SafeFS: A Modular Architecture for Secure User-Space File Systems

(One FUSE to rule them all)

Rogério Pontes[**], Dorian Burihabwa[*], Francisco Maia[**], João Paulo[**], Valerio Schiavoni[*], Pascal Felber[*], Hugues Mercier[*], and Rui Oliveira[**]

[*]University of Neuchatel, Switzerland
[**]INESC TEC & University of Minho, Portugal

ABSTRACT

The exponential growth of data produced, the ever faster and ubiquitous connectivity, and the collaborative processing tools lead to a clear shift of data stores from local servers to the cloud. This migration occurring across different application domains and types of users—individual or corporate—raises two immediate challenges. First, outsourcing data introduces security risks, hence protection mechanisms must be put in place to provide guarantees such as privacy, confidentiality and integrity. Second, there is no "one-size-fits-all" solution that would provide the right level of safety or performance for all applications and users, and it is therefore necessary to provide mechanisms that can be tailored to the various deployment scenarios.

In this paper, we address both challenges by introducing SafeFS, a modular architecture based on software-defined storage principles featuring stackable building blocks that can be combined to construct a secure distributed file system. SafeFS allows users to specialize their data store to their specific needs by choosing the combination of blocks that provide the best safety and performance tradeoffs. The file system is implemented in user space using FUSE and can access remote data stores. The provided building blocks notably include mechanisms based on encryption, replication, and coding. We implemented SafeFS and performed in-depth evaluation across a range of workloads. Results reveal that while each layer has a cost, one can build safe yet efficient storage architectures. Furthermore, the different combinations of blocks sometimes yield surprising tradeoffs.

CCS Concepts

•**Information systems** → **Hierarchical storage management;** Cloud based storage; •**Hardware** → *External storage;* •**Security and privacy** → Management and querying of encrypted data;

SYSTOR '17 May 22–24, 2017, Haifa, Israel

DOI: http://dx.doi.org/10.1145/3078468.3078480

General Terms

Design, Security, Standardization, Experimentation

Keywords

Software Defined Storage, Privacy at Rest, Data Confidentiality, FUSE

1. INTRODUCTION

We are living in the era of digital information, where uninterrupted and almost instantaneous access to worldwide data is expected. Generated data is growing at an unprecedented rate and latest projections predict a total of 44 *zettabytes* by 2020, a 10× increase from 2013 [44].

This overwhelming amount of data is quickly migrating from private computers and servers to third-party cloud providers (*e.g.*, Dropbox or Google Drive) to leverage large and scalable infrastructures. This paradigm shift raises two important challenges. First, client applications need to handle heterogeneous and incompatible interfaces. Second, storing data on remote sites implies losing control over it, a serious matter for applications that deal with sensitive data [41].

The first problem can be partially solved by providing well-known and extensively used abstractions on top of third-party interfaces. One of the most widespread and highest level abstractions offered atop storage systems is the file system. The practicality and ease of use of such abstraction has spurred the development of a plethora of different file system solutions offering a variety of compromises between I/O performance optimization, availability, consistency, resource consumption, security, and privacy guarantees for stored data [45, 37, 50, 25]. Additionally, developers can leverage the FUSE [9] (filesystem in user-space) framework to implement a POSIX-like [48] file system on top of a multitude of local and remote storage systems in a fairly straightforward manner. Nevertheless, each file system implementation is different and specifically designed for certain use cases or workloads. Choosing which implementations to use and combining them in order to take advantage of their respective strong features is far from trivial [54].

The second problem, securely outsourcing sensitive data, has also been the subject of intensive work. There are several file system implementations providing privacy preserving mechanisms [50, 30, 7, 13]. However, similarly to storage systems, a single approach does not address the specific privacy needs of every application or system. Some require higher levels of data privacy while others target performance

at the price of lower privacy guarantees. Furthermore, these approaches lack a clear separation between privacy preserving mechanisms and the file system implementation itself. This prevents an easy combination of different privacy preserving mechanisms with other file system properties (*e.g.*, caching, compression, replication, etc).

In this paper we tackle both challenges simultaneously. Inspired by software-defined storage (SDS) design principles [43, 23], we introduce SafeFS, a novel approach to secure user-space stackable file systems. We advance the state of the art in two important ways by providing **two-dimensional modularity** and **privacy à la carte**.

First, the SafeFS two-dimensional modular design allows easy implementations of specialized storage layers for security, replication, coding, compression and deduplication, while at the same time allowing each layer to be individually configurable through plug-in software drivers. SafeFS layers can then be stacked in any desired order to address different application needs. The design of SafeFS avoids usual pitfalls such as the need for global knowledge across layers. For instance, for size-modifying layer implementations (*e.g.*, encryption with padding, compression, deduplication), SafeFS does not require a cross-layer metadata manager to receive, process, or redirect requests across layers [51].

Second, SafeFS design allows us to easily combine any FUSE-based file system implementation with several cryptographic techniques and, at the same time, to leverage both centralized [19, 17] and distributed storage back-ends [25, 14]. For example, it is easy to integrate an existing FUSE-based file system with secret sharing on top of distributed storage back-ends using SafeFS simply by adapting the system APIs. To the best of our knowledge, SafeFS is the first user-space file system to offer this level of flexibility.

To show the practicality and effectiveness of our approach, we implemented a full prototype of SafeFS that, as in recent proposals [24, 14, 10, 46], resorts to the FUSE framework. With thorough experimental evaluation, we compare several unique configurations of SafeFS, each combining different privacy-preserving techniques and cryptographic primitives. We evaluate the performance by resorting to state-of-the-art benchmarks. SafeFS is open-source[1] and available to ease the reproducibility of our evaluation.

The remainder of this paper is organized as follows. We present related work in Section 2. Section 3 illustrates the design goals of SafeFS, while Section 4 details its architecture. The implementation details are given in Section 5. Section 6 presents our extensive evaluation of the SafeFS prototype, before concluding with the road ahead.

2. RELATED WORK

There is a large body of literature and numerous deployed systems to which SafeFS relates to. This section presents a survey of related file systems, both in kernel- and user-space, with a special focus on other file systems with privacy-aware features. We wrap up this section with related work from the software-defined storage domain [27].

Kernel-space solutions. Stackable file systems [31] decouple data processing into multiple layers that follow a common interface. As each layer is independent and isolates a specific data processing stage, these file systems can be extended with additional layers. Kernel-space stackable file systems apply this concept by extending the classic *vnode* interface [34]. This design is used by CryptFS and NCryptfs [52, 50] to enhance file systems with access control, malware detection, and privacy-aware layers. NCryptfs extends the security features presented in Cryptfs, while both are built atop WrapFS [53], which proposed *stackable templates* to ease the development of stackable file systems. Stackable templates have a limited API and are bound to a single platform. These limitations were mitigated by FiST [54], a high-level language to define stackable file systems and a correspondent cross-platform compiler. Nevertheless, even with a high-level language abstraction, implementing layers with complex behavior (*e.g.*, predictive caching, replication) at the kernel level is an extremely challenging task. Furthermore, when any of the layers changes the size of the file being processed (*e.g.*, compression/decompression layers, cryptographic primitives with padding), these systems require a system-wide global index that tracks the offsets and the size of data blocks written at the final filesystem layer. This indexing requires a global metadata structure to keep track of the real block and file size, thus introducing dependencies between layers.

