TRUSTFS: An SGX-enabled Stackable File System Framework

Tânia Esteves, Ricardo Macedo, Alberto Faria, Bernardo Portela*, João Paulo, José Pereira and Danny Harnik†
INESC TEC & University of Minho
*INESC TEC & University of Porto
†IBM Research - Haifa

***Abstract*—Data confidentiality in cloud services is commonly ensured by encrypting information before uploading it. However, this approach limits the use of content-aware functionalities, such as deduplication and compression. Although this issue has been addressed individually for some of these functionalities, no unified framework for building secure storage systems exists that can leverage such operations over encrypted data.**

We present TRUSTFS, a programmable and modular stackable file system framework for implementing secure content-aware storage functionalities over hardware-assisted trusted execution environments. This framework extends the original SAFEFS architecture to provide the isolated execution guarantees of Intel SGX. We demonstrate its usability by implementing an SGX-enabled stackable file system prototype while a preliminary evaluation shows that it incurs reasonable performance overhead when compared to conventional storage systems. Finally, we highlight open research challenges that must be further pursued in order for TRUSTFS to be fully adequate for building production-ready secure storage solutions.

***Index Terms*—Storage Systems, Software-Defined Storage, Intel SGX**

I. INTRODUCTION

Storage systems are a fundamental component of cloud computing solutions, being exposed as full-fledged services that offer durable, available, and scalable storage capabilities (*e.g.*, Amazon S3, Microsoft Azure). Such services enable efficient and resilient storage solutions by resorting to the combination of data-oriented functionalities such as caching, deduplication, compression, replication, and encryption. However, developing, integrating, reusing, and maintaining all these functionalities is a non-trivial task. Software-Defined Storage, in particular stackable storage systems, is an active research topic that addresses such challenges with a programmable and layered storage approach, easing the development and integration of different storage functionalities [1], [2]. Briefly, the storage stack is divided into multiple interoperable layers, each implementing an independent functionality and following a common API. Such stacking configuration enables a programmable and flexible storage solution that can be dynamically configured to fit the requirements of different applications and workloads.

On the other hand, reports of user data disclosure, government pressure on cloud-based companies, and hacking vulnerabilities, make the security and privacy of sensitive information stored at third-party cloud providers an increasingly pressing concern [3]. This issue has triggered new privacy directives, such as the European General Data Protection Regulation. The common approach for achieving storage privacy is to encrypt sensitive data at the client or company premises before storing it remotely [4]. However, traditional encryption schemes prevent processing over the encrypted data, disabling critical features such as deduplication, compression, and other content-specific functionalities. The importance in allowing for these features has motivated the proposal of alternative cryptographic protocols [5].

The emergence of trusted hardware platforms, such as Intel SGX [6], suggests a novel paradigm for the development of security-critical solutions. Specifically, these platforms enable isolated execution environments allowing for applications to run in isolation from external interfaces (including co-located software, or a potentially malicious Hypervisor/OS), and provide a mechanism for the cryptographic verification of computed outputs. The exploration of these technologies in the context of stackable storage solutions can allow for developers to leverage these security and isolation properties for implementing essential storage features while maintaining performance and security of stored data. For instance, this integration would allow boosting the number of implementations for trustworthy deduplication and compression schemes, which would validate their adoption in production deployments [7].

We address this challenge by introducing TRUSTFS, a novel storage framework that combines the modularity and programmability of state-of-the-art stackable file systems with the security and integrity guarantees provided by the Intel SGX trusted hardware technology. In more detail, this framework extends the original design of SAFEFS, a stackable file system framework, to take advantage of the isolation guarantees ensured by SGX-equipped storage servers, and enable black-box secure operations to be performed over protected data. Moreover, SGX can be integrated at different levels of the TRUSTFS framework, enabling a fine-grained distribution on the processing tasks of both trusted and untrusted environments. Such an approach allows extensible configurability and flexibility, while minimizing the need to reimplement existing storage solutions. To the best of our knowledge, this work presents the first proposal towards the integration of trusted hardware technologies with stackable file systems.

