Diplomarbeit

Design and Implementation of a Trustworthy File System for L4

Carsten Weinhold

23. März 2006

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dipl.-Inf. Alexander Warg

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 23. März 2006

Carsten Weinhold

Acknowledgments

I would like to thank Prof. Dr. Hermann Härtig for the opportunity to work on such an interesting and challenging project. I also want to thank my tutor Alexander Warg for his greatly appreciated support and guidance. Furthermore, my thanks go to Michael Roitzsch, Thomas Friebel, and Björn Döbel for most helpful discussions and welcome distraction, and to all other people in the Operating-System Research Group at TU Dresden who supported me.

Contents

List of Figures

List of Tables

1. Introduction

Motivation: Users store all kinds of data on their computers and mobile devices. The spectrum ranges from contact information in address books over email correspondence and digital photographs to highly critical data such as patient records. These data, and also the necessary applications, may have different security requirements, nevertheless, users often keep them on the same computer. Traditional commodity operating systems (OSes) such as Microsoft Windows and Linux store the user's data and applications in the file system and protect them using built-in access control mechanisms. To ensure data confidentiality even in case an attacker gains physical access to the computer, users can optionally utilize file-system encryption software. I shall give an overview about existing solutions when discussing related work in Chapter 3.

However, being based on isolation at the level of user accounts, these protection mechanisms may not be effective to protect sensitive data in the file system against spyware and trojan horses, which often run covertly under the user's account. Recent studies [1] showed that the risk of computers to become infected with this kind of malicious software is still real. Security reports [2] also document that not only commonly used OSes, but standard software as well, contains vulnerabilities that attackers can exploit to break into computers.

It is difficult to harden traditional commodity OSes and the various application programs that users run. These software components consist of several million lines of code, yet the user relies on their correctness with regard to the protection of his files. However, it is generally observed that, as software gets more complex, it gets less secure, too. Functional complexity is a major cause of software errors that can compromise security and error rates are even higher for device drivers included in OSes [3].

Subject of the Thesis: Given this situation, it is hard for a user to trust in the ability of the OS and its file-system implementation to protect his most critical data from attacks. Therefore, it is the objective of my work to design and implement a file system that consequently addresses the aforementioned problems.

The most important goal to reach is *trustworthiness,* hence, I chose the name *TrustedFS* for the file-system implementation that is subject of my work. Trustworthiness here means that a user, either a real person or an application program, can trust the file-system implementation to store data securely and reliably. Of course, the meaning of these adjectives depends on the specific circumstances and on the data that is to be stored. In my work, I shall focus on the protection goals (1) *confidentiality,* (2) *integrity,* and (3) *recoverability.* I shall define these terms precisely in Chapter 2.

To make the implementation secure and reliable, and thus trustworthy, it is necessary to keep its complexity low. However, if the overall complexity of the file-system implementation is to be minimized, it will most likely also have limited functionality. An alternative approach that does not impose such a limitation is based on the observation that the security-critical parts of an application often represent only a small fraction of its overall code base. So, the general idea is to split the implementation of TrustedFS into two isolated components, of which one is small and trusted and the other one untrusted. The former implements all functionality that is critical to security.

However, as I discussed previously, the complexity of the OS is relevant for overall security as well. Therefore, I chose a system architecture as the basis of my work that is specifically designed to minimize complexity. In the research paper *The Nizza Secure-System Architecture* [4], Hermann Härtig and colleagues describe an architecture that implements the previously mentioned split-application approach. The main objective of the Nizza architecture is to minimize an application's trusted computing base (TCB), which comprises all code that is essential to meet certain functional and security-related requirements.[1]

The foundation of the Nizza architecture is a small microkernel, which is supported by a set of trusted server tasks that provide basic services. Thus, the complexity of the underlying OS, which always belongs to the TCB of any application, is also small. Another important aspect of this architecture is the possibility to run existing commodity OSes *(legacy OSes)* on top of the microkernel.

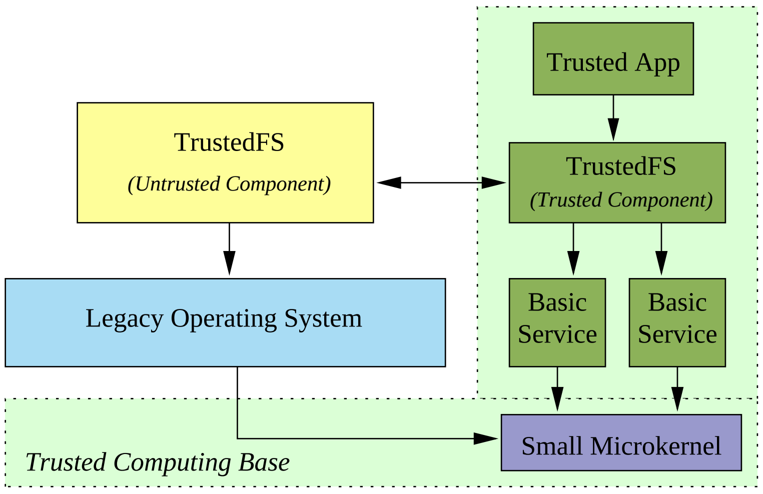

Figure 1.1.: TrustedFS based on the Nizza Secure-System Architecture.

For TrustedFS, I decided on a design as illustrated in Figure 1.1, where a minimal trusted component implements the security-critical functionality of the file system. This trusted component can reuse services provided by an untrusted part of the implementation through a *trusted wrapper*. The untrusted component runs on top of the legacy OS, thus, it can reuse the already existing functionality of the legacy OS including its file system. That is, the legacy OS can provide untrusted storage for all file-system contents.

For obvious reasons, the trusted component has to protect the contents of the file system in untrusted storage, otherwise the user's data could be subject to successful attacks regardless of the splitting of the code base. It can ensure confidentiality and integrity using cryptographic means such as encryption and collision-resistant hash functions. However, an attacker who gained control over the legacy OS can still delete the encrypted data. To make sure that the file system can be restored in such a case, TrustedFS must be able to create backups.

[1] For timing-critical applications, the TCB must also ensure timeliness.

Outline: In Chapter 2, I shall elaborate on the requirements that have to be met so that TrustedFS can reach the aforementioned protection goals. I shall also discuss how to leverage trusted-computing technology such as authenticated booting and sealed memory for this purpose. Related work and cryptographic basics are subject of Chapters 3 and 4. Chapter 5 will discuss the design of TrustedFS and how to use the cryptographic algorithms. Chapter 6, gives more insight into important aspects of the implementation, which I shall evaluate with regard to performance and complexity in Chapter 7. Finally, I conclude my thesis with a summary of the achieved results after discussing future work in Chapter 8.

2. Basics

2.1. Context of the Thesis

The starting point for my work on a trustworthy file system is the implementation of the Nizza architecture that is being developed by the Operating-System Research Group at TU Dresden. It is based on the L4/Fiasco microkernel [5], which supports L^{4}Linux [6], a para-virtualized Linux, running on top of the microkernel as a legacy operating system (OS). As I already outlined in the introduction, the trusted component of the file-system implementation will take care of protecting file-system contents by using cryptography. The untrusted component can reuse existing infrastructure provided by L^{4}Linux. In particular, it is a design goal to allow trusted L4 applications to reuse the file system of an untrusted L^{4}Linux instance through a trusted wrapper.

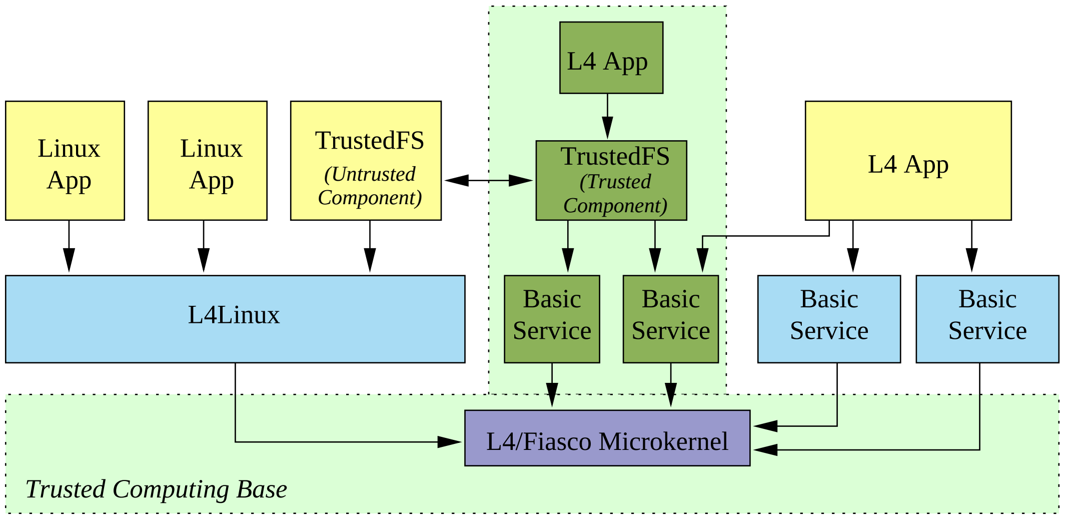

Figure 2.1.: TrustedFS based on the Nizza Secure-System Architecture using L4/Fiasco and L^{4}Linux.

2.2. Protecting Data in Untrusted Storage

To be trustworthy, a general-purpose file system has to make sure that the following protection goals can be reached:

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

Integrity: Any data that the file-system implementation provides to the user is correct, complete, and up to date. If the data lacks either of these properties, then the implementation must be able detect this.

Recoverability: In case of a modifying attack or data loss after a system failure, the file-system contents can be restored to a previous valid state. Changes to the file system that were made after saving this state may be lost after recovery.

By ensuring confidentiality and integrity as defined previously, TrustedFS does not impose any limitation on the type of data that it can store securely. However, the third protection goal, recoverability, is weaker than the the more general goal *availability.* The latter means that, with regard to file-system contents, an authorized user can access data *whenever* he needs to. For obvious reasons, any design that uses untrusted components is unsuited to ensure availability, because untrusted components may be subject to denial-of-service attacks. So, the best that TrustedFS can achieve is recoverability, on which I shall elaborate in Section 2.4.3.

Other goals, such as *real-time capability* or *confinement* of information, are not subject of my work. Being based on the reuse of untrusted components, the described design does not even meet the necessary requirements. Neither can it ensure timeliness, which is a prerequisite for real-time capability, nor is it possible to prevent unwanted flow of information between trusted and untrusted components, as required for confinement.

With all file-system contents being stored in the untrusted storage provided by the legacy OS, the trusted component is responsible for protecting this data with regard to the protection goals. To ensure confidentiality, it must encrypt the contents of the file system and keep the encryption key secret.

To detect unauthorized modifications of the file-system contents, the trusted component uses collision-resistant hash functions, on which I shall elaborate in Section 4.2. This kind of hash functions allows to calculate small checksums *(hash sums)* of large amounts of data in such a way that even the slightest modification to the data is detectable with high probability:

$$hash\_sum := H(data)$$

Modifying the input data of the collision-resistant hash function H, for example manipulating an encrypted file in untrusted storage, causes the hash sum to change. Thus, the trusted component can ensure integrity, if it can store a reference copy of the correct hash sum securely.

2.3. Attacker Model

2.3.1. Software-based Attacks

With regard to software, the Nizza architecture partitions the execution environment into the trusted computing base (TCB) and untrusted code. Figure 2.1 on page 5 gives an overview about trusted and untrusted components relevant for TrustedFS. Particularly, the following components belong to the TCB:

- Microkernel
- Memory pagers
- Application loader

- Authenticated-booting, remote attestation, and sealed-memory services
- Trusted file-system component

Of course, the user must also trust the applications he uses to access the file system. As TCBs are application specific, any other components running on the same system are untrusted. Nevertheless, they may be part of the TCBs of other applications. In the remainder of my thesis, I shall also use the term *trusted domain* to denote both the TCB and trusted storage, whereas *untrusted domain* denotes untrusted software components and untrusted storage.

Having two separate domains of trust only makes sense, if an attacker can be restricted to the untrusted domain. Therefore, I assume that the attacker cannot penetrate the TCB or access data in trusted storage. However, attacks can completely compromise untrusted components, even the legacy OS and its kernel. For example, an attacker could gain remote access to the L^{4}Linux instance using a trojan horse. If he has physical access to the computer, he might be able to circumvent Linux' security barriers even more easily. So, assuming that an attacker can do virtually anything inside the untrusted domain, the following attacks on the file system have to be considered possible:

- **Attacks on Untrusted Software Components:** Untrusted components can be modified to try to exploit weaknesses in trusted components, 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 who is in control of the untrusted domain can modify or delete this data.

These attacks can happen at any time, even while the file-system implementation is active. However, with the protection goals defined in Section 2.2 in mind, this attacker model leaves an attacker operating in the untrusted domain with a denial-of-service attack as his only way to cause damage.

2.3.2. Hardware-based Attacks

As I already indicated in the introduction, TrustedFS, and also the Nizza architecture itself, relies on trusted-computing technology, which in turn requires a trusted platform module (TPM) as specified by the *Trusted Computing Group* [7, 8]. In Section 2.4.2, I shall discuss this requirement in greater detail. Regarding possible attacks on hardware components, I assume that the TPM is tamper resistant[1], which means an attacker cannot compromise it without substantial effort.

Because the file system is also intended to be used in mobile devices, physical destruction or theft of such a device must be considered a likely threat as well.

2.4. Platform Requirements

2.4.1. General Requirements

The concept of TCBs requires trusted software components to be isolated from untrusted ones. Therefore, the trusted and the untrusted part of the file-system implementation

[1] Apart from the TPM, other hardware components, such as the CPU and the memory subsystem, need to be trusted, too.

must live in distinct address spaces. Nevertheless, they must still be able to communicate with each other. Because both components need to transfer large amounts of data between their address spaces, they also need to be able to share a certain amount of memory[2] to avoid unnecessary copy operations.

With regard to TrustedFS, trust is mutual. Not only the user must be able to put trust in the file-system implementation, but the implementation needs to trust its users as well. Otherwise, it cannot fulfill the user's expectation to store data securely. Therefore, the system platform must provide user-authentication infrastructure, so that only authorized and thus trusted users can access the file system.

2.4.2. Authenticated Booting, Remote Attestation, and Sealed Memory

In Section 2.3, I derived an attacker model that is based on the assumption that any software component or storage, except those in the trusted domain, can be subject to successful attacks. This assumption will hold as long as the microkernel, the base services, and the trusted file-system component and its user applications can withstand software-based attacks originating from untrusted components. Address spaces and well-defined communication interfaces allow for this kind of robustness, if the complexity of the trusted software components is small enough, so that programmers can create a secure implementation.

However, an attacker who has physical access to the computer, either a desktop computer or a mobile device, can modify the trusted software stack. He might even be able to access sensitive data in trusted storage, such as the encryption key for the file system, if it is not protected by additional measures. Furthermore, a legitimate user might, if he is not aware of his computer being manipulated, disclose even more information, for example passwords.

Authenticated Booting and Remote Attestation. To be able to protect trusted software components, the system platform requires hardware support. Specifically, the Nizza architecture relies on a TPM, as these hardware components combine relatively low costs with good security properties. The TPM is used as a basis to implement *authenticated booting* and *remote attestation*. Authenticated booting cannot stop the previously described attacker from tampering with trusted software components.[3] However, it allows, when used in conjunction with remote attestation, for a user to determine whether a computer indeed runs the software he expects.

In [9], Bernhard Kauer described design and implementation of a TPM-based infrastructure for L4, which provides authenticated booting, primitives necessary for remote attestation, and sealed memory.

Sealed Memory. Being able to protect trusted software, which does not change often, is not sufficient. TrustedFS, and other components as well, need to store frequently changing data persistently. Although it uses untrusted storage for file-system contents, there is also data that the trusted component cannot encrypt and store outside the trusted domain:

[2] Memory that a trusted component shares with an untrusted component must not be used to store unprotected data.

[3] A similar technology, *secure booting*, can enforce that only certified software runs on a computer. However, secure booting makes the user completely dependent on hardware and software manufactures, whereas authenticated booting allows him to decide for himself whether or not to trust a software stack.

- The secret encryption key that is used to ensure confidentiality.
- The cryptographic hash sum to verify the integrity of the file-system contents.

This data requires only little storage capacity, a maximum of 32 bytes each. To comply with the aforementioned attacker model and with the Nizza architecture's demand for minimal TCBs, the system platform must provide sealed memory for this kind of data. Sealed memory enforces access restrictions to stored data, so that only authorized applications running on top of an authorized OS can use it. Therefore, sealed memory can only work with either secure booting or authenticated booting.

A sealed-memory implementation does not necessarily have to store data in nonvolatile hardware-implemented storage. TPM-based solutions, such as the implementation mentioned in [9], encrypt all data using an application-specific encryption key, which is derived from a key private to the TPM and trusted information about the running software stack including the application itself. If an attacker modifies a component in the software stack, the sealed-memory implementation can no longer decrypt sealed data, thus leading to detectable denial of service.[4]

With this approach, applications can store private data of arbitrary size in untrusted storage. For example, TrustedFS can easily use the legacy OS's file system for this purpose. However, encryption does not protect against *replay attacks* on sealed data. That is, a trusted application cannot detect that it is using outdated data, if an attacker replaced it with a previous version. The property that data is up to date is often referred to as *freshness* [10, 11]. To allow for freshness guarantees, the sealed-memory implementation can provide applications with per-application version numbers for sealed data. Provided that these version numbers cannot be manipulated by an attacker, an application can always verify whether it got an up-to-date version of its private data unsealed. In principle, a sealed-memory implementation could also encapsulate this functionality behind its application programming interface.

For TrustedFS, the system platform must provide both an authenticated boot process and sealed memory. The latter also needs to be resistant against replay attacks. Using this technology, access to the most critical data, the file-system encryption key and the anchor to ensure integrity, can effectively be restricted to a minimal TCB. Assuming a sufficiently secure user-authentication mechanism, file-system contents are even secure, with regard to confidentiality and integrity, if an attacker gains physical access to the computer.

2.4.3. Infrastructure required for Recoverability

Reusing the untrusted legacy OS's file-system infrastructure allows to minimize TCBs, however, it also leaves the file-system contents with cryptography as their only protection. An attacker can still damage the data in untrusted storage, as this kind of attack on availability cannot be prevented by cryptographic means. To ensure recoverability, TrustedFS must keep a backup copy of all file-system data in an external trusted storage.

