VPFS: Building a Virtual Private File System with a Small Trusted Computing Base

Carsten Weinhold
weinhold@os.inf.tu-dresden.de

Hermann Härtig
haertig@os.inf.tu-dresden.de

Department of Computer Science
Technische Universität Dresden
01062 Dresden, Germany

In this paper we present the lessons we learned when developing VPFS, a virtual private file system that is based on both a small amount of trusted storage and an untrusted legacy file system residing on the same machine. VPFS' purpose is to provide secure and reliable storage to highly sensitive applications running on top of a microkernel, which may concurrently execute untrusted software. The confidentiality and integrity guarantees of VPFS do not only apply to file contents, but also to all meta data including integrity of the directory structure.

We explored design alternatives that allow us to securely reuse untrusted infrastructure and thereby minimize the complexity that a file-system implementation adds to the trusted computing base. VPFS is split into two isolated components. A small trusted component implements all security-critical functionality, whereas the untrusted part reuses an existing file-system implementation provided by a virtualized legacy operating system that can be untrusted. In our VPFS prototype, alternative configurations of the trusted component comprise only between 4,000 and 4,600 lines of code, which is at least an order of magnitude smaller than existing commodity file-system stacks.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: [Access Controls, Information Flow Controls, Security Kernels]

General Terms

Reliability, Security, Design

Keywords

Trusted File System, Virtualization, Legacy Reuse

1. INTRODUCTION

To motivate our work on VPFS, we first discuss a use case. Consider a user who runs an Internet-banking application on his laptop, which he also uses for other tasks such as playing computer games and browsing the web. The user must be able to trust the banking application to keep his credentials and financial information confidential. Unfortunately, current commodity operating systems (OSes) such as Microsoft Windows or Linux cannot effectively protect individual applications and the data they process. For example, a game program that the user downloaded off the Internet can – when running under his account – access files that were created by the banking application. Although the banking application could encrypt these files, malicious software – which the user may not even be aware of – could still eavesdrop confidential authentication information, for instance using spy-ware techniques.

Isolation Architectures. Existing solutions that allow for fine-grained access control based on roles such as SELinux [8] still suffer from high complexity of the underlying commodity OSes, which consist of millions of lines of code. Frequently found security-related bugs [11] enable attackers to circumvent access control boundaries. Based on this observation, recent research [21, 30, 31] suggests to virtualize the commodity OS and isolate sensitive applications such as the aforementioned banking application by running them outside the legacy OS's virtual machine (VM); for example using a *microkernel* such as L4 [5, 9]. The key argument for this approach is that running the application directly on top of a microkernel platform – or a on *small hypervisor* [22] and a stripped-down OS in a dedicated VM – results in a smaller trusted computing base (TCB) for the isolated application. Because the commodity OS and other software running on top of it are no longer part of this TCB, the attack surface becomes smaller, too. For certain types of applications, size and complexity of their TCBs can be reduced even further by extracting and isolating the security-critical parts only: in the order of 100,000 lines of code [30] compared to millions of lines of code in a commodity OS. For example, an email client can be split such that the functionality required for digitally signing as well as encrypting and decrypting messages can be encapsulated in a separate small application; this includes cryptographic algorithms, key handling functionality, and a trusted viewer for signed and decrypted messages.

Private Storage. In both the banking and email use case, the isolated application requires persistent storage (i.e., for financial data and cryptographic keys, respectively). However, the user trusts these applications only for their specific purpose, so it follows that each application requires its own storage. With VPFS, we address the problem of private secure storage for isolated applications or extracted components (we denote them as *applications* in the remainder of our paper). The design of our file system is based on a split implementation built upon on a small virtualization-capable platform as described above. Figure 1 shows the two components of which VPFS consists: a small trusted component is responsible for security-critical operations and the other one is built on top of an untrusted legacy OS providing persistent storage. This approach allows us to take advantage of existing file-system infrastructure for the untrusted part of the implementation. Particularly, the disk driver, the buffer cache and parts of the file-system implementation of the legacy OS can be reused, thus major functionality for accessing persistent storage is no longer part of the TCB. By wrapping an existing general-purpose file system, VPFS can provide a similarly powerful interface to its client applications, thus making it suitable for all kinds of data.

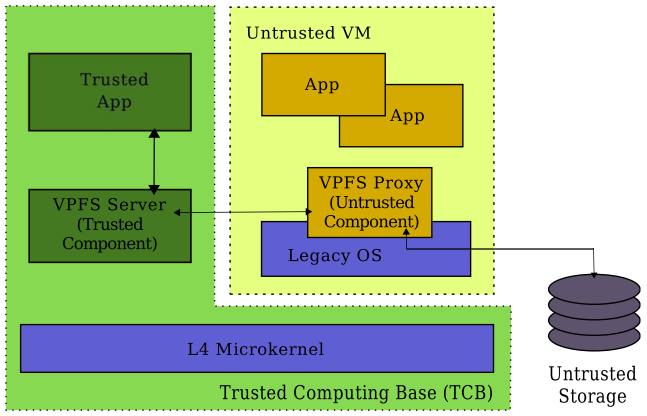
Untrusted VM
Trusted App
App
App
VPFS Server (Trusted Component)
VPFS Proxy (Untrusted Component)
Legacy OS
L4 Microkernel
Trusted Computing Base (TCB)
Untrusted Storage

Figure 1: Architectural idea behind VPFS: Following the principle of split applications, VPFS consists of two isolated components running in different protection domains. (Additional services running on top of the microkernel are not shown in this illustration.)

The trusted component must protect the file-system data stored in the untrusted legacy file system. Firstly, we use symmetric encryption to keep file-system contents accessible only to the trusted part and its client applications. Secondly, to ensure integrity of the file system, VPFS uses collision-resistant hash functions. It hashes not only the data contents but also any file-system meta data. Additionally, the hash sum includes freshness so as to avoid replay attacks. VPFS leverages trusted computing support built on top of the hypervisor. We use this technology to bind the secret encryption key to a specific hardware platform and software configuration as well as to provide a small trustworthy storage for the aforementioned hash sum that protects file-system integrity.

As the *main contributions* of the work presented on the following pages, we show how to reuse major parts of an untrusted file-system implementation, while meeting high security requirements. Unlike previous work [24, 31], we strive for full integrity and freshness of a complete, local file system and for protection against both online and offline attacks. We investigate and thoroughly discuss the implications of securely reusing untrusted infrastructure to implement naming and file lookup; we compare this strategy to the approach of including all directory support in the TCB and evaluate the alternative configurations of our VPFS prototype. We show that reusing optimized legacy infrastructure is possible with strong security guarantees. However, as a negative result, we learned that it has functional limitations and rarely provides a practically relevant performance advantage over integrating simple naming and lookup algorithms into the TCB. With either approach, VPFS enlarges an application's TCB by only a few thousand lines of code, compared to tens or even hundreds of thousand lines of code comprising the file-system stack of a commodity OS.

Synopsis. The remainder of this paper is structured as follows: in the next section we present our threat model and platform requirements and define the security policies that VPFS is designed to accommodate for. In Sections 3 and 4, we discuss our design decisions and a prototype implementation of VPFS. In Section 5, we evaluate the code complexity and performance of our implementation. We conclude our paper after discussing related work in Section 6.

2. BACKGROUND

2.1 Threat Model

Software-based Attacks. We assume that all system and application software in the untrusted legacy VM can be fully compromised. This is a plausible assumption, because, due to their enormous size and complexity, the commodity OS and many of the applications running on top of it are likely to contain weaknesses that an attacker can exploit. We consider the following attacks to be possible:

- **Attacks on Untrusted Software** Untrusted software components including the legacy OS kernel can be modified to behave maliciously or they can be shut down.
- **Attacks on Untrusted Storage** Although all file-system data in untrusted storage is protected by cryptographic means, an attacker can still modify or delete this data.

