A Fine-grained General Purpose Secure Storage Facility for Trusted Execution Environment

Luigi Catuogno and Clemente Galdi
Università Degli Studi di Salerno, Italy

Keywords: Trusted Execution Environments, BYOD, Secure Storage, Enterprise Rights Management.

Abstract: In this paper we address the problem of enforcing data access control over the storage area of a mobile device running different and independent third party applications. To this end, we present the design of a general purpose secure file system that allows to guarantee file-grained data confidentiality at OS level. Data encryption, key management and policy enforcement are based on Trusted Execution Environment (TEE) facilities. We describe a prototype implementation and discuss preliminary performance results.

1 INTRODUCTION

Modern ICT infrastructures and business models entail the possibility that personal mobile devices execute software and handle data owned by multiple providers, e.g., front-end applications for subscription services along with the user access credentials and possible locally stored/cached contents.

Consider, in particular, the increasingly spreading "Bring Your Own Device" (BYOD) paradigm, where employers may leverage employees' devices (having installed the required applications and data) for the sake of any corporate process. The fact that both employees' and corporate applications and private data coexist on the same platform, raises the need of protecting each from the others' interferences. Locally stored information pertaining to each service is generally considered private to its interface application and is assumed not to be available to other applications.

Thus, for every service, related security policies and agreements are locally enforced by the respective front-end application on the basis of both subscriber's and device's credentials. However, access to user's personal data, e.g., photos and videos, that are intended to be used with different legacy applications (e.g., camera, photo gallery, file manager), is managed through the native OS-level mechanisms.

Nowadays several scenarios require advanced file access control mechanisms that: (a) enable any application to keep the data it owns isolated from the others (b) enable any application to share such data according to *ad hoc* access policies, possibly depending on the *context* in which the device is used; (c) ensure that the policy established for a file is enforced even if the file is transferred to another device through the network or with an SD card. Off-the-shelf OSes, rarely support this kind of mechanisms.

Enterprise Right Management (ERM) systems feature fine-grained access control on protected documents at fruition time. However, such systems rely on centralized security authorities whereas data protection is mainly implemented at application level as protected data fall in a limited number of types and formats. Moreover, ERM systems are rather closed solutions and rarely offer advanced interoperability features.

GlobalPlatfom's (GlobalPlatform, 2011) Trusted Execution Environment (TEE) is forthcoming standard architecture for Trusted Computing on mobile and embedded devices. Its specifications define a double sided environment composed of a *Rich Execution Environment* which runs legacy *Rich* applications, and a Trusted Execution Environment in which a *Trusted OS* takes place.

The TEE architecture feature strong resources isolation for applications running in the Trusted OS (Trusted Applications - TA). Through their TAs deployed on the device, remote service providers are enabled to locally enforce access control over their data. However, the TEE architecture provides a quite strict model of file system.

In facts, the TEE's Secure Storage facility is mainly devoted to store generic "data objects" (keys, certificates, etc.) rather than offering a full featured file system-like interface to the file store. In addition, every data object can be accessed exclusively by the

Catuogno, L. and Galdi, C.
A Fine-grained General Purpose Secure Storage Facility for Trusted Execution Environment.
DOI: 10.5220/0007578605880595
In *Proceedings of the 5th International Conference on Information Systems Security and Privacy (ICISSP 2019)*, pages 588-595
ISBN: 978-989-758-359-9
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

TA that owns it, while other TAs can not.

Finally, TEEs lets multiple TAs handling and sharing secure objects lying on secure removable storage devices through the TEE Secure Elements API. However, this interface is conceived for the sake of providing an APDU-based transport layer to ISO IEC 7816-4 modelled file systems (like smart cards). Both these features are too strict to fulfil our requirements.

In this paper we present the design and the implementation of a full featured secure file system implemented as an additional feature of the Trusted OS.