I expect my file-system implementation also to be used in mobile devices, which can get lost or stolen. So, the physical location of the trusted external storage has to be distinct from that of the mobile device. Still, the backup copy in the external storage has to be synchronized with the file-system contents on the computer. Nowadays, mobile devices

[4] The TPM can only unseal the data, if its *platform control registers* contain the hash value of the software stack's accumulated binary code.

are connected to the Internet, so a trusted server seems like an optimal solution to provide the required storage.

Now, the important question is: What kind of trust does a user have to put in the server? A naive answer would be that this trust must be ultimate, because, in case the device including the encryption key is no longer available, the server must store the backup either as plaintext or in encrypted form together with the encryption key. However, the trusted server would still have to ensure that only authorized users can retrieve a backup, so all users were required to prove their authorization using a credential. For such a credential, other means of backup would be required.[5]

I therefore assume the availability of an alternate backup mechanism for small amounts of static data such as the aforementioned credential, or even for the file-system encryption key. For example, the user could keep a backup copy on a CD or USB stick.

As a consequence, a user just needs to trust the server to store an encrypted backup reliably, which means ensuring the backup copy's availability. By using the same method as to ensure integrity of file-system contents in untrusted storage, the trusted component of the file-system implementation can also verify correctness and completeness of a recovered file system. To make sure that the file-system contents are up to date, which is the third requirement for integrity, it needs to store an additional hash sum for the latest backup in sealed memory. If a file system needs to be recovered, and the data stored in sealed memory is lost, too, a user also needs to trust the server always to offer the latest backup.

Again, the trust relationship between a user and the server, in which he puts his trust, is mutual:

1. A user must trust the server to store backups reliably.
2. The server must trust the user to send only valid backups, otherwise it cannot fulfill the first requirement.

The second requirement implies that only the trusted component of TrustedFS can be allowed to send data to the server. Therefore, both sides need to authenticate each other. In addition to exchanging credentials, mutual attestation of the software stacks running on the server and the user's computer can enhance security further.

Communication between the trusted component and the backup server requires a trustworthy communication channel. Based on the design principle of trusted wrappers, it is possible to reuse untrusted network infrastructure for this purpose as well. However, in the context of my thesis, I shall discuss neither the details regarding the communication infrastructure, nor shall I elaborate on design and implementation of the trusted backup server. Nevertheless, I shall specify interfaces and requirements that the aforementioned components have to meet so that recoverability can be ensured.

2.4.4. Random Numbers and Key Generation

The degree of security that TrustedFS can offer does not only depend on its design and implementation, but also on the quality of the encryption key being used. Cryptographic

[5] Of course, the credential could also be a password, but then an attacker could compromise confidentiality, integrity, and availability of the backup copy, if he knew this password. Because a significant number of users choose relatively weak passwords, the trusted server, which is always exposed to any attacker who has access to the Internet, would become the weakest link.

keys are generated from random numbers, so that they are hard to guess. Thus, to generate the secret encryption key, the trusted file-system component needs a source for high-quality random numbers. In [12], Hans Marcus Krüger investigated ways to provide L4 applications with random numbers that can be used for cryptography. He designed and implemented an L4 server that can, among other applications, be used for TrustedFS.

3. Related Work

3.1. Protected Storage

Existing solutions that provide cryptographically protected storage are based on various approaches. Low-level solutions, such as Linux' Crypto Loop or its successor dm_crypt [13], provide transparently encrypted containers. These containers can host any file system supported by the operating system (OS). BestCrypt [14], which is also available for Microsoft Windows, is one of many commercial solutions that work essentially in the same way. However, none of the available solutions can ensure the integrity of the encrypted data.

Another commonly used approach is to add a security layer on top of an existing file-system implementation. The Cryptographic File System (CFS) [15] for Unix reuses infrastructure of the Network File System (NFS). Its main component is an NFS server that runs on the user's computer and transparently encrypts filenames and file contents. The Transparent Cryptographic File System (TCFS) [16] is essentially a network file system. It is implemented as a kernel-space NFS client, which provides access to encrypted files on a, not necessarily remote, NFS server. In contrast to CFS, this file system can also ensure data integrity by using cryptographic hash functions. However, it cannot detect replay attacks on block or file level.

NCryptfs [17] and EncFS [18], which are only available for Linux, implement a more direct approach, as they are wrappers for local file systems. The latter operates in user space and can ensure confidentiality and limited integrity. That is, EncFS can detect modifications inside a block, but it will accept any block or even file that has been replaced by an older version of itself. NCryptfs is a kernel-space extension that ensures confidentiality. In a technical report [19], the authors briefly describe how to ensure integrity using cryptographic hash functions. However, I suspect that their implementation is susceptible to replay attacks as well.

The general idea of extending an existing file-system implementation with cryptographic security mechanisms is similar to the trusted-wrapper approach that I chose for TrustedFS. However, the aforementioned solutions do not meet all security requirements that I set for my implementation.

Microsoft Windows 2000 has been the first release of Windows with built-in support for cryptographic file protection. However, the Encrypting File System (EFS) only takes care of transparent encryption. The upcoming Windows release, Windows Vista, will offer two features called BitLocker and Secure Startup [20]. The former implements transparent encryption of the system partition, whereas the latter intents to ensure the integrity of the boot process. Using secure booting based on a trusted platform module (TPM), Secure Startup protects the master encryption key, and thus the EFS keys as well, and the integrity of the boot loader. The available documentation [20] vaguely mentions that, once the boot loader passed control to the validated OS, the OS will check the integrity of each executable that is to be run. However, the documentation does not specify any details.

BitLocker and Secure Startup only address off-line attacks. In contrast, TrustedFS is designed to handle on-line attacks on untrusted storage as well.

There are also solutions that, just like TrustedFS, have been designed specifically to operate on untrusted storage. The Protected File System (PFS) [21] is an in-kernel extension for existing journaling file systems. It can ensure integrity, but not confidentiality, of user and meta data. PFS is intended for use on servers that store file systems in an external untrusted storage, such as a storage area network. It is vulnerable to replay attacks.

The Secure Untrusted Data Repository (SUNDR) [10] is a network file system that stores data on untrusted remote servers. It ensures confidentiality as well as integrity of the file system, including meta data. SUNDR guarantees *relative freshness* of file-system contents. That is, it allows a user to view all changes made by other users of the distributed file system, or, if the server misbehaves, no changes at all.

SiRiUS [11] is another network file system operating on untrusted remote storage. It implements access control using cryptographic means and it can ensure confidentiality and integrity. SiRiUS also guarantees the freshness of meta data. That is, it can detect replay attacks.[1] Under certain circumstances, when multiple clients wrote to the same directories, an extended version called SiRiUS-U is able to ensure freshness of user data in these directories as well.

PFS, SUNDR, and SiRiUS operate on remote storage and assume the local system to be fully trusted, whereas TrustedFS only relies on the integrity of the microkernel-based OS, but not on the legacy OS. Being based on traditional commodity OSes and platforms, the security that any of the aforementioned solutions can provide is limited:

1. All previously mentioned solutions are designed for traditional monolithic OSes. Thus, they rely on large trusted computing bases (TCBs) in the magnitude of several million lines of code. Because of their enormous size, it is likely that the TCBs contain exploitable weaknesses, which an attacker can use to penetrate the system, and thus compromise file-system security.

2. Solutions that purely rely on traditional local storage, such as NCryptfs and EncFS, cannot ensure freshness. They have no way to store integrity information in such a way that it is secure from software-based attacks, in this case replay attacks.

TrustedFS consequently addresses these problems. The Fiasco microkernel and the base services of the Nizza architecture consist of merely tens of thousands of lines of code.[2] With the trusted component itself being minimal, it is much more realistic that the TCB can be made secure. By storing integrity information in sealed memory, as I specified in Section 2.4.2, TrustedFS can detect replay attacks, and thus ensure freshness of file-system contents.

[1] The freshness of the meta data is essential to prevent users from regaining previously revoked access rights, which are granted by per-user encrypted keys included in the meta data. These keys are necessary to decrypt file contents or to generate valid signatures upon write accesses.

[2] These figures appear in *The Nizza Secure-System Architecture* [4].

The Trusted Database System (TDB) [22] addresses the aforementioned problems regarding small TCBs and secure storage for critical data as well. It operates on untrusted storage and ensures confidentiality and integrity of all user and meta data. TDBs integrity guarantees also include freshness, for which the design explicitly requires a small tamper-resistant storage.[3] This storage contains a single hash sum that authenticates the root node of a hash tree [23], which in turn authenticates the whole database. To protect itself from software-based attacks, TDB also requires a trusted processing environment. In the TDB paper [22], Maheshwari and coworkers mention a dedicated CPU and memory, however, a TPM-based solution using authenticated or secure booting can ensure similar protection.

TDB does not provide functionality as rich as offered by other database systems. In contrast, TrustedFS implements a trusted wrapper, which allows it reuse as much as possible of available legacy infrastructure, so that it can offer rich functionality with a minimal TCB.

The Trusted Computing Group (TCG) [8] currently works on a specification for trusted storage devices. At the time of this writing, the TCG Storage Work Group [24] has not yet published this specification. Among others, it is expected to cover use case such as (1) protection of sensitive data, (2) encryption of data for a specific host, (3) logging for forensic purposes, and (4) secure firmware upgrades.

3.2. Backup and Recovery

TDB performs incremental backups of the database in untrusted storage. It creates consistent snapshots including integrity information that it can use for recovery. However, the backup storage is untrusted as well and it must be permanently available during operation. An attacker could delete data in both storages to destroy the database. In contrast, TrustedFS creates consistent snapshots on a trusted backup server as described in Section 2.4.3. Thus, it can ensure recoverability.

SUNDR uses an untrusted server to store file-system contents and relies on the server operator to take care of backups. Upon recovery, SUNDR can also restore parts of the file system from client-local caches, if the restored backup on the server is too old. SiRiUS relies on server operators to create backups as well, for example using standard backup software. Nevertheless, both SUNDR and SiRiUS make no explicit assumptions on availability or recoverability.

pStore [25] is a secure peer-to-peer backup system using storage provided in a network of untrusted peers. It uses cryptographic means to ensure confidentiality and integrity as well as to implement access control and per-user name spaces. pStore splits files into chunks that are replicated on a certain number of peers. Only authorized users, who are in possession of a secret key, can create valid chunks in their name space or delete existing ones from it. In the pStore paper [25], Batten and coworkers performed a reliability study of their work. It showed that, when distributing four copies of each chunk among the remote peers, at least 25 out of 30 nodes need to be available to reach one hundred percent recoverability. Although increasing the number of copies will allow for more nodes failing, it also means increased bandwidth requirements for backup.

[3] Alternatively, TDB can use tamper-resistant hardware-implemented version counters to check whether integrity information is up to date.

The Distributed Internet Backup System (DIBS) [26] stores backups in a peer-to-peer network as well. It assumes storage provided by peers to be untrusted with regard to confidentiality and integrity. DIBS encrypts and digitally signs files, which it then distributes to all participating peers. It bases its recoverability guarantees on Reed–Solomon error correction [27]. Each peer receives a differently encoded chunk of a file, such that it is possible to reconstruct the whole file, if at least half of the originally distributed chunks are available at recovery time.

The DIBS approach requires less network bandwidth than pStore-like solutions. Nevertheless, distributed backup strategies require a significant number of peers to be available for both backup and recovery. These approaches might be well suited for peers in a local area network. However, because a backup and recovery solution in the context of my work would need to ensure full recoverability for mobile devices as well, the reachability of a critical mass of peers could be a relevant problem. The solution using a dedicated trusted backup server that I chose for my work also requires less network bandwidth.

TPM key backup is a problem on a lower level, nevertheless, it is related to my work because TPM-based authenticated booting, remote attestation, and sealed memory are prerequisites for my file-system implementation.

A TPM has only a small nonvolatile storage, which cannot store all cryptographic keys and all sealed data that are needed by the OS and the applications. Therefore, most of these keys and the sealed data remain, cryptographically protected by root keys residing in the TPM, in an untrusted persistent storage. Obviously, it is necessary to back up the data in this storage as well. Furthermore, it must be ensured that the encrypted keys and the sealed data can be recovered even if the TPM containing the root keys is no longer available. The TCG conceptually solved this problem by allowing TPM keys to be migrated to other TPMs, or to be exported for backup purposes. In the *Interoperability Specification for Backup and Migration Services* [28], the TCG specifies means for key backup.

Regarding sealed application data that does not change often, the user can maintain backup copies in a safe place. For example, he might keep the sealed encryption key for the trustworthy file system on commodity backup media, which he can lock into a vault or give to a trusted person.

3.3. File Access on the L4 Platform

The L4 Virtual File System (L4VFS) [29] implements an input–output infrastructure with POSIX-like semantics. It consists of a set of client-side libraries and servers, which provide objects in a hierarchical name space. Applications can access files provided by L4VFS file servers using functions like `open()` and `read()`. Currently available file servers allow either read-only access to persistently stored files or read–write access to temporary files residing in main memory. However, these servers cannot ensure confidentiality and integrity. Nevertheless, the L4VFS infrastructure is suitable to allow existing applications to use the trustworthy file system, if the implementation provides the necessary L4VFS server interface.

4. Cryptography to Protect File-System Contents

4.1. Confidentiality

Almost all the storage solutions that I presented in Section 3.1 use a symmetric block cipher, such as the Advanced Encryption Standard (AES) [30], Blowfish, Twofish, or Triple-DES, to keep data confidential. These algorithms have been subject to extensive review and are considered secure. However, block ciphers can only operate on small data blocks, usually with a length of 128 bit. To overcome this limitation, you can use the block cipher in *cipher-block–chaining* mode (CBC mode), which is also what most encrypting storage systems do. Disk sectors, or file-system blocks, are divided into n cipher blocks that are encrypted as follows:

$$\begin{aligned}
&c_0 := E_k(p_0 \oplus iv) \\
&c_1 := E_k(p_1 \oplus c_0) \\
&c_2 := E_k(p_2 \oplus c_1) \\
&\ldots \\
&c_n := E_k(p_n \oplus c_{n-1})
\end{aligned}$$

c_i is the ciphertext block created by encryption algorithm E using key k. The input data of each iteration is the result of the *xor* operation applied to the plaintext block p_i and the previous ciphertext block c_{i-1}. For the first iteration, an initialization vector (IV) is used instead of the previous ciphertext block, which would be part of the previous sector. Decryption works reversely.

If the IV being used is of high quality[1], a block cipher in CBC mode allows to securely encrypt any data block with a size that is a multiple of the cipher-block size. However, many implementations of encrypting storage systems use weak IVs. For example, the widely used Linux–Crypto-Loop implementation encrypts disk sectors using their plain sector numbers as IVs, thus making itself vulnerable to related-IV attacks[2] such as *watermark attacks* [31]. By placing a number of bit patterns (watermarks) in a file, an attacker can detect the presence of that file in an encrypted container with high probability, although he does not know the encryption key.

To prevent related-IV attacks, an encrypting storage system can make use of a cryptographic hash function to calculate IVs. Hashing not only the sector number n, but also a secret salt s, makes it impossible for an attacker to predict the IV for a certain block based on its sector number:[3]

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

[1] Ideally, an initialization vector is used only once, so that encrypting the same data several times always results in different ciphertexts.

[2] Related-IV attacks are based on the knowledge, how IVs change from one encryption run to another.

[3] Hash functions take byte streams as their input data. The $\parallel$ sign denotes concatenation.

If the cryptographic hash function has the *Avalanche Property*[4], an attacker who does not know the secret salt s is also unable to tell how one IV differs from another, thus he cannot mount a related-IV attack. Newer versions of the Linux-dm_crypt implementation, which is the successor of Crypto Loop, use this approach with the secret salt being the encryption key.

For TrustedFS, I decided to use the CBC mode in conjunction with the IV-generation method described previously and with AES as the block cipher. AES is widely accepted, considered both secure and fast, and there are open source implementations available. Another important reason is that hardware vendors started to implement instructions to accelerate basic AES operations into their processors, creating the potential for a considerable performance gain.

AES officially supports key sizes of 128, 192, and 256 bits. Guidelines published by the U.S. NIST in [32] recommend any of these key sizes to be used for encryption for at least the next ten years.

4.2. Integrity

The attacker model that I derived in Section 2.3 implies that the encrypted file-system contents in the untrusted storage cannot be protected from unauthorized tampering. The only protection the trusted component of the file-system implementation can achieve, is that it detects any unauthorized modification, so that falsified data will not be used.

To confirm the authenticity of any data that is read from untrusted storage, the trusted component must calculate a checksum of the data and check whether it matches a reference copy. The reference copy is critical to provide integrity, therefore, it must be stored in a secure nonvolatile storage, such as sealed memory. I discussed this requirement in Section 2.4.2.

With the reference checksum being stored securely, an attacker operating outside the trusted domain cannot exchange it. However, it is also necessary to ensure that he cannot modify file-system contents in such a way that the resulting checksum remains unchanged. Collision-resistant hash functions being used to calculate the checksum meet this requirement. Another approach to detect unauthorized modifications of file-system data would be to use digital signatures [33]. However, the main advantage of digital signatures, allowing to identify the party that issued a signature, is not relevant in my work. Also, because digital signatures are based on asymmetric cryptographic algorithms, they have higher run-time costs in terms of CPU time. Therefore, I decided to use a collision-resistant hash function to ensure file-system integrity.

Specifically, I chose SHA-1 [34] from the Secure Hash Algorithm family. SHA-1 provides good performance and it is widely used and well studied. At the moment, it is considered to be secure, however, there have been reports about a successful attack on SHA-1. In [35], Wang and colleagues claim to have developed a collision attack[5] that has a lower computational complexity than a brute force attack. For the purpose of authenticating

[4] In cryptography, the avalanche effect means that small changes in a cryptographic algorithm's input data cause its output to change significantly. The *Strict Avalanche Criterion* requires that a one-bit change in the input data results in each bit of the output to change with a probability of 0.5.

[5] In a collision attack, an attacker tries to find two different messages that have the same hash value.

file-system contents, it is important that the hash function in use is not vulnerable to preimage attacks.[6]

Currently, the aforementioned attack does not seem to weaken SHA-1 with regard to preimage attacks. Also, to mount a preimage attack, an attacker must be able to choose the whole message he wants to fake. However, if the message also consists of a secret part that is unknown and inaccessible to the attacker, preimage attacks are no longer possible. This construction effectively turns SHA-1 into a keyed hash function similar to HMAC [36].