Applications using VPFS need only rely on the correctness of the hypervisor and its isolation mechanisms and the trusted part of VPFS. Being at least an order of magnitude smaller than a legacy OS, these components are assumed to be significantly harder to penetrate.

Hardware-based Attacks. We consider certain physical attacks on the user's device to be possible. We assume that an attacker can remove a storage device such as a hard disk and get raw access to the untrusted storage; or he can steal the device including all software and data on it. However, VPFS relies on the trusted computing hardware to be tamper resistant.

2.2 Protection Goals

Based on our threat model and the described split architecture, we define VPFS' protection goals as follows:

Confidentiality Only authorized applications can access file-system data. Unauthorized attempts to read the data must not reveal any useful information.

Integrity Any data that VPFS provides to an application is correct, complete, and up to date (i.e., fresh). If the data lacks any of these properties, then VPFS must be able to detect this and report an *integrity error*.

Recoverability In case of data loss (e.g., after system failure or a modification attack) the file-system contents can be restored to the last committed backup.

In addition to the first two protection goals, VPFS is designed to allow for recoverability. Recoverability is weaker than the more commonly used *availability* goal, which requires working access to file-system contents at the time it is demanded. However, a design that uses untrusted components for storing data is unsuited to ensure availability, because these components may stop working (e.g., as a result of a denial-of-service attack). Unfortunately we do not have enough space to discuss VPFS' recoverability mechanisms in this paper and therefore focus on confidentiality and integrity.

2.3 Platform Requirements

In this section we give an overview of functionality that a system platform needs to provide for VPFS. We built VPFS on top of these concepts and technologies, however, their actual implementation is out of the scope of this paper.

Isolation and Communication. The trusted and the untrusted parts of VPFS as well as trusted and untrusted applications need to be executed in different protection domains (i.e., isolated tasks on a microkernel and legacy VMs). However, the isolation must be breached such that the two parts of VPFS are still able to communicate with each other. In particular, the split-implementation approach used for VPFS relies on a high-bandwidth communication channel to transfer data between the trusted and the untrusted part (e.g., based on shared memory), as well as on a fast signaling mechanism.

Access Control. VPFS relies on the system platform's ability to allow and disallow communication between VMs or applications and instances of VPFS' trusted component (e.g., based on capabilities [19]). Since VPFS is intended to provide private storage for isolated applications, it is sufficient to assume that only one principal is using a virtual private file system. VPFS therefore does not enforce additional security policies such as per-file access control. This functionality could be added at the expense of a more complex implementation. The platform must further ensure that only authorized users can use the sensitive applications and the VPFS instances bound to them. Thus, the system platform must implement user authentication and provide a trusted path to the applications; for example, by means of a secure user interface [16].

Trusted Computing Technology. Trusted software components must be protected against offline attacks (e.g., an attacker who has physical access to the computer). A tamper-resistent Trusted Platform Module [13] as specified by the Trusted Computing Group [12] can be used as a basis to implement *authenticated booting* [23] and *remote attestation* [18]. Authenticated booting cannot keep an attacker from tampering with trusted software components. However, it allows – when used in conjunction with remote attestation – for a user to determine whether a computer indeed runs the software he expects.

VPFS reuses untrusted storage in a legacy OS to store file-system contents persistently. However, there is also data that must remain in trusted storage: (1) the secret encryption key and (2) the reference hash sum for validating file-system integrity. A TPM implementing *sealing* and *monotonic counters* as specified in [13] provides the necessary storage: Firstly, sealing allows the TPM to restrict access to the storage to a specific software stack on a specific machine. Secondly, monotonic counters can be used to ensure *freshness* [24], which means that sealed data cannot be replaced by an old copy. We subsequently refer to such a storage as *sealed memory*.

Furthermore, a TPM provides a high-quality random-number generator, which can be used for generating the secret encryption key and random nonces in VPFS. We envision that a separate trusted component of the system platform provides the described TPM-based functionality to higher-level services such as VPFS. For example, a TCB-efficient service implementing authenticated booting and sealed memory is described in [23].

3. DESIGN AND IMPLEMENTATION

When designing VPFS, our main concern was minimizing the amount of code and complexity that VPFS adds to the trusted computing base (TCB) of a specific application. We therefore chose to split the file-system implementation. The small trusted part, which we will call *VPFS server* for the remainder of our paper, implements only functionality that is critical to meet our security requirements in the context of a general-purpose file system. Hence, it only takes care of confidentiality, integrity, and recoverability, while reusing an untrusted legacy file system through the untrusted *VPFS proxy*. (Due to space constraints, recoverability is not discussed in this paper.) Before elaborating on the details of our design, we shall first give an overview of the general approach.

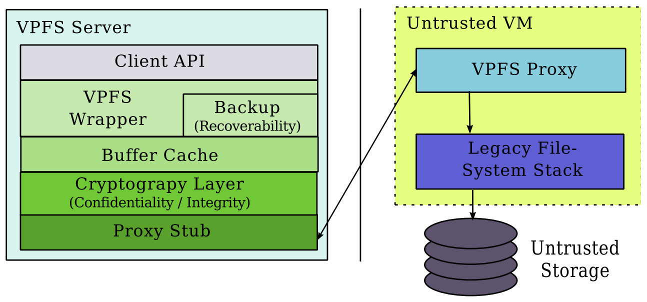

Figure 2: Layered architecture of the trusted VPFS server and the untrusted part. The VPFS server implements all security-related functionality, whereas the untrusted proxy interfaces with a legacy file-system implementation.

3.1 Design Overview

Figure 2 shows the internal structure of the VPFS server and the VPFS proxy. In essence, the VPFS server implements a cryptographic wrapper for an existing legacy file system. It includes a buffer cache, which holds decrypted and authenticated data. We quickly found that a cache in the trusted part is vital for both performance and functionality reasons; it will be discussed in Section 4.1. VPFS' untrusted proxy depends only on a POSIX-like API in order to allow for flexibility in the choice of the legacy file system being reused (even if the source code is not available). This approach does not only allow us to keep large parts of the file-system stack out of the TCB, but VPFS can also benefit from optimizations in the existing file system (e.g., caching, read ahead, clustered writes, and block allocation strategies).

From a high-level point of view, VPFS maps individual files in the virtual private file system to cryptographically protected *file containers*. File containers are regular files in the file system of the untrusted legacy OS. VPFS encrypts both the contents and the filenames of file containers so that it can ensure confidentiality. Apart from ensuring per-file integrity using hash sums, the VPFS server maintains a hash tree [27] spanning the complete virtual private file system. The root of this hash tree is kept in sealed memory and the leaves are data blocks in the file containers, which are stored in untrusted storage. Regarding integrity, we consider all meta data such as file sizes, time stamps, and the directory structure itself to be as important as file contents. The sealed memory, in which the VPFS server also stores the secret encryption key, is provided by a separate service that is part of the system platform as outlined in Section 2.3. Both the encryption key and the hash sum are sealed against the TCB of the VPFS server in order to restrict unauthorized access to this highly-critical data.

The architecture we outlined above describes a security layer that is built on top of an existing file-system interface. This wrapper approach is not only highly flexible, but it also allows for a smaller TCB than solutions based on a single protected container such as Linux' dm_crypt, which then hosts a standard file system. For example, the size of the Ext2 file system and its dependencies (e.g., the Linux Virtual File System (VFS) layer) is in the order of tens of thousands of lines of code not counting a buffer cache and cryptographic algorithms. The gains in terms of flexibility, reuse, and a smaller code base come at the price that information such as the number of files and file sizes cannot be kept secret, because the untrusted storage is populated with file containers. However, we consider this limitation with regard to confidentiality to be acceptable in a great number of use cases.