Our file system builds on top of standard TEE Internal Core API specification and it is implemented by means of a pool of Trusted Applications intended to provide it as a service to other TAs, by means of the so called Internal Client APIs. Our design approach makes possible to deploy both the file system components and forthcoming "client" Trusted Application onboard of devices featuring legacy Trusted OSes.

Our prototype features a cryptographic filesystem providing data encryption and access control at file-level and serves multiple *Trusted Applications* (*TA*) by means of a filesystem-like interface wrapped into the TEE Internal API. The filesystem is implemented in the Trusted OS. Data I/O and access control are performed by means of a TA acting as stand-alone server (the filesystem back-end). Access Control Policies can be based on roles as well as "environmental" factors such as time, events and location. In addition, we have developed an interface which enables legacy applications (a.k.a. *Rich Applications* or *RA*) to access the secure storage transparently.

2 RELATED WORKS

Cryptographic file systems (Blaze, 1993; Cattaneo et al., 2001; Kallahalla et al., 2003) enjoyed a certain success mainly in the first decade of the century. In model "trusted client" versus "untrusted server", these projects pushed data encryption functionalities beneath the virtual file system abstraction layer so that data confidentiality could be ensured independently from the type of data and the legacy application which handled it. Their evolution (Yun et al., 2009; Castiglione et al., 2014) leverages cloud-based outsourced storage facilities or passive mobile storage devices.

Mobile computing poses at least two main problems: sensitive data are stored on devices that might be stolen or tampered with; mobile devices require enriched specifications for access policies as they store data belonging to multiple stakeholders.

The Proof-Carrying File System (PCFS) (Garg and Pfenning, 2010; Geambasu et al., 2011; Peters et al., 2015; Catuogno et al., 2014) leverages formal proofs to enforce file access control policies. However, the PCFS features a quite complicated and error-prone policy-making process and a rather invasive interface.

A full featured cryptographic file system for Android is presented and analysed in (Wang et al., 2012). It essentially aims at protecting data stored on micro SD cards exchanged amongst different devices and does not natively support any trusted computing facility to protect encryption keys.

The synergy between Trusted Computing and virtualisation technologies have boosted the development of OS-level solutions which ensure data security to multiple parties by means of workloads isolation. The architecture in (Catuogno et al., 2016; Catuogno et al., 2018) use biometric identification and key binding, in conjunction with workload isolation in order to enforce the security of documents over a system comprising multiple devices owned by different actors. The architecture of VPFS (Weinhold and Härtig, 2008) feature an insecure compartment, that executes untrusted applications, and a trusted one, providing trusted data management.

Our solution provides a full featured cryptographic file system for applications running in both trusted and untrusted compartments and is intended to be fully compliant with the GlobalPlaftorm TEE standard (GlobalPlatform, 2011).

2.1 Trusted Execution Environments

GlobalPlatform, a non-profit organization, issued a set of standard specifications documents defining the concept of Trusted Execution Environment (GlobalPlatform, 2011).

GlobalPlaform's platform features a secure virtualization infrastructure which puts side by side a *Trusted Execution Environment* (TEE) and a *Rich Execution Environment* (REE). The latter executes untrusted *Rich Applications* (RA), *i.e., user applications*, whereas the former executes *Trusted Applications* (TA) that provides on-demand access to privileged resources and sensitive data to RAs, by means of a client-server interface.

The Operating System running in the REE, the Rich OS, can be considered as a legacy operating system provided of a *TEE Client API* (GlobalPlatform, 2010) library which enables RAs to interact with TAs. TEE Client APIs implement a session-oriented communication protocol with the *TEE Kernel*, the heart of the operating system running in the TEE, which is in charge of handling the communication be-

tween the two worlds. In particular, through the TEE Client API, the RAs can use possibly available cryptographic devices. Sessions are always started by RAs that act as *TEE Clients*.