In Section 5.2.2, I will present a design for the cryptographic part of the trusted file-system component that uses SHA-1 in this way to ensure integrity. SHA-1 also has all the properties that are necessary so that the trusted component can use it to calculate initialization vectors as described in Section 4.1.

Should further research point out additional weaknesses in SHA-1, it is recommended to replace it with a stronger hashing algorithm, possibly SHA-256, SHA-384, or SHA-512. However, all currently available trusted platform modules also use SHA-1. So, at the present time, using a more secure algorithm for the TrustedFS would not enhance security in general.

Using two different algorithms to provide for confidentiality and integrity, for example AES in CBC mode (AES-CBC) and SHA-1, is often referred to as the *generic-composition approach*. But there also exist modes of operation for block ciphers that take care of both encrypting and authenticating the processed data. These *authenticated encryption modes* can be divided into two classes: (1) *One-pass modes* and (2) *Two-pass modes*. They represent an alternative to generic composition, nevertheless, I decided not to use such a mode in the implementation of TrustedFS for various reasons. Two-pass modes are slower than AES-CBC and SHA-1, as they double the number of block-cipher operations. One-pass modes, such as Offset Code Book (OCB) mode can outperform an implementation based on generic composition, but most of them are patented. Galois/Counter mode (GCM) is free of such intellectual property claims, but its performance is comparable to AES-CBC and SHA-1.

I also came to the conclusion that authenticated encryption modes provide less flexibility than the generic-composition approach, which allows encrypting and hashing to be performed independently from each other. For example, if the contents of a certain file do not need to be kept confidential, the generic-composition approach allows to omit encryption and decryption, thus improving performance significantly.

[6] In a preimage attack, an attacker tries to find a message that, when hashed, results in a given hash value. Preimage attacks are more difficult than collisions attacks, in which any two messages with the same hash value constitute a collision.

5. Design

5.1. General Design

Splitting the implementation of TrustedFS into a trusted and an untrusted part is vital to minimize the trusted computing base (TCB). The untrusted component can provide all functionality that is not critical for maintaining security. To take advantage of existing legacy software, it should also reuse as much of the available infrastructure as possible. The trusted component, on the other hand, is responsible for ensuring confidentiality, integrity, and recoverability, as stated in Section 2.2. That is, it must (1) encrypt and decrypt file-system contents, (2) be able to detect unauthorized modifications to the file system, including replay attacks, and (3) it must send backups to the trusted backup server. It should not implement functionality that is not necessary with regard to these requirements. However, file-system performance cannot be ignored, so every design decision is also a tradeoff between a minimal TCB and reasonable performance.

Untrusted Component: Currently, L^{4}Linux is the only legacy operating system (OS) available on the L4 platform.[1] Processes running inside an L^{4}Linux instance and L4 tasks can communicate using L4 inter-process communication (IPC). Therefore, I decided to implement the untrusted component, the *untrusted block server*, as an L^{4}Linux process. Thus, it can reuse the Linux file system as untrusted storage.

Trusted Component: To allow TrustedFS to be used for a variety of purposes, I chose to encapsulate the implementation of the trusted part in a server library. A front end, which is not part of the library itself, provides a specific application programming interface (API). The library can be used as the basis of an L4 server and it is even possible to link the library directly into an application, giving it exclusive access to the file-system contents.[2]

Figure 5.1 gives an overview of the trusted component's layered architecture. As indicated in this figure, the trusted component also implements a buffer cache. It adds considerable complexity to the TCB, but these costs cannot be avoided. A buffer cache holding decrypted and authenticated data can reduce the number of data transfers between the trusted and the untrusted domain to a great extent. Thus, it allows to reduce the run-time overhead introduced by the cryptographic operations, which need to be performed whenever data enters or leaves the trusted domain. Nevertheless, a cache is also necessary to reduce the run-time costs to ensure integrity to an acceptable level, as I shall explain in Section 5.6.1.

Reusing Untrusted Storage: The probably most important design decision is the one regarding the way how the untrusted storage is to be used. There is a whole spectrum of

[1] Unmodified legacy OSes can run inside virtual machines, as provided by *L4VM*, but there is currently no way for L4 tasks to communicate with processes running in such a virtual machine.

[2] In this scenario, access control is enforced by the authenticated boot process. Controlling access to the files provided by a separate file server requires additional measures.

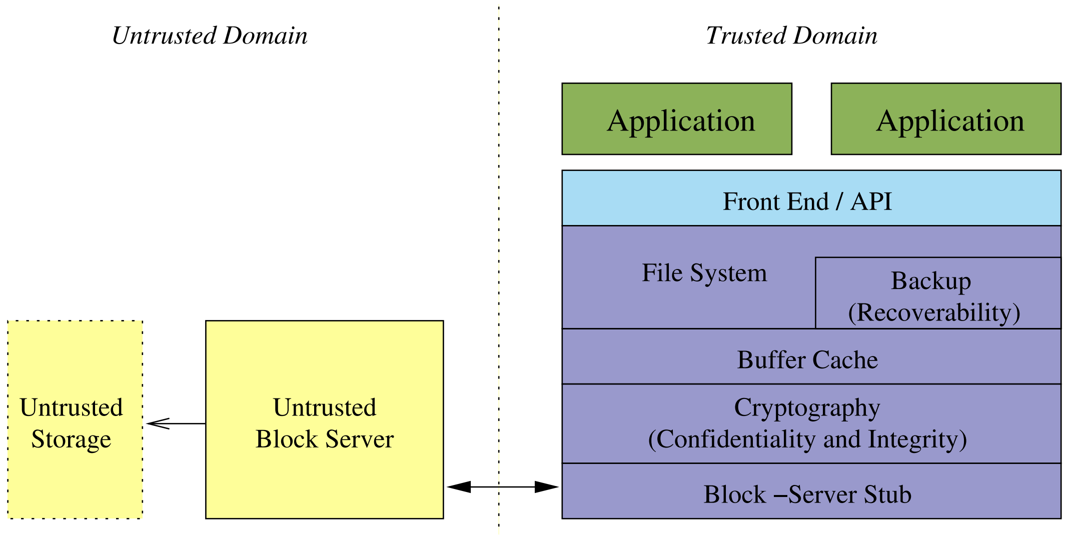

Figure 5.1.: Layered Architecture of the Trusted File-System Component.

design alternatives. The following two approaches are extremes, of course, other solutions that are a mixture of both exist as well:

1. **Complete File System in the Trusted Component:** The trusted part implements all functionality needed to provide files and directories, which are stored in a single container in the untrusted domain. The legacy OS only takes care of persistently storing the container on disk, thus only its disk driver is reused. Such a solution is similar to Linux' Crypto Loop or dm_crypt [13].

2. **Minimal File-System Wrapper in the Trusted Component:** A file-system wrapper in the trusted part only ensures confidentiality, integrity, and recoverability of the file-system contents. The legacy OS takes care of how to store files and directories and may also provide file lookup routines. This alternative is much like the approach taken by EncFS [18] on Linux.

Any of the design alternatives uses encryption and cryptographic hash sums to ensure confidentiality and integrity. The first approach requires only the file system container to be protected, whereas the second one has to provide confidentiality and integrity for indiviual files and also for the directory structure.

It is a complex task to design and implement a new file system, especially, if it is expected to provide high performance. By adapting an existing code base instead, you can reduce the amount of time that is needed. However, both size and complexity of the trusted component will grow considerably, if it contains a complete file-system implementation.

Reusing the file system of the legacy OS according to the second approach will result in a less complex trusted component. Such a design still allows efficient read and write accesses to files and it is also possible to reuse existing functionality for directory handling and file lookup. On the downside, the number of files is revealed, as well as their sizes and location in the directory hierarchy. Special care must also be taken to ensure the integrity of the directory tree itself.

Reusing the untrusted legacy OS including its file-system infrastructure via a trusted wrapper seems more promising with regard to the goal of minimizing the TCB. Therefore, I chose to use a design that is close to this approach as the basis of TrustedFS. In the following sections of this chapter, I shall examine to which extent the legacy OS can be reused and where limitations prevent reuse of certain functionality.

5.2. Protecting Individual Files

TrustedFS allows files to have an arbitrary size, so they cannot always fit into the buffer cache. Thus, the trusted file-system component must be able to cache just parts of a file, while it still needs to ensure confidentiality and integrity. To meet these requirements, TrustedFS treats file containers in untrusted storage as chains of data blocks, which, when decrypted and authenticated, can be stored in cache buffers.

5.2.1. Confidentiality

By using a block cipher in CBC mode, as described in Section 4.1, the trusted component can securely encrypt and decrypt data blocks independently from each other. However, the method to calculate IVs must be adapted for use with multiple files. Basically, there are two ways to do that, either by making the secret salt s a per-file secret, or by making block numbers unique among files.

Introducing per-file secrets would increase file-creation costs, because the trusted component would need to obtain a high-quality random *nonce* for each newly created file. Alternatively, the secret can remain global for all files, but instead of just using the block number to make an IV unique, you also hash a unique file number u:

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

These file numbers have to be unique among all files ever created, so they must not be reused once a file has been deleted.

I chose the second alternative because of its simplicity. It is easy to make sure that IVs are unique, for example by incrementing the assigned file number for each newly created file. This approach ensures the same degree of resistance against related-IV attacks as the original IV-calculation method described in Section 4.1.

5.2.2. Integrity

Ensuring integrity of file contents is more difficult. Because file-system data in untrusted storage can be subject to unauthorized tampering at any time, the trusted component has to ensure the integrity of each data block it reads from a file container. To verify that data is indeed unmodified, the trusted component makes use of the hashing method described in Section 4.2. Particularly, the following requirements have to be met:

- Modifications inside a data block or replacing a data block with an older version of itself must not go unnoticed.
- Moving data blocks, adding new ones, or deleting them must not go unnoticed.

The trusted component needs to know where each data block belongs to and which hash sum its content is supposed to have. This information must be trustworthy, however, sealed memory cannot provide sufficient storage capacity, as the number of data blocks in a file system is virtually unlimited. Therefore, the untrusted file container must store all information regarding data-block integrity in such a way that a single hash sum suffices to authenticate all data blocks. In Section 5.3.3, I shall elaborate on how to ensure the integrity of a file container's contents using these per-file hash sums.

As calculating a single hash sum that authenticates all data blocks at once is not possible, there remain two alternatives for block-level integrity:

1. **Authenticate Data Blocks In Place:** Per-block hash sums authenticate a block's content and its location, which is the tuple consisting of the unique file number, the block number, and a version number. The version number is necessary to prevent replay attacks on block level. By hashing the total number of blocks in the file container and all version numbers, the trusted component can meet all aforementioned requirements for block-level integrity.

2. **Authenticate Data Blocks Using a Hash Tree:** Data blocks in a file container are *nodes* in a self-protecting hash tree [23], as illustrated in Figure 5.2. Starting at the root node, all nodes along a path authenticate their child nodes. Leaf nodes at the lowest level of the tree finally store the user's data. The hash sum of the root node is sufficient to authenticate all data blocks that are part of the tree.

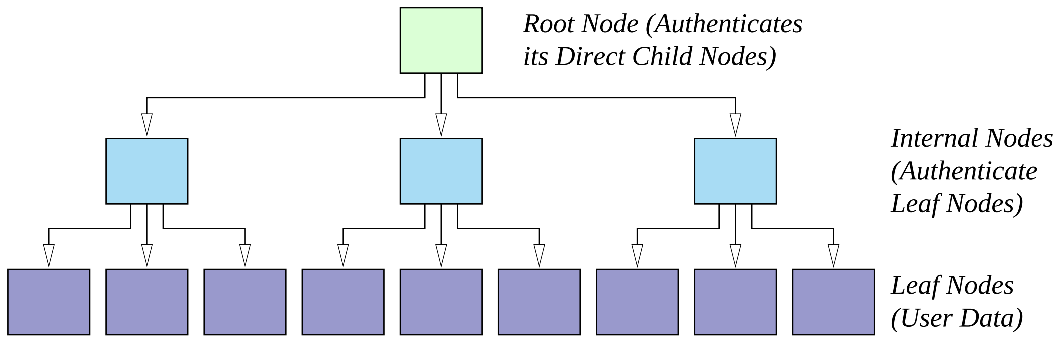
Root Node (Authenticates its Direct Child Nodes)
Internal Nodes (Authenticate Leaf Nodes)
Leaf Nodes (User Data)

Figure 5.2.: Ensuring integrity on block level using a hash tree.

The first solution is extremely simple and therefore tempting. Unfortunately, authentication costs scale linearly with a file's size. Before it can load any data block, the trusted component has to load and authenticate all version numbers. Although this approach allows for an extremely small TCB because of its simplicity, it is also limited to use cases in which only small files with large block sizes are needed.

Therefore, I chose the second alternative as it is better suited for a general-purpose file system. However, it is also more complicated, because it induces parent–child relations among data blocks. As parent nodes authenticate their child nodes, the trusted component's block-server layer cannot read a child node before its parent is in the buffer cache. In Section 5.6.1, I shall discuss this difficulty in detail.

5.2.3. Recoverability

The decision to embed a self-protecting hash tree into file containers also has advantages with regard to recoverability. To allow recovery in the first place, the trusted file-system component must send backups to the trusted server, as explained in Section 2.4.3. With mobile devices in mind, I must assume that the network connection to the server is not permanent, expensive in terms of money, and that bandwidth is limited. Therefore, it is crucial that only the differences made since a previous backup *(checkpoint)* are sent to the server.

With regard to files, the trusted component must keep track of all data blocks that were modified after creating a certain checkpoint. It can easily identify those, if both data blocks and checkpoints are assigned version numbers. Therefore, the trusted component stores a block's version number in its parent node, whereas the trusted server takes care of the checkpoint's version information.

Assuming that n is the checkpoint version, each data block with a version less than or equal to n has not been modified. Thus, to create a new checkpoint, the trusted component needs to send all blocks that have a version greater than n to the server. After synchronizing the backup copy with the primary copy on the user's computer, the server sets m as the version of the newly created checkpoint, where m is the highest version number that exists in the primary copy as reported by the trusted file-system component. When the user's actions cause data blocks to be modified after the new checkpoint has been created, the trusted component will tag all those blocks with version $m + 1$.

In Section 5.5, I shall elaborate on the complete backup protocol.

5.2.4. Meta Data and Per-File Integrity Anchors

Regarding meta data, such as file size and time stamps, the trusted file-system component cannot rely on the untrusted legacy OS. It is not even possible to hide this kind of information from an observing attacker who has access to the file containers.[3]

To provide client applications with trustworthy meta data, the trusted component must implement the necessary functionality. Meta data itself has to be stored in the file containers, however, storing it inside the embedded tree is infeasible. Therefore, I decided to reserve the first data block of each file container for it.

To ensure the integrity of the meta data, the trusted component must hash it, too. For practical reasons, I decided to use this hash sum to authenticate all contents of a file container. To fully establish the chain of trust, I added the hash sum of the embedded tree's root node to the meta data, as well as information on where to find the reference hash sum for the file container in the first place.

To be able to calculate IVs, the trusted component also requires the unique file number to be included in the meta data.[4]

5.2.5. Summary

Using the measures that I described in the previous sections, TrustedFS can provide files with arbitrary sizes. With a minimal TCB, it can take care of all desired protection goals and it reuses already existing untrusted infrastructure. Thus, it does not need to take care of low-level file handling, such as block allocation, fragmentation avoidance, or storing data persistently on a hard disk.

[3] If unused bytes in the last block of a file container are not truncated, an attacker can only learn that a file fits into a certain number of fixed-size blocks.

[4] For the first data block containing the meta data, the trusted component does not need to know the unique file number in advance. Instead, the plaintext meta data can be prepended by a precalculated IV itself.

5.3. Protecting Directories and Directory Structure

5.3.1. General Considerations

The idea of a trusted wrapper reusing untrusted legacy infrastructure works well for individual files. However, the question whether a wrapper can also be useful to implement trustworthy directory handling and lookup is not trivially answered.

Storing each file that belongs to the trustworthy file system in its own untrusted file container already imposes limitations on confidentiality, because the number of files and the file sizes cannot be hidden. Even more, reusing directory-handling and lookup functionality of the legacy OS will disclose the directory structure. Nevertheless, the filenames of file containers can still be encrypted.

The second critical protection goal in this context is integrity. On file and directory level, the trusted wrapper would have to meet the following requirements (the terms *file* and *filename* also apply to directories):

1. Performing a filename-based operation must always result in accessing the correct file. Replay attacks on files must be detectable.
2. Files that never existed, or that the user deleted previously, must not be accessible.
3. It must be detectable, if a file is inaccessible due to an attack (e.g., because its file container has been deleted).
4. Reading directory contents must result in a correct, complete and up-to-date list of filenames.

Pure Wrapper. If the trusted file-system component reuses the legacy OS's lookup functionality as outlined previously, it can fulfill the first two requirements with little additional infrastructure. Particularly, it needs trustworthy information to verify content and location of each file in the file system. The trusted component can store this information in a special file, which in turn is protected by its hash sum being stored in sealed memory. This kind of infrastructure is necessary in any case.

Requirements 3 and 4 depend on a redundant trustworthy list of filenames for each directory, because the legacy OS's routines, in this case L^{4}Linux' `open()`, `stat()`, and `readdir()` functions, cannot be trusted to provide correct information.

Wrapper with Complete Name Service. So, the pure-wrapper approach requires directory functionality to be duplicated in the trusted component. Also, reusing untrusted lookup routines necessitates the disclosure of the directory structure.

Thus, to overcome the limitations regarding confidentiality, I considered the possibility of implementing a trusted file-lookup mechanism as well. Based on such a design, the trusted component could reuse L^{4}Linux' file system as a flat name space. For example, it could use plain numbers, which are unrelated to the the real directory structure in the trusted domain, to name file containers.

However, I had to realize that even this approach cannot always ensure the desired degree of confidentiality. After all, an attacker can still observe access patterns regarding individual data blocks in the naming database. Whenever the trusted component has to resolve a pathname, it must access at least one of these data blocks to look up each path element. So, it is not possible to avoid disclosing information about the directory structure.

