deployments.

To sum up, the lessons to take away from our study are that there is no single combination of security measure that performs best for all applications. Instead, users need to carefully chose the minimum set of mechanisms that can match their security requirements. In turn, cloud provides need to provide security measures that can be freely combined instead of proposing a "complete package", as every additional security layer comes with an associated cost. We hope that our experimental results will provide valuable insights to both service providers and their users, and can be instrumental for improving cloud-based storage systems and developing applications that can best leverage them.

## VIII. Acknowledegements

The research leading to these results has received funding from different sources. Rogério Pontes is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) as part of project UID/EEA/50014/2013. The remainder authors received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 653884.

## References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, 2010.

[2] G. Zhao, C. Rong, J. Li, F. Zhang, and Y. Tang, "Trusted data sharing over untrusted cloud storage providers," in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CLOUDCOM)*, 2010.

[3] "European commission data protection." 2015. [Online]. Available: http://ec.europa.eu/public_opinion/archives/ebs/ebs_431_en.pdf

[4] A. J. Duncan, S. Creese, and M. Goldsmith, "Insider attacks in cloud computing," in *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, 2012.

[5] M. T. Sandikkaya and A. E. Harmanci, "Security problems of platform-as-a-service (paas) clouds and practical solutions to the problems," in *IEEE 31st Symposium on Reliable Distributed Systems (SRDS)*, 2012.

[6] R. Zhao, C. Yue, B. Tak, and C. Tang, "SafeSky: A Secure Cloud Storage Middleware for End-User Applications," in *IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, 2015.

[7] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: Dependable and secure storage in a cloud-of-clouds," in *Proceedings of the Sixth Conference on Computer Systems (EuroSys)*, 2011.

[8] H. Tang, F. Liu, G. Shen, Y. Jin, and C. Guo, "UniDrive: Synergize Multiple Consumer Cloud Storage Services," in *Proceedings of the 16th Annual Middleware Conference (Middleware)*, 2015.

[9] R. Denning and D. Elizabeth, *Cryptography and Data Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1982.

[10] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure Untrusted Data Repository (SUNDR)," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI)*, 2004.

[11] W. Vogels, "Beyond server consolidation," *ACM Queue*, vol. 6, pp. 20–26, 2008.

[12] L. M. Kaufman, "Data security in the world of cloud computing," *IEEE Security & Privacy*, vol. 7, no. 4, pp. 61–64, 2009.

[13] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, "Enabling Security in Cloud Storage SLAs with CloudProof," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2011.

[14] T. Okamoto and J. Stern, "Almost uniform density of power residues and the provable security of ESIGN," in *Advances in Cryptology (ASIACRYPT)*. Springer, 2003.

[15] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *ACM Transactions on Computer Systems (TOCS)*, 2011.

[16] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Proceedings of the 14th International Conference on Financial Cryptograpy and Data Security*. Springer-Verlag, 2010.

[17] M. Vrable, S. Savage, and G. M. Voelker, "BlueSky: A cloud-backed file system for the enterprise," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[18] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: A High-availability and Integrity Layer for Cloud Storage," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[19] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall, "Metasync: File synchronization across multiple untrusted storage services," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[20] S. Lin and D. J. Costello, *Error Control Coding*, 2nd ed. Paerson Prentice Hall, 2004.

[21] A. Shamir, "How to share a secret," *Communincations of ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[22] H. Mercier, M. Augier, and A. K. Lenstra, "STeP-archival: Storage integrity and anti-tampering using data entanglement," in *Proceedings of the 2015 International Symposium on Information Theory*, Hong Kong, 14-19 2015, pp. 1590–1594, extended version submitted to the IEEE Transactions on Information Theory.

[23] S. Frankel, R. Glenn, and S. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec," RFC Editor, RFC 3602, September 2003.

[24] C. Sanchez-Avila and R. Sanchez-Reillol, "The Rijndael block cipher (AES proposal) : a comparison with DES," in *IEEE 35th International Carnahan Conference on Security Technology*, 2001.

[25] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Advances in cryptology*. Springer, 1984, pp. 10–18.

[26] C. K. Wong and S. S. Lam, "Digital signatures for flows and multicasts," *IEEE/ACM Transactions on Networking*, vol. 7, pp. 502–513, Aug 1999.

[27] R. Rivest, "The MD5 Message-Digest Algorithm," RFC Editor, RFC 1321, April 1992.

[28] D. Eastlake and P. Jones, "US secure hash algorithm 1 (SHA1)," RFC Editor, RFC 3174, September 2001.

[29] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

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

[31] 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.

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

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

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

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

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

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

[38] ApacheBench, https://httpd.apache.org/docs/2.4/programs/ab.html.

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

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

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

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

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

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