SafeFS relies on the FUSE API for layer stacking, which provides a richer set of operations and avoids the aforementioned problems. Each layer is independent from its adjacent layers while still supporting size-changing processing.

User-space solutions. It is possible to implement user-space file systems by exploiting the FUSE software framework and the corresponding kernel module. User-space solutions require considerably lower implementation efforts. A recent study [42] observed that at least 100 user-space file systems have been published between 2003 and 2015. On the contrary, only about 70 kernel-based systems appeared in the last 25 years. The same report showed that porting file systems functionalities to user-space reduced the time required to develop new solutions and improve their maintainability, reliability, extensibility, and security. An obvious advantage for user-space implementations is to have direct access to efficient user-space software libraries

The extensive micro-workload evaluation in [42] inspired the evaluation of SafeFS, which we present later in Section 6. Those results show that with the recent advances in hardware processing, storage, and RAM capabilities, the throughput of FUSE-based file systems has improved significantly and now offers an acceptable I/O overhead. Several applications are now adopting this approach [24, 14, 10].

The SafeFS design brings a modular architecture for FUSE-based file systems that enables easy and almost transparent combination and enrichment to the plethora of existing systems in a single yet configurable solution.

Privacy-aware file systems. Data encryption is an increasingly desirable counter-measure to protect sensitive information. This becomes crucial when the data is being stored on third-party infrastructures where data is no longer in control of its rightful owners. However, the widespread adoption of these systems is highly dependent on the performance, usability, and transparency of each approach.

As shown in Table 1, several proposals address data encryption to tackle the aforementioned challenges. Cryptfs [52], NCryptfs [50], StegFS [33], PPDD [16], EFS [6], and TCFS [28] reuse the implementations (kernel-code or the NFS kernel-mode client) of existing file systems to integrate a data encryption layer. These systems support dis-

[1] https://github.com/safecloud-project/SafeFS

Type	Name	Stackable Layers			Multi-Backend	
		Single-L	Multi-L	PnP-Privacy	Multi-B	PnP-B
Kernel	PPDD [16]	×	×	×	×	×
	StegFS [33], EFS [6], TCFS [28], BestCrypt [1], eCryptfs [30]	×	×	√	×	×
	Cryptfs [52]	√	×	×	×	×
	NCryptfs [50]	√	×	√	×	×
	dm-crypt [3]	√	×	√	√	×
User	CFS [26], LessFS [10], MetFS [13], S3QL [18], Bluesky [47]	×	×	×	×	×
	SCFS [25]	×	×	×	√	×
	EncFS [7], CryFS [2]	×	×	√	×	×
	SafeFS	√	√	√	√	√

Table 1: Survey of related file systems. We categorize them by type (*Kernel*=kernel-space, *User*=user-space), if they are stackable and if the stacking architecture contemplates a single-layer (Single-L) or multi-layer (Multi-L) hierarchy, if they provide a plug & play design for easily switching between different privacy-preserving libraries (PnP-Privacy), if they can leverage multiple storage backends (Multi-B), and if the multiple backend algorithms (data replication, load-balancing, etc) are pluggable (PnP-B).

tinct symmetric key cryptography algorithms such as Blowfish [36], AES [32], and DES [38].

Similarly, dm-crypt [3] is implemented as a device-mapper layer and uses Linux Crypto API [12] to support different symmetric cyphers. BestCrypt [1] provides a kernel solution to export encrypted volumes. It supports several cypher algorithms such as AES, CAST [21], and Triple DES. The kernel-space eCryptfs [5] further extends BestCrypt with the support of additional cyphers (*e.g.*, Twofish).

Another set of solutions is focused on user-space implementations. CFS [26] provides a user-level NFS server modification that uses DES+OFB to protect sensitive information. EncFS [7], lessfs [10], MetFS [13] and CryFS [2] provide FUSE-based file systems that cypher data with techniques ranging AES, Blowfish, RC4, and AES-GCM.

Proposals such as S3QL [18], Bluesky [47], and SCFS [25] focus on remote FUSE-based file systems that store data in one or multiple third-party clouds. These systems ensure data confidentiality with AES encryption.

As shown in Table 1, with the exception of Cryptfs, NCryptfs and dm-crypt, all the previous solutions are monolithic systems, similar to traditional distributed file system proposals [22, 40] with additional data confidentiality features. Many proposals only support one type of encryption scheme, and even in the systems that allow the user to chose among a set of schemes, it remains unclear what are the trade-offs of using different approaches in terms of system performance and usability.

Only a small number of these proposals provide a design that supports the distribution of data across different storage back-ends, for replication, security, or load balancing purposes. Also, none of these proposals enable to easily switch across the distribution algorithms (column PnP-B) and have a stackable architecture that support stacking layers with this distribution behavior (column Multi-L). For instance, it is not possible to provide a stackable design where a replication layer replicates data across several subsequent processing layers that encrypt data replicas with different encryption algorithms before storing them.

In the next sections we show how SafeFS tackles all these challenges. Moreover, we show how FUSE-based security file systems can be integrated in a stackable solution while at the same time comparing the performance of several cryptographic schemes. Finally, we compare our approach with several existing kernel- and user-space solutions.

The discussion presented in this section is based on the techniques to protect sensitive information. Nevertheless, it is important to recall that the mentioned systems also provide other security functionalities such as key management, access control, data integrity, and even functionalities not related with security such as compression, de-duplication, and snapshotting. While we do not focus on such features, the flexible design of SafeFS allows us to easy integrate additional layers supporting them.

Software-defined storage. Our design draws inspiration from recent work in software-design storage (SDS) and network (SDN) systems. These proposals make a clear distinction between control and data stage planes. The control plane is logically centralized, and it has a global view of the storage infrastructure in order to dynamically manage all data stage layers that correspond to heterogeneous storage components. This is the case for IOFlow [43], whose goal is to ensure a given quality of service (QoS) for requests from virtual machines (VMs) to storage servers. A common set of API calls is implemented by both control and data stage layers. Through these calls, the control plane is able to enforce static policies to a set of VMs. Notable policy examples are for prioritizing I/O requests, defining a minimum bandwidth or maximum latency usage, and routing I/O requests through specific data stage layers.

The control plane can also be used to enforce policies not directly related to QoS metrics [23]. Notably, data stage components can be stacked and configured in an workload-aware fashion, thus supporting distinct storage workloads even in a dynamic setting. Moreover, SDS proposals are able to dispatch data requests toward multiple storage layers or devices in a flexible and transparent way for applications using the storage stack [39].

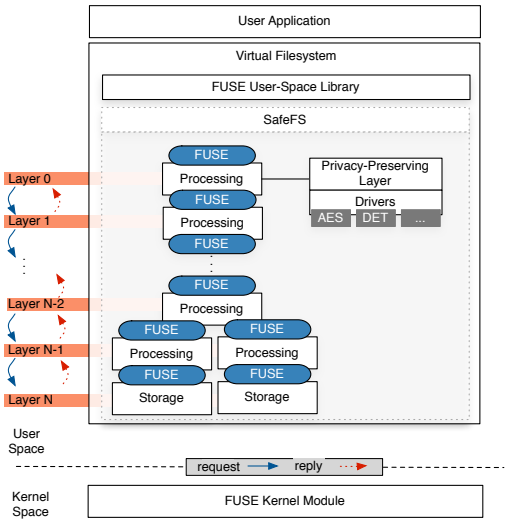

Figure 1: Architecture of SafeFS.