In any case, an implementation of a complete naming infrastructure in the trusted component would be complex. Although a simple solution to do file lookups will enlarge the TCB only moderately, a competitive implementation has to satisfy stronger demands. I identified the following key requirements:

- **Naming Database:** The whole naming database, which contains the tree of filenames, must be stored in a single file container. Using per-directory containers is infeasible because opening such a directory file to look up each path element is too expensive.
- **Fragmentation Avoidance:** With the contents of all directories stored in a single file container, fragmentation becomes a relevant problem when reading directory contents and, depending on the search strategy, when performing lookups.
- **Efficient Lookup:** Resolving a pathname causes a filename lookup for each path element, so lookup performance is critical. The lookup algorithm should avoid unnecessarily loading data blocks from the naming database, because decrypting and authenticating them is expensive. For example, the implementation could use *B-Trees*[5], which offer logarithmic lookup complexity, or hash tables.

Given the time frame for my thesis and the little gain in secrecy, I could not justify the engineering costs for such an implementation. Instead, I decided on the pure-wrapper approach, which allows to reuse at least parts of the highly-optimized naming infrastructure provided by L^{4}Linux. It is also more interesting from a scientific point of view, as it allows to test to which extent the design principle of trusted wrappers can be applied in practice.

Nevertheless, a solution that fully implements naming in the trusted component should be subject of a future work.

5.3.2. Confidentiality

As each file is stored in its own file container, untrusted components are able to do lookups based on a filename. So, if it assigns encrypted filenames to the file containers, the trusted component can make use of L^{4}Linux' optimized lookup routines and directory caches.

Reusing the legacy OS's functionality in this way requires the names of the file containers in untrusted storage to reflect the directory structure in the trustworthy file system. As I shall explain in Section 5.3.3, the trusted component does not need to be able to decrypt filenames that are used in the untrusted domain. Therefore, it is possible to use a collision-resistant hash function to conceal plaintext filenames from unauthorized parties.[6] Doing so makes sure that all encrypted filenames have the same length.

Hashing only the filename is insecure, because an attacker can still guess the plaintext filename and then calculate its hash to verify whether his guess was correct. To prevent such known plaintext attacks, you can use a method similar to the one used for calculating initialization vectors:

[5] B-Trees are often used for file systems and databases, for which, due to slow storage, the costs for accessing a node are significantly higher than those for searching within a node.

[6] In theory, it might happen that hashing two distinct filenames results in identical encrypted names, however, the probability for such a collision is extremely small.

$$encrypted\_name := H(s \parallel plaintext\_name)$$

Again, s is a secret salt unknown to the attacker, so he is no longer able to verify the correctness of a guessed plaintext filename.

Unfortunately, hashing identical plaintext filenames, even in different directories, will always result in the same encrypted filename. Using per-directory salts would solve this problem, but this solution is equivalent to fully implementing file lookup in the trusted component. The only remaining solution that reuses the legacy OS's lookup infrastructure is to make the encrypted filename of each path element dependent on all preceding path elements. For example, the path `/dir/subdir/file` would be encrypted as follows:

$$encrypted\_name := H(H(H(s \parallel "dir") \parallel "subdir") \parallel "file")$$

This solution would even allow to store all file containers in a single directory of the untrusted storage, because they would have distinct filenames. So, it seems that the real directory structure could be hidden entirely. However, with this approach, renaming a directory will invalidate all encrypted filenames in the sub tree below this directory. Thus, all affected file containers would have to be renamed as well, which would not only cause severe performance degradation, but it would also reveal to an observing attacker that all these files belong to the same sub tree.

Therefore, this approach to encrypt filenames is only feasible, if a file-system implementation does not allow to rename or move directories. However, I consider this limitation to be too restrictive for a general-purpose file system, so I decided to use the first hashing method to keep filenames secret.

Nevertheless, the hash sum of all path elements is useful to ensure one of the requirements for file-level integrity. In Sections 5.3.3 and 5.4.2, I shall discuss this aspect in greater detail.

5.3.3. Integrity

Shadow Directories: As I already mentioned in Section 5.3.1, the trusted component has to maintain redundant lists of directory contents to ensure integrity. These lists *(shadow directories)* do not need to be referenced just to open a file or to resolve its pathname. However, the trusted component needs to read from a shadow directory, if a client application wants to scan directory contents. Also, if an `open()` operation fails, the trusted component must consult the relevant shadow directory, so that it can determine whether the file indeed does not exist, or whether an attacker deleted the corresponding file container.

Obviously, creating new files or removing existing ones from the trustworthy file system necessitates shadow directories to be updated. Also, whenever the trusted component creates a file, it has to make sure that the filename given by the user is not already in use. Thus, an efficient lookup mechanism for shadow directories would be beneficial.

Nevertheless, to keep the TCB small and the implementation costs in terms of time low, I decided on a simple solution. I chose to represent shadow directories as unsorted lists that are stored in per-directory file containers. As the trusted component needs to touch at most one shadow directory per file operation, I consider the overhead for opening its file container acceptable.

Unfortunately, this approach also means that creating a new file requires the trusted component to do a linear search in the shadow directory. However, in practice, I still

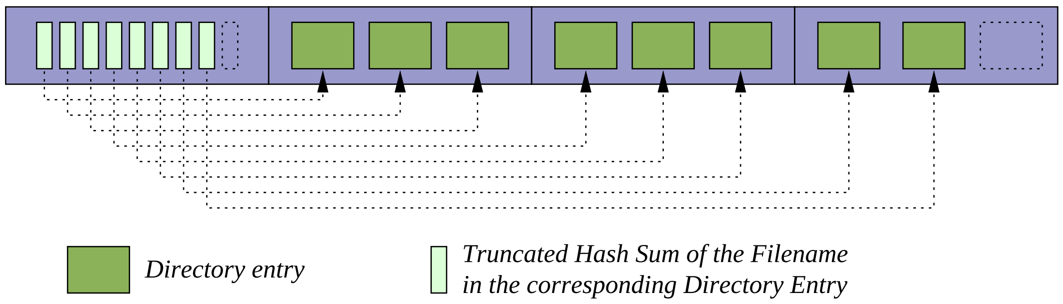
Directory entry
Truncated Hash Sum of the Filename in the corresponding Directory Entry

Figure 5.3.: A great number of short hash sums in only few data blocks of a shadow directory give hints regarding the position of a certain directory entry.

expect acceptable performance. Because it is necessary to hash the filename anyway, a shadow directory can also contain truncated hash sums (e.g., 32 bit in length) that are, densely packed, stored in only few data blocks. Using these truncated hash sums, the trusted component does not need to load all blocks of the shadow directory into the buffer cache just to look for a certain filename. This solution is extremely simple, but also most efficient for directories with lots of entries, when considering the high costs of loading data blocks into the buffer cache. Figure 5.3 illustrates this idea.

Master Hash File: Per-file hash sums authenticate the contents of file containers, including shadow directories. These hash sums have to be stored in untrusted storage, as it is not possible to keep all of them in sealed memory.

Therefore, I introduced the master hash file, which I briefly mentioned in Section 5.3.1. It contains data records *(master entries)* that serve as per-file integrity anchors. A master entry contains the hash sum that is needed to authenticate a file container's first data block, which in turn contains the hash sum to verify the integrity of the embedded tree.

The per-file hash sums allow to detect modifications inside a file, as well as replay attacks on file level. However, the trusted component must also be able to detect, whether the untrusted block server indeed opened the correct file container. Therefore, a master entry also contains data describing a file's location. I decided to encode this information by hashing the file's full pathname, as I described in Section 5.3.2. Just the hashed pathname is not sufficient in all situations, therefore additional information is necessary. I shall elaborate on it in Section 5.4.2.

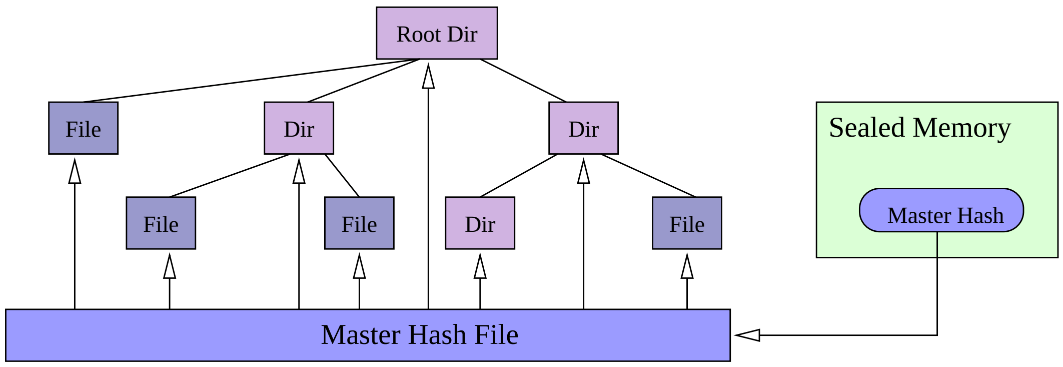

Figure 5.4.: The master hash file authenticates all files and shadow directories.

With its own hash sum being stored in sealed memory, the master hash file can ensure the integrity of all file-system contents, as illustrated in Figure 5.4.

5.3.4. Recoverability

During the backup procedure, the trusted component sends only data that is newer than the checkpoint stored at the trusted backup server. To allow it to detect efficiently, whether a file has indeed been modified, I added a per-file backup version to each master entry. In Section 5.5.1, I shall discuss the strategy to find modified files based on their master entries in greater detail.

5.3.5. Summary

The pure-wrapper approach allows to implement directory-handling and lookup functionality with a minimal TCB. It reuses existing infrastructure of both the legacy OS and the trusted component itself, as it implements shadow directories and the master hash file using self-protecting file containers. It fully ensures integrity and, with limitations as described in Sections 5.3.1 and 5.3.2, confidentiality.

5.4. Trusted File-System Wrapper

5.4.1. Opening and Authenticating Files

Any design approach that reuses file containers in the untrusted storage requires the trusted component to perform additional integrity checks after the untrusted part of the implementation opened the file container. It must test the following conditions:

1. **Are the File Container's Contents Valid?** The trusted component must decrypt the meta data from the first data block, lookup the corresponding master entry using the information in the meta data, and compare the reference hash sum in the master entry with the hash sum of the previously decrypted meta data. If the hash sums do not match, the trusted component must assume an integrity violation.

2. **Is the File Container the Requested One?** According to the pure-wrapper approach, the trusted component must calculate the hash sum of the full pathname used for lookup and compare it with the hashed pathname that is stored in the master entry.

If a directory has been moved or renamed, the second test may fail for files in that directory or one of its subdirectories, even if file-system integrity has not been compromised. I already discussed the cause of this problem in Section 5.3.2. In Section 5.4.2, I shall present an efficient solution, which allows hashed pathnames to be used to ensure integrity nonetheless.

The untrusted component may also fail to open the requested file container in the first place. With the pure-wrapper approach, there are two possible causes for such an error condition:

1. The file does not exist.

2. The file container does not exist or the untrusted component did not provide its contents for other reasons.

To determine the real cause of the lookup error, the trusted component must recursively look up each element of the full pathname in the corresponding shadow directories.[7] If it encounters a sub path that does not exist based on the information in the shadow directories, the file does indeed not exist. However, if the filename of a path element is listed in the corresponding shadow directory, the trusted component must report an integrity violation to the client application. Integrity has also been compromised, if the trusted component could not open any of the shadow directories in the path and finally fails to open the root directory, which must always exist.

5.4.2. Moving and Renaming Files and Directories

To move or rename a regular file, the trusted component must update shadow directories and it must instruct the untrusted component to move or rename the file container accordingly. Additionally, it has to update the hashed pathname in the master entry (path hash) to reflect the file's new location. However, I already explained in Section 5.3.2 that moving or renaming a directory invalidates all path hashes that point into this directory.

For performance reasons, it is infeasible to update all affected path hashes immediately when renaming or moving a directory. However, the trusted component can also reconstruct the correct pathname of a file that has been moved implicitly when opening it. That is, it can lazily revalidate the path hash of any file that changed its location in the directory tree because of a directory being renamed or moved. Particularly, each master entry must contain the following information in addition to the path hash:

- **Parent Pointer:** The parent pointer specifies the master entry of the shadow directory that represents the file's parent directory.
- **Hash Sum of the Filename:** The hashed filename that the trusted component provides for lookup of path elements in the untrusted domain must also be stored in the master entry.

It is trivial to provide this information. The trusted component can use it to determine, whether the untrusted part of the implementation indeed opened the correct file container. Using the parent pointers, it retrieves the hashed filenames for each path element from the corresponding master entries and compares them with those used for lookup. If the hash sums match for all path elements up to the root directory, the trusted component is indeed using the correct file container. Otherwise, it must report an integrity violation. In case the hashed pathname in the requested file's master entry, or those of any subdirectory in the path, has just gotten out of date, the trusted component can update them once it established the validity of the pathname.

For example, assuming that the user created a file with the pathname `/A/B/C` and later renamed the directory `B` to `X`, the master entry for `C` would contain an outdated path hash still specifying `/A/B/C` instead of `/A/X/C` as `C`'s location in the directory tree. So, when opening `C` using the new pathname, the trusted component would compare the hashed filenames in the encrypted path with those found in the master entries of `C`, `X`, and `A`. Figure 5.5 illustrates this example.

[7] For performance reasons, the trusted component should do this lookup in reverse order, because often only the last path element is invalid.

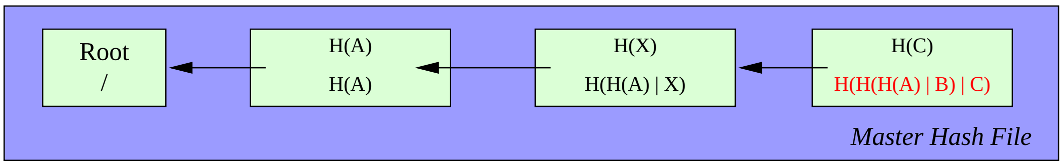

Figure 5.5.: Hashed pathnames in the master entries after renaming a parent directory.

As an optimization, the trusted component can also determine the validity of a sub path instead of checking all path elements up to the root directory. It can do so by calculating the path hash for the sub path as well and comparing it with the hashed pathname in the corresponding master entry. That is, in the previous example, the hashed pathname for `C`'s parent directory `/A/X` is already up to date, so the trusted component can stop revalidation at this point.

Unfortunately, there is one case left that leaves the possibility of a successful attack from the untrusted domain. For example, opening the previously mentioned file `C` using the invalid pathname `/A/B/C` might trick the trusted component into accepting the file, if it did not yet update the path hash in the master entry. For this attack to be successful, the untrusted component would have to open the previously moved or renamed file container, even though it should report a lookup failure instead.

To prevent this kind of attack, the trusted component must consider hashed pathnames in the master entries to be out of date after renaming or moving a directory. To enforce this behavior, it can assign version numbers to them. Whenever it moves or renames a directory, it increases the current version number, so that it can detect an out-of-date path hash and revalidate it using the method that I described previously.[8] This solution causes increased latency when opening a file for the first time after renaming or moving a directory. However, with the pure-wrapper approach, it is necessary to ensure full integrity even if client applications try to access files using invalid pathnames.

5.4.3. Deleting Files

To delete a file, the trusted component must remove the filename from the shadow directory. The untrusted component has to delete the file container. However, it is even more important to invalidate the file's master entry, so that future attempts to authenticate the file will fail in the open phase.

5.5. Backup and Recovery

5.5.1. Basic Backup Protocol and File-System Scanner

In Sections 5.2.3 and 5.3.4, I described how the trusted file-system component can use version numbers to keep track of changes made to file-system contents. These version numbers are a critical requirement for minimizing the amount of data that needs to be transferred to the trusted backup server. Especially with regard to mobile devices, which most likely cannot rely on high-bandwidth Internet connectivity, it is important that TrustedFS only sends a small set of changes.

[8] In this case, revalidating a path hash also involves updating its version number.

To be able to identify all differences between the primary copy of the file system and the backup, the trusted component needs to know the version number of the checkpoint stored on the backup server. Such a checkpoint does not necessarily have to be the most recent one. In Section 5.5.2, I shall discuss this aspect of the protocol in detail.

Again, to keep the TCB small, I decided on a simple protocol to communicate file-system changes to the trusted backup server:

- **Modified Files:** Prefixed by the encrypted pathname, the file length in data blocks, and the encrypted meta data, all modified blocks and their block numbers are sent to the server. If the file in the backup copy is larger than the communicated length, the backup server can decide on its own where to truncate it.

  If a file has been moved or renamed after creating the last checkpoint, it is necessary to transmit all data blocks.

- **New Files:** New files can be treated the same way as modified files, so no special care must be taken.

- **Deleted files:** The file system itself does not keep information about files that have been deleted after creating a previous checkpoint.[9] However, the backup server cannot learn about obsolete file containers or even subdirectories on its own, because all shadow directories are encrypted. Therefore, the trusted component needs to send a current list of all file containers in a specific directory, whenever it detects that the corresponding shadow directory has been modified.

To find modified files in the first place, the trusted component could scan the whole directory tree recursively. However, I decided not to do so for various reasons:

- **Efficiency:** There is no efficient way to propagate the highest backup version number in a sub tree of the file-system hierarchy into directory entries in upper-level directories. Therefore, the trusted component would have to open all shadow directories, because it cannot know whether a subdirectory contains modified files or not.

- **File-Handle Consumption:** When scanning shadow directories recursively, the scanner would require one file handle for each element of a full pathname. If the user decides to create a new backup checkpoint while client applications still have files opened, there may not be enough file handles to descend to deeper levels of the directory tree.

- **Unnecessary Enlargement of the TCB:** Because untrusted components are responsible for file-container lookup, the implementation of a recursive directory scanner would only be used for backup tasks.

To solve these problems, I chose an alternative scanning strategy. The trusted component can scan all entries of the master hash file. By evaluating the backup version number of each master entry, it can find modified files more efficiently. It can easily reconstruct the full encrypted pathname using the method that I described in Section 5.4.2. Such a scanner is easier to implement and it requires only one file handle to be available.

[9] The run-time costs for maintaining a list of deleted files are unacceptable. Because a new file can have the same name as a previously deleted file, this list would have to be updated whenever creating or deleting a file.

5.5.2. Extended Backup Protocol for Integrity

Obviously, a checkpoint stored on the trusted backup server must not be damaged, even if the server received a corrupted or incomplete change set. So, with unstable network connections in mind, the communication protocol must provide robustness against unexpected network failure. But even with a stable connection, the trusted component might be forced to abort the backup procedure. For example, because it detected that the file system's integrity has been compromised.