To demonstrate the usability of our proposal, we implemented a stackable file system using TRUSTFS that provides SGX-enabled compression. This prototype shows that the stackable and pluggable design of TRUSTFS allows for existing

978-1-7281-4255-5/19/$31.00 ©2019 IEEE
DOI 10.1109/SRDSW49218.2019.00012

FUSE-based file systems to easily be combined with the trusted execution primitives of SGX. Preliminary results show that the proposed prototype incurs reasonable performance overhead under different workloads when compared to conventional storage systems, with throughput degradation ranging from 6.5% up to 31.3%. We also highlight future research directions that must be considered for TRUSTFS to be effectively leveraged by both academia and industry to implement a new generation of secure storage solutions.

The paper is structured as follows. §II presents the TRUSTFS framework. §III describes the prototype file system implemented using TRUSTFS and the preliminary results obtained. §IV presents some of the open issues and the future directions. §V surveys related work. §VI concludes the paper.

II. DESIGN & IMPLEMENTATION

TRUSTFS extends the SAFEFS framework to provide integration with hardware-assisted trusted execution environments (Intel SGX). In this section, we begin by outlining SAFEFS' architecture and then detail how TRUSTFS builds on it.

A. SAFEFS

SAFEFS [2] is an open-source framework based on FUSE that enables the development of stackable POSIX-compliant file systems. It targets four major goals: (1) reducing the cost of implementing storage functionalities by resorting to self-contained, stackable, and reusable layers; (2) providing simple integration of existing FUSE-based file system implementations as individual layers; (3) allowing flexible stacking configurations to address the varying requirements of different storage workloads and applications; and (4) being transparent to client applications by exposing a POSIX file system interface.

SAFEFS' architecture is illustrated in the *Trusted Premises* part of Figure 1. Operations performed by client applications on a SAFEFS-based file system (*e.g.*, `open`, `read`) are intercepted by the FUSE kernel module (Figure 1-❶) and redirected to the corresponding SAFEFS user-space daemon by the FUSE user-space library (Figure 1-❷). Operations are then handled by a sequence of *processing layers*, that process file data and/or metadata (*e.g.*, compress, encrypt) and then forward the corresponding requests to the next layer (Figure 1-❸). Requests are finally handed to a *terminal layer*, which is responsible for persisting the required data and metadata in designated storage back-ends, such as local storage hardware, networked storage, or cloud-based storage services (Figure 1-❹).

Layers expose an interface compatible with the FUSE library API, improving flexibility in defining SAFEFS stacks and promoting code reuse, in particular allowing existing FUSE-based file systems to be integrated as layers. These stacks are defined by the user and configured at mount time.

Many storage functionalities, such as replication or encryption, may be implemented by resorting to different algorithms and/or schemes (*e.g.*, full-data replication *vs.* erasure coding; probabilistic *vs.* deterministic encryption). SAFEFS provides the notion of *drivers* to further improve code reusability, accommodating different algorithms and schemes in a configurable and modular fashion. Briefly, each layer can implement and load different drivers by respecting an API specific to the layer's implementation and purpose.

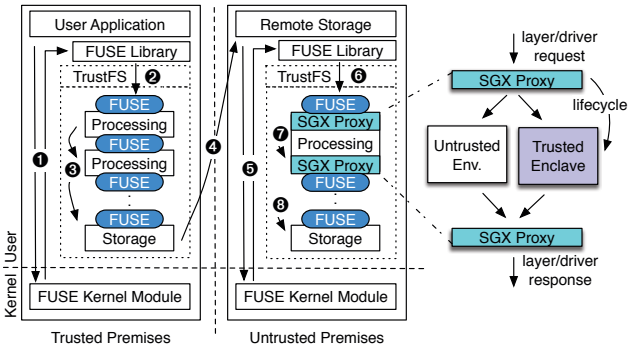

Fig. 1: TRUSTFS' architecture.