This paper focuses on the vertical data stage to provide a flexible solution for stackable file systems. This design enables (1) easy integration of existing file system implementations and (2) significant speedup in the development of novel file system implementations that require a specific set of storage optimizations. In the future, we plan to integrate and extend SafeFS with the control plane design and an extensive set of policies, similar to the OpenStack Swift storage component [29].

3. DESIGN GOALS

SafeFS is a framework for flexible, modular and extensible file system implementations built atop FUSE. Its design allows to stack independent layers, each with their own characteristics, optimizations, etc. These layers can then be integrated with existing FUSE-based file systems as well as restacked in different order. Each stacking configuration leads to file systems with different traits, suitable to different applications and workloads. Keeping this in mind, the four pillars of our design are:

- *Effectiveness.* SafeFS aims to reduce the cost of implementing new file systems by focusing on self-contained, stackable, and reusable file system layers.
- *Compatibility.* SafeFS allows us to integrate and embed existing FUSE-based file systems as individual layers.
- *Flexibility.* SafeFS can be configured to fit the stack of layers to the applications requirements.
- *User-friendliness.* From a client application perspective, SafeFS is transparent and usable as any other FUSE file system.

4. ARCHITECTURE

Figure 1 depicts the architecture of SafeFS. The system exposes a POSIX-compliant file system interface to the client applications. Similar to other FUSE systems, all file system related operations (*e.g.*, `open`, `read`, `write`, `seek`, `flush`, `close`, `mkdir`, etc.) are intercepted by the Linux FUSE kernel module and forwarded to SafeFS by the FUSE user-space library. Each operation is then processed by a stack of layers, each with a specific task. The combined result of these tasks represents a file system implementation.

We identify two types of SafeFS layers serving different purposes. Upon receiving a request, *processing* layers manipulate or transform file data and/or metadata and forward the request to the next layers. Conversely, *storage* layers persist file data and metadata in designated storage backends, such as local disks, network storage, or cloud-based storage services. All layers expose an interface identical to the one provided by the FUSE library API, which allows them to be stacked in any order. Requests are then orderly passed through all the layers such that each layer only receives requests from the layer immediately on top of it and only issues requests to the layer immediately below. Layers in the bottom level must be of storage type, in order to provide a functional and persistent file system.

This stacking flexibility is key to efficiently reuse layer implementations and adapt to different workloads. For example, using compression before replicating data across several storage back-ends may be acceptable for archival-like workloads. In such settings, decompressing data before reading it does not represent a performance impairment. On the other hand, for high-throughput workloads it is more convenient to only apply compression on a subset of the replicated backends. This subset will ensure that data is stored in a space-efficient fashion and is replicated to tolerate catastrophic failures, while the other subset will ensure that stored data is uncompressed and readily available. In these scenarios, one storage stack would use a compression layer before a replication layer, while a second storage stack would put the compression layer after the replication and only for a subset of storage backends. Layers must be stacked wisely and not all combinations are efficient. An obviously bad design choice would be to stack a randomized privacy layer (*e.g.*, standard AES cypher) before a compression layer (*e.g.*, gzip): by doing so, the efficiency of the compression layer would be highly affected since information produced by the above layer (the randomized encryption) should be indistinguishable from random content. While such malformed scenarios can indeed happen, we believe that *with great power comes great responsibility* and operators who deploy the system need to take care of the appropriate layer ordering.

Finally, the SafeFS architecture allows us to embed distributed layers as intermediate layers. This is depicted in Figure 1, where layer $N-2$ (*e.g.*, a replication layer) stores data into two different sub-layers $N-1$. SafeFS supports redirection of operations toward multiple layers, while at the same time maintaining these layers agnostic from the layer above that transmits the requests.

4.1 A day in the life of a write

To illustrate the I/O flow inside a SafeFS stack, we consider a `write` operation issued by the client application to the virtual file system (`read` operations are handled similarly). Each request made to the virtual filesystem is handled by the FUSE kernel module (Figure 2-❶) that immediately forwards is to the user-space library (Figure 2-❷). At this point the request reaches the topmost layer of the stack (Figure 2-❸), called *Layer 0*. After processing the request according to its specific implementation, each layer issues a write operation to the following layer. For example, a privacy-preserving layer responsible for ciphering data will take the input data, cipher it according to its configuration, and emit a new `write` operation with the encrypted data to the underlying layer. This process is repeated, according

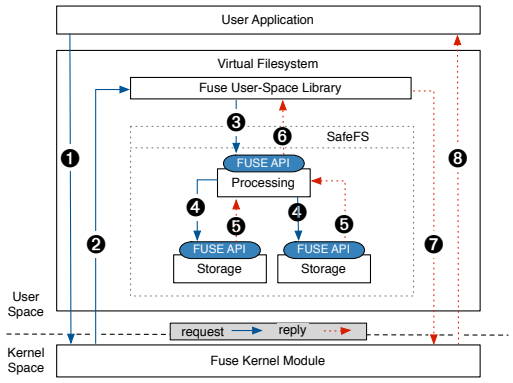

Figure 2: Execution flow of a `write` request.

to each layer implementation, until the operation reaches a *storage* layer, where the data is persisted into a storage medium (Figure 2-❹). The reply request stating whether the operation was successfully executed or not takes the reverse path and is propagated first to *Layer 0* (Figure 2-❺), and eventually backward up to the application (Figure 2-❽).

When using distributed layers (*e.g.*, with replication), `write` operations are issued to multiple sublayers or storage back-ends. These distributed layers can break some of the assumptions made by the applications. For instance, `rename` and `sync` operations must be atomic. To ensure correct semantics of the operations, a distributed layer should contain a synchronization mechanisms that ensure that an operation is only committed if successful in every backend. Otherwise, the operation fails and the file state must not be changed. A possible solution would be a block cache that stores blocks before any operation is applied.

We have discussed so far how layers modify data from read and write operations. The behavior for layers that modify the attributes and permissions of files and folders is similar. For instance, a layer providing access control to files shared among several users will add this behavior to the specific FUSE calls that read and modify the files. This design paves the way for layer reuse and for interesting stacking configurations. Individual layers do not need to implement the totality of the FUSE API: if a layer only needs to manipulate files, it only needs to wrap the FUSE operations that operate over files. FUSE operations over folders can be ignored and passed directly to the next layer without any additional processing. Layers can support the full FUSE API or a restricted subset, and this allows for a highly focused layer development cycle.

4.2 Layer integration

Besides the standard FUSE API, each SafeFS layer implements two more functions. First, the *init* function initializes metadata, loads configurations, and specifies the following layer(s) in the specific SafeFS stack. Second, the *clean* function frees the resources possibly allocated by the layer.