On the TEE side, the Trusted OS executes each TA independently from the others, guarantees their isolation and allows each TA to access only its own data. GlobalPlatform provides specification for every aspects of TA development, in particular: the *TEE Internal APIs* to define interaction between a TA and other applications; the *TEE Trusted User Interface APIs* to enable applications to interact with users through a trusted path amongst the platform's I/O devices; and *Secure Elements APIs* to communicate with hardware secured resources and devices, called *Secure Elements* such as smart card readers, NFC controllers, crypto accelerators or hardware keys.

2.2 Commercial Mobile OSes

Mobile devices, such as tablets and smartphones, have been considered for a long time "inherently" single-user. With the continuous performance improvement for such devices, some currently available operating systems have switched from the single-user approach to a multi-user one.

Currently available Android and Windows Mobile devices provide actual multi-user support at the operating system level. Conversely, iOS virtually provides multi-user support via the cloud-based feature, *Shared Ipad* (Apple Inc., 2016), that reconfigures the device at login time.

A similar evolution has occurred regarding the file encryption functionalities. *Windows Mobile* provides a device encryption functionality (Microsoft Corp., 2017). Current OS version allows to uniquely pair SD cards to specific devices and to store personal or corporate apps and data securely either in the internal memory or on secured SDs. The Windows Information Protection (WIP), by means of *englighted* apps, transparently keeps corporate data protected and personal data private.

iOS. Apple devices include the Secure Enclave (Apple Inc., 2018), a security dedicated coprocessor that provides full support for data protection and key management. Each such device is equipped with keys that are fused or compiled in the device at manufacturing time and that are not directly accessible. In this way, data stored in memory chips are bound to a specific device. In addition to hardware security support, iOS provides File Data Protection, with each file encrypted with a random key that is itself encrypted using the file system key.

Android provides Full Disk Encryption (FDE) and File Based Encryption (FBE). FDE encrypts the whole user-data partition in a disk with a password protected encryption key. The user needs to unlock the whole disk partition before any data on it can be accessed. FBE protects each file independently and allows the FBE-aware apps to unlock each file independently. Key management is executed by means of a TEE. In all cases, data stored on external devices, such as SD cards, cannot be encrypted and should be stored in clear.

3 PRELIMINARIES

3.1 Entities and Roles

Our system features extended Role Based Access Control to protected resources by *entities* living on the device. An entity is an instance of an application which runs on the device on behalf of a subscriber. Entities are identified through the composition of a *Domain ID* and at least one amongst:(a) the UUID of the application; (b) the unique device identifier (*e.g.* a hardware-bound serial number, the mac-address) the application is running on; and (c) the subscriber credentials of the user is running the application.

Each entity can activate multiple roles. Our system features a set of default roles including *owner*, *administrator*, and *guest*, though it makes possible the definition of new per-domain, per-device, per-file roles.

Policy rules assign a set of permissions (read, write, execute) to every role. Role enforcement leverages public key encryption. To this end, every role is associated to a public-secret keypair (role keypair in short). Each file is associated to at least one *proprietary roles* and, possibly, *additional roles*. A proprietary role can both access the file content and modify its access policy while additional ones are only allowed to access file contents.

At glance, we define a *Domain* as the scope within unambiguous namespaces for entities and roles are established. Every *Domain member* (such as any device, application, subscriber), registered by the *Domain authority*, is provided of appropriate unique identifiers along with the related cryptographic credentials including digital certificates and possibly a required set of domain-wide role key pairs.

Any device owner or operator (*user* in short) is registered as *subscriber* and receives her *subsID* along with her authentication credentials (*e.g.* PIN/password, crypto-tokens, etc.). Devices can be registered by placing identifiers and credentials on

board through different procedures that depend on the platform technological characteristics. For example, credentials could be saved on a SIM, or "burnt" as a firmware update. Applications IDs and credentials are assigned according to their registration "scope": either device-wide or domain-wide.

Role keypairs, IDs and credentials are stored locally on each device at registration time. For certain entities, this choice is inherently static. For example device identifiers and credentials are deployed by the equipment manufacturer in the factory; trusted applications may be installed in the device firmware, along with their private data, once and for all. On the other hand, the choice of subscriber identities and domain-wide roles to be active on any device is likely to change over times. To this end, our architecture provides a mechanism to securely import/export keys and credentials from/to other devices.