We shall now first elaborate on the protection of individual file containers. We then explore ways to ensure confidentiality and integrity of the whole file system based on them. In particular, we thoroughly discuss how to securely reuse the optimized lookup routines of the legacy file system; we present the alternative of implementing a trusted naming infrastructure thereafter.

3.2 Protecting File Contents

Because we want VPFS to be a general-purpose file system, we must not make any assumptions regarding file sizes or access patterns. Therefore, the VPFS server treats file containers in untrusted storage as chains of fixed-size data blocks, which can be loaded into the buffer cache without the need to have the whole file cached.

3.2.1 *Confidentiality*

VPFS ensures confidentiality by encrypting file contents. Individual data blocks are encrypted independently of each other, so that adequate random-access performance can be reached. We use the well-studied AES [3] algorithm in cipher-block-chaining (CBC) mode with unique initialization vectors (IVs). The VPFS server calculates these IVs based on a secret salt s, a unique file number u, and the block number n, which is local to a file container:

$$iv := H(s \parallel u \parallel n)$$

H is a collision-resistant hash function such as SHA-1 [2] that makes the IVs unpredictable for an attacker who does not know the secret s. The salt is derived from a high-quality random number and we keep it secret by storing it in a special file container. Thus, with the master encryption key for the file containers being protected by the sealed memory and the authenticated boot process (see Section 2.3), VPFS can indeed ensure confidentiality of file contents. The described IV generation scheme also effectively prevents related-IV attacks, for example watermark attacks as described by Saarinen in [29]. A similar approach is also used in Linux' dm_crypt [10].

3.2.2 *Integrity*

Based on our threat model, we assume that file-system data in untrusted storage can be subject to unauthorized tampering at any time. As such, the VPFS server has to ensure the integrity of each individual data block that it reads from a file container. At the lowest level, the trusted part of VPFS calculates and validates hash sums generated using a collision-resistant hash function in order to verify that data is indeed unmodified. We chose a hash-tree [27] based approach, which has also been demonstrated in the Trusted Database System [25]. That is, data blocks in a file container are *nodes* in a self-protecting hash tree, with a layout as illustrated in Figure 3. Only leaf nodes at the lowest level of the tree actually store the user's data; the higher-level nodes authenticate their direct child nodes.

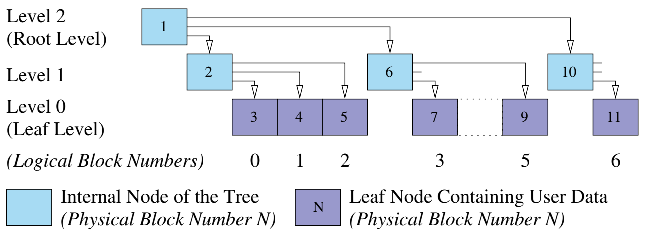
Level 2
(Root Level)
Level 1
Level 0
(Leaf Level)
(Logical Block Numbers)
1
2
6
10
3
4
5
7
9
11
0
1
2
3
5
6
Internal Node of the Tree
(Physical Block Number N)
N
Leaf Node Containing User Data
(Physical Block Number N)

Figure 3: Block layout of a file container with an embedded hash tree.

The root node's hash sum is sufficient to authenticate all data blocks in a file container. In Section 3.3, we discuss how to store this hash sum securely by applying the hash-tree approach to the complete file system. So, based on the assumption that the integrity of the root node can be guaranteed by the VPFS server, the hash tree fulfills all our integrity requirements as defined in Section 2.2:

- **Correctness and Freshness** Because there is a chain of hash sums leading to each leaf node, an attacker cannot modify user data (or replace it with an older version) without being detected.
- **Completeness** By construction, the hash tree makes it impossible for an attacker to move data blocks, neither can he add or remove nodes successfully.

However, the tree has two disadvantages: (1) it induces parent–child relations among data blocks, and (2) sub trees can become inaccessible after an unclean shutdown if not all nodes were written to persistent storage (e.g., due to a system crash). The first difficulty can be solved by reading and writing nodes in the correct order (i.e., reading the parent node before reading the child, etc.). We shall describe our solution when discussing the buffer cache in Section 4.1. The second problem can be solved using a journaling mechanism such as the one described in [25]. However, designing a split logging mechanism is part of future work on VPFS and out of the scope of this paper.

We considered simpler solutions than using a hash tree, however we found that they do not meet our requirements with regard to data integrity and performance. For example, keyed hashing [14] of independent data blocks enables fast random access, but it cannot protect against replay attacks at the block level. On the other hand, using a single hash sum to authenticate all data blocks of a file at once is impracticable for a large file, because VPFS would need to load it completely in order to calculate the hash sum; this would impose slow performance for random access patterns, or it might even be impossible if a file exceeds the size of the buffer cache.

3.3 Protecting Per-File Meta Data

Simple Meta Data. Meta data describing files could not be relied upon, if it were provided by VPFS' untrusted components. Fortunately, the hash trees in the file containers already encode trustworthy information on the file sizes. However, for time stamps, it turns out to be necessary to implement the required functionality in the trusted VPFS server. The costs in terms of additional complexity in the TCB are low, though, and the time stamps themselves can be stored securely as a part of file containers. Thus, VPFS can ensure integrity for the aforementioned types of meta data. It is however impractical to hide this information completely from an attacker operating in the untrusted domain. Although the VPFS server hides the exact file size by always writing fixed-size encrypted data blocks, an attacker can still learn about time stamps by inspecting the file containers in the legacy file system.

Integrity Anchors. What remains to be solved, is the problem of secure storage for the per-file hash sums that authenticate the contents of file containers. It is impractical to keep all these hash sums in sealed memory, because the number of files can be arbitrarily high. Therefore, we added a level of indirection: VPFS stores all the per-file hash sums in a single file container, which we call *master hash file*. For each file or directory in the file system, the VPFS server maintains one record *(master entry)* containing the hash sum of the corresponding file container's root node. As illustrated in Figure 4, only the root hash sum of the master hash file needs to be stored in sealed memory, which therefore becomes the integrity anchor of a hash tree spanning the whole file system.

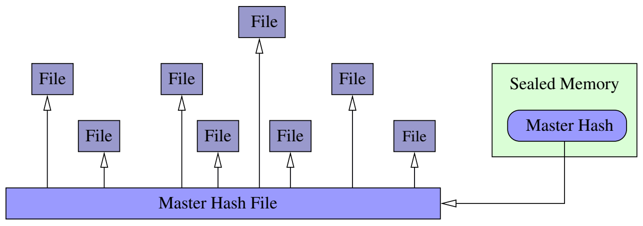

Figure 4: The master hash file authenticates the contents of all file containers.

There are a number of important aspects to mention with regard to the security properties of master entries: a file container is bound to its master entry by means of the cryptographic hash sum in it; this hash sum includes the unique file number (also used to calculate per-block IVs as explained Section 3.2.1). It effectively prevents replay attacks at the file level. That is, an attacker cannot restore a file once it has been deleted from the virtual private file system, because its master entry containing the hash sum (and also the unique file number) does no longer exist. As a master entry is logically part of the file container itself, all per-file meta data such as the aforementioned time stamps can be kept in the master entry (i.e., similar to an inode in Unix).

Retrieving Master Entries. Retrieval of the specific master entry that is required to authenticate a file container is part of the lookup process. In a very simple file system that supports files only, but no directories, a pointer identifying the master entry can easily be stored inside the file container itself. Alternatively, a minimalist file system could directly map the master entry's location inside the master hash file to the filename of the file container (i.e., both objects are named by the same number). In the following section, we will discuss the problem of naming and lookup in the context of a general-purpose file system.