The integration of existing FUSE-based implementations in the form of a SafeFS layer is straightforward. Once the *init* and *clean* are implemented, a developer simply needs to link its code against the SafeFS library instead of the default FUSE. Additionally, for a layer to be stacked, delegation calls are required to forward requests to the layers below or above. The order in which layers are stacked is flexible and is declared via a configuration file.

Finally, SafeFS supports layers that modify the size of data being processed (*e.g.*, compression, padded encryption) without requiring any global index or cross-layer metadata. This is an advantage over previous work [50], further discussed with concrete examples in Section 5.

4.3 Driver mechanism

Some of the privacy-preserving layers must be configured with respect to the specific performance and security requirements of the application. However, these configurations do not change the execution flow of the messages. From an architectural perspective, using a DES cipher or an AES cipher is strictly equivalent.

With this observation in mind, we further improved the SafeFS modularity by introducing the notion of *driver*. Each layer can load a number of drivers by respecting a minimal SafeFS driver API. Such API may change according to the layer specialization and characteristics, as further discussed in the next section. Drivers are loaded according to a configuration file at file system's mount time. Moreover, it is possible to change a driver without recompiling the file system, to re-implement layers, or to load new layers. Naturally, this is possible provided that the new configuration does not break compatibility with the previous one. For instance, introducing different cryptographic techniques will prevent the file system from reading previous data.

Consider the architecture depicted in Figure 1, with a privacy-preserving layer having two drivers, one for symmetric encryption via AES and another for asymmetric encryption with RSA. The driver API of the layer consists of two basic operations: `encode` and `decode`. In this scenario, the cryptographic algorithms are wrapped behind the two operations. When a `write` request is intercepted, SafeFS calls `encode` on the loaded driver and the specific cryptographic algorithm is executed. Similarly, when a `read` operation is intercepted, the corresponding `decode` function is called to decipher the data. In order to change the driver it is sufficient to unmount the file system, modify the configuration file, and remount the SafeFS partition.

The driver mechanism can be exploited by layers with diverse goals, such as those targeting compression, replication, or caching. In the next section we discuss the current implementation of SafeFS and illustrate its driver mechanism in further details.

5. IMPLEMENTATION

We have implemented a complete prototype of SafeFS in the C programming language. Currently, it consists of less than 4,200 lines of code (LOC), including headers and the modules to parse and load the configuration files. Configuration files are used to describe what layers and drivers are used, their initialization parameters, and their stacking order. The code required to implement a layer is also remarkably concise. For example, our cryptography-oriented layer only consists of 580 LOC. SafeFS requires a Linux kernel that natively supports FUSE (v2.6.14). To evaluate the benefits and drawbacks of different layering combinations, we implemented three unique SafeFS layers, as depicted in Figure 3. These layers are respectively concerned with data size normalization (*granularity-oriented*), enforcing data privacy policies (*privacy-preserving*) and data persistence (*multiple backend*). Since they are used to evaluate

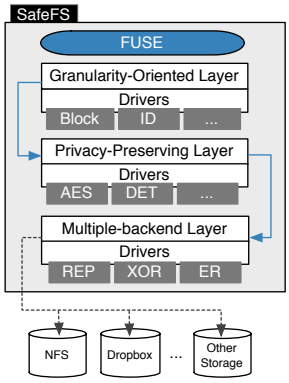

Figure 3: SafeFS: chain of layers and available drivers.

SafeFS, we detail them in the remainder of this section.

5.1 Granularity-oriented layer

It is important to be able to stack layers that operate on data at different granularity levels, *e.g.*, with different block sizes. For example, one might need to stack a layer that reports dynamic sizes for file write and read operations over a layer that only works with fixed-sized blocks ([50, 7]).

As a more concrete example, the FUSE user-space library reports file write and read operations with dynamic sizes. Yet, many cryptographic algorithms only work with fixed-size block granularity and hence require a *block size translation* mechanism. Such translation is provided by the *granularity-oriented* layer. This layer opens the way to exploit block-based encryption, instead of whole-file encryption, which is more efficient for many workloads where requests are made only for small portions of the files. For instance, if only 3 bytes of a file are being read and the block size is 4KB, then only 4KB must be deciphered while a whole-file approach could require the entire file to be deciphered only to recover those same 3 bytes of data.

In more details, the translation layer creates a logical abstraction on top of the actual data size being read, written, or modified. This is achieved by processing data write and read requests from the upper layer and manipulating the offsets and data sizes to be read/written from underneath layers. The manipulation of the offsets and sizes is done using two functions: `align_read` and `align_write`. The drivers of the layer must implement both function calls to define distinct logical views for read (for `align_read`) and write (for `align_write`) operations. Operations on directories or file attributes are redirected to adjacent layers pristine.

Our prototype implements two drivers for the translation layer: a *block* and an *identity* driver. The *block* driver creates a logical block representation for both file write and read requests, which will be used transparently by the following layers. This block abstraction is fundamental for layers whose processing or storage techniques rely on block-based requests (*e.g.*, block-based encryption, de-duplication, etc.) [10]. Block size is configured on driver initialization. On the other hand, the *identity* driver does not change the offset or the buffer size of the bytes read or written. We use this driver as a baseline to understand the overhead of our block-oriented approach and the layer itself.

5.2 Privacy-preserving layer

The goal of this layer is to protect sensitive information in a transparent way for applications and other layers of SafeFS. As explained in Section 4.3, file data being written or read is intercepted by the layer and then ciphered (`encode`) or deciphered (`decode`). We implemented three drivers: standard AES encryption (*AES*), deterministic encryption (*Det*), and *Identity*.

The *AES* driver leverages the OpenSSL's own AES-128 block cipher in *CBC* mode [15]. Both key and initialization vector (`IV`) size of the AES cipher are parameters defined during the initialization of the driver. Our design follows a block-based approach for ciphering and deciphering data. Hence, the *block* driver of the *granularity-oriented* layer is crucial to transparently ensure that each `encode` and `decode` call issued by the *AES* driver receives the totality of the bytes for a given block. Each block has a random IV associated, generated in the *encode* function, that is stored as extra padding to the cipher text.[2]

The *Det* driver protects the plaintext with a block-based deterministic cipher (AES-128). This cipher does not need a new random `IV` for each encoded block and is hence faster than randomized encryption. Despite compression algorithms being more efficient in plain-text, this driver helps detect data redundancy over cipher-text, otherwise impossible to find with a standard randomized encryption scheme.

Both drivers resort to padding (16 bytes from the AES padding plus 16 bytes for storing the `IV`). For example, a 4KB block requires 4,128 bytes of storage. Manipulating block sizes must be done consistently across filesystem operations. Every size modifying layer must keep track of the real file size and the modified file size so no assumption is broken for the upper and lower layers. For instance, if a layer adds padding data, it only reports the original file size without the extra padding to the previous stack layer.

Finally, we implemented an *Identity* driver, which does not modify the content of intercepted file operations and is used as an evaluation baseline, similarly to the *granularity-oriented* Identity driver. We note that drivers for other encryption schemes (*e.g.*, DES, Blowfish, or RSA) could be implemented similarly.