In consequence, the server must write all data that it receives from the user's computer or mobile device *(the client)* to a temporary log. It must not modify the backup copy, before the client verified that the server received all data.

To make sure that the server will only commit a correct change set, I decided to let both the client and the server calculate the hash sum of the transferred change set. If the server can reply with a correct hash sum, the client can be sure that the peer successfully received all data.

After requesting the current checkpoint version number from the backup server, the client can start sending data, while both peers calculate the hash sum of the transmitted data stream. When the client sent all data, the following steps have to be executed:

1. The client instructs the server to send the calculated hash sum.
2. If both the server's hash sum and the one calculated by the client match, the client sends a *COMMIT* message including the version number of the new checkpoint. Otherwise, it sends an *ABORT* message to the server and communication ends.
3. The server commits the change set in the temporary log, updates the checkpoint version number, and sends a *COMMITED* message back to the client. If an error occurs during commit, it sends an *ABORT* message and communication ends. To ensure availability of a valid checkpoint, the server must perform a rollback of the partly applied change set.
4. If the client receives a *COMMITED* message, or no message, it increases the backup version number that is used for future changes to file system.[10] Communication ends.

The described protocol ensures that the backup server will not create invalid checkpoints, even if communication is interrupted, or if the client cannot send a complete change set.

As the only state regarding checkpoints on the client's side are version numbers as described in Sections 5.2.3 and 5.5.1, the client can generate valid change sets to upgrade any previously created checkpoint. Thus, it is also possible to bring a backup copy up to date, if, for example due to system failure, the latest checkpoint on the server has been lost. A user can also switch to another trusted backup server, if it becomes necessary.

5.5.3. Communication Channel to Trusted Backup Server

In section 2.4.3 I already discussed the necessity of mutual authentication between the client device and the trusted backup server. Design and implementation of the required

[10] If the client does receive neither a *COMMITED* message nor an *ABORT* message, it cannot know whether the server committed the change set. So, for a future attempt to create a new checkpoint to succeed, the client must use a new version number for any changes made to the file system after the possibly failed backup attempt.

infrastructure, including the network communication channel, is out of the scope my thesis. Nevertheless, I shall specify the functionality and interfaces that are required for TrustedFS.

The communication channel between client and server must be bidirectional. Preferably, this channel should have semantics similar to the Secure Socket Layer (SSL) [37, 38]. However, because all data blocks that need to be sent to the server are encrypted, it does not need to ensure confidentiality. In principle, the previously described communication protocol could even be implemented, if the communication channel can only ensure integrity and authenticity of independently transmitted messages. However, such an approach would be impracticable with regard to remote attestation and it is likely that other applications require SSL-like semantics anyway.

Particularly, I assume the availability of an interface providing the following communication primitives:

1. **Handshake and Open:** During handshake both client and server identify themselves using credentials and they mutually attest their software stacks via remote attestation. If both accept the other's identity and software stack, the communication channel is opened.

2. **Sending/Receiving:** Both client and server can securely send and receive data, either as a byte stream or as packets of user-defined size.

3. **Shutdown:** The communication channel is closed.

Regarding the client side, the trusted file-system component requires an interface to obtain the following information for the handshake and open phase:

- A credential, for example a certificate including the private signing key.
- A public key to validate the server's credential.
- A specification for a valid software stack on the server.

The trusted backup server must have access to equivalent information.

5.5.4. Recovery

If data in untrusted storage has been damaged, it will not pass the integrity tests when the trusted component tries to authenticate it at a later time. Thus, the data becomes useless. However, depending on the extent of the damage, other parts of the file system might still be accessible. In this situation, the user has the following options:

1. **Full Recovery:** The user can request the file system's contents to be reverted to the latest checkpoint that is available on the trusted backup server. All changes made after creating the checkpoint will be lost. However, to minimize data loss, the user can manually copy files that are still intact to another storage, for example another file-system instance, before reverting the file-system state.

   If not only untrusted file containers, but also the encryption key in sealed memory, the client's credential, or the authentication keys that I mentioned in Section 5.5.3 are lost, additional recovery measures must be taken.

2. **Partial Recovery:** As long as the master hash file is still intact, it is also possible to repair the file system by replacing damaged files with their backup copies. TrustedFS has to retrieve undamaged versions of the affected file containers and the corresponding version of the master hash file from the trusted backup server. With these file containers being stored in a separate area of untrusted storage, it is possible to create a second instance of the file system containing just the recovered files, which the user can copy to the original file system.[11]

To recover files using the backup copy on the trusted backup server, TrustedFS needs to manipulate untrusted storage. Therefore, it is possible to leave the retrieval of file containers entirely to the untrusted component.[12] In principle, you can even reuse existing file-transfer software, such as *rsync* [39], to synchronize the data in untrusted storage with the encrypted backup copy.

Nevertheless, the trusted component must still be involved. With regard to full recovery, which implies replacing the master hash file, the trusted component must inform the user, when it detects that the master hash file has been reverted. Otherwise, an attacker could successfully perform a replay attack by replacing the data in untrusted storage with the latest checkpoint on the backup server.

Partial recovery is more complex and it might also require the user to make decisions based on the specific situation. For example, he might prefer to extract at least parts of valuable data from a damaged file, instead of replacing it with an out-of-date backup copy. Therefore, support for partial recovery should be implemented as a separate trustworthy utility program that uses primitives provide by the trusted file-system component. The details regarding an implementation supporting partial recovery are subject to future work, especially the question how to deal with damaged shadow directories, which are the only source for trustworthy information on directory contents.

5.6. Buffer Cache

5.6.1. Requirements

In essence, the buffer cache has to manage a pool of cache buffers, which hold decrypted and authenticated data from files. Each of these cache buffers stores exactly one data block. Apart from the usual requirements for a cache implementation, such as efficient lookup of cached objects, additional demands arose in the context of my work.

It quickly turned out that it is necessary to map the parent–child relationships among the data blocks in a file container to the cache buffers that contain these data blocks. Because it must ensure integrity on block level, the trusted component cannot load a data block from untrusted storage into the cache, unless it knows the reference hash sum of the data block. Therefore, the corresponding parent node, the data block that contains the hash sum, must already be in the cache.

If the cache flushes a dirty buffer, it is even more important for the trusted component to have the parent node cached, as this operation requires the reference hash sum to be

[11] The user must delete the damaged files first, because overwriting them might fail due to integrity errors.

[12] Because all data that is stored on the trusted backup server is protected by strong encryption, user authentication is not strictly necessary to retrieve the backup copy. Nevertheless, restricting access to authorized users can at least make denial-of-service attacks against the backup server harder. For example, the backup server could ask for a password or a one-time credential, which the trusted component could acquire prior to recovery.

updated. Of course, the cache must also flush dirty buffers, if there are no more free buffers available. If the parent node is not in the cache at this time, it would have to be read from the file container, which would require another buffer to be made available just to be able to free the first one. The same problem can occur there as well, it might even be necessary to load all nodes along a path in the embedded hash tree into the cache. Obviously, the resulting loss of performance is unacceptable.

A second major requirement is that the trusted component must be able to flush all dirty buffers that are allocated to a specific file. I shall elaborate on the reason for this demand in Section 6.3.3. In Section 6.3.2, I shall discuss the implications for the buffer-cache implementation regarding the aforementioned necessity to respect parent–child relationships.

5.6.2. Looking up Cache Buffers

The unique file number and the block number, when being used as a tuple, form a unique name for a certain data block. Hence, they can be used as a key for finding any block in the cache. However, the name space containing these keys is huge. In my implementation, each key has a length of 128 bits, so a simple array of buffer addresses with a key being used as array index cannot be used. The common solution to look up an object in a cache is to use a hash table or a search tree. For example, the Linux page cache uses a Radix Tree [40] for finding physical page frames.

When choosing an efficient cache lookup mechanism for TrustedFS, the special requirements in the context of my work have to be taken into account:

1. Key collisions cannot be allowed, because they can cause arbitrary data blocks to be evicted from the cache.
2. Finding all data blocks or those belonging to a specific file has to be efficient. Preferably the logical order of the retrieved blocks should be preserved.
3. The TCB is to be kept small.

The first requirement intents to make sure that no data block is evicted as long as there are any of its child nodes still in the cache. The second one is there, because it must be possible to flush buffers belonging to a specific file. With these demands in mind, a hash table would not be a good choice, because it does not meet the second requirement.

Search trees, such as radix trees and binary search trees, can accommodate all these demands. Fortunately, there exists an efficient implementation of a self-balancing binary search tree, which is already part of the TCB of L4Env-based applications. The L4 Region Mapper package uses an AVL tree [41] to determine, which dataspace manager it must call to resolve a page fault in a specific virtual-memory region. With modifications in the order of approximately one hundred lines of code, both the region mapper and the buffer cache can use the AVL-tree implementation.

An AVL tree, just like any other self-balancing tree, may require rebalancing when keys are inserted or removed. Therefore, these operations are often slower when compared to a radix tree, or *tries*[13] in general. However, the influence on the overall performance of TrustedFS is negligible, because the cryptographic operations that need to be performed

[13] A *trie*, unlike a binary search tree, does not explicitly store keys in its nodes. Instead, the key encodes the path that has to be taken to reach the desired node.

when loading a block into the cache, or when evicting a block from it, take considerably more time. The AVL tree's lookup performance is comparable to a trie in this scenario, because the keys are simple integer numbers. So, in favor of a small TCB, I decided to adapt the existing implementation of the AVL tree.

5.7. Untrusted Block Server and Trusted Block-Server Layer

5.7.1. Untrusted Block Server

The discussions in previous sections of this chapter already showed that large parts of the legacy OS's infrastructure can be reused for TrustedFS. Thus, the untrusted component only needs to implement high-level functionality to handle file containers and, most importantly, it must provide an interface to access data blocks.

Although the trusted component implements its own buffer cache, the block-server interface has influence on the overall performance, too, especially regarding throughput when reading or writing large amounts of data. An early prototype implementation used L4 IPC to coordinate the transfer of single data blocks in a shared memory buffer. However, this solution does not provide for optimal performance. A main reason are high IPC costs between the trusted component and the untrusted block server, which is an L^{4}Linux process. IPC between native L4 tasks and L^{4}Linux processes is not only expensive because of address-space switches, but also because of the L^{4}Linux process's *alien status*.[14]

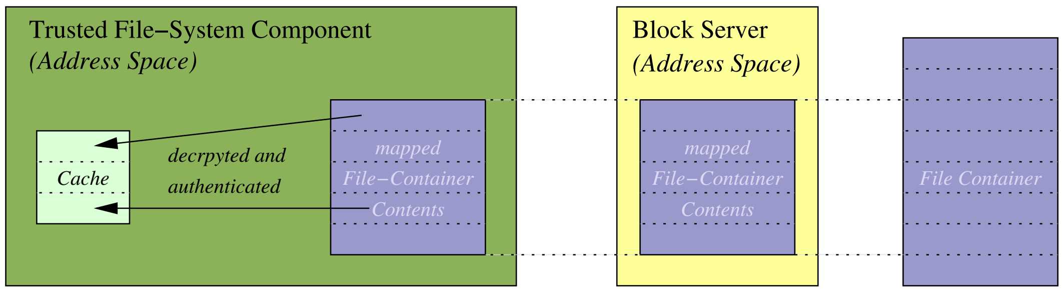

Figure 5.6.: Mapping of cryptographically protected file-container contents into address spaces.

To allow for maximum performance, I decided on a design as illustrated in Figure 5.6. Instead of using `read()` and `write()` to move data blocks between the file container and the shared-memory buffer, the block server maps a file container's contents into its address space. The trusted component, in turn, maps the contents of the file container from a dataspace-provider thread in the untrusted block server.

The block server can implement optimizations, such as paging in data blocks in advance based on observed access patterns. It can give hints to the trusted component regarding which data blocks in the mapped file container are probably needed next. The trusted component can use these hints to request the block server to map the corresponding memory pages in a single operation, thus saving expensive IPC calls to the L^{4}Linux process.

[14] L^{4}Linux tasks that perform a native L4 system call cause the Fiasco microkernel to reflect exceptions into the L^{4}Linux server task. By evaluating these exceptions, L^{4}Linux can prevent the calling thread of the L^{4}Linux process from being scheduled while the thread is blocked in the L4 system call.

5.7.2. Block-Server Layer of the Trusted Component

In contrast to the block server itself, the block-server layer of the trusted component is part of the TCB. However, I consider the enlargement of the TCB due to an implementation as described in Section 5.7.1 acceptable. The L4 Region Mapper already provides the necessary infrastructure to map dataspaces into the trusted component's address space. Only little additional code is necessary to map file-container contents, to make use of the block server's hints, and to unmap a dataspace when the trusted component eventually closes a file.

5.7.3. Virtual–Address-Space Consumption

Unfortunately, mapping the contents of all currently opened file containers requires plenty of virtual address space. It is likely that, even with a small number of client applications, the address spaces of both the block server and the trusted component cannot provide enough room for all mappings. It is also conceivable that a single file is larger than the virtual address space.

Therefore, it must be possible to map just parts of a file container. However, the decision on which parts to map can be left to the untrusted block server. Using the aforementioned hinting mechanism, it can inform the trusted component about advantageous mappings.

5.8. Application Programming Interface

5.8.1. Function Primitives of the Server Library

The core of the trusted component is the server library, which encapsulates all basic file-system functionality. In essence, it is a wrapper for the POSIX file-system interface [42], so it is also based on the same well-understood primitives. Thus, it provides `open()` and `close()` as well as functions that behave like `fstat()`, `rename()`, `unlink()`, and `readdir()`.

However, because the trusted component implements its own buffer cache, there is no need to force client applications to use a certain method to access file contents. The library merely provides mechanisms to change the size of a file and to access its contents on block level. A front end can offer higher-level methods, for example using `read()` and `write()` functions. For maximum performance, a front end can also map cache buffers containing file data directly into a client application's address space, because cache buffers are the same size as memory pages.

To support memory-mapped file access, the buffer-cache implementation must know whether a certain buffer is mapped into another address space. I shall discuss this aspect of the implementation in Section 6.3.

5.8.2. Low-Complexity API Front End

Although it is possible to link the server library into an application binary, its public interface is not designed to be used directly by applications. To offer programmers a more convenient API, a front end is necessary.

I decided not to use L4VFS as the primary front end, because just its client-side infrastructure would add about 2,900 lines of code to the TCB.[15] However, the server library already provides an interface that could be used by client applications, if it were complemented by mechanisms to conveniently access file contents. Thus, I chose to implement a minimal-complexity front end that basically wraps the server library's public functions using IDL-implemented L4 IPC. The server task, which implements the trusted file-system component, also supports memory-mapped file access using a separate dataspace-provider thread. To map those files into an address space, the client-side library reuses existing infrastructure of the L4 Region Mapper.

In the API documentation generated from the library source code, you can find a comprehensive description of all functions and data types that are available to client applications.

5.8.3. Error Handling

Because TrustedFS relies on untrusted components that can be subject to attacks, error handling by both the trusted component and its client applications is extremely important. Particularly, the trusted component must immediately inform a client application, if it detects that the integrity of data in untrusted storage has been compromised.

I consider it a crucial design requirement that client applications do not crash in case of errors, not even integrity errors. They should be given the chance to report errors to the user or to recover gracefully.

All functions of the previously described client API, as I outlined in Sections 5.8.1 and 5.8.2, operate synchronously. Therefore, error conditions can be reported trough their return codes. By consequently evaluating those return codes, an application can meet the aforementioned stability requirement. However, it is much more difficult for an application to handle error conditions gracefully, if the server task cannot resolve a page fault due to an integrity error.

During implementation, I discovered that the L4 Region Mapper already provides a mechanism to handle unresolvable page faults. Based on a callback function, it is possible to implement a simple and yet powerful exception handling. With this mechanism, which I shall discuss in detail in Section 6.4.3, client applications can handle errors that occur while accessing contents of memory-mapped files.

[15] The L4VFS name server accounts for about 1,500 lines of code and runs in its own address space. The remaining L4VFS code implements functionality that is already implemented in the trusted file-system component, so that it can function as a wrapper.

6. Implementation

In this chapter, I shall give insight into the implementation of TrustedFS. Particularly, I shall discuss implementation details regarding file containers, the file-system wrapper, and the buffer cache. I shall also elaborate on the application programming interface (API) and reuse of already existing source code.

6.1. Thread Structure

The implementation of the trusted file-system component, including the front end described in Section 5.8.2, uses three threads:

- **API Thread:** The API thread performs synchronous operations, such as opening files.
- **Dataspace-Provider Thread:** The dataspace provider supplies mappings for client applications to resolve page faults in virtual-memory regions that represent files.
- **Flush Thread:** The flush thread is responsible for writing back dirty cache buffers asynchronously.

Because all these threads may use functions offered by the server library's public API concurrently, the implementation must be thread safe. For the time being, I implemented mutual exclusion in the front-end layer using a single lock for all API functions. However, fine-grained locking is subject to future work.

6.2. File-System Wrapper and Organization of File Containers

In this section, I shall discuss important aspects of the implementation with regard to file containers and the file-system wrapper. I shall give an overview about the structure of embedded trees in file containers and I shall elaborate on the algorithms to use and manipulate these trees.

6.2.1. Embedded-Tree Structure

File-Container Layout: In Section 5.2.4, I discussed the need for an extra data block that contains the meta data of a file. Meta data is stored in the first data block. User data and data blocks that belong to higher levels of the embedded tree are stored right behind it.