3.2 File System Encryption

Policy enforcement leverages file system level encryption in which every single file is encrypted with a randomly generated *file key*. Only entities enjoying the required roles can "unlock" such key (through a mechanism we describe later) and access the file content. Per-file policy rules are included into file metadata. The file system ensures data confidentiality, integrity and authenticity. Integrity is guaranteed by means of message authentication codes (MAC). In particular, the file key is used both to encrypt the file content (along with its metadata) and to compute its MAC fingerprint. Data authentication is provided by means of digital signatures. Each entity is provided of a set of private keys that can be used to sign the content of each file it creates or modifies.

The cryptographic file system as a whole is implemented as a local stand-alone *file server*. The file server stores protected data (in encrypted form) on a dedicated storage area lying on the device's file system or on an external storage device (*e.g.,* SD cards, USB sticks). Entities willing to access protected files firstly authenticate themselves with the file server and then issue their requests through a file-system-API shaped IPC protocol. This component is introduced, in details, in Section 4.4.3.

4 SYSTEM ARCHITECTURE

The system we propose can be seen as an Enterprise Right Management (ERM) system, i.e., a set of tools, techniques and practices which concern

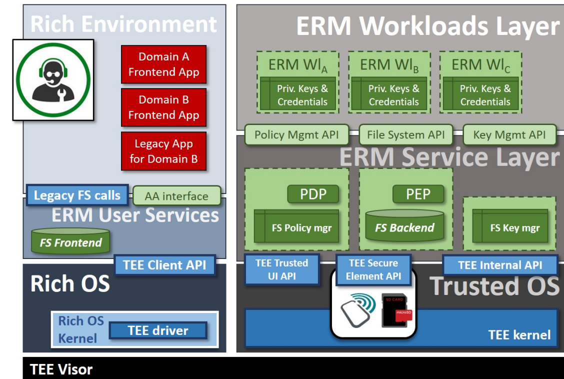

Figure 1: System architecture of maintainers' devices.

of ensuring the confidentiality of sensitive information regardless to where it is stored or is transferred through (Catuogno et al., 2016). Figure 1 describes the overall architecture. Since our architecture is modelled on top of the TEEs specifications, the "ERM interface" is split into two components: the *front-end* application, which runs as *Rich Application (RA* and its *back-end* counterpart, which is a Trusted Applications (TA) which lives in the Trusted OS along with all its private data. With *ERM workload*, we denote the ERM application "as a whole", *i.e.*, meaning both its front-end and back-end.

Components of our architecture are also implemented with Trusted Applications and protocols which are compliant to the TEE specifications.

In the following sections we introduce each component, its design highlights and the services it provides.

4.1 Front-end Applications

Operators access protected resources by means of dedicated *front-end applications* running in the Rich Environment. Such application may be of two different kinds: TEE-aware and legacy ones.

TEE-aware applications are aware of the security architecture and mechanisms deployed on the mobile device. According to the TEE standard, each such application consists of two parts, a first one running in the Rich OS and its corresponding back-end counterpart running in the Trust OS. The communication between these two components occurs via the TEE APIs. The back-end application interacts with the trusted applications implementing our security architecture by means of TEE internal APIs. It is clear that, such applications have to be specifically designed and implemented using the TEE architecture as a reference.

Legacy applications, such as standard web browsers and multimedia players, are developed by

third parties. Such applications are completely *unaware* of the security mechanisms deployed on the device. Nevertheless, the proposed architecture makes available protected data through a *file system front-end* where legacy application can access them, once the user/subscriber has been successfully authenticated and authorized.