The SAFEFS project is implemented in C and provides three built-in layers. (1) A *granularity-oriented* layer allows stacking layers that operate on data at different granularities. For instance, many cryptographic algorithms consider fixed-size blocks, while other layers or even client applications may issue read and write requests with variable block sizes. This layer thus acts as a transparent middleware that promotes code reuse and avoids the implementation of this granularity translation at each layer. (2) A *privacy-preserving* layer transparently encrypts sensitive information by resorting either to probabilistic or deterministic encryption drivers. (3) A *multi-backend* layer allows persisting and retrieving file data and metadata from one or multiple storage back-ends, and provides drivers for data replication and erasure coding.

B. TRUSTFS *Design*

TRUSTFS extends the architecture of the SAFEFS framework to provide the isolated execution guarantees of Intel SGX. Its design enables content-aware storage functionalities to be leveraged in potentially untrusted environments. For instance, untrusted servers where TRUSTFS is deployed receive encrypted data from multiple remote clients over the network, and TRUSTFS resorts to SGX isolated execution environments, known as *enclaves*, to perform the required content-aware computations. This allows for computation to be performed over the original plaintext (*e.g.*, compression) or even to enable secure encrypted file sharing across multiple clients.

This paper tackles the architectural challenges of integrating Intel SGX in SAFEFS, as well as on building new layers and modifying existing ones to become SGX compliant. To be SGX compliant, TRUSTFS introduces a new middleware component — *SGX proxy* — that acts as an intermediary between the untrusted execution environment and the secure enclaves. As depicted in Figure 1, the proxy provides a transparent mechanism that promotes code reuse of existing layers and drivers, and allows executing content-aware operations either at the untrusted execution environment or at a trusted SGX enclave, according to the deployment requirements.

Since the SGX technology comes with hardware limitations (*e.g.*, memory constraints for the Enclave Page Cache size [6])

and only supports a restricted set of libraries, it is important to have a flexible design where the SGX proxy handles either the entirety of a layer's functionality, or a subset thereof. In the first case, the proxy transparently redirects incoming layer requests to an isolated SGX enclave. After completing the necessary secure computation steps (by reusing the original layer code, whenever possible, and deploying it in an enclave), the proxy handles the output from the enclave and propagates each request to the next layer. In the second case, the proxy can be used to run the code of specific drivers (*e.g.*, different compression algorithms) in a secure enclave and to handle the I/O payload of such computations. This design still promotes code reuse and avoids unnecessarily deploying operations at the enclaves, which may have a negative impact on the system's performance or functionality [8]. Moreover, the SGX proxy also manages the enclave's lifecycle (*i.e.*, creation and destruction).

C. TRUSTFS *Request Flow*

The flow of requests for a TRUSTFS stack follows a similar path as in the SAFEFS file system. To illustrate this, lets consider us consider the scenario of a write operation issued by a client application to the file system. After the initial setup (key exchange between the client and a server-side enclave), client write requests are encrypted by a privacy-preserving layer (Figure 1-❷), forwarded to a terminal layer (Figure 1-❸), and sent to the server via a remote storage protocol (Figure 1-❹). The untrusted server runs a server-side daemon for that protocol (*e.g.*, a NFS server), while data is stored and retrieved from another TRUSTFS stack (Figure 1-❺).

Requests then reach the topmost layer of the stack, for example a compression layer (Figure 1-❻). Since data is encrypted, write requests are handled by the SGX proxy, which redirects them to a trusted SGX enclave that decrypts, compresses, and re-encrypts the compressed data (Figure 1-❼). The request is then passed to the subsequent untrusted layers until the operation reaches a terminal layer, where the data is persisted in a storage medium (Figure 1-❽). The request reply, stating whether the operation was successfully executed or not, takes the reverse path and is propagated through all the layers back to the remote storage daemon, then to the user machine and, finally, back to the client application.