3.4 Protecting the Directory Structure

We consider filenames and the directory tree in which they are organized to be first-class meta data in a file system. Thus, the name of a file is security critical with regard to confidentiality and integrity, as is the information whether or not a specific file is located in a certain directory. Especially with regard to integrity, previous work either did not address the problem of meta-data protection at all, or implemented all naming and lookup functionality within the TCB. The latter is the case for secure network file systems such as SUNDR [24] as well as for single-container based solutions such as Linux' dm_crypt [10]. We argue that a fast and space efficient implementation of a complete naming infrastructure (e.g., based on B-trees for efficient lookup and directory updates) is too complex to be part of the TCB. The analysis of existing file-system implementations indicates that fast algorithms would contribute thousands of lines of code to the TCB (e.g., 2,300 lines of code in the case of the Ext3 file system). On the other hand, all the sophisticated algorithms and caching mechanisms are already available within the untrusted commodity OS's file-system stack.

We found the possibility to reuse this untrusted infrastructure while at the same time ensuring integrity and confidentiality of the directory tree highly intriguing. Unfortunately, information provided by the legacy file-system implementation cannot be trusted (e.g., directory listings). Nevertheless, we explored this direction in order to find out exactly, how much functionality can safely be reused or complemented by additional security checks and where limitations are. We eventually implemented a prototype based on our findings, which we describe in the following Section 3.4.1. It allows for *untrusted naming* of file containers and uses simple proofs for post-operation validation.

As an alternative approach, we also implemented a simple naming system using directories based on file containers. In Sections 3.4.2 and 3.4.4, we compare the functional and security properties of both approaches.

3.4.1 Maximizing Reuse: Untrusted Naming

The motivation for reusing untrusted naming infrastructure is twofold: to keep complexity outside the TCB and to allow VPFS to benefit from existing optimizations in legacy code. We first explain what kind of functionality we want to achieve.

The basic idea behind the untrusted-lookup approach is to pass a pathname to the VPFS proxy and let it perform a lookup operation using the file-system API of the untrusted legacy OS. Those pathnames can be encrypted and hence filenames – and directory names in the path – remain secret. The challenge is to ensure integrity, which in this context means the following: (1) always the correct file is opened, (2) no file gets overwritten by creating a new file or renaming an existing one, (3) failure to look up an existing file is correctly recognized as a deletion attack, (4) deleted files are no longer accessible (i.e., no replay attacks), and (5) any directory listing is correct, complete, and up to date. Because only the VPFS server can be trusted to enforce these requirements, it follows that it must possess trustworthy information to verify the results of operations carried out by the untrusted part.

Unfortunately, the fifth integrity requirement regarding directory listings needs the VPFS server to maintain a representation of what a directory is and which files it contains. It basically means that directories need to be implemented in the VPFS server, which is exactly what untrusted naming is intended to avoid. We therefore cannot support reading of directory contents in conjunction with untrusted naming. However, this is a functional limitation that might be acceptable for many applications; we consider the first four requirements to be more important as they are security critical.

Proofs for Untrusted Information. VPFS already maintains a trustworthy data structure to ensure integrity: the master hash file. Hence, we considered reusing the master entries to make untrusted naming secure to be the best option to keep complexity low. The functionality for our integrity checks is split between the VPFS server and the untrusted VPFS proxy. Basically, the VPFS proxy maintains information on where to find a specific file container's master entry and provides hints – or *proofs* – to the VPFS server such that it can verify the correctness of a result reported by the untrusted part.

Implicit Proofs: Path Hashes. The support implemented in the VPFS server requires additional data fields in each master entry, which we will explain in the following. The first element we added is the *path hash*, which is the hash sum of the full pathname of a file. Thus, path hashes allow us to represent file locations in short fixed-size byte string. The VPFS server uses a path hash to verify that the untrusted proxy indeed opened the file container originally requested. It is easy to calculate them based on a secret salt s and the pathname:

$$path\_hash := H(s \parallel "/dir/subdir/file")$$

This authentication scheme is simple and mainly depends on functionality that is part of the VPFS server anyway (i.e., a collision-resistant hash function and routines to normalize the pathname provided by the client application). The path hash also serves another purpose: it is used as the encrypted filename, which the VPFS proxy uses to lookup the corresponding file container. Therefore, VPFS can hide the directory structure of the virtual private file system, because each file container will be named different.

Explicit Proofs. The second field that we added to each master entry is the *next pointer*. It points to the master entry that contains the lexicographically smallest path hash that is greater than the path hash of its own master entry. That is, master entries form a linked list with all elements sorted by their path hashes in ascending order. The ordering is strict and we exploit this property to allow the VPFS proxy to specify *proofs of existence* and *non-existence*.

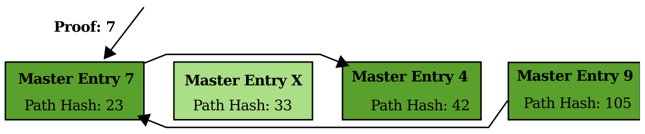

Figure 5: Master entries form a linked list sorted by their path hashes. Their strict order allows the VPFS server to determine the existence or non-existence of a file by examining only two master entries.

A proof is a pointer to a master entry, which in turn contains a *next pointer* to a second master entry. By evaluating the path hashes stored in those two master entries and knowing that both of them are directly linked, the VPFS server can easily verify whether or not a file exists under a certain pathname. Those proof pointers specify the master entry that contains the largest path hash that is smaller than the one whose existence or non-existence is to be proven. Figure 5 visualizes the general idea (path hashes are represented as small numbers for illustration purposes). Assume that a client application wanted to open a file whose pathname is transformed into the path hash 33; further assume that such a file does not exist and the VPFS proxy needed to report a lookup failure. In this case the proxy would specify a pointer to master entry 7 containing path hash 23, whose successor in the list contains path hash 42. Because the path hash 33 of the non-existing file would need to be inserted between those two master entries, the VPFS server can verify the reported lookup failure to be valid. If there were a master entry X with path hash 33 linked from master entry 7, the specified pointer to master entry 7 would proof that a file with path hash 33 indeed exists.

The untrusted component of VPFS provides a proving master-entry pointer for all operations that take an encrypted filename as argument (i.e, file open, create, rename, and delete operations), however, there are two special cases: firstly, in case of file creation, the VPFS server additionally tells the proxy which newly created master entry is linked to the path hash of the file to be created. Secondly, when opening a file, the VPFS proxy reports the master entry belonging to the requested file container instead of its predecessor in the master-entry list. Updates to the list (i.e., insertions and deletions) are performed at the position that the VPFS proxy reports after the VPFS server checked that the strict ordering is not violated and that the two master entries specified by the proof bracket the path hash derived from the application-provided pathname. A similar approach was used in [33].

Security Properties of Untrusted Naming. It is easy to see that the presented approach meets the integrity requirements that we stated at the beginning of this subsection: (1) the evaluation of path hashes stored in master entries guarantees that always the correct file is accessed, (2) files cannot get silently overwritten, because the VPFS server checks the proof of non-existence for the path hash of the target pathname, (3) file deletion attacks can be safely detected by an invalid proof of non-existence, and (4) a previously deleted file container will fail to validate, because its master entry got invalidated (or overwritten) in the delete operation that was performed at an earlier point in time.

Untrusted-Naming Support of the VPFS Proxy. In our prototype implementation, the VPFS proxy uses the popular Berkeley DB [1] to maintain a database of master-entry pointers. Naturally, the path hashes associated with the master entries are used as keys to access individual database records; the database allows the VPFS proxy to efficiently find the predecessor of a specific path hash according to the lexicographic sort order. We consider the use of Berkeley DB to be an efficient solution and it also fits our goal of reusing existing legacy infrastructure in the untrusted domain.