5.3 Multiple back-end layer

The storage layers directly deal with persisting data into storage back-ends. In practice, these back-ends are mapped to unique storage resources available on the nodes (machines) where SafeFS is deployed. The number of storage back-ends is a system parameter. They may correspond to local hard drives or remote storage back-ends, such as NFS servers accessible via local mount points. The drivers for this layer follow the same implementation pattern described previously, namely via `encode` and `decode` functions. The encode method, upon a write request, is responsible for generating a set of data pieces to be written in each storage back-end from the original request. The decode implementation must be able to recover the original data from a set of data pieces retrieved from the storage back-ends.

Our evaluation considers three drivers: replication (*Rep*), one-time pad (*XOR*), and erasure coding (*Erasure*). The *Rep* driver fully replicates the content and attributes of files and folders to all the specified storage back-ends. Thus, if

[2]The `IV` is important for decoding the cipher-text and returning the original plain-text but keeping it public after encryption does not impact the security of the system [35].

one of the back-ends fails, data is still recoverable from the others. The *XOR* driver uses one-time pad XOR to protect files. The driver creates a secure block (secret) by applying the one-time pad to a file block and a random generated block. This operation can be applied multiple times using the previous new secret as input, thus generating multiple secrets. The original block can be discarded and the secrets safely stored across several storage back-ends [35]. The content of the original files can only be reconstructed by accessing the corresponding parts stored across the distinct storage back-ends. Finally, the *Erasure* driver uses erasure codes such as Reed-Solomon codes [49] to provide reliability similar to replication but at a lower storage overhead. This driver increases data redundancy by generating additional parity blocks from the original data. Thereafter, a subset of the blocks is sufficient to reconstruct the original data. The generated blocks are stored on distinct back-ends, thus tolerating the unavailability of some of the back-ends without any data loss. As erasure codes modify the size of data being processed, this driver resorts to a metadata index that tracks the offsets and sizes of stored blocks on a per-file basis. The index allows containing the size-changing behavior of erasure-codes within the layer, thus not affecting any other layer.

5.4 Layer stacks

The above layers can be configured and stacked to form different setups. Each setup offers trade-offs between security, performance, and reliability. The simplest SafeFS deployable stack consists of the *multiple back-end* layer plus the *Rep* driver with a replication factor of 1 (file operations issued to a single location). This configuration offers the same guarantees of a typical FUSE loopback file system.

Increasing the complexity of the layer stack leads to richer functionalities. By increasing the replication factor and the number of storage back-ends for the simplest stack, we obtain a file system that tolerates disk failure and file corruption. Similarly, replacing the *Rep* with the *Erasure* driver, one may achieve a file system with increased robustness and reduced storage overhead. However, erasure coding techniques only work in block-oriented settings thus requiring the addition of the *granularity-oriented* layer to the stack.

When data privacy guarantees are required, one simply needs to include the *privacy-aware* layer into the stack.[3] Note that when *AES* and *Erasure* are combined, the file system stack only requires a single *block* oriented layer. This layer provides a logical block view for requests passed to the *privacy-aware* layer. These requests are then automatically passed as blocks to the *multiple back-end* layer.

Using the *XOR* driver provides an interesting privacy-aware solution, since trust is split on several storage domains. This driver exploits a bitwise technique not dependent on previous bytes to protect information, thus it does not require a block-based view as *privacy-aware* drivers.

6. EVALUATION

This section presents our comparative evaluation of the SafeFS prototype. First, we present the third-party file systems against which we compare SafeFS in Section 6.1. Then, in Section 6.2, we describe the selected SafeFS stack configurations and their tradeoffs. Section 6.3 presents the evaluation setup and the benchmark tools. Finally, Section 6.4 focuses on evaluation results.

6.1 Third-party file systems

Since our SafeFS prototype focuses on privacy preserving file systems, we deployed and ran our suite of benchmarks on four well-known open-source file systems with encryption capabilities. More precisely, we evaluate SafeFS against the CryFS [2], Metfs [13], and LessFS [10] user-space file systems. We further include eCryptfs [30], a kernel-space file system available in the Linux mainstream kernel. We selected those for being widely used, freely available, adopted by the community, and offering different security tradeoffs. Their characteristics are summarized in Section 2. This section deals with details relevant for the evaluation.

CryFS [2] (v0.9.6) is a FUSE-based encrypting file system that ensures data privacy and protects file sizes, metadata, and directory structure. It uses AES-GCM for encryption and is designed to integrate with cloud storage providers such as Dropbox. For evaluation purposes, we configure CryFS to store files in a local partition.

EncFS [7] (v1.7.4) is a cross-platform file system also built atop FUSE. This system has no notion of partitions or volumes. It encrypts every file stored on a given mounting point using AES with a 192-bit key. A checksum is stored with each file block to detect corruption or modification of stored data. In the default configuration, also used in our benchmarks, a single `IV` is used for every file, which increases encryption efficiency but decreases security.

We also evaluate MetFS [13] (v1.1.0) that uses a stream cipher (RC4) for encryption. When unmounted, the MetFS partition only stores a single blob file.

LessFS [10] (v1.7.0) supports deduplication, compression (via QuickLZ used in our experiments) and encryption (BlowFish), all enabled by default.

eCryptfs [30] (v1.0.4) includes advanced key management and policy features. All the required metadata are stored in the file headers. Similar to SafeFS, it encrypts the content of a mounted folder with a predefined encryption key using AES-128.

6.2 SafeFS configurations

Our benchmarks use a total of 7 different stack configurations (Table 2). Each exposes different performance tradeoffs and allows us to evaluate the different features of SafeFS. The chosen stacks are divided in three groups: baseline, privacy, and redundancy.

The first group of configurations, as the name implies, serve as baseline file system implementations where there is no data processing. The FUSE stack is a file system loopback deployment without any SafeFS code. It simply writes the content of the virtual file system into a local directory. The *identity* stack is an actual SafeFS stack where every layer uses the *identity* driver. It corresponds to a pass-through stack where the storage layer mimics the loopback behavior. These two stacks provide means to evaluate SafeFS framework overhead and individual layer overhead.

The privacy group is used to evaluate the modularity of SafeFS and measure tradeoffs between performance and privacy guarantees of different privacy preserving techniques. In our experiments we used three distinct techniques. The *AES* stack and *Det* stacks correspond respec-

[3]Due to the block-based nature of the *Rep* and *Erasure* drivers, the *granularity-oriented* is also required.

Groups	Stack	Granularity-Oriented		Privacy-Preserving			Multiple-Backend		
		Block	Id	AES	Det	Id	Simple	XOR	Erasure
Baseline	FUSE	×	×	×	×	×	√,1	×	×
	Identity	×	√	×	×	√	√,1	×	×
Privacy	*AES*	√	×	√	×	×	√,1	×	×
	Det	√	×	×	√	×	√,1	×	×
	XOR	×	×	×	×	×	×	√,3	×
Redundancy	*Rep*	×	×	×	×	×	√,3	×	×
	Erasure	√	×	×	×	×	×	×	√,3

Table 2: The different SafeFS stacks deployed in the evaluation. Stacks are divided in three distinct groups: Baseline, Privacy, Redundancy. The table header holds the three SafeFS layers. Below each layer we show the respective drivers. For each stack, we indicate the active drivers (the √ symbol). Layers without any active drivers are not used in the stack. The indices for Multiple-Backend drivers indicate the number of storage backends used to write data.

tively to a standard and a deterministic encryption mechanism. The *AES* stack is expected to be less efficient than *Det* as it generates a different IV for each block. However, *Det* has the weakest security guarantee. The third stack, named *XOR*, considers a different trust model where no single storage location is trusted with the totality of the ciphered data. Data is stored across distinct storage back-ends in such a way that unless an attacker gains access simultaneously to all back-ends, it is impossible to recover any sensitive information about the original plaintext.