4.2 ERM User Services

4.2.1 The AA Interface

Users authentication is performed through trusted channels such as PIN/password insertion dialogs through TEE Trusted UIs or leveraging cryptographic hardware based authentication systems. Once the user is successfully authenticated, she starts her *session* that lasts until she logs out. Throughout the session, protected data available to the user are made available through the file system front-end, and legacy application access such files enjoying the privileges of the user's active roles.

4.2.2 File System Front-end

The file system front-end allows legacy application to handle protected data through file system-shaped APIs. To this end, we use a virtual file system layer that wraps the TEE Client APIs to properly interact with the *file system back-end* in the Trusted OS.

Specifically, whenever an *untrusted* application requires a file system operation on a secured file, the file system front-end in the Rich OS forwards such request to an underlying translation module. Such module will contain no relevant application logic but it will just encode the requests in a proper way and send it to a file system back-end running on the TEE side. In our prototype, this component is implemented with a FUSE file system layer (FUS,).

4.3 ERM Workloads Layer

ERM back-end applications along with their private data take place in the ERM Workloads layer. Such applications are designed to trustworthily interact with their owners' premises in order to carry out their tasks. As TAs, ERM back-ends communicate with the user and remote facilities through the TEE APIs and the secure network protocols and are provided of a private local storage for e.g. temporary data and keying materials. Moreover, such applications handle secured files lying in the File System Backend related services by means the APIs they expose. Through such interfaces, different ERM back-ends are enabled to access and share their secure files with other TAs, according to arbitrary policy rules. Legacy TAs may co-exist with ERM workloads.

4.4 ERM Service layer

4.4.1 The Policy Manager

Our architecture features access control policies which are based on a extended Role Based Access Control (RBAC) approach which is thought to include temporal, spatial (GPS based) and event-driven policies (Bonatti et al., 2015; Aich et al., 2009). Fine-grained policy rules are attached to the file metadata.

The *Policy Manager* (PolicyMGR) maintains the *role-entity* database. For each entity "living" on the device, the role-entity database contains an entry which relates its identifier with the list of the roles it is enrolled into. The PolicyMGR API provides the functions to query such database: `check_role(entity-id, role-id)` which returns true whether (a) the entity has activated role-id or (b) the entity is allowed to play that role due to some occurring events or environmental conditions.

The function `check_policy(policy,role-id)` returns a bit-string denoting the granted access mode (like the unix-like octal digit representing read, write and execute permission) that the policy rules allow to the role *role-id*.

Entries related to domain-wide identities and roles are added to the role-entity database at registration time and can not be modified by the device owner. However, the device owner can create new "local" roles and entities, concerning the definition of policy rules for local files.

Each file is associated to the following metadata. A list of proprietary and additional roles, $r_1, \ldots, r_n$; the actual plaintext size; the specification of cryptographic algorithms, including their operational parameters; the encryptions of the file key under the public keys of authorized roles, i.e., $Enc_{r_1}(k) \ldots Enc_{r_n}(k)$; the encryption of the file access policy under the file key k, $Enc_k(Policy)$; $FILEMAC = MAC_k(File\ Content)$; LastRoleID of the last role that modified the file along with her signature on $FILEMAC$. The integrity of above data is guaranteed by $MAC_k(Metadata)$ computed by LastRoleID.

4.4.2 The Key Manager

The Key Manager (KeyMGR) handles the cryptographic keys on behalf of every entity and role present in the system.

For the sake of access control enforcement, our architecture holds a public-secret key pair for each

role defined by the domain authority. Access control leverages data symmetric encryption. Each file is encrypted with a random key (the file key) Hence, for each enabled role, the system attaches to the file a copy of its file key, encrypted with the corresponding public key.

When any entity accesses to a protected file, the file system back-end requires the KeyMGR to decrypt the file key, by using a private key associated to one of the roles the requesting entity has activated. In addition, role keypairs are used to sign/verify file data and metadata.

In order to fulfill such tasks, the KeyMGR API provides the following functionalities: *filekey encrypt/decrypt* and *data sign/verify*.