I decided on a static structure as illustrated in Figure 6.1, where parent nodes are followed by a fixed number of direct child nodes. As the trusted component must load parent nodes first, this layout allows for optimal performance when reading user data from consecutive data blocks. However, when writing data, the trusted component must first

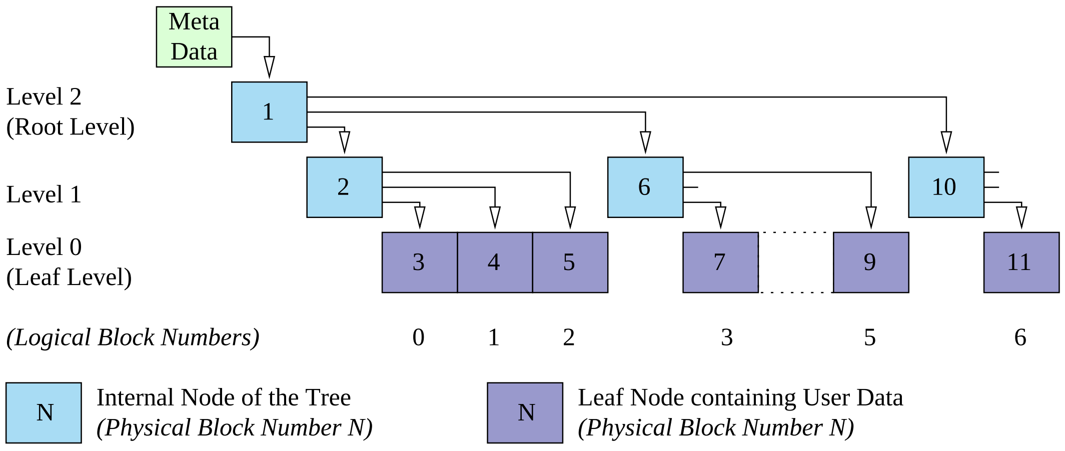
Meta Data
Level 2 (Root Level)
Level 1
Level 0 (Leaf Level)
(Logical Block Numbers)
1
2
6
10
3
4
5
7
9
11
0
1
2
3
5
6
N
Internal Node of the Tree (Physical Block Number N)
N
Leaf Node containing User Data (Physical Block Number N)

Figure 6.1.: Block layout of file containers.

write all child nodes before it can update the parent node. In practice, this strategy does not necessarily cause a performance problem, because the legacy operating system (OS), or even the hard disk itself, caches encrypted data blocks and it is most likely able to write them consecutively.

Logical and Physical Block Numbers: Nodes that are not on the lowest level of the tree (leaf level) are unknown to client applications. That is, the trusted component completely hides the embedded tree. Hence, data blocks on the leaf level that ultimately contain user data are assigned logical block numbers that differ from their physical block numbers.

Nevertheless, the layout of data blocks allows to calculate physical block numbers easily. For a tree with maximum depth d and m children per node, you can use the following formula to map a logical block number l to a physical block number p:

$$p := \underbrace{1 + (d-1)}_{Offset} + \sum_{h=0}^{d} \lfloor \frac{l}{m^h} \rfloor$$

The offset accounts for the first data block containing the meta data and the nodes leading to the left-most leaf, which has logical block number 0. The sum specifies the number of higher-level nodes that are interleaved with user data that is stored in front of the leaf with logical block number l. The depth d is a maximum, therefore, the offset is fixed. Thus, depending on the actual depth of the tree, there may be unused data blocks at the beginning of a file. This layout allows the embedded tree to grow in height as needed, I shall elaborate on this aspect in Section 6.2.3.

6.2.2. Loading Data Blocks into the Buffer Cache

To load a data block from a file container into the buffer cache, the trusted component must know the reference hash sum to check the data block's integrity. Retrieving the hash sum for the root node of the embedded tree is trivial, as it is part of the meta data.[1]

[1] Although a file's meta data is stored in its own data block, it requires only a fraction of the storage capacity that is available in there. Such a solution is acceptable for file containers in untrusted storage,

However, to load any other data block, the trusted component has to make sure that the corresponding parent node is already in the buffer cache. There are two strategies to ensure that:

1. **Top-Down Approach:** For each data block that is to be loaded into the buffer cache, the trusted component descends in the embedded tree. Starting at the root node of the file container, it looks up all nodes along the path and, if the lookup fails, it reads the missing node from untrusted storage and authenticates it using the hash sum found in the direct parent node.
2. **Backward-Recursive Approach:** Once data blocks are in the buffer cache, their contents have been authenticated. The trusted component optimistically tries to exploit this fact by assuming that the parent node of the requested data block is already present in the cache. Starting with the direct parent node, it recursively looks up nodes along the path up to the root node. Recursion ends if either a node is found in the cache or after processing the embedded tree's root node. The trusted component loads nodes that are not yet cached in top-down order when unwinding the recursion stack.

The first approach is a simple and naive solution that illustrates the general idea. However, it does not provide good performance, because it requires many unnecessary buffer lookups.

In favor of efficiency, I decided on the second alternative, which is only moderately more complex. To load any data block that has been requested by a client application, the trusted component must calculate the corresponding physical block number as described in Section 6.2.1. To retrieve the authenticating hash sum, it needs the following information about the parent node:

- The physical block number of the parent node.
- An index to find the required hash sum inside the parent node.

An internal node of the embedded tree, that is, a node that can be parent node, contains only a vector of data records that describe each of its child nodes. Vector indices correspond to the relative location of the child nodes. For a data block on the leaf level, it is easy to calculate the index i based on its logical block number l:

$$i := l \mod m$$

Again, m is the number of child nodes that a node of the embedded tree can have. Using the data block's physical block number p and the index i, you can easily calculate the physical block number of the parent node:

$$p_{parent} := p - i - 1$$

In generalized form, the trusted component can use these formulae to precisely calculate the location of all per-node authentication data along the path in in the embedded tree:

$$i := \lfloor \frac{l \mod m^{h+1}}{m^h} \rfloor \qquad p_{parent} := p - i \cdot (m+1)^h - 1$$

In these formulae, the parameter h is the height in the tree, where $h = 0$ specifies the leaf level. This calculation method allows to retrieve parent lookup information iteratively in a tree of any depth.

however, allocating a cache buffer for about one hundred bytes of meta data is not. Therefore, the trusted component keeps meta data of currently opened files in a dedicated array in memory.

6.2.3. Adapting Depth and Breadth of the Embedded Tree

The required depth of an embedded tree depends on the size of the file. Non-leaf nodes store a 20-Byte hash sum and an 8-Byte backup version number for each of their child nodes. With each data block having a size of 4 KB, which is the hardware page size on IA-32 hardware, a node can have no more than 146 children. Thus, a tree of depth 3 would suffice for files with a size of about 12 GB. Some use cases might require more than that, so a depth of 4 is the minimum for a general-purpose file system.

However, a vast majority of files is much smaller, in the order of tens or hundreds of kilobytes. Using a tree of depth 4 would mean considerable overhead with regard to the number of required cache buffers and latency upon opening files. For example, opening a short file such as a shadow directory would require that the trusted component loads, decrypts, and authenticates four data blocks.

Unfortunately, TrustedFS cannot know in advance how large a file will eventually be. That is, it cannot decide on the optimal depth of the embedded tree when creating a new file. Therefore, I implemented support for growing and shrinking embedded trees at run time based on the file size set via the API function `tfs_set_file_size()`.

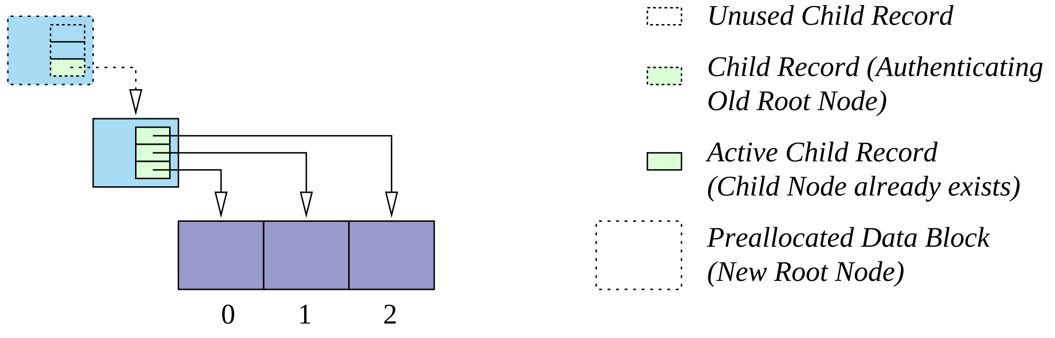
Unused Child Record
Child Record (Authenticating Old Root Node)
Active Child Record (Child Node already exists)
Preallocated Data Block (New Root Node)
0
1
2

Figure 6.2.: Growing the embedded tree in a file container.

Growing Embedded Trees: As I already mentioned in Section 6.2.1, the trusted component pre-allocates a number of data blocks at the beginning of a file container. These data blocks are nodes in the left-most path of the tree that, depending on the tree's actual depth, may be unused. To grow an embedded tree that has not yet reached its maximum depth, the trusted component adds one or more of these nodes on top of the old root node. That is, trees grow from the leaf level, as illustrated in Figure 6.2. This solution has the major advantage that the block numbers of already existing, and possibly cached, data blocks do not change. The Trusted Database System (TDB) [22] uses the same approach.

Shrinking Embedded Trees: To shrink an embedded tree, the trusted component removes the nodes in the left-most path down to the new root node of the shrunk tree. All other nodes that have a greater physical block number than those in the shrunk tree will be removed as well. That is, the trusted component instructs the untrusted block server to truncate the file container behind the right-most node that should remain in the tree. It must also invalidate all truncated data blocks in the buffer cache. However, finding these data blocks based on their unique file number and block number is efficient. The cache provides suitable lookup functionality, on which I shall elaborate in Section 6.3.5.

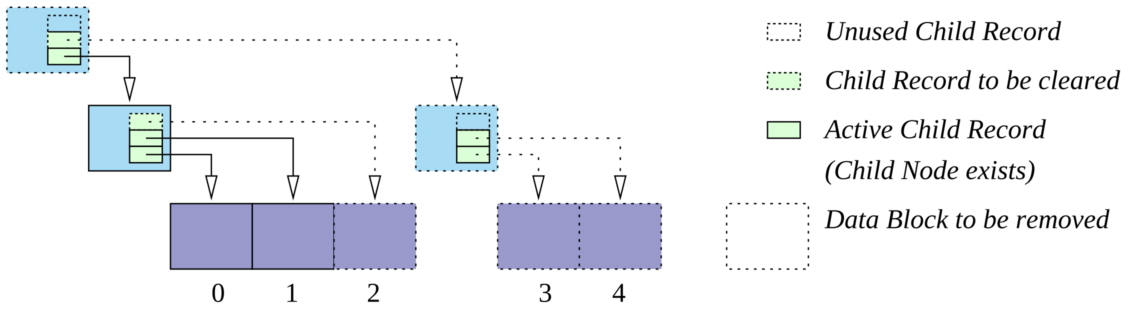
0
1
2
3
4
Unused Child Record
Child Record to be cleared
Active Child Record (Child Node exists)
Data Block to be removed

Figure 6.3.: Shrinking the embedded tree in a file container.

Removing higher-level nodes only makes the tree shrink vertically. To reflect reduction in breadth as well, the trusted component traverses all nodes along the right-most path of the remaining tree and clears all child records that belong to truncated nodes. Similarly, when growing the tree vertically, the trusted component creates new nodes with all their child records invalidated, except for those leading to the old root node of the tree. Figures 6.2 and 6.3 also illustrate this aspect.

Unfortunately, the code to grow and shrink embedded trees is quite complex. Although the concept is simple, the implementation must handle several corner cases, which mainly result from the tight coupling between the file-system wrapper and the buffer cache. Nevertheless, I consider the complexity added to the TCB to be justified, as this solution improves performance for short files significantly.

6.2.4. Sparse Files

If the trusted file-system wrapper encounters an invalid child record in a parent node, it will not try to read the node from untrusted storage. Instead, it simply allocates a cache buffer and clears its content.[2] Invalid child records can also appear at locations other than the right-most path in the tree. In such a case, they mark a gap in the file. Thus, TrustedFS supports sparse files.

Sparse-file support is necessary to allow memory-mapped file access. Because client applications can write to pages representing a mapped file in random order, files can, at least temporarily, have holes. However, the trusted component leaves the most difficult task, namely allocating file-system blocks, to the untrusted legacy OS.

6.3. Buffer Cache

In this section, I shall elaborate on the implementation of the buffer cache. I shall discuss requirements that arose from the necessity to ensure integrity and from the use of file containers in untrusted storage. I shall also give insight into the replacement and flushing strategies.

[2] Invalid child records can occur for both internal nodes and leaf nodes. An internal node with zeroed content implicitly contains invalid child records.

6.3.1. Replacement Strategy

I chose the *second-chance algorithm* as the buffer cache's replacement strategy. The algorithm itself is simple and its implementation enlarges the TCB by only 50 lines of code. Nevertheless, its efficiency is acceptable and it has been used in various Unix-like OSes, for example in Linux 2.4.

The second-chance algorithm requires information on whether a certain cache buffer has been referenced or not. The front end can provide this information to the buffer-cache implementation, whenever it requests a data block using the `_tfs_get_block()` function. However, pages containing cache buffers can also be mapped into other address spaces. On most hardware architectures, the required information is available at little costs in form of a *reference bit* in the corresponding page-table entry.

When I implemented the buffer cache, the stable version of the Fiasco microkernel could not provide applications with reference-bit values. Therefore, I decided on a user-level solution, which is implemented in a similar way in L^{4}Linux as well. The buffer cache uses software-implemented reference bits. Clearing these bits also involves unmapping the corresponding buffer page from other address spaces. Clients implicitly provide reference information when the file-system server is asked to restore a mapping upon a page fault.

Future versions of Fiasco that implement the *L4.Sec Interface Specification* [43, 44] provide a mechanism to read reference bits from hardware page tables. The experimental *X2 Interface* [45] provides similar functionality. Although these interfaces will not accelerate the buffer-cache implementation itself, they can at least save the client applications unnecessary page faults.

6.3.2. Making the Buffer Cache Aware of Parent–Child Relationships

To make the buffer cache aware of the parent–child relationships among data blocks, I added additional attributes for each cache buffer. I introduced a reference counter for each buffer to make it possible to keep track of all child blocks that are present in the cache. The buffer cache will not free a buffer unless its reference counter is zero. A parent pointer, which I also added for each buffer, allows the cache implementation to find a buffer's parent. Thus, updating reference counters requires only minimal run-time overhead.

A second reference counter prevents buffers from being flushed (without evicting data blocks) while there are any child buffers with their dirty flag set. Flushing such buffers would cause unnecessary input–output traffic, because the buffer's contents would have to be written back again, once all its child buffers got updated. Also, flushing parent buffers does not guarantee the consistency of the data in untrusted storage, unless all nodes along a path in the embedded tree and in the master hash file are flushed as well. Finally, the master hash sum in sealed memory would have to be updated, too. Obviously, the even further increased input–output load caused by such an approach would be beyond acceptable limits.

The first reference counter effectively pins parent nodes as long as there are child nodes in the cache. This approach implicates an important constraint regarding the replacement strategy. Pinned data blocks must not be chosen for eviction. Their number is relatively small, less than one percent for files larger than 2 MB, although this ratio increases for files that are only a few kilobytes in size. Nevertheless, I decided to omit a test for the number of child buffers from the replacement algorithm. Instead, the function `alloc_buffer()` keeps calling the second-chance search function until it returns a buffer that contains a node without cached children. The run-time costs for this simple solution are negligible compared to the overall costs of evicting a possibly modified data block.[3]

6.3.3. File-Handle Consumption Caused by the Buffer Cache

As every file in the trusted file system corresponds to a file container in the untrusted storage, it is unavoidable that the trusted and the untrusted component share a common file handle for each opened file container. Thus, the maximum number of open files is also limited by the legacy OS. Running multiple client applications can cause the number of available file handles to deplete, however, tweaking the legacy OS or running several instances of the untrusted block server can solve this problem.

Unfortunately, even a single client making small changes to a sufficiently large number of files in a short time can cause the file-system wrapper to run out of file handles. For each of these files, the cache will contain at least one dirty buffer that is associated with a distinct file handle. However, although the client application may have closed all files properly, the trusted file-system component cannot free the file handles while the cache still contains modified data from these files.

If the trusted file-system wrapper runs out of file handles in such a situation, it must instruct the buffer cache to flush dirty buffers, so that a file handle becomes available. The buffer cache can easily flush buffers of a specific file using the approach that I shall describe in Sections 6.3.5 and 6.3.7.

6.3.4. Flushing Single Cache Buffers

Each cache buffer contains a single data block. Both data blocks and cache buffers are exactly the same size as a memory page. If the contents of a buffer have been modified, the trusted component must write them back to untrusted storage eventually. For each buffer, this flush operation is performed in three steps:

1. If the buffer is mapped into other address spaces, the cache revokes write permissions for all these mappings.
2. The block-server stub calculates the authenticating hash sum of the cached data block, writes it to the data block's parent node, updates the backup version number in the parent node, and encrypts the data block.
3. The block-server stub transmits the encrypted data block to the untrusted block server.

These steps are performed atomically with regard to the server library's API and page-fault resolution.

6.3.5. Flushing Multiple Cache Buffers at Once

If the cache runs out of free buffers, it must evict data blocks. In such a situation, the second-chance algorithm chooses the buffers that are to be evicted. Thus, it can happen that the cache quickly flushes many data blocks that come from various locations in

[3] The trusted file server never maps internal nodes of an embedded tree to client applications, thus, it does not need to perform an expensive unmap operation when clearing the software-implemented reference bit.

untrusted storage. However, random write access to file containers is slow. To improve performance, the cache flushes dirty buffers in advance while the file-system implementation is idle. Doing so decreases latency when it becomes necessary to evict modified data blocks from the cache at a later time.

For maximum performance, even during idle periods, the cache must write back data blocks in their natural order. Nevertheless, it must respect the parent–child relationships among data blocks. To allow the cache to find the buffer that needs to be flushed next efficiently, I implemented a dedicated scanning routine that is based on a special lookup function for the AVL tree.

I extended the AVL-tree implementation by adding the function `avlt_find_greater()`. It finds the node with the smallest key that is greater than the key given as the function's argument. As I explained in Section 5.6.2, tuples of the form (u, n) uniquely identify data blocks, where u is the unique file number and n the block number. Given a greater-than relation that is defined on these tuples as follows, it is possible to use them as keys for the AVL tree:

$$(u_1, n_1) > (u_2, n_2) \Leftrightarrow (u_1 > u_2) \vee ((u_1 = u_2) \wedge (n_1 > n_2))$$

The aforementioned scanning routine uses `avlt_find_greater()` to find the, not necessarily immediate, successor of a given data block, which may also be the first data block of another file according to the defined order.