The flow of requests is similar for data reads and metadata operations. Read operations at the compression layer require calling the SGX enclave to decrypt, decompress and re-encrypt the uncompressed data that will be sent back to the client.

III. TRUSTFS IN PRACTICE

TRUSTFS extends the original SAFEFS implementation and provides a new component (SGX proxy) that can be used to transparently run layer and driver code in secure SGX enclaves. Transparent request handling requires adapting the proxy interface according to the interface provided by the original layers and drivers. Similarly, the enclave's interface and code need to be defined according to the computation that will run on the trusted environment. The enclave's interface is defined by using the SGX's Enclave Definition Language, while the actual enclave's implementation may reuse code from existing layers/drivers enabling the required functionalities.

To showcase and validate the applicability of the TRUSTFS framework, we next present a prototype that enables secure compression using the SGX enclaves. The main goal of this use case is to show that TRUSTFS can ease the implementation of different content-aware storage systems that leverage the Intel SGX technology.

A. SGX-enabled Compression Layer

Compression is a space reduction technique commonly applied in storage systems. Standard encryption nullifies the efficacy of compression, due to the unstructured nature of the ciphertext. However, by leveraging the isolation guarantees of SGX enclaves, we can find redundancy in the original plaintext without compromising data privacy of client data. The main goal is to provide a file system instantiation where data is encrypted at the client premises, before being stored in an untrusted server. Simultaneously, at the server, it is desirable to leverage secure compression for the files being stored, as this can reduce the required storage space.

To promote reusability, we incorporated *FuseCompress* [9], an existing FUSE-based implementation supporting data compression, as a novel TRUSTFS layer. FuseCompress enables a mix of online and offline compression for file blocks, while the size of these blocks can be defined as a configuration parameter. Blocks being written to the file system are intercepted along the I/O path and compressed before being stored (online compression). When a file is re-opened for writing or reading, all the blocks of that file are decompressed, which allows for quicker access to the data and reduces the impact in I/O operations. Then, the file can be compressed again by resorting to a background (offline) compression feature that can run, for instance, upon unmounting the file system. FuseCompress was integrated in TRUSTFS as a terminal layer, which required defining an initialization function that exports the layer's FUSE API as callbacks for other layers. This required modifying less than 230 of a total of 5276 LoC.

At the client premises, requests go through the granularity-oriented layer, are encrypted at the privacy-preserving layer, and forwarded, by the multi-backend layer, to a NFS client directory. At the server-side, requests are handled by a NFS server daemon and forwarded to a privacy-preserving layer that re-encrypts data with a secure server key. Re-encryption requires decrypting data with the corresponding client encryption key, which was exchanged through the previously established secure channel with a specific SGX enclave, and encrypting this data with a shared secure server key. Since plaintext data is temporarily disclosed during this operation, such must be done inside the enclave. This approach allows having a deployment that supports multiple clients, with independent encryption keys, that want to write and read shared files at the storage server.

After being re-encrypted with the secure server key, blocks are sent to the compression layer. The solution is to enable compression and decompression over the original plaintext, which must only be disclosed on a secure enclave environment

when the server is untrusted. Briefly, the compression driver decrypts each block with the server key, uses the LZO algorithm to compress data (in chunks of 128 KiB), encrypts the resulting data (with the same deterministic scheme and server key), and passes the encrypted compressed data back to the layer to be written in the local storage device. When data is being read, the driver decrypts the compressed data, decompresses this data and encrypts it back with the secure server key. At the re-encryption layer, data is decrypted with the server key and encrypted back with the encryption key of the client that wants to read the data.

Turning this insecure compression layer into an SGX-compliant one required adding less than 200 LoC. Also, this deployment demonstrates that it is possible to support multiple enclaves in the same storage stack.