Finally, the two remaining stacks deal with data redundancy. The *Rep* stack fully replicates files into three storage back-ends. In our configuration, two out of three back-ends can fail, while still allowing the applications to recover the original data. The *Erasure* stack serves the same purpose but relies on erasure codes for redundancy instead of traditional replication. Data is split into 3 fragments (2 data + 1 parity) over 3 back-ends for a reduced storage overhead of 50%, with respect to replication. This erasure configuration supports the complete failure of one of the back-ends. These stacks provide an overview of the costs of two different redundancy mechanisms.

6.3 Experimental Setup

The experiments run on virtual machines (VM) with 4 cores and 4GB of RAM. The KVM hypervisor exposes the physical CPU to the guest VM with the `host-passthrough` option [20]. The VMs run on top of hard disk drives (HDD) and leverage the `virtio` module for better I/O performance. We deployed each file system implementation inside a Docker (v1.12.3) container with data volumes to bypass Docker's AUFS [4] and hit near-native performance.

We conducted our experimental evaluation using two commonly used benchmarking programs: `db_bench` and `filebench`. The `db_bench` benchmark is included in `LevelDB`, an embeddable key-value store [11]. This benchmark runs a set of predefined operations on a local key-value store installed in a local directory. It reports performance metrics for each operation on the database. The `filebench` [8] tool is a full-fledged framework to benchmark file systems. It emulates both real-world and custom workloads configured using an workload modeling language (WML). Its suite of tests includes simple micro-benchmarks as well as richer workloads that emulate mail- or web-server scenarios. We leverage and expand this suite throughout our experiments.

6.4 Results

We ran several workloads for each considered file system (4 third-party file systems and 7 SafeFS stacks). The results have been grouped according to the workloads. First, we present the results of using `db_bench`, then `filebench` and, finally, we describe the results of running latency analysis for SafeFS layers.

Microbenchmark: db_bench. We first present the results obtained with `db_bench`. We pick 7 workloads, each executing 1M read and write operations on `LevelDB`, which stores the data on the selected file systems. The *fill100K* test (identified by ①) writes 100K entries in random order. Similarly, the entries are written in random order (*fillrandom*, ②) or sequentially (*fillseq*, ③). The *overwrite* (④) test completes the write-oriented test suite by overwriting entries in random order. For read-oriented tests, we considered 3 cases: *readrandom* (⑤), to randomly read entries from the databases, *readreverse* (⑥) to read entries sequentially in reverse order, and finally *readseq* (⑦) to read entries in sequential order.

Figure 4 presents the relative results of each system against the same tests executed over a `native` file system (ext4 in our deployment). We use a colors to indicate individual performance against native: red (below 25%), orange (up to 75%), yellow (up to 95%), and green (>= 95%). MetFS results were discarded due to a lock issue which prevented the database to initialize correctly.

We observe that all systems show worse performance for write-specific workloads (①–④) while performing in yellow class or better for read-oriented workloads (⑤, ⑥, and ⑦). The results are heavily affected by the number of entries in the database (*fill100K* ① vs *fillrandom* ②). As the size of the data to encrypt grows, the performance worsens. For instance, the SafeFS `XOR` configuration (the one with the worst performance) drops from 21% to 0.5%. The same observation applies for CryFS (the system with best performance) that drops from 79.78% to 12.33%.

The results for the *fillseq* ③ workload require a closer look as they have the worst performance in every file system evaluated. Since `db_bench` is evaluating the throughput of `LevelDB` which is storing its data on the evaluated file-systems, it is necessary to understand an important property of `LevelDB`. The database is optimized for write operations, which results for *fillseq*, in high throughputs on native file systems contrasting with the selected file systems, where the throughput is significantly lower. As a matter of example, comparing throughputs for *fillseq* vs *fillrandom* on `native`

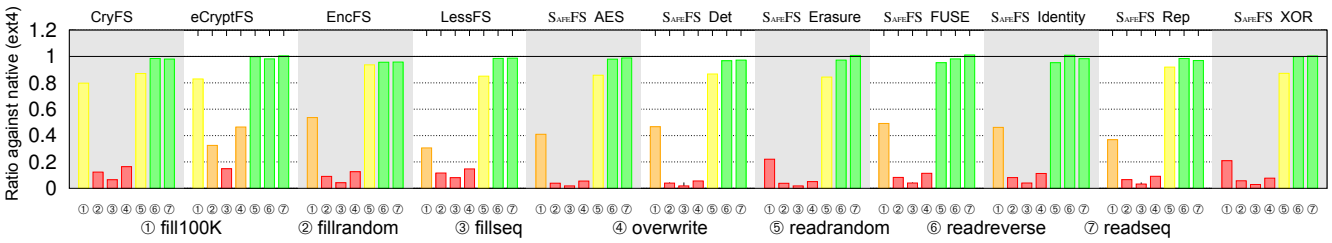

Figure 4: Relative performance of `db_bench` workloads against `native`.

(17.4 MB/s vs. 7.74 MB/s) and CryFS (1.14 MB/s vs. 0.94 MB/s) shows how much of the initial gains provided by `LevelDB` are lost.

While the processing of data heavily impacts the writing workloads, reading operations (⑤, ⑥ and ⑦) are relatively unaffected. The results for *readrandom* range from 85.06% with LessFS up to 99.81% for eCryptFS. Moreover, in experiments that switch the reading offset, the results are even better. In more details, the results never drop below 95.67% (*readreverse* on EncFS) independently of whether the reading is done from the beginning or the tail of the file.

Overall, the different SafeFS stacks perform similarly for the different database operations. The privacy stacks (see Table 2) performs comparably to the other file systems on most operations. Only the *fill100K* test shows significant differences, in particular against CryFS and eCryptFS. As expected, the deterministic driver provides a better performance (46.69%) against *AES* (40.96%) and *XOR* (20.98%). The redundancy stacks perform similarly. The erasure driver is slightly less efficient (22.11% of the native performance) due to the additional coding processing.

Microbenchmark: `filebench`. Next, we look at the relative performance of various workloads from `filebench`. Figure 5 depicts our results. We use the same color scheme as for `db_bench`. The seven workloads, executed over the different file systems and configurations, can be separated in two sets: application emulations (file-server ①, mail-server ②, web-server ③) and micro-workloads (④ to ⑦). This classification also introduces 3 major differences.

First, the application benchmarks last for 15 minutes, while the micro-workloads terminate once a defined amount of data is read or written. Second, the number of threads interacting with the system is respectively set to 50, 16, and 100 for the three workloads ①, ②, and ③, while micro-workloads are single-threaded. Third, the focus of micro-workloads is to study the behavior of a single type of operation while the application emulations usually run a mix of read and write operations.