Role keys are physically stored (indexed by role-id) in the *Key Repository*, which is a private storage area cryptographically bound to the KeyMGR, and never leave this component during normal operations. KeyMGR's API includes the functions needed to handle keys and keypairs generation, import/export and disposal. The role/role keys lifecycle within domains, KeyMGR's API includes the functions to handle keypair generation, import/export and disposal.

The device user *generates* new "local" roles (and their respective role key pairs) in order to setup any ad-hoc policy rule to govern the access to a locally created file. To this end, the KeyMGR enables the user to engage in a secure protocol for the generation of a local role key-pair and triggers the PolicyMGR to add the new role-id to role-entity database (along with its enrolled entities).

The *import* functionality can occur at different times and has the effect of adding new role/role key pairs to the *Key Repository*. At device registration time, domain and device authorities can populate the *Key Repository* with the key pairs related to every pre-defined roles according to every installed application and pre-loaded files and data. In addition, at any time, the user can import a newly generated key pair from an external source. External sources include mainly *Secure Elements* as defined by the GlobalPlatform standard, such as a smart-card, or nearby trusted devices, connected through NFC.

Key pairs generation can be accomplished according different strategies. In the simplest one, the application vendor provides at least the keypair for the domain-wide reserved role of a given application. This guarantees the highest possible portability of encrypted contexts. On the other hand it also allows, in principle, the vendor to access the data generated by every single installation of its application.

Frequently protected files are transferred between device (*e.g.*, through the network or by means of SD cards). Whenever none of the roles enabled to access the files are present on the receiving device, it is necessary to transfer the related key pairs between the devices. In this case, the endpoints have to establish a trusted channel through which the key are exported.

Local role key pair disposal occurs whenever every entity operating on the device should no longer enabled to play the corresponding role.

4.4.3 The File System Back-end

The file system back-end (FSB) is the TA which acts as a stand-alone file-server and Policy Enforcement Point (PEP). To this end, the FSB exhibits a number of interfaces implemented by means of the entry points defined in the TEE internal API. Trusted applications (including ERM backends) access the encrypted storage by using a library which resembles usual file system calls (open, read, write, etc.), along with the functions devoted to entity authentication and authorization. Such a library wraps the TA client API and is intended to provide a high level interface to the FSB for third party TA application developers.

Access requests for encrypted files make the ERM system services components interact each others through the APIs introduced above.

The FSB features a file system-like interface to encrypted data stored on its *private storage*. Through this layer, data are organized and made available as *logical files*. The way in which any logical file, along with its metadata and status information is structured, handled and made available underneath of such an interface, depends on the nature of the underlying OS (the Trusted OS).

5 PROTOTYPE IMPLEMENTATION

Our PEP/FS works on top of a *ext4* filesystem, which is the last representative in a family of filesystems of widespread use on Linux systems.

In our architecture, we assume that information related to any logical file is split amongst at least two different *low level files*, one for its contents and one (or more) for its metadata.

For the purpose of managing open files, the FSB maintains two different data structure: the *open files table (OFT)* and the *Virtual File Descriptor table (VFD)*.

The OFT contains an entry for each logical file currently in use by at least one entity. When an entity opens a file, the FSB creates a new entry in the OFT. Each entry features: (a) the logical file descriptor; (b)

the data structure that references the file's low level counterparts; and (c) file's metadata and status information (including e.g. the reference counter which reports the number of entity which opened the file).

The VFD features an entry for each entity that has opened a file and maintains it until such an entity closes the file. Each entry (whose reference is the virtual file descriptor the `open` returns) links the entity with the logical file in use and contains: (a) a reference to the entry of the OFT that represents the logical file; (b) file key k; (c) entity's current role-id and credentials; and (d) access mode information.

In order to change its current role, an entity invokes the FSB *claim* function, providing the subject virtual file descriptor and the new role. The FSB, in turn, issues a *check_role* call to the PolicyMGR for the new role and, if successful, updates the current role of the VFD.