B. Preliminary Evaluation

We conducted a preliminary performance evaluation to assess the performance of TRUSTFS. Here, we describe the adopted evaluation methodology and discuss the obtained results.

a) Evaluated deployments: As the baseline, we considered a `Native` NFS-v3 deployment, with the NFS client and server running in separate machines. Client applications perform operations directly on the NFS client's file system, which is configured to perform writes asynchronously, while the NFS server persists data in an ext4 file system residing in a solid-state drive. Since the proposed TRUSTFS prototype relies on FuseCompress, we also considered a deployment where the NFS server persists data to the FuseCompress file system, in turn backed by the ext4 file system. This `Vanilla` setup allowed us to measure the overhead imposed by FuseCompress. To gain insight into the performance cost of integrating existing FUSE file systems as a TRUSTFS layer, we also evaluated a `Layered` setup where FuseCompress is integrated into TRUSTFS as an independent layer without contemplating any security guarantees. Finally, we considered an `SGX` deployment that contemplates the SGX-enabled stackable prototype previously described, which allows understanding the direct impact of the security primitives employed in the prototype.

b) Workloads and collected metrics: All deployments were evaluated using two dumps: (1) a set of 21 ISO images of Ubuntu releases (22.3 GiB), and (2) a set of 20 Linux kernel source code releases (4.5 GiB). We measured the throughput achieved by the client when writing and reading the dumps' contents to and from each deployment. We refer to these workloads as *ISOs write*, *ISOs read*, *Kernels write*, and *Kernels read*. For each experiment, we observed the CPU and memory utilization of both client and server machines. We performed a minimum of 3 runs for each combination of evaluated deployment and workload, subsequently computing the mean of each metric and the corresponding 95% confidence intervals.

c) Experimental environment: Experiments were conducted by resorting to client nodes with one Intel Core i3-7100 CPU, clocked at 3.90 GHz, with 2 physical and 4 logical cores; 8 GiB of DDR3 RAM, clocked at 1600 MHz; and one 128 GB, SATA-III, SK hynix SC311 SSD. Server nodes had one Intel Core i3-4170 CPU, clocked at 3.70 GHz, with 2 physical and 4 logical cores; 8 GiB of DDR3 RAM, clocked at 1600 MHz; and one 128 GB, SATA-III, Samsung MZ7LN128 SSD. Nodes were interconnected by a switched Gigabit Ethernet network.

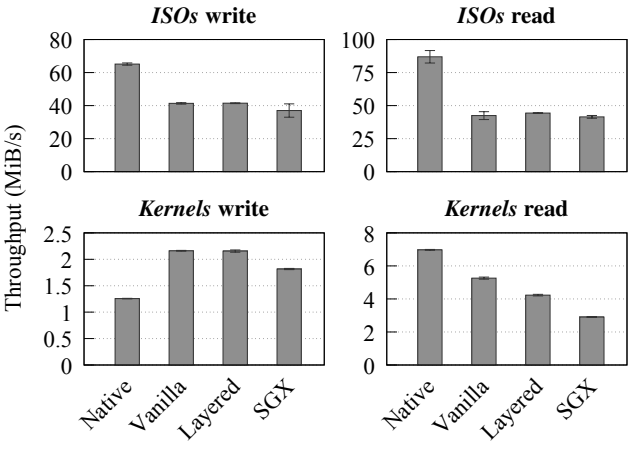

Fig. 2: Throughput results under dump workloads.

d) Results: The throughput achieved by the `Native`, `Vanilla`, `Layered`, and `SGX` deployments are depicted in Figure 2 (error bars represent the aforementioned confidence intervals). Regarding the *ISOs* dump, when compared to the `Native` results, all setups observed throughput degradation. For the `Vanilla` setup, throughput decreased by 36,5% and 51.2% for writes and reads respectively. The `Layered` deployment exhibits similar performance. Results for the `SGX` deployment report a throughput of 37.0 and 41.5 MiB/s for write and read operations, respectively, which reflects a degradation of 10.8% and 6.5% when compared to the performance achieved by the `Layered` prototype. All compression setups achieved space savings of 67.4%.