We discarded LessFS results from this part of the evaluation as the system exhibits inconsistent behavior (*i.e.*, unpredictable initialization and run time ranging from minutes to hours) leading to timeouts before completion.

In the micro-workloads (④–⑦), we observe the performance of the tested solutions in simple scenarios. Reading workloads (④ and ⑥) are most affected by the reading order. Surprisingly, our implementation performs better than the baseline with `Det` at 104.68% on random reads. These observations contrast with the results obtained for sequential reads where the best performing configuration in this case is SafeFS `fuse` (94.24%). On the writing side, micro-workloads ⑤ and ⑦ also display different results. For sequential writes (⑦), SafeFS `Identity` tops the results at 55.56% of the native performance. On the other end of the scale, SafeFS `XOR` stalls at 9.14%. The situation does however get a little better when writing randomly: `XOR` then jumps to 14.98%. An improvement that contrasts with the case of erasure coding (that has to read all the existing data back before encoding again) where the performance dramatically drops from 29.93% to 0.7% when switching from sequential to random writes.

On the application workloads side, the mixed nature of the operations gives better insights on the performance of the different systems and configurations. The systems that make use of classical cryptographic techniques consistently experience performance hurdles. As the number of write operations diminishes, from ① to ③, the performance impact decreases accordingly. Another important factor is the use of weaker yet faster schemes (such as the deprecated stream cipher RC4 for MetFS or re-using IVs for SafeFS `Det`). As expected, those provide better results in all cases. Indeed, MetFS reaches 39.56% for ① and 55.65% for ③, and `Det` tops at 38.74% for ②. Resorting to more secure solutions can still offer good results with SafeFS `AES` (①: 28.40%, ②: 32.32%, and ③: 32.79%) but the need for integrated access control management should not be neglected for an actual deployment. Figure 5 presents the remaining SafeFS stacks for redundancy. These also exhibit signs of performance degradation as the data processing intensifies.

Beyond the specifics of the data processing in each layer, the performance is also affected by the number of layers stacked in a configuration. As evidence, we observe that the `Identity` stack has a small but noticeable decrease of performance when compared with other `FUSE` stacks.

Overall, the privacy-preserving stacks of SafeFS with a single back-end have a better performance than the other available systems across the workloads. Only MetFS is capable of providing a better performance in some workloads. This benchmark suggests that user-space solutions, such as those easily implementable via SafeFS, perform competitively against kernel-based file systems.

Microbenchmark: Layers breakdown. In addition to using `db_bench` to study the performance degradation introduced by SafeFS, we use some of its small benchmarks (*fill100K*, *fillrandom*, *fillseq*, *overwrite*, *readrandom*, *readreverse*, and *readseq*) to measure the time spent in the different layers as the system deals with read and write operations. To do so, SafeFS records the latency of both operations in every layer loaded in the stack. The results obtained are presented in Figure 6. We note that for all these benchmarks, the initialization phase is part of the record and that the time stacks show the sum of a layer's inherent overhead and the time spent in the underlying layers.

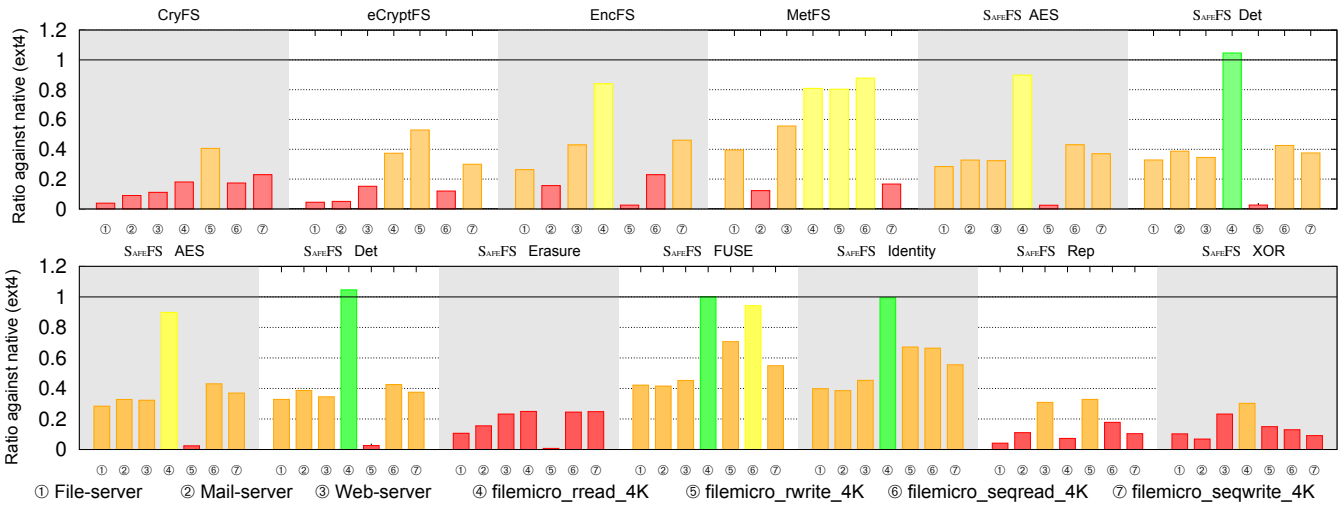

Figure 5: Relative performance of `filebench` workloads against `native`.

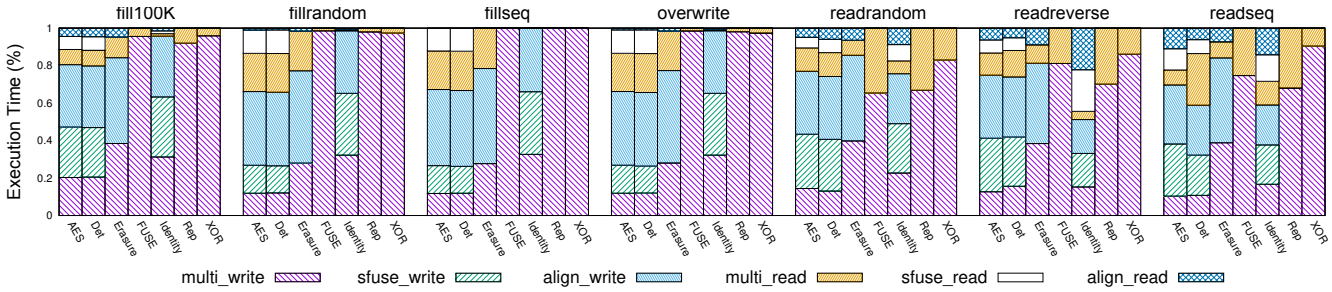

Figure 6: Execution time breakdown for different SafeFS stacks.

As expected, the time spent in each layer varies according to the tasks performed by the layers. The 3 most CPU-intensive stacks (`AES`, `Det` and `Erasure`) concentrate their load over different layers: `sfuse` for the first two and `multi` for the last one. Indeed, more time is spent in `multi`, the lowest layer, in non-privacy-preserving configurations. Another noticeable point is the increase in time spent reading data back for `Erasure` in the `multi` layer (11.03% for *fill100K*, 21.19% for *fillrandom*, and 21.53% for *fillseq*) compared to the decrease for `Rep` (respectively 8.02%, 1.93%, and 0.05%) and `XOR` (4.03%, 2.59%, and 0.05%) stacks.