The FSB provide some ad hoc APIs to allow proprietary roles to modify the file's access policy rules. To this end, the entity opens the file and pushes the policy changes into the policy metadata then re-encrypts it with the file key and signs it. Our system is intended to be neutral with respect to the policy representation format. We envision the employment of any standard format such as extensions of XACML.

We build our prototype system using the *Open-TEE* which is a software emulator of a GlobalPlatform compliant TEE environment for the Linux operating system. Open-TEE provides a framework that can be used to develop and test prototypes of TEE-enabled applications, keeping apart the problems and challenges raised by the real hardware target platform. However, Open-TEE ensures a high level of source-compatibility with real-world TEE development frameworks.

The Open-TEE environment is composed of two stand-alone servers and a set of dynamic libraries which implements the GlobalPlatform APIs. Rich Applications are native Linux Applications written in C/C++, whereas Trusted Applications are dynamic shared objects registered with the launcher and executed in separated threads. It is clear that Open-TEE only reproduces the development environment but does not fulfills any security requirement addressed by GlobalPlatform specifications.

6 PERFORMANCE EVALUATION

We have run preliminary performance measurements of our prototype implementation. There are a number of variables that come into play when evaluating the performance of our solution. Some examples are the

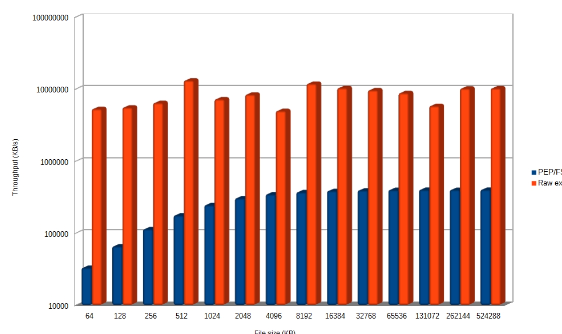

Figure 2: Comparison proposed file system vs Plain ext4 for *random read* operations.

complexity of the access policy, the presence of the (optional) en/decryption operations, the block size, the complexity related to the translation of metadata into metadata storage interfaces of the underlying file system.

We highlight that the purposes of these experiments are validating the approach and verifying whether the TEE framework APIs are suitable for the implementation of our architecture. In the following, we briefly summarize some performance results we achieved in our early experiments.

Currently, such results are clearly far even from the ones of the ext4 file system for a number of reasons. The first one is that we build on top of such a file system and, thus, our measurements consist of the time needed by our software structure and by the time needed by the ext4 file system to actually execute the operations. Secondly, we highlight that the Open-TEE environment has not been written by focusing on efficiency but as a proof of concept of the architecture.

Due to space constraints, we only report some of the experiments with have done. As stated above, we have implemented our system on an ext4 file filesystem by its extended attributes for storing file metadata. The complexity of the access policies is another variable that strongly depends on the specific policy. Since we needed to evaluate the usability of our system, we have run the first experiments using 'flat' spatio-temporal policies, i.e., allow everytime-everywhere. We note that complex access policies verification may have a huge impact on the *open* operation. On the other hand, once the file is opened, the status of the policy condition can be asynchronously monitored by the PDP in order to reduce the response time for each subsequent request.

We have run the following tests:

- Read/Randomized Read: This test measures the performance of reading an existing file (at randomly selected locations).

- Write/Randomized Write: This test measures the performance of writing a new file (at randomly selected locations).

Each test only considers the time needed to execute read/write operations in files and does *not* consider the time needed to open or close the file. In Figures 2 we report the performance of our filesystem (compared with a plain ext4 file system) in the random read operations into files with size ranging from $64KB$ to $512MB$. The performance of the read and write operations are similar to the corresponding randomized versions and are omitted.

7 CONCLUSIONS

In this paper propose the design of a full featured cryptographic file system, intended as general purpose facility for Trusted Execution Environment (TEE)-compliant platforms.