For the *Kernels* dump, when compared to `Native`, all other setups observed throughput improvements for write operations and degradations for read operations. For the `Vanilla` setup, throughput increased by 71.9% for write operations, while degrading by 24.6% on reads. The improvement for write operations can be justified by the space savings achieved by compression, namely 46.5%, which reduce the amount of data being written to the storage medium. Note that for the *ISOs* dump this improvement is not visible as the current deployment is dealing with large files that are re-opened for writing several times. This action triggers the decompression of data that is only compressed again in an offline fashion. As for the `Layered` setup, write operations performed similarly when compared to `Vanilla`, while degrading throughput by 19.6% for read operations. We hypothesize this to be due to external factors, namely performance variations at the underlying file system, operating system, and/or hardware [10]. Results for the `SGX` deployment show a throughput degradation of 15.7% and 31.3% for write and read operations, respectively, when compared to the performance achieved by the `Layered` configuration.

e) Discussion: Results show that the chosen FUSE-based compression file system (`Vanilla` setup) has a noticeable

impact, in most workloads, when compared to the `Native` deployment. This overhead is partly due to the additional context-switching between kernel and userspace required by FUSE. Such a problem has been previously addressed in the literature, and those solutions can be applied to TRUSTFS. Further, as data redundancy increases, write workloads performance can benefit from space reduction techniques since less data needs to be persisted at the storage mediums, and the `Vanilla` setup may exhibit higher write throughput than the `Native` one. By comparing the results for the `Vanilla` and `Layered` setups, it is possible to conclude that the integration of FuseCompress as a TRUSTFS layer has a small impact in the performance of the different workloads. The results for the proposed `SGX`-enabled compression prototype show that it is possible to provide secure compression while keeping the performance overhead between 6.5% to 31.3%, when compared to the `Layered` setup that does not contemplate any privacy guarantees. Indeed, the performance impact is more noticeable for the *Kernels* workloads that deal with smaller files. This happens because our current implementation exhibits better performance when handling larger requests with multiple blocks to be encrypted/decrypted. This is a future aspect to be optimized in TRUSTFS. Finally, RAM and CPU resource monitoring for the different workloads shows that resource utilization is mostly unaltered, even on the SGX-enabled setup. To summarize, experimental results indicate that TRUSTFS can ease the implementation of SGX-enabled file systems while incurring a small overhead for most of the evaluated workloads.

IV. OPEN ISSUES AND FUTURE DIRECTIONS

This section discusses open issues and future directions to achieve a more practical and robust TRUSTFS framework.

A. Block Padding

The SGX-enabled prototype evaluated in this paper resorts to a deterministic encryption scheme that preserves the original size of the plaintext data in the resulting cyphertext. This approach provides useful insights on the performance impact of the TRUSTFS framework, however in a production-ready scenario it would be desirable to resort to stronger probabilistic cryptographic schemes that require adding an extra padding to the resulting cyphertext, for instance, an Initialization Vector and a Message Authentication Code. However, simply concatenating this pad (*e.g.*, 32 bytes) to the encrypted block (*e.g.*, 4096 bytes) and storing the resulting chunk (*e.g.*, 4128 bytes) directly into the storage medium can have a deep impact in the overall I/O performance. Namely, file system implementations are typically optimized to process data in chunks whose size is a power of 2 (4 KiB, 8 KiB, *etc.*). Changing the chunk size to include the padding information introduces significant complexity to the I/O path and requires extra storage operations, resulting in an unwanted performance penalty. An alternative approach, to be explored in the future, would be to store this extra padding information as auxiliary metadata that must be persisted and retrieved efficiently across the different TRUSTFS layers.