3.4.2 Limitations of Untrusted Naming

At the beginning of the previous Section 3.4.1, we pointed out that the untrusted-naming approach cannot provide the concept of directories. We shall now discuss this problem and its functional implications in greater detail and explain why we rejected possible solutions.

Hierarchical Name Space. Despite the lack of explicit directory support, path hashes and the way how they are calculated allow client applications to lay out files in a hierarchical name space: for example, an application can create a file under the pathname `/dir/subdir/file`. We consider this an important facility as many applications expect directories to be available. On the other hand, it can be observed that the ability to read directory contents is not required in a significant number of usage scenarios we are targeting with VPFS. An example might be an e-mail application that stores many files in maildir folders, but maintains index files that provide a comprehensive list of all individual files in those folders.

Hard Links. Due to the use of path hashes that are associated with file containers, hard links cannot be supported. However, they are required in rare cases only.

3.4.3 Moving Directory Trees

A feature usually offered by a general-purpose file system is the ability to move whole sub trees of a directory tree in a single operation (i.e., moving the root directory of the sub tree). However, VPFS with untrusted naming can move individual files only. We invested considerable effort exploring solutions that would eliminate this limitation and even implemented one of them. Unfortunately, we had to learn that (1) VPFS' confidentiality and integrity properties with regard to the directory structure would become weaker, (2) additional run-time overhead would be introduced, and (3) the little gain in functionality is hard to justify.

Regarding the first problem, VPFS would need to reveal the directory structure of the virtual private file system by storing file containers in subdirectories of the untrusted legacy file system; it could no longer hide it in a flat name space. Otherwise, the VPFS proxy could not move a directory and its contents. In such a design, confidentiality would be weakened even further, because each path element of an encrypted pathname needs to be encrypted individually without taking names of higher-level directories into account. If they were not, the VPFS proxy could not look up any file container below a directory that has been moved. This means, that files with the same name that exist in different directories would have the same encrypted filename. We considered using per-directory salts, however we dropped this alternative because it would neutralize the benefits of untrusted lookup – retrieving the salts would require each directory in the pathname to be accessed.

For the same reason that necessitates each path element to be encrypted individually, moving a directory invalidates the path hashes for all files stored below that directory. Revalidation of path hashes is possible at the expense of additional run-time costs. However, the VPFS proxy could no longer provide information that proofs the existence or non-existence of a specific file. As a result, VPFS' ability to detect integrity violations would be limited to verifying that an opened file container is the one specified by the path hash. Hence, we decided not to support the concept of directories.

3.4.4 Maximizing Functionality: Trusted Directories

Considering that the untrusted-naming approach is simple enough to be implemented in less than 200 lines of trusted code, we consider it a viable and elegant option for an implementation of a virtual private file system. Nevertheless, it cannot be denied that certain applications need complete directory support including the ability to read directory contents.

Trusted Directories. In a modified prototype, we did a straightforward implementation of trustworthy directories within the VPFS server, which then stores unsorted lists of filenames in per-directory file containers. File lookup is done by traversing all directories in the pathname. That is, the VPFS server opens the directory's file container, searches for the directory entry with the current path element's name, and closes the file container thereafter. We kept each directory in its own file container, because it saved us the need to implement efficient allocation algorithms. Also, we did not implement sophisticated lookup and update strategies, but used simple linear search. The reason for both decisions is that we did not want to include complicated algorithms in the TCB; VPFS' implementation of directory support enlarges the code base by approximately 350 lines of code. The overhead of accessing at least one, but often multiple, file containers while resolving a pathname is high. Fortunately, it can be reduced significantly thanks to a simple lookup cache, which we will describe in Section 4.2.

The integrity properties of file containers as explained in Section 3.2.2 allow VPFS to guarantee the file contents to be correct, complete, and up to date. Therefore, it follows that the directory support built on top of them provides the same guarantees for the directory structure and hence all file-system meta data. This alternative to the previously described untrusted-naming approach enables equally strong confidentiality guarantees with file containers being layed out in a flat name space in the untrusted legacy file system.

3.5 Recoverability

VPFS protects data in the untrusted storage using cryptography, however, an attacker can still damage or delete file containers. We addressed this problem by integrating a backup mechanism into VPFS, so that a user can recover data that he lost due to an attack or accident. Encrypted backups are stored on an external trusted server, which is physically decoupled from the user's device and connected to it through the Internet only (physical separation is necessary because VPFS may be used in a mobile device that can get lost or destroyed). Unfortunately, we do not have enough space in this paper to discuss recoverability in detail. More details regarding the backup and recovery protocols and required infrastructure are discussed in a separate report [32].

4. OPTIMIZATIONS

In this section, we will have a closer look at certain key areas of our prototype that we optimized for performance. The benefit of these optimizations will be evaluated quantitatively in Section 5.

4.1 Buffer Cache

The largest performance-enhancing component in VPFS is only partly an optimization: the trusted buffer cache, which holds decrypted and authenticated data blocks. We early included it into our design, even though it contributes of a significant portion of complexity to the VPFS server. The motivation for this decision is twofold. First, the buffer cache greatly improves performance for workloads with high locality; even the PostMark [7] benchmark (see Section 5) whose locality is relatively small suffered a performance loss of 70 percent, when we reduced the cache size to only a few hundred kilobytes in a quick test. With mobile devices in mind, this slowdown translates into higher power consumption because the cryptographic routines are executed more often as more data needs to be transferred between the trusted and the untrusted domain. The second important function of the the buffer cache is that it serves as a trustworthy repository for per-block authentication information, a direct consequence of using a hash tree to ensure integrity. We discuss this aspect in the following paragraphs.

Loading Data Blocks. The VPFS server can retrieve a data block from a file container only after it read the parent node, which contains the hash sum that is required to verify the data block's integrity. Our approach to meet this central requirement is based on recursion. The main idea is that, once data blocks are in the buffer cache, their contents have been authenticated. The VPFS server tries to exploit this fact by optimistically assuming that the requested data block's parent node is already present in the cache. Starting with the direct parent node, it recursively looks up nodes along the path in the embedded tree up to the root node. Recursion ends if either a node is found in the cache or after processing the root node. VPFS will then load nodes that are not yet cached in top-down order when unwinding the recursion stack.

Replacement. Because a single non-leaf node in the hash tree contains the hash sums for a large number of its direct child nodes, VPFS can draw greater benefit from cached parent nodes than from having a data block containing user data cached. We therefore reflect the parent–child relations among data blocks in cache-internal data structures. The cache uses per-buffer reference counters to keep track of buffers that contain child nodes. They are used to ensure that the buffer cache will not free a buffer unless there are no more child nodes cached (i.e., nodes with children are pinned). Fortunately, we found that the replacement algorithm itself does not need to take this constraint into account. Experiments showed that the fraction of pinned buffers is in the order of 2 to 5 percent in most use cases, so the simple approach of calling the replacement algorithm's search function multiple times is acceptable.

4.2 Other Optimizations

Plaintext Contents In File Containers. We designed VPFS as a general-purpose file system without any restrictions on the kind of data that it can store. On the other hand, there are contents that require their integrity to be preserved, but do not need to be kept confidential (e.g., application binaries or legal texts). We therefore added an opt-in interface allowing applications to create plaintext file containers. On a standard PC platform, disabling encryption reduces the CPU time consumed in the cryptographic routines by 75 percent, at the cost of adding 10 lines of code to the TCB.