7. CONCLUSION

The design, implementation, and evaluation of file systems is a complex problem as applications using them have different requirements in terms of fault-tolerance, data privacy, or pure performance. This paper proposes SafeFS, a modular FUSE-based architecture that allows system operators to simply stack building blocks (*layers*), each with a specific functionality implemented by plug-and-play *drivers*. This modular and flexible design allows extending layers with novel algorithms in a straightforward fashion, as well as reuse of existing FUSE-based implementations.

We compared several SafeFS stacking configurations against industry-battled alternatives and demonstrated the trade-offs for each of them. Our extensive evaluation based on real-world benchmarks hopefully shed some light on the current practice of deploying custom file systems and will facilitate future choices for practitioners and researchers.

We envision to extend SafeFS along three main directions. First, we plan to smooth the efforts to integrate any existing FUSE file system as a SafeFS layer, for example by exploring Linux's LD_PRELOAD mechanism, thus avoiding any recompilation step. Second, we envision a context- and workload-aware approach to choose the best stack according to each application's requirements (*e.g.* storage efficiency, resource consumption, reliability, security) leveraging SDS control plane techniques that enforce performance, security, and other policies across the storage vertical plane stack [43]. Finally, we intend to improve the driver mechanism to allow for dynamic, on-the-fly reconfiguration.

8. ACKNOWLEDGMENTS

This research was supported by the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 653884. This work 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-00696 ≫, and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia as part of project ≪UID/EEA/50014/2013≫.

9. REFERENCES

[1] BestCRYPT. http://www.jetico.com/products/personal-privacy/bestcrypt-container-encryption.

[2] CryFS. https://www.cryfs.org/.

[3] DM-Crypt. http://www.saout.de/misc/dm-crypt/.

[4] Docker and aufs in practice. https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/#/aufs-and-docker-performance.

[5] eCryptfs. http://ecryptfs.org/.

[6] EFS. https://technet.microsoft.com/en-us/library/bb457065.aspx.

[7] EncFS. https://github.com/vgough/encfs.

[8] FileBench. https://github.com/filebench/filebench.

[9] FUSE. https://sourceforge.net/projects/fuse/.

[10] LessFS. http://www.lessfs.com/wordpress/.

[11] Leveldb. http://leveldb.org/.

[12] Linux Crypto API. https://kernel.org/doc/html/latest/crypto/.

[13] MetFS. http://www.enderunix.org/metfs/.

[14] Mountable HDFS. https://wiki.apache.org/hadoop/MountableHDFS.

[15] OpenSSL. https://www.openssl.org.

[16] PPDD. http://linux01.gwdg.de/~alatham/ppdd.html.

[17] S3FS. https://github.com/s3fs-fuse/s3fs-fuse.

[18] S3QL. http://www.rath.org/s3ql-docs/.

[19] SshFS. https://github.com/libfuse/sshfs.

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

[21] C. M. Adams and S. E. Tavares. Designing S-boxes for ciphers resistant to differential cryptanalysis. In *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography, Rome, Italy*, pages 181–190, 1993.

[22] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, 2002.

[23] A. Alba, G. Alatorre, C. Bolik, A. Corrao, T. Clark, S. Gopisetty, R. Haas, R. I. Kat, B. S. Langston, N. S. Mandagere, D. Noll, S. Padbidri, R. Routray, Y. Song, C. H. Tan, and A. Traeger. Efficient and agile storage management in software defined environments. *IBM Journal of Research and Development*, 58:5:1–5:12, 2014.

[24] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 21:1–21:12, 2009.

[25] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: A Shared Cloud-backed File System. In *Proceedings of the USENIX Annual Technical Conference*, pages 169–180, 2014.

[26] M. Blaze. A Cryptographic File System for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[27] M. Carlson, A. Yoder, L. Schoeb, D. Deel, C. Pratt, C. Lionetti, and D. Voigt. Software Defined Storage. *Storage Networking Industry Assoc. working draft, Apr*, 2014.

[28] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 199–212, 2001.

[29] R. Gracia-Tinedo, P. García-López, M. Sánchez-Artigas, J. Sampé, Y. Moatti, E. Rom, D. Naor, R. Nou, T. Cortés, W. Oppermann, and P. Michiardi. IOStack: Software-Defined Object Storage. *IEEE Internet Computing*, 20(3):10–18, 2016.

[30] M. A. Halcrow. eCryptfs: An enterprise-class encrypted filesystem for linux. In *Proceedings of the 2005 Linux Symposium*, volume 1, pages 201–218, 2005.

[31] J. S. Heidemann and G. J. Popek. File-system Development with Stackable Layers. *ACM Trans. Comput. Syst.*, 12(1):58–89, Feb. 1994.

[32] S. Heron. Advanced Encryption Standard (AES). *Network Security*, 2009(12):8–12, 2009.

[33] A. D. McDonald and M. G. Kuhn. StegFS: A Steganographic File System for Linux. In *Information Hiding: Third International Workshop, IH'99, Dresden, Germany, September 29 - October 1, 1999 Proceedings*, pages 463–477, 2000.

[34] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[35] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[36] B. Schneier. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *International Workshop on Fast Software Encryption*, pages 191–204. Springer, 1993.

[37] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[38] N. Standard. Data Encryption Standard (DES). *Federal Information Processing Standards Publication*, 1999.

[39] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sRoute: Treating the Storage Stack Like a Network. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, pages 197–212, 2016.

[40] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, Wide-area Storage for Distributed Systems with WheelFS. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 43–58, 2009.

[41] H. Takabi, J. B. D. Joshi, and G.-J. Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security and Privacy*, 8:24–31, 2010.

[42] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok. Terra Incognita: On the Practicality of User-Space File Systems. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.

[43] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196, 2013.

[44] V. Turner, J. F. Gantz, D. Reinsel, and S. Minton. The digital universe of opportunities: Rich data and the increasing value of the internet of things. *IDC Analyze the Future*, 2014.

[45] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: A High-throughput File System for the HYDRAstor Content-addressable Storage System. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, pages 17–17, 2010.

[46] B. K. R. Vangoor, V. Tarasov, and E. Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, pages 59–72, 2017.

[47] 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*, pages 19–19, 2012.

[48] S. R. Walli. The POSIX Family of Standards. *StandardView*, 3(1):11–17, 1995.

[49] S. B. Wicker and V. K. Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[50] C. P. Wright, M. C. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the USENIX Annual Technical Conference*, 2003.

[51] E. Zadok, J. M. Andersen, I. Badulescu, and J. Nieh. Fast Indexing: Support for Size-Changing Algorithms in Stackable File Systems. In *Proceedings of the General Track: USENIX Annual Technical Conference*, pages 289–304, 2001.

[52] E. Zadok, I. Badulescu, and A. Shender. CryptFS: A stackable vnode level encryption file system. Technical report, Technical Report CUCS-021-98, Computer Science Department, Columbia University, 1998.

[53] E. Zadok, I. Badulescu, and A. Shender. Extending File Systems Using Stackable Templates. In *Proceedings of the USENIX Annual Technical Conference*, pages 5–5, 1999.

[54] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–16, 2000.