B. Chunk Splitting

Another challenge that must be considered is that data chunks are manipulated individually by each TRUSTFS layer, and as such it is possible for chunks to be split across the I/O path. For example, if the privacy-preserving layer encrypts data with a chunk size of 128 KiB and a subsequent layer partitions the chunk into smaller parts, the decryption at the SGX enclaves will fail if only a subset of the original encrypted chunk is considered. Therefore, some TRUSTFS layers will require proper mechanisms to detect split chunks and prefetch/wait for the remaining content.

C. Integration of Existing Storage Solutions

TRUSTFS allows the straightforward integration of existing FUSE implementations as layers. However, one needs to ensure that those solutions behave as expected, especially when combined with other storage processing layers. As future work, it would be valuable to have a testing tool for assessing that a specific implementation follows desired functional and performance requirements to be integrated as a TRUSTFS layer.

D. Key Exchange and Management

TRUSTFS assumes the establishment of a secure channel between client machines and corresponding enclaves, located at the untrusted servers. This secure channel is then used to exchange encryption keys so that SGX enclaves are able to decrypt client data. The implementation of this mechanism is contemplated as future work and can leverage solutions such as the ones presented in [11].

In the present TRUSTFS implementation, client and server encryption keys are managed by the enclave and securely persisted in a storage medium by using the SGX sealing mechanism [12]. This mechanism ensures that encryption keys are only managed by the SGX enclave and are never disclosed to the untrusted server software. However, if the SGX hardware fails, it becomes impossible to recover the original content of sealed data. It would be interesting to study the trade-offs of using sealing versus a more traditional approach for encryption key management, such as the one proposed in [13].

E. SGX Side-channel Attacks

Another important issue to consider is that of side-channel resistance, which has been demonstrated to break the security of cryptographic protocols in SGX enclaves [14]. As future work, it would be essential to support such protection, for example, resorting to the LibSodium [15] library, a high-assurance cryptographic library with a strict constant-time policy, a very effective side-channel countermeasure that is often not ensured by standard cryptographic implementations.

V. RELATED WORK

Several studies resort to Intel SGX to port, deploy, and run unmodified applications in trusted execution environments [16], [17]. Such systems provide confidentiality and integrity guarantees by isolating and protecting applications and their data from access by unauthorized entities, such as the operating

system and hypervisor. Another line of research focuses on applying the SGX technology to database and data analytics systems [18]–[20], enabling secure transactional and analytical operations in untrusted infrastructures. Unlike previous work, the present paper addresses stackable storage systems and their integration with trusted hardware technologies.

The performance impact of using SGX enclaves to decrypt data, perform different data transformation operations, and then re-encrypt it in an end-to-end encrypted storage system, is studied in [21]. Relying on a trusted anchor to perform data reduction techniques over protected data has also been previously studied. The approach of [5] relies upon a trusted component for performing content-aware computation over encrypted data and for handling key exchanges between users.

In [22], the authors propose an SGX-enabled framework that system developers can use to implement FUSE-based file systems that are deployed entirely in a single enclave. Briefly, this work proposes a new module that intercepts FUSE library calls and redirects these to run in an SGX enclave. In TRUSTFS, the main goal is to provide a framework that enables developers to specify exactly what code is to be run in a secure enclave. This design allows developers to implement complex systems and balance the resulting performance, functionality, and security trade-offs. Moreover, TRUSTFS provides a framework for easily stacking different storage functionalities, each of which may be developed by different projects, thus promoting code reuse and avoiding the need to implement FUSE-based file systems from scratch.

In short, and to the best of our knowledge, there is no prior work that relies on SGX to achieve a secure and programmable storage stack. TRUSTFS is the first SGX-enabled stackable file system framework that allows building, deploying, and fine-tuning secure and generic content-aware file systems.