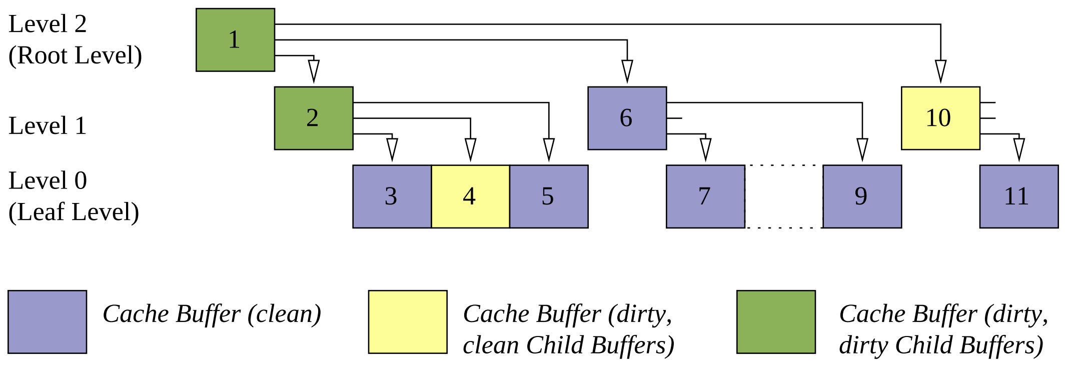
Level 2
(Root Level)
Level 1
Level 0
(Leaf Level)
1
2
6
10
3
4
5
7
9
11
Cache Buffer (clean)
Cache Buffer (dirty, clean Child Buffers)
Cache Buffer (dirty, dirty Child Buffers)

Figure 6.4.: Parent–child relationships among data blocks from the same file in the buffer cache (clean and dirty buffers).

Additionally, the implementation skips data blocks that do not need to be flushed because they were not modified. As I explained in Section 5.6.1, the parent–child relationships among data blocks directly map to the cache buffers that contain these data blocks. Because a clean cache buffer can, by construction, never have dirty child buffers, the scanning routine can use `avlt_find_greater()` to skip whole sub trees of clean nodes. Figure 6.4 illustrates the idea. Originating from the current node, the scanning routine decides on where to continue based on the following three cases:

1. **Current Node is a Leaf Node:** If the current node's buffer is marked as dirty, the cache flushes the buffer. The scanning routine will continue with the successor of the current node.

2. **Current Node has Only Clean Child Nodes:** If the current node's buffer is marked as dirty, the cache flushes the buffer. The scanning routine will skip all child nodes.

3. **Current Node has at Least One Dirty Child Node:** The cache does not flush the current node's buffer. The scanning routine will continue with the successor of the current node.

If the buffers that need to be flushed form trees with a maximum depth of d, the scanning routine must be executed at most d times to flush all buffers. In each pass, the implementation will flush the lowest level containing dirty buffers of each tree or sub tree.

6.3.6. Flushing All Dirty Cache Buffers

Flushing all dirty cache buffers during idle periods is the responsibility of a dedicated thread. This thread periodically looks for dirty buffers using the scanning routine that I described in Section 6.3.5. It starts scanning with the first block of the master hash file, which has the lowest unique file number, and continues until `avlt_find_greater()` returns no more buffers.

Because the internal functions of the server library are not yet thread safe, the flush thread must acquire the global API lock before it can start to flush buffers. To minimize interference with client requests, I implemented a knocking mechanism in the front-end layer of the trusted file server. Whenever a client application calls the trusted server, either directly or indirectly because of a page fault, the front end atomically increments a global counter right before it requests the API lock. While holding the API lock, the flush thread periodically checks this counter's value and, if the value is greater than zero, it stops flushing buffers, releases the API lock, and sleeps for a short time. After the server library completed a client application's request, the front end atomically decreases the counter and releases the API lock. Unless further client requests come in, the flush thread can resume its work when it awakes.

6.3.7. Flushing a Specific File

Flushing cache buffers that belong to a specific file works essentially the same way as flushing all dirty buffers in the cache. The only difference is that the scanning routine will only look for buffers containing data blocks with a specific unique file number.

6.4. Error Handling

6.4.1. Types of Errors

While using TrustedFS, two types of errors can occur: (1) *normal errors* and (2) *integrity errors.*

Normal errors can occur even if the untrusted component functions as expected. For example, opening a file will fail if it indeed does not exist in the file system.

All errors that are caused by falsified or missing data from the untrusted domain are considered to be integrity errors. For example, if the untrusted block server could not open a file container that is supposed to exist, the trusted component will report an integrity error.

6.4.2. Server Library

The implementation of the server library checks function arguments that a client application passes in through the front end and it reports error conditions regarding these arguments immediately to the application.

Obviously, the library implementation must validate all data and information it receives from the untrusted component. If the implementation detects that data from untrusted storage has been falsified, it aborts the currently ongoing operation. If necessary, it performs a rollback of all changes that it did to the file-system contents since the operation started. It then reports the error condition to the client application.

Except for information about currently opened files, the public API of the server library is stateless. That is, the library does not record any information about errors that occurred previously. Even if an integrity error occurred, the implementation tries to perform any operation that a client application requests at a later time. Thus, a user can manually backup current file-system contents that have not been damaged, before recovering the file system from the backup copy stored on the trusted backup server.

6.4.3. Client Library

As I already explained in Section 5.8.3, client applications can easily detect error conditions when calling functions of the library API. However, the trusted file-system server provides file contents as dataspaces, which client applications must map into their address spaces. Thus, accessing file contents causes page faults that the application's region-mapper thread must resolve by calling the dataspace-provider thread in the server task.

To prevent an application from crashing, the client library installs a callback function, which the region mapper calls in case it cannot resolve a page fault. This callback function uses `l4rm_lookup()` to determine, whether the unresolvable page fault occurred in a virtual-memory region that contains memory-mapped file contents. If the region indeed represents a file, the function calls `l4_thread_ex_regs()` to set the faulting thread's instruction pointer to an exception handler. The exception handler in turn uses the C-library function `longjmp()` to resume thread execution at a user-defined entry point, which has been set previously using `setjmp()`.

The library provides the preprocessor macro `TFS_BEGIN_EXCEPTION_AND_CATCH`, which encapsulates `setjmp()`:

```
int exception_file;

TFS_BEGIN_EXCEPTION_AND_CATCH(exception_file);
if (exception_file != 0) {
    /* handle exception in file 'exception_file' */
}

/* code section accessing memory-mapped file contents */

TFS_END_EXCEPTION; /* catch no more exceptions */
```

The corresponding macro `TFS_END_EXCEPTION` ends a code section in which page-fault exceptions should be caught and handled in the `if`-branch. These code sections can also be nested, like `try` and `catch` blocks in C++.

6.5. Reused Components

TrustedFS uses the Advanced Encryption Standard (AES) to encrypt file contents and the Secure Hash Algorithm (SHA-1) to calculate hash sums. I chose implementations from the Linux kernel for both algorithms, as they have no dependencies to other source code. I implemented a thin adaptation layer, which allows them to be used in the server library with almost no changes.

Currently, the server library contains a modified copy the L4 Region Mapper's AVL-tree implementation. This modified version should be moved into a separate L4 package, so that only one implementation needs to be maintained.

6.6. Issues of the System Platform

Using currently available hardware, it is not possible to prevent legacy OSes from violating address-space boundaries, if they use direct memory access (DMA). Therefore, an untrusted L^4Linux instance using DMA can access physical memory that belongs to trusted components.

A software-based solution would be to let L^4Linux use the disk controller through a separate trusted driver, for example using the Linux Device Driver Environment [46]. In his PhD thesis *Kapselung von Standard-Betriebssystemen* [47], Frank Mehnert presented another software-based solution for this problem. There also exist hardware solutions, for example, Christian Böhme developed a PCI-to-PCI bridge [48] that allows to enforce DMA restrictions. Upcoming hardware-virtualization technology, such as AMD's *"Pacifica"* technology [49], can solve DMA-related problems as well.

7. Evaluation

In this chapter, I shall discuss the performance of TrustedFS and I shall evaluate the implementation with regard to size and complexity of the trusted computing base (TCB).

7.1. Performance

When evaluating the performance of a file-system implementation, it is common practice to use a standard benchmark utility such as bonnie++ [50]. However, this approach is not an option to evaluate the performance of TrustedFS, because these benchmarks only work on the operating system for which they have been created. Instead, I developed three custom benchmarks to examine the performance with regard to (1) throughput, (2) page-fault resolution, and (3) latency when opening files. These aspects are of particular interest, because they allow to determine the performance overhead introduced by the trusted file-system wrapper and the cryptographic operations.

7.1.1. Test Environment

Hardware: I ran all performance tests on a system with an AMD Athlon64 2800+ processor, which operated at a clock rate of 1800 MHz. The system was equipped with a Via K8T800 chipset, 512 MB DDR RAM, and an 80 GB hard disk (model Maxtor 98196H8). For all test series, I used a single disk partition, on which I created a fresh *ReiserFS*[1] file system [51] before performing any write tests.

Using the *hdparm* utility on a native Linux 2.6.14, I measured a maximum transfer rate of 28.1 MB/s for read operations. However, when reading an unfragmented file of roughly 260 MB from the test partition, the hard disk could deliver only 19.5 MB/s (the file was not in the native Linux' page cache). For the latter measurement, I used the system utility *dd* with `/dev/null` as the output file.

Software: The test system was running the L4/Fiasco microkernel and L^{4}Linux 2.6.15. L^{4}Linux was using the L4 graphical console and it was given 128 MB of physical memory. Fiasco and all user-space applications including L^{4}Linux were configured to use system call entry code in the kernel info page. Due to the kernel extensions required by L^{4}Linux, the assembler IPC shortcut was disabled. Except for the throughput benchmarks, all tests were performed with the kernel option 'fine-grained CPU time' enabled.

I compiled the TrustedFS file server with all assertions and logging statements disabled and I configured it to use 64 MB of physical memory as its trusted buffer cache.

7.1.2. Throughput

Throughput measurements allow for getting an impression of the overall performance when reading and writing large amounts of data. However, the benchmark results discussed

[1] ReiserFS version 3.6.

in this section are preliminary, as I did not yet implement the memory-mapped block-server interface that I described in Section 5.7. Instead, the implementation uses a shared memory buffer to transfer single data blocks synchronously between the trusted component and the untrusted block server.

I measured read and write throughput for both encrypted and plaintext files. When using plaintext files, the trusted component just needs to copy data between its buffer cache and the memory region that it shares with the untrusted block server. Nevertheless, it must still calculate each data block's checksum to be able to ensure integrity. For encrypted files, the trusted component uses the encryption and decryption routines rather than the `memcpy()` function to move data between the two domains of trust.

The benchmark consisted of two phases: (1) writing a new file of 256 MB in size and (2) reading the same file back into memory. Both tests were performed with empty caches (I rebooted the test system before performing the read benchmark). Figures 7.1 and 7.2 visualize the throughput that the benchmark application measured for each megabyte of data that it has read or written.

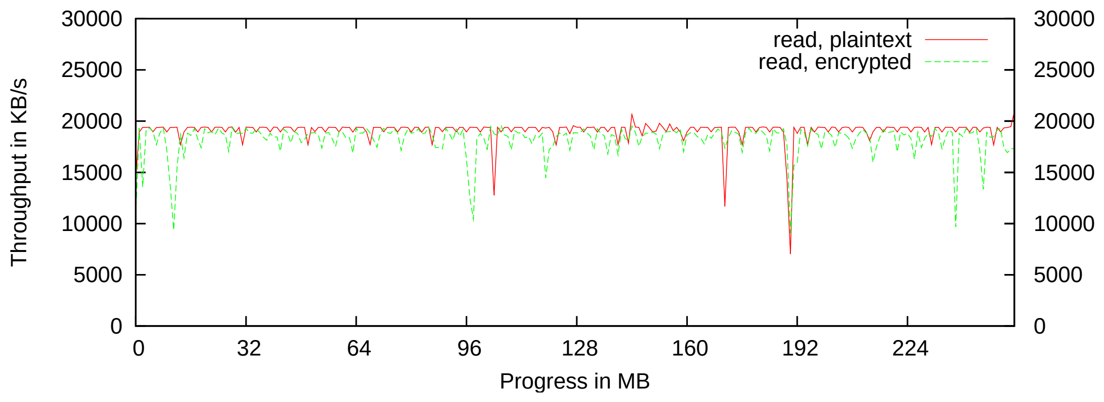

Figure 7.1.: Throughput when reading a file (encrypted contents and plaintext contents).

Based on the results of the micro benchmark that I shall discuss in Section 7.1.3, I determined the maximum throughput that TrustedFS can achieve on the test system. For encrypted file contents, the cryptographic operations limit the transfer rate between the trusted and the untrusted domain to 22.7 MB/s. The theoretical maximum when reading or writing plaintext files is 87.9 MB/s. Given the hard disk's transfer rate of 19.5 MB/s, which I mentioned in Section 7.1.1, you would expect the hard disk to be the limiting factor when reading data.

TrustedFS performs slightly better with plaintext files, which the benchmark application could read at 18.5 MB/s compared to the average read rate of 17.6 MB/s for encrypted files. Clearly, throughput for plaintext files is limited by the hard disk. However, for encrypted files, the CPU seems to be the limiting factor, although the measured throughput is only 78 percent of the theoretical maximum. You can see from Figure 7.1 that the plot for the encrypted file is below the plot from the plaintext benchmark. Also, the transfer rate is not constant, but the diagram shows negative peaks. These anomalies are not caused by the trusted file-system wrapper, but L^{4}Linux itself. I did not have the time to determine the exact cause. A possible explanation is that the file containers got fragmented while flushing the buffers during the write phase of the benchmark, however, this assumption has to be verified in a more detailed analysis.

Throughput	Encrypted file contents	Plaintext file contents
Read	17.6 MB/s	18.5 MB/s
Write (average)	23.7 MB/s	32.0 MB/s
Write (cache not full; no flushing)	261.0 MB/s	261.4 MB/s
Write (cache full; with flushing)	18.3 MB/s	24.9 MB/s

Table 7.1.: Throughput when reading and writing files.

The average throughput when writing files is 23.7 MB/s for encrypted and 32.0 MB/s for plaintext files. However, these results are not representative, because they are highly influenced by both the buffer cache of the trusted file server and L^{4}Linux' page cache. They depend on the the size of the file that is written and on the size of the caches. The trusted component does not transmit any data to the untrusted block server, as long as there are unused buffers available in its buffer cache, which has a capacity of 64 MB. Thus, at the beginning of the write benchmark, throughput is determined by the performance of page-fault resolution including allocation of a new cache buffer. The measured throughput is roughly 261 MB/s.

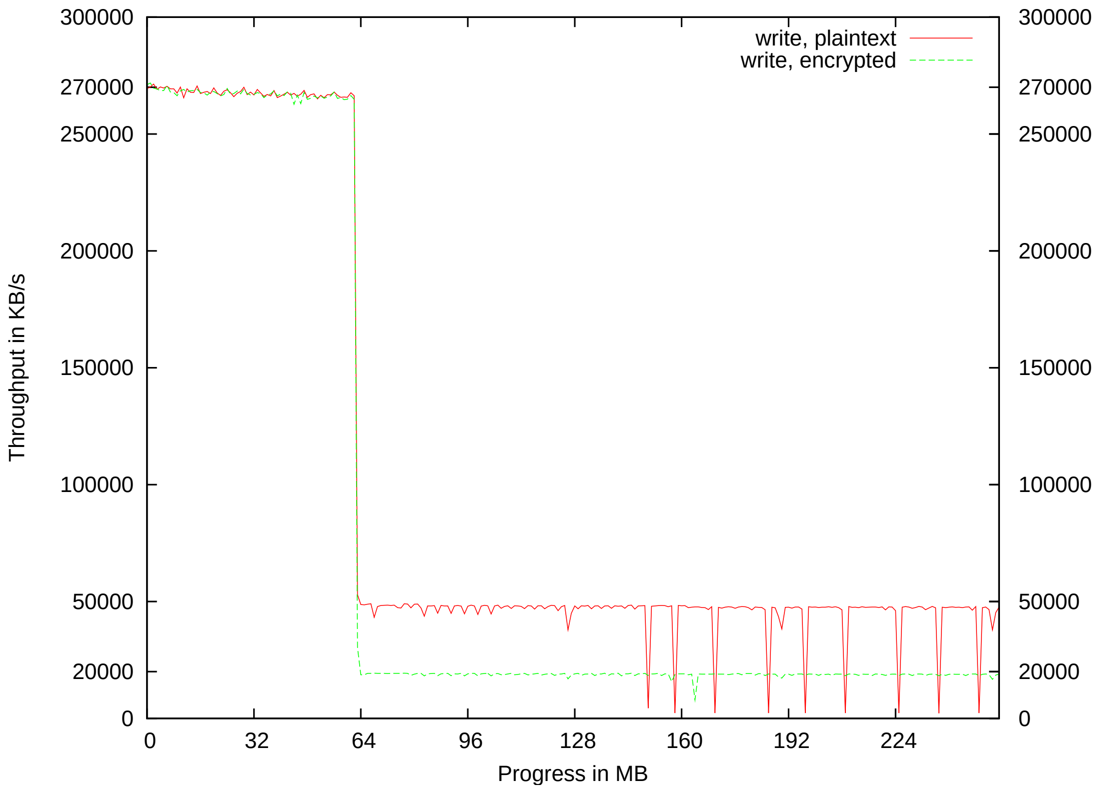

Figure 7.2.: Throughput when writing a file (encrypted contents and plaintext contents).

When writing an encrypted file, the trusted component flushes the corresponding buffers at an almost constant rate of 18.3 MB/s. Figure 7.2 also shows that it is possible to flush buffers of plaintext files at approximately 47 MB/s. In both cases, these figures include necessary overhead caused by page-fault resolution and transferring the evicted data block to the untrusted block server. I shall give a more detailed analysis regarding this overhead in Section 7.1.3.

As the flush rate for encrypted data blocks is close to the transfer rate of the hard disk, L^{4}Linux is able to write the received data blocks to disk quickly enough. However, the increased flush rate when writing the plaintext file causes L^{4}Linux' page cache to overflow. As I assigned only 128 MB of physical memory to L^{4}Linux, the first of these overflows occurred after the benchmark application wrote about 150 MB of data, of which 64 MB were still in the trusted buffer cache. To free buffers in its page cache, L^{4}Linux writes large chunks of data to disk. This operation takes considerable time, thus, the untrusted component blocks in the `write()` system call. The diagram in Figure 7.2 shows the corresponding negative peaks. As a result, the average flush rate for plaintext files, 24.9 MB/s in this benchmark scenario, is lower than the maximum rate of approximately 47 MB/s.

Nevertheless, I consider the performance of TrustedFS to be acceptable regarding both read and write throughput.