Data may either lie on a device internal storage (and bound to that device) or can be transferred amongst different devices (*e.g.,* by means of micro SD cards rather than trusted communication channel). In this case, a protocol to transfer access privileges to the receiving device is provided.

The file system is fully compliant to the TEE standard specifications. As far as we know, this is the first project of this kind which offers this feature. We present a prototype based on OpenTEE and the FUSE filesystem. Although preliminary experiments are likely to significant improvements and optimizations, they look quite promising and make the approach worthy of further investigation.

REFERENCES

FUSE: Filesystem in Userspace. http://fuse.sourceforge.net.

Aich, S., Mondal, S., Sural, S., and Majumdar, A. (2009). Role based access control with spatiotemporal context for mobile applications. In *Trans. on Computational Science IV*, volume 5430 of *LNCS*, pages 177–199.

Apple Inc. (2016). Apple Shared Ipad. https://developer.apple.com/education/shared-ipad/.

Apple Inc. (2018). iOS Security Guide - White Paper. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

Blaze, M. (1993). A cryptographic file system for unix. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 9–16. ACM.

Bonatti, P., Galdi, C., and Torres, D. (2015). Event-driven RBAC. *Journal of Computer Security*, 23(6):709–757.

Castiglione, A., Catuogno, L., Del Sorbo, A., Fiore, U., and Palmieri, F. (2014). A secure file sharing service for distributed computing environments. *The Journal of Supercomputing*, 67(3):691–710.

Cattaneo, G., Catuogno, L., Sorbo, A. D., and Persiano, P. (2001). The design and implementation of a transparent cryptographic file system for unix. In *USENIX Annual Technical Conference*, pages 199–212.

Catuogno, L., Galdi, C., and Riccio, D. (2016). Flexible and robust enterprise right management. In *IEEE Symposium on Computers and Communication, ISCC 2016, Messina, Italy, June 27-30, 2016*, pages 1257–1262.

Catuogno, L., Galdi, C., and Riccio, D. (2018). Off-line enterprise rights management leveraging biometric key binding and secure hardware. *Journal of Ambient Intelligence and Humanized Computing*.

Catuogno, L., Löhr, H., Winandy, M., and Sadeghi, A.-R. (2014). A trusted versioning file system for passive mobile storage devices. *Journal of Network and Computer Applications*, 38:65–75.

Garg, D. and Pfenning, F. (2010). A proof-carrying file system. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 349–364. IEEE.

Geambasu, R., John, J. P., Gribble, S. D., Kohno, T., and Levy, H. M. (2011). Keypad: an auditing file system for theft-prone devices. In *Proceedings of the sixth conference on Computer systems*, pages 1–16. ACM.

GlobalPlatform (2010). TEE Client API Specification v1.0. http://globalplatform.org.

GlobalPlatform (2011). TEE System Architecture v1.0. http://globalplatform.org.

Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., and Fu, K. (2003). Plutus: Scalable secure file sharing on untrusted storage. In *Prof. of the USENIX Conference on File and Storage Technologies*, pages 29–42.

Microsoft Corp. (2017). Windows 10 mobile deployment and management guide. https://docs.microsoft.com/en-us/windows/client-management/windows-10-mobile-and-mdm.

Peters, T., Gondree, M., and Peterson, Z. N. J. (2015). DEFY: A deniable, encrypted file system for log-structured storage. In *22nd Network and Distributed System Security Symposium, NDSS*. The Internet Soc.

Wang, Z., Murmuria, R., and Stavrou, A. (2012). Implementing and optimizing an encryption filesystem on android. In *Proc. of IEEE Mobile Data Management*, pages 52–62.

Weinhold, C. and Härtig, H. (2008). Vpfs: Building a virtual private file system with a small trusted computing base. *ACM SIGOPS Operating Systems Review*, 42(4):81–93.

Yun, A., Shi, C., and Kim, Y. (2009). On protecting integrity and confidentiality of cryptographic file system for outsourced storage. In *Proc. of the ACM workshop on Cloud computing security*, pages 67–76.