Variable Depth of the Hash Tree. Furthermore, we optimized the handling of the hash trees in the file containers. The required depth of such a tree is determined by the size of the file itself. In our implementation, the embedded tree's non-leaf nodes can store information describing 146 child nodes (each node is 4 KB in size and stores 28 Bytes per child). Thus, a tree with a depth of three would suffice for files with a size of about 12 GB; we consider a maximum depth no smaller than four to be sensible. However, making the depth of the tree fixed would imply that the VPFS server must load, decrypt, and authenticate as many parent nodes as there are levels in the tree. This would lead to unacceptable overhead when initially reading data from a small file after it has been opened (e.g., a directory). We therefore implemented functionality to grow and shrink an embedded tree dynamically at run time. This functionality enlarges the VPFS server's code base by 115 lines.

Lookup Cache. The VPFS server must keep a file container open as long as there are dirty cache buffers containing data from this particular file container, even though the application might have released the file handle already. To benefit from this behavior, we implemented a simple lookup cache, which maps path hashes to handles of currently opened file containers. When requested to open a file, the VPFS server first consults the lookup cache and calls the untrusted VPFS proxy only if the file container is not currently open. In case of a hit in the lookup cache, several thousand CPU cycles can be saved; this optimization reduces the overhead caused by directory updates to a fraction of the otherwise required open–close cycle (e.g., when creating a large number of small files). The implementation costs are 40 lines of code, because we could reuse code originally implemented for the buffer cache.

5. EVALUATION

We based our prototype implementation of VPFS on the Fiasco [9] microkernel from the L4 family [5]. This platform is capable of running a paravirtualized Linux kernel called L^4Linux [6], which can be allowed to access the host's hardware. We used an L^4Linux instance as the virtualized legacy OS and allowed it to access the hard-disk controller.

The VPFS proxy is implemented as a virtualization-aware Linux application, which uses the standard POSIX file-system API. The VPFS server is implemented as native application running in its own address space on top of L4/Fiasco. Those two components of VPFS communicate with each other via shared memory and synchronous message passing. Applications are running in their own address spaces as well and interact with their respective instances of the VPFS server using a client-side library. This library encapsulates all inter-process communication between the client application and VPFS server.

5.1 Code Complexity

A main objective of our work was to minimize the complexity that a file system contributes to the TCB of an application. We used two different metrics to measure how complex our implementation of the VPFS server became: source lines of code (SLOC) and cyclomatic complexity (CCM) [26]. SLOC measures the code size excluding comments, whereas CCM is a metric for functional complexity; it basically counts the number of possible execution paths that result from conditional statements. For both metrics, higher numbers indicate higher complexity.

Table 1 shows the detailed results[1] of our size and complexity analysis. The largest part is the VPFS server itself, whose core functionality comprises almost 2,400 lines of code including the buffer cache and VPFS-proxy stub. This part is mandatory, as are the cryptographic algorithms

[1]These SLOC figures were generated using David A. Wheeler's 'SLOCCount'.

	SLOC	CCM
VPFS Server Core	$\sum$ **2,369**	
Core and utility functions	1,723	2.9
Buffer cache	468	2.9
Proxy stub	178	2.2
API Extensions (optional)		
Untrusted Naming / Trusted Dirs	180/351	4.7/4.8
read()/write()	124	4.0
mmap()	214	2.6
Backup (incomplete)	191	4.5
Lookup Cache (optional)	40	4.5
Standard Algorithms	$\sum$ **1,354**	
Cryptographic Algorithms	667	2.4
AVL Tree (used by caches)	687	7.1

Table 1: Source lines of code (SLOC) and cyclomatic complexity (CCM) for individual subsystems of VPFS' trusted code base.

and the AVL-tree implementation. VPFS requires the AVL tree to look up cache buffers efficiently. Fortunately, those standard algorithms might already be part of the TCB in same usage scenarios; for example, applications running on our L4/Fiasco platform also use the AVL-tree implementation to manage their virtual address-space layout. Furthermore, VPFS can easily be tailored to the needs of a specific application by including required functionality only. For example, file contents can be accessed using `read()` and `write()` or an `mmap()`-like mechanism or both of them. Typical configurations using either untrusted naming or trusted-directory support will therefore comprise between 4,000 and 4,600 lines of trusted code including the client-side library to access the VPFS server.

In total, we found VPFS trusted code base to be at least an order of magnitude smaller than the Linux file-system stack: including dependencies, more than 30,000 lines of kernel code are required for block-device support and a specific disk driver. The Ext3 file system and the Linux Virtual File System (VFS) layered on top comprise more than 10,000 lines of code each.

Furthermore, we measured the average cyclomatic complexity of popular Linux file-system implementations and compared the results to the CCM value obtained for the VPFS server. In Linux 2.6.14, the FAT, Ext2, and ReiserFS file systems have average cyclomatic complexities of 5.9, 5.6, and 6.5 respectively. The wrapper approach we used for VPFS results in an average CCM value of 3.1. This figure indicates that the functions implementing the simple algorithms in VPFS' trusted code base are – in average – less complex than a complete file-system. Table 1 contains a breakdown of the cyclomatic complexity for the individual subsystems. Subsystems that actually implement functionality rather than wrapping the legacy file system have higher CCM values (e.g., the code to validate proofs for untrusted naming or the support for trusted directories).

5.2 Performance

5.2.1 Test Setup

We compared the performance of VPFS and the widely-used Linux dm_crypt [10] implementation as well as

EncFS [20]. Dm_crypt is implemented in kernel space and therefore does not require expensive address-space switches. EncFS is a user-space implementation of a cryptographic file system, which – unlike dm_crypt – also ensures per-file integrity using hash sums. Thus, it is quite similar to VPFS, although it is not designed to operate across VM boundaries. We configured the security features of both systems to be as close to the capabilities of VPFS as possible (i.e., we enabled salted IVs and hash-based integrity checks if available). In order to determine the virtualization overhead, we performed the dm_crypt and EncFS benchmarks on both a paravirtualized L^{4}Linux and a native Linux instance. Native Linux accessing a plaintext disk partition containing the legacy file system serves as a baseline for reference purposes. VPFS was benchmarked with both trusted-directory support and untrusted naming; we also measured the effectiveness of some of our optimizations.

Test Environment. We chose a mix of different benchmarks, which we ran under all of the aforementioned configurations. However, although VPFS offers an API that is similar to the standard POSIX file-system interface, it turned out to be too difficult and time consuming to port the standard UNIX benchmarks to the microkernel environment. We therefore developed a framework for recording and replaying traces of file-system operations. We ran each benchmark on a plain Linux installation once and recorded all system calls to the Linux kernel using the *strace* utility. Using a tool from our framework, we then transformed the obtained trace into source code that we could compile into platform-specific programs that replayed the recorded I/O operations. This way, we could be certain that the measurements would not be influenced by differences in the specific operating environments (e.g., differences in the C library's memory management or time-related functions).

Each OS environment had the same resources[2] available, including 256 MB of main memory, which also contained a small RAM disk in which the complete test environment was installed. In the case of VPFS, we split the amount of memory and allocated 64 MB and 192 MB to the VPFS server's buffer cache and the L^{4}Linux instance, respectively. The underlying legacy file system was ReiserFS 3.6 in all benchmarks.

5.2.2 Workloads

The benchmarks cover three different workloads: (1) writing and reading large files sequentially using traces of the Bonnie++ [15] benchmark, (2) performing a large number of transactions on many small files by playing a trace of the PostMark [7] benchmark, and (3) simulating a string search in files of varying sizes. The data set used for the third workload are a number of maildir folders produced by a standard e-mail application.