VI. CONCLUSION

This paper presented TRUSTFS, a stackable file system framework that leverages hardware-assisted trusted execution environments for building secure content-aware storage systems. TRUSTFS builds on SAFEFS, maintaining its modularity and programmability, and supports integration with Intel SGX. A preliminary evaluation of a compression prototype built using TRUSTFS shows that it incurs reasonable performance overhead under most workloads when compared to conventional storage systems, with throughput degradation ranging from 6.5% up to 31.3%. We believe that TRUSTFS can be of great utility for file system developers to both vest existing insecure storage functionalities with trusted properties, and to develop novel secure and flexible storage systems.

ACKNOWLEGMENTS

This work was supported by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within the project UID/EEA/50014/2019, and by FCT/MCTES through project HADES (PTDC/CCI-INF/31698/2017).

REFERENCES

[1] 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 *24th ACM Symposium on Operating Systems Principles*, 2013.

[2] R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni, P. Felber, H. Mercier, and R. Oliveira, "SafeFS: A Modular Architecture for Secure User-space File Systems: One FUSE to Rule Them All," in *10th ACM International Systems and Storage Conference*, 2017.

[3] C. A. Ardagna, R. Asal, E. Damiani, and Q. H. Vu, "From Security to Assurance in the Cloud: A Survey," *ACM Computing Surveys*, vol. 48, no. 1, 2015.

[4] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds," in *6th European Conference on Computer Systems*, 2011.

[5] N. Baracaldo, E. Androulaki, J. Glider, and A. Sorniotti, "Reconciling End-to-End Confidentiality and Data Reduction In Cloud Storage," in *6th ACM Workshop on Cloud Computing Security*, 2014.

[6] V. Costan and S. Devadas, "Intel SGX Explained," *IACR Cryptology ePrint Archive*, vol. 2016, no. 86.

[7] Y. Shin, D. Koo, and J. Hur, "A Survey of Secure Data Deduplication Schemes for Cloud Storage Systems," *ACM Computing Surveys*, vol. 49, no. 4, 2017.

[8] H. Dang and E.-C. Chang, "Privacy-Preserving Data Deduplication on Trusted Processors," in *10th International Conference on Cloud Computing*.

[9] M. Svoboda, A. Aagaard, and U. Hecht, "FuseCompress," 2008, (retrieved June, 2019). [Online]. Available: https://github.com/hexxellor/fusecompress

[10] Z. Cao, V. Tarasov, H. P. Raman, D. Hildebrand, and E. Zadok, "On the Performance Variation in Modern Storage Stacks," in *15th USENIX Conference on File and Storage Technologies* , 2017.

[11] R. Pass, E. Shi, and F. Tramer, "Formal Abstractions for Attested Execution Secure Processors," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2017.

[12] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *2nd International Workshop on Hardware and Architectural support for security and privacy*, vol. 13. ACM New York, NY, USA, 2013.

[13] D. Harnik, P. Ta-Shma, and E. Tsfadia, "It Takes Two to #MeToo-Using Enclaves to Build Autonomous Trusted Systems," *arXiv preprint arXiv:1808.02708*, 2018.

[14] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[15] "LibSodium documentation," (retrieved June, 2019). [Online]. Available: https://libsodium.gitbook.io/doc/

[16] C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *USENIX Annual Technical Conference*, 2017.

[17] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[18] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, "HardIDX: Practical and Secure Index with SGX," in *IFIP Annual Conference on Data and Applications Security and Privacy*, 2017.

[19] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A Secure Database using SGX," in *IEEE Symposium on Security and Privacy*. IEEE, 2018.

[20] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy Data Analytics in the Cloud using SGX," in *IEEE Symposium on Security and Privacy*, 2015.

[21] D. Harnik, E. Tsfadia, D. Chen, and R. Kat, "Securing the Storage Data Path with SGX Enclaves," *arXiv preprint arXiv:1806.10883*, 2018.

[22] D. Burihabwa, P. Felber, H. Mercier, and V. Schiavoni, "SGX-FS: Hardening a File System in User-Space with Intel SGX," in *IEEE International Conference on Cloud Computing Technology and Science*, 2018.