Bonnie++ Traces. We measured throughput using two Bonnie++ traces that either wrote or read two 1 GB files sequentially. The hard disk itself is usually the bottleneck in such a case, however, Figure 6 clearly shows that the performance of the encrypting file systems we tested on our machine is limited by CPU speed. As expected, dm_crypt performs best with a difference of 4 to 5 MB/s compared to the baseline (i.e., native Linux with no disk encryption at all). It shows only a small performance degradation when running under the paravirtualized Linux, which is what we expected, because the dm_crypt code is running in kernel context. Compared to dm_crypt, both EncFS and VPFS perform additional cryptographic operations for their integrity checks and they also require address-space switches by design. Yet we were surprised by the low write throughput of EncFS. It seems that either EncFS or the FUSE [4] framework on which it is based has a performance bug; read operations are not affected, however, we did not look into this issue any further.

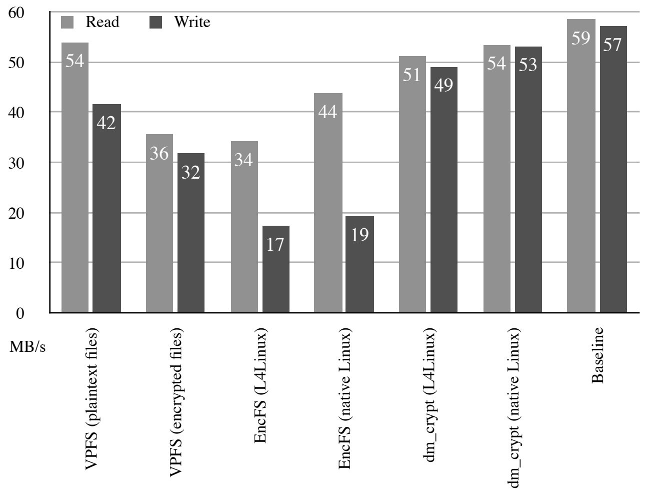

Figure 6: Layered architecture of the trusted VPFS server and the untrusted part. The VPFS server implements all security-related functionality, whereas the untrusted proxy interfaces with a legacy file-system implementation.

Nevertheless, EncFS allowed us to get an impression of the virtualization overhead introduced by L^{4}Linux: we observed a 20-percent slowdown relative to native Linux. Even though VPFS slightly outperforms EncFS on the paravirtualized kernel, we can consider the costs observed for EncFS to be comparable to the communication overhead introduced by the split architecture of VPFS. The first reason is that the paravirtualized Linux kernel runs in its own virtual address space and hence makes context switches to the EncFS user-level service equally expensive to switching between the VPFS server and the paravirtualized legacy OS. Secondly, the amount of cross address-space communication for transferring a single 4 KB data block is the same in both case: six messages.

PostMark Traces. The PostMark trace did 5,000 transactions on several hundred newly created files with a size of up to 10 KB; the amount of data written was about 20 MB, which fit into the buffer cache. We believe that this is a sensible performance test, as bursty file-system requests that can be satisfied by a buffer cache are common for many end-user applications.

[2]Our test machine was equipped with two 2.0 GHz dual-core Opteron processors (however, we used only one core), 2 GB DDR RAM in dual-channel configuration, and an 80 GB SATA hard disk (Samsung HD080HJ, maximum transfer rate of 58.5 MB/s measured using Bonnie++ benchmark).

PostMark	Time
VPFS (directories, lookup cache)	2.52 s
VPFS (directories)	2.95 s
VPFS (untrusted naming)	2.50 s
EncFS (L^4Linux)	3.57 s
EncFS (native Linux)	2.72 s
dm_crypt (L^4Linux)	0.57 s
dm_crypt (native Linux)	0.42 s
Baseline (native Linux, plaintext)	0.45 s

Table 2: PostMark trace replayed on VPFS, L^4Linux, and native Linux. The benchmark tested cache performance for a burst of transactions on many small files.

The figures in Table 2 show that VPFS and EncFS achieve only one fifth of dm_crypt's performance. On the other hand, EncFS has proven to perform sufficiently well in production environments. We therefore think that VPFS – which even outperforms EncFS – is fast enough. We intuitively expected to see a performance advantage of our untrusted-naming implementation over trusted directories. However, this is not the case if VPFS was configured to use its lookup cache. This simple caching mechanism proved to be highly effective for write-dominated workloads with many files.

Maildir Search	Cold	Warm
VPFS (directories, lookup cache)	7.54 s	2.17 s
VPFS (directories)	7.77 s	2.48 s
VPFS (untrusted naming)	7.32 s	2.12 s
EncFS (L^4Linux)	4.80 s	2.04 s
EncFS (native Linux)	4.41 s	1.76 s
dm_crypt (L^4Linux)	(23.1 s)	0.12 s
dm_crypt (native Linux)	(10.9 s)	0.07 s
Baseline (native Linux, plaintext)	3.07 s	0.6 s

Table 3: Maildir search trace. The benchmark simulates searching several maildir folders for messages; we tested performance with cold and warm buffer caches. (For unknown reasons, the underlying ReiserFS 3.6 file system did not cope well with the large number of small files in the dm_crypt test case; this did not happen in the baseline test.)

Maildir Search Traces. The maildir trace replayed by the benchmark application read more than 3,000 files from several directories; file sizes were between a few hundred bytes and several megabytes and we ran the benchmark with both cold and warm buffer caches. As can be seen in Table 3, ReiserFS has a serious problem with small files: it reproducibly spent tens of seconds with disk seek operations, possibly due to inefficient layout of the on-disk data. Interestingly, we could not observe this behavior in the baseline test, whose setup is identical except that the dm_crypt layer is not being used. We ignored this artifact, because, in a quick test under exactly the same conditions, but with Ext3 as the legacy file system, we measured performance within our expectations.

The interesting result of this read-only benchmark is that VPFS is always slower than EncFS. We determined an inefficiency in the VPFS prototype to be the cause: the VPFS server prepends a 4 KB block containing meta data to each file container. Because the vast majority of files in the data set was smaller than the block size of 4 KB, the VPFS server performed an additional call to the untrusted VPFS proxy and therefore processed two data blocks in cases where EncFS had to handle only one of them. This deficiency resulted in an average slowdown of up to 60 percent in this benchmark; at the time of this writing, the relevant parts of the VPFS code base are being adapted to eliminate the described performance problem.

Another side effect of the described deficiency is that small files require additional 4 KB of disk space in the untrusted file system. Because of the embedded hash tree, VPFS will always require more space than dm_crypt or EncFS. However, the relative overhead drops below 1 percent for files larger than 400 KB.

6. RELATED WORK

We shall now have a look at previous work in the area of cryptographic storage systems and discuss how it relates to VPFS.

Proxos [31] is a framework that allows applications originally written for commodity OSes to be isolated in VMs with only minor changes to the application source code. It redirects security-critical system calls to a specialized trusted OS, which may also provide a private file system. Proxos' private file-system implementation encrypts and hashes all file contents and – much like VPFS – stores the cryptographically protected files in the file system provided by an untrusted commodity OS running in another VM. The authors of Proxos claim that their private file system comprises less than 2,000 lines of code. However, this code still relies on a secure block device provided by the underlying virtualization layer in order to store the hash sums of the individual files. As such, a disk driver and a block-device layer becomes part of the TCB, too. Unlike VPFS, the Proxos private file system does not seem to protect any meta data except file sizes.

The Trusted Database System (TDB) [25] has security properties similar to those of VPFS. It operates on untrusted storage and ensures confidentiality and integrity of all user and meta data. It maintains a hash tree [27] spanning the whole database and stores the root node in a small tamper-resistant storage, such that it can ensure freshness. TDB implements all its functionality in a small and trusted code base consisting of approximately 6,000 lines of code; as a result, it cannot provide a feature set as rich as other database systems do. In contrast, VPFS reuses an untrusted legacy file system, such that it can offer all the functionality of a general-purpose file system based on an even smaller TCB.

The Secure Untrusted Data Repository (SUNDR) [24] is an example for a network file system that stores data on an untrusted remote server. It ensures confidentiality and integrity of the file system, including meta data. SUNDR could be used locally to provide private storage to isolated applications. However, it is not designed to ensure freshness in such a setup and its untrusted block server is targeted at multi-disk storage solutions, which mobile devices for end users cannot offer.

There is a large number of readily available solutions that protect local file storage in traditional commodity OSes. They either encrypt a complete block device or they are built on top of an existing file system. Examples for the former class are Linux' dm_crypt [10] and Bit Locker [28] in Microsoft Windows Vista. The latter approach is used for stacked file systems such as NCryptfs [34] or EncFS [20] on Linux and the Encrypting File System (EFS) introduced in Microsoft Windows 2000. However, these solutions are designed for and tightly integrated into commodity OSes. EncFS is an exception, as it is operating in user space but it requires a Linux or BSD kernel.

As such, these solutions rely on code bases that are several orders of magnitude larger then the reliance set of VPFS. Because of such enormous sizes, it is likely that these code bases contain a large number of exploitable weaknesses that an attacker can use to penetrate the system and compromise file-system security. Furthermore, the aforementioned storage systems are primarily designed to ensure confidentiality; only few such as EncFS address integrity as well, however, even those cannot provide freshness or detect the unauthorized deletion of files.

7. CONCLUSIONS

We presented VPFS, a virtual private file system designed to provide secure and reliable local storage to highly-sensitive applications running on top of a microkernel. VPFS leverages trusted computing technology and is able to ensure confidentiality and integrity – including freshness – of all file-system data and meta data. The main design goal we strived for was to keep the size and complexity that VPFS adds to the TCB of an application as small as possible. We therefore split VPFS into a small trusted part implementing only security-critical functionality and an untrusted part that reuses a file system provided by a virtualized and untrusted legacy OS.

Reusing untrusted functionality allows VPFS to provide features expected from a general-purpose file system nonetheless. To our knowledge, we are the first to thoroughly discuss how and to which extent untrusted infrastructure of a legacy file system can be reused without jeopardizing security. We implemented a VPFS prototype with *untrusted naming* and compared it to the alternative approach of implementing all naming and lookup within the TCB. In all configurations, the size of VPFS' trusted code is well below 5,000 lines of code, which is at least an order of magnitude smaller than commodity file-system stacks.

We evaluated several configurations of our VPFS prototype and found its overall performance to be acceptable and in some cases even superior to other cryptographic file systems that have weaker security properties than our system.

8. FUTURE WORK

A major goal of future work on VPFS is improved robustness in the event of an unclean shutdown, which can currently cause inconsistencies in the on-disk file system (i.e., recovery from a previously created backup snapshot may become necessary). We are exploring ways to split a journaling mechanism such that it allows for secure recovery of corrupted hash trees in file containers with most of the recovery procedure being executed by untrusted code. Furthermore, we are investigating how recently published work on user-defined dependencies at the file-system level [17] can be used for efficient ordering of write accesses in the untrusted file system so as to improve journaling performance.

9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their most valuable feedback and suggestions for improvement. Thanks also goes to the Operating System Research Group at Technische Universität Dresden, especially to Alexander Warg and Michael Roitzsch for helpful comments and productive discussions. The research on VPFS is partially funded through the European project OpenTC (EC contract number IST-027635). This work has also been supported by Intel Corp.

10. REFERENCES

[1] Berkeley DB - Oracle Embedded Database. Located at: `http://www.oracle.com/database/berkeley-db/`.

[2] Federal Information Processing Standards Publication 180-1: Secure Hash Standard. Available from: `http://www.itl.nist.gov/fipspubs/fip180-1.htm`.

[3] Federal Information Processing Standards Publication 197: Announcing the Advanced Encryption Standard. Available from: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[4] FUSE: Filesystem in Userspace. Located at: `http://fuse.sourceforge.net/`.

[5] L4HQ – The L4 Headquarters. Located at: `http://www.l4hq.org/`.

[6] L4Linux. Located at: `http://os.inf.tu-dresden.de/L4/LinuxOnL4/`.

[7] PostMark Filesystem Performance Benchmark. Located at: `http://www.netapp.com/tech_library/3022.html`.

[8] Security-Enhanced Linux. Located at: `http://www.nsa.gov/selinux/`.

[9] The Fiasco Microkernel. Located at: `http://os.inf.tu-dresden.de/fiasco/`.

[10] The Linux Kernel Archives. Located at: `http://www.kernel.org/`.

[11] The Month of Kernel Bugs (MoKB) Archive. Located at: `http://projects.info-pull.com/mokb/`.

[12] Trusted Computing Group. Located at: `https://www.trustedcomputinggroup.org`.

[13] Trusted Computing Group: TPM. Located at: `https://www.trustedcomputinggroup.org/groups/tpm/`.

[14] M. Bellare, R. Canetti, and H. Krawczyk. Message Authentication Using Hash Functions: the HMAC Construction. *CryptoBytes*, 2(1):12–15, 1996.

[15] R. Coker. Bonnie++. Located at: `http://www.coker.com.au/bonnie++/`.

[16] N. Feske and C. Helmuth. A Nitpicker's Guide to a Minimal-Complexity Secure GUI. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.

[17] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized File System Dependencies. In *SOSP '07: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 307–320, New York, NY, USA, 2007. ACM.

[18] K. Goldman, R. Perez, and R. Sailer. Linking Remote Attestation to Secure Tunnel Endpoints. In *STC '06: Proceedings of the First ACM Workshop on Scalable Trusted Computing*, pages 21–24, New York, NY, USA, 2006. ACM Press.

[19] L. Gong. A Secure Identity-Based Capability System. In *IEEE Symposium on Security and Privacy*, pages 56–65, Los Alamitos, CA, USA, May 1989. IEEE Computer Society.

[20] V. Gough. EncFS Encrypted Filesystem. Located at: `http://arg0.net/wiki/encfs`.

[21] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *Proceedings of CollaborateCom*, 2005.

[22] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB Size by Using Untrusted Components — Small Kernels versus Virtual-Machine Monitors. In *Proceedings of the Eleventh ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[23] B. Kauer. Authenticated Booting for L4, November 2004. Available from: `http://os.inf.tu-dresden.de/papers_ps/kauer-beleg.pdf`.

[24] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, San Francisco, CA, Dec. 2004.

[25] U. Maheshwari, R. Vingralek, and B. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 135–150, San Diego, CA, Oct. 2000.

[26] T. J. McCabe. A Complexity Measure. *In IEEE Transactions on Software Engineering*, SE2(4):308–320, December 1976.

[27] R. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[28] Microsoft Corporation. Secure Startup – Full Volume Encryption: Technical Overview. Available from: `http://www.microsoft.com/whdc/system/platform/pcdesign/secure-start_tech.mspx`.

[29] M.-J. O. Saarinen. Encrypted Watermarks and Linux Laptop Security. In C. H. Lim and M. Yung, editors, *Information Security Applications, 5th International Workshop*, volume 3325 of *Lecture Notes in Computer Science*, pages 27–38. Springer, 2004.

[30] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. In *EuroSys '06: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 161–174, New York, NY, USA, 2006. ACM.

[31] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, November 2006.

[32] C. Weinhold. Design and Implementation of a Trustworthy File System for L4. Master's thesis, TU - Dresden, 2006. available at: `http://os.inf.tu-dresden.de/papers_ps/weinhold-diplom.pdf`.

[33] J. Wires and M. J. Feeley. Secure File System Versioning at the Block Level. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 203–215, New York, NY, USA, 2007. ACM Press.

[34] C. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, June 2003.