7.1.3. Page-Fault Resolution

To precisely determine how much overhead the trusted file-system wrapper and the cryptographic operations cause with regard to throughput, I also performed micro benchmarks. I instrumented the code path that is responsible for mapping data blocks into a client application's address space with performance sensors, which collected timing information. For this purpose, I used the *Ferret* performance-measurement framework, which Martin Pohlack develops internally at the Operating-System Research Group at TU Dresden.

When reading data from from a memory-mapped file on disk, the trusted file server resolves page faults as follows:

1. Upon a page fault in a client applications address space, the microkernel switches to the corresponding region-mapper thread, which calls the dataspace-provider thread of the trusted file server.

2. After looking up the parent node of the requested data block, the trusted file server instructs the untrusted block server to load the data block.

3. The trusted file server decrypts (or copies) the data block into its buffer cache and verifies the hash sum of the data block.

4. The trusted file server sends the mapping for the cache buffer containing the requested data block back to the client application's region-mapper thread.

Again, I performed the benchmark for both an encrypted and a plaintext file. The benchmark application read 8 MB of data from the files that were created during the write phase of the throughput benchmark. For the last 1024 data blocks (4 MB of data), the benchmarking code collected timing information.

Table 7.2 shows the average execution times in CPU cycles that were measured for each of the involved components. It is not surprising that the cryptographic operations account for most of the CPU time consumed by the trusted file server. The discrepancy of approximately 1,600 CPU cycles in the table row denoted as 'Other' is caused by

	Encrypted file contents	Plaintext file contents
Wrapper (total)	**316,452**	**84,881**
Cryptography	309,892	79,979
Other	6,561	4,902
Block sever/L^{4}Linux (total)	**58,176**	**263,859**
IPC (total)	**20,709**	**20,291**
IPC (client and file server)	9,216	9,029
IPC (file server and block server)	11,492	11,262
Page-fault resolution (total)	**385,765**	**366,367**

Table 7.2.: Performance data of page-fault resolution.

cryptographic overhead as well. These figures include the costs to decrypt and authenticate parent nodes and to allocate a cache buffer.

However, the trusted trusted file server waited 72 percent of the average time to resolve a page fault until the untrusted block server retrieved a plaintext data block. When reading an encrypted file, it had to wait approximately 58,000 CPU cycles, or 15 percent, yet the average throughput is comparable in both cases, as mentioned in Section 7.1.2. These results show that L^{4}Linux can effectively read data in advance while the trusted file server performs CPU demanding cryptographic operations.

Nevertheless, as you can also see from the raw performance data in Appendices B.1 and B.2, the average time that it took to retrieve one data block does not reveal all performance-related characteristics of the block-server interface. The block server could provide more than 95 percent of the plaintext data blocks within 22,500 CPU cycles, including IPC costs of approximately 11,200 CPU cycles. The costs for retrieving an encrypted data block are comparable. However, the average costs are significantly higher, because the untrusted block server itself occasionally has to wait several million CPU cycles for the hard disk to provide the requested data block.

With regard to the overhead caused by the trusted file-system wrapper, only the best-case time of approximately 22,500 CPU cycles is relevant. Thus, with the influence of the hard disk eliminated, TrustedFS requires approximately 116,000 CPU cycles to map a plaintext data block into the benchmark application's address space, if the data block is not yet in the buffer cache. For the encrypted file, the costs are 348,000 CPU cycles. In both cases, these costs include the time required to allocate a cache buffer, to retrieve the parent node if necessary, and the IPC[2] between the benchmark application and trusted file server. Thus, for encrypted files, the cryptographic operations account for 89 percent of the overall time to resolve a page fault. Copying a plaintext data block into the buffer cache and hashing it requires 69 percent accordingly.

As the cryptographic operations and the IPC costs cannot be avoided, there is little room for optimizations, if the user requires files to be encrypted. In this case, retrieving the data block takes six percent of the overall time. For plaintext files, a more efficient block-server interface could improve performance notably, as retrieving a data block accounts for 19 percent of the overall time. With the memory-mapped block-server interface that I described in Section 5.7, the implementation could potentially reduce these costs. The

[2] These costs include the time required to switch to the region mapper, to determine and contact the dataspace provider, marshalling and unmarshalling of the transferred information, and eventually send the mapping back to the region mapper.

savings could also increase the flush rate of the trusted buffer cache, however, as the hard disk is most likely the limiting factor when reading and writing data blocks, such an improvement might only reduce CPU utilization.

7.1.4. Opening Files

To ensure integrity on file level, the trusted component must retrieve a file's master entry using the meta data in the corresponding file container, whenever a client application opens a file. I explained the details in Sections 5.3.3 and 5.4.1. To be able to quantify the overhead caused by TrustedFS' security measures, I performed another micro benchmark, for which I instrumented the code path that is executed when opening a file.

While opening 50 distinct files in the root directory of the file system, the benchmarking code measured the time required to execute the relevant sections of this code path. To allow for an estimation of the average costs for retrieving uncached master entries, I modified the trusted component, so that it stored each master entry in its own data block in the master hash file. However, in this scenario, I assume that it is not necessary to revalidate the hashed pathname stored in the file's master entry, as described in Section 5.4.2. The purpose of this benchmark is to evaluate both worst-case and best-case performance. At the beginning of the benchmark, both the buffer cache of the trusted file server and L^{4}Linux' page cache were empty. After closing all files, the benchmarking application repeated the benchmark with warm caches. Table 7.3 shows the average time in CPU cycles that the involved components required to execute the steps necessary to open one file.

	Cold caches	Warm caches
Wrapper (total)	**331,945**	**18,686**
Cryptography	322,802	14,575
Other	9,143	4,111
Block server/L^{4}Linux (total)	**5,385,160**	**27,927**
Open file container	18,979	7,518
Read meta data	5,349,793	5,386
Retrieve master entry (total)	**2,066,865**	**852**
IPC (total)	**22,155**	**18,871**
IPC (client and file server)	9,767	6,703
IPC (file server and block server)	13,106	12,168
Open file (total)	**7,481,640**	**52,186**

Table 7.3.: Latency when opening files (cold and warm caches).

With cold caches, the untrusted block server requires most of the time to open the file container and to read the meta data. In average, performing these tasks takes approximately 5.4 of the 7.5 million CPU cycles that the benchmarking application measured for the `tfs_open()` call in total. Opening the file container itself is fast. It seems that, when opening the master hash file before starting the benchmark, L^{4}Linux already cached the inodes and directory entries of the other file containers. However, reading the meta data takes significantly more time, because L^{4}Linux has to wait for the hard disk. The hard disk in turn cannot read the data before its read–write head reached the disk sector that contains the meta data. Although this seek operation increases latency when opening an uncached file container, it does not cause a slowdown in practice, because it is necessary when accessing a file's contents in any case.

Loading the data block of the master hash file that contains the corresponding master entry takes roughly 2.1 million CPU cycles, of which the trusted file server requires approximately 306,000 CPU cycles to decrypt and authenticate the data block. Thus, the costs of retrieving a master entry in this benchmark are 28 percent of the overall time that is required to open a file. Although this percentage might increase under different work loads, I consider the overhead to be acceptable given the integrity guarantees, for which it allows.

As you would expect, opening a file with warm caches is much faster. Because master entries were cached, too, the average CPU time consumed by the trusted file server decreased from approximately 332,000 down to 18,686 CPU cycles, of which 78 percent were required for cryptographic operations. Particularly, these costs result from pathname encryption and decrypting and hashing of the meta data that the untrusted block server supplied.

There is much room for optimizations. For example, when creating or removing a large number of files in the same directory, the trusted component needs to open and close the corresponding shadow directory for each of these file operations. In the current implementation, the trusted component always passes the encrypted pathname of the shadow directory to the untrusted block server to retrieve the meta data, which it then decrypts and authenticates. This is inefficient, because the file container of the shadow directory is still opened, if there are dirty buffers left in the buffer cache. I explained this aspect of the implementation in Section 6.3.3.

A simple lookup cache could save the costs involved in contacting the untrusted block server. This cache could map the hashed pathnames of currently opened file containers such as the aforementioned shadow directory to the corresponding file handles. With regard to the described scenario, I estimate that the costs for reopening a shadow directory can be reduced to approximately 2,000 CPU cycles. Client applications that reopen a file might benefit from such an optimization as well. However, in this case, IPC costs for communication between the client application and the trusted file server and the costs for calculating the hash sum of the full pathname have to be taken into account as well.[3] For client applications, the costs could be reduced to approximately 15,000 instead of 52,000 CPU cycles.

I estimate that it is possible to implement this lookup cache in about one hundred lines of code by reusing the AVL-tree implementation.

7.2. Code Complexity

The main objective of my work has been to create a trustworthy file-system implementation, which should be implemented with only a small trusted computing base (TCB). In this section, I shall analyze size and complexity of TrustedFS' source code. However, as the implementation is not yet complete, I can only evaluate its current state, which I shall discuss further in Sections 8.1 and 8.2.

The code base of TrustedFS is split into two major parts: (1) the trusted component and (2) the untrusted component. The former is divided into the server library, which

[3] When opening a shadow directory internally upon file creation or removal, the trusted component reuses parts of the target file's encrypted pathname.

is the core of the file-system implementation, and a font end that allows client applications to access the file system conveniently. Table 7.4 summarizes the complexity for all components and their subsystems.[4] These figures do not include debugging code such as assertions or logging statements.

	Lines of code
Server library (total)	**3,369**
File-system wrapper	1,713
Cryptography layer	789
Buffer cache	455
Block-server layer	181
Other	231
API Front end (total)	**613**
Client-side API library	236
File Server (API)	162
File Server (dataspace manager)	215
AVL-tree implementation	953
Untrusted block server	**808**

Table 7.4.: Source-code complexity of TrustedFS.

Source Lines of Code: In its current state, the file-system wrapper accounts for about half of the server library's overall code base, which consists of 3,369 lines of code. Thanks to the efficient reuse of already existing code, the implementation of the buffer cache has a size of only 455 lines of code. However, although there currently is a copy of the AVL-tree implementation in the source tree of TrustedFS, I do not consider it part of the server library. The reason is that the TCB of all L4Env applications already includes the AVL tree, as I mentioned in Section 5.6.2.

The front end consists of both the trusted file server, which is based on the server library, and a client-side library that wraps the application programming interface (API). The file server also implements the dataspace manager that is required to support memory-mapped file access. In total, the front end consists of 613 lines of code. Thus, the current implementation of both the server library and the front end add only about 4,000 lines of code to an application's TCB.

In general, the presented figures prove that it has been the right decision to base the design of TrustedFS on a wrapper that reuses the file-system infrastructure of an untrusted legacy operating system (OS). Approximately 450 of the 1,713 lines of code that belong to the wrapper implement the naming infrastructure required by the pure-wrapper approach. This code implements pathname encryption, shadow directories, and error handling. Of course, the complexity of the file-system wrapper will increase, once I added the missing functionality, on which I shall elaborate in Section 8.2. I estimate that the final version will have a size between 2,500 and 2,700 lines

However, the trusted file-system wrapper does not only provide access to the functionality that is provided by untrusted components, but it is also responsible to ensure confidentiality, integrity, and recoverability. Thus, I consider it hard, or even impossible, to

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

implement a complete file system inside the TCB that provides comparable functionality and performance with as little code as I estimated previously for the file-system wrapper of TrustedFS. Reusing an existing implementation seems even more difficult. For example, the source tree of the Linux kernel contains file-system back ends that are implemented in less than 2,000 lines of code. However, these file systems provide only limited functionality[5] and their implementations depend on Linux' Virtual-File-System (VFS) layer, which consists of more than 10,000 lines of code.

Cyclomatic Complexity: Thomas J. McCabe introduced another commonly used complexity metric [52], which measures the cyclomatic complexity of source code. In essence, the cyclomatic complexity of a function is the number of possible execution paths that result from conditional statements in the source code. Thus, higher numbers indicate higher functional complexity.

Using the tool *pmccabe* [53], I measured the average cyclomatic complexity of all functions in the source code of TrustedFS. The result of 3.1 is low. In contrast, I measured average cyclomatic complexities of 5.9, 5.6, and 6.5 for the implementations of the FAT file system, Ext2, and ReiserFS respectively in the Linux kernel 2.6.14. Thus, according to the McCabe metric, TrustedFS' functional complexity is lower than those of commonly used existing file-system implementations.

Code Reuse: Although the untrusted component does not necessarily have to be small, it has been a design goal of my work to reuse as much as possible of the already available infrastructure. The implementation of the untrusted block server, which I implemented in about 800 lines of code, fulfills this demand. Thus, TrustedFS can benefit from the efficient file-system implementation provided by the legacy OS.

[5] Only the Minix files system and the MS-DOS file system have stable write support. However, they impose unacceptable restrictions on file size and the supported length of filenames.

8. Conclusion and Outlook

8.1. Current State of the Implementation

Unfortunately, I could not complete the implementation within the time frame of my thesis. Nevertheless, the core functionality has been implemented. It is possible to create files and directories as well as to remove them from the file system. Also, applications can resize files arbitrarily.

I implemented all encryption and integrity checks that I discussed in Chapters 4 and 5. When creating a file, an application can specify, whether the file should be encrypted or not.[1] Thus, TrustedFS can provide better performance, if confidentiality of file contents is not required. However, it is neither possible nor desirable to turn off integrity protection.

The implementation of the buffer cache is complete. I also implemented the client-side library including error handling as described in Sections 5.8 and 6.4. This library encapsulates all communication with the trusted file server, which provides the functionality of the server library trough a small front end. Client applications can map file contents into their address spaces using existing functionality of the L4 Region Mapper.

8.2. Open Tasks

The following functionality has not yet been implemented or completed:

- **Moving and Renaming Files:** Currently, it is not possible to move or rename files or directories. Also, I did not yet implement path revalidation as described in Section 5.4.2.

- **Backup and Recovery:** I implemented the scanning routine to collect modified data blocks in a file. However, the server library currently lacks the functionality that is necessary to find modified files based on the backup version number in their master entries. It cannot reconstruct a file's encrypted pathname either, as this functionality depends on path revalidation.

- **Sealed-Memory Support:** Currently, the encryption key is hard coded into the source code. Also, the implementation ignores the calculated hash sum of the master hash file's contents.

- **Optimizations:** I did not yet implement the optimizations for shadow directories as described in Section 5.3.3. The lookup cache to optimize reopening files that I described in Section 7.1.4 it not implemented either. Also, the communication interface between the untrusted block server and the trusted component does not yet use the mapping method that I described in Section 5.7.

[1] Encryption is enabled by default. Applications must explicitly request the creation of plaintext files.

8.3. Outlook

Apart from the open tasks that I mentioned in Section 8.2, there are other areas of future research. In Section 5.3.1, I outlined requirements for a file-system implementation that includes a complete naming infrastructure. By fully implementing file lookup in the trusted computing base (TCB), it would become possible to enforce access restrictions on file and directory level. Naming, file lookup, and access control could be moved into a second trusted component, whereas the first one just provides functionality that is necessary to handle files and to access file contents.

Robustness against system crashes is also subject to future work. Partial recovery using the backup copy on the trusted backup server, as described in Section 5.5.4, is one simple solution for repairing a file system. Nevertheless, writing a transaction log, which the trusted component can use to recover from a system crash, could limit data loss further. An interesting question in this context is how much of the necessary functionality can be excluded from the TCB, and implemented in the untrusted component instead.

8.4. Summary

It has been the main objective of my work to design and implement a trustworthy file system. A user should be able to trust this file-system implementation to store his data securely and reliably. Within the context of my thesis, I developed TrustedFS, which is able to ensure confidentiality of file contents and filenames. TrustedFS also gives strong integrity guarantees including freshness of all file-system contents and it can ensure recoverability of the user's data in case of data loss.

The design of TrustedFS is based on the Nizza Secure-System Architecture. The implementation is split into a small trusted component and an untrusted part, which reuses existing infrastructure provided by a legacy operating system (OS). Particularly, TrustedFS stores all file-system contents in the legacy OS's untrusted file system and protects this data using cryptographic means. With the trusted component running on top of a small microkernel-based system platform, this architecture allows to keep the TCB small. The file-system implementation leverages trusted-computing technology such as sealed memory and remote attestation, so that it can ensure confidentiality, integrity, and recoverability.

TrustedFS is a general-purpose file system that allows the user's trusted application programs to store arbitrary data. It is not only suitable for use in desktop computers, but also in mobile devices. However, the implementation is not yet complete. I discussed the current state and open tasks in Sections 8.1 and 8.2. Nevertheless, the performance measurements that I evaluated in Section 7.1 have shown that TrustedFS can provide acceptable performance despite the use of cryptography that is necessary to fulfill its security requirements.

Appendix A.

Glossary

CPU: Central Processing Unit.

PCI: Peripheral Component Interconnect. PCI is a input–output bus system used in most desktop and laptop computers.

IA-32: A processor architecture introduced by Intel Corporation that is used in most desktop and laptop computers. Also known as *x86* architecture.

Inode: Index node. In Unix file systems, the inode contains all meta data describing a file such as ownership information, access rights, or file size. It also contains information on where to find the data blocks (or clusters) containing the user data. It does not contain the filename, which is stored in directories.

L4Env: L4 Environment. L4Env comprises of a set of basic servers and libraries used for developing applications on top of the L4/Fiasco microkernel.

L4/Fiasco: The L4/Fiasco microkernel is a second generation microkernel that implements, among others, the L4v2 interface. It is largely written in C++ and supports real-time applications.

Appendix B.

Raw Data from Measurements

B.1. Page-Fault Resolution (Encrypted File Contents)

Total	Total CPU time FS	IPC to FS	Find parent	Alloc buffer	BS read	IPC to BS	IPC to FS	Crypt	Hash		IPC to client	Consume full page
348838	313336	3731	1497	1688	21505	7987	3394	247590	61039	241	10751	6542
348222	317272	3721	1467	1582	22413	8303	3387	251685	61031	256	5312	6678
346268	314538	3747	1424	1463	23136	7947	3445	248892	61135	365	5374	6689
344579	313271	3686	1484	1592	22594	7948	3407	247607	61033	310	5533	6646
344808	312932	3730	1437	1396	23087	8114	3384	247571	61033	250	5577	6587
343749	312879	3715	1398	1284	22408	7982	3378	247625	61040	276	5267	6469
348891	317598	3645	1398	1420	22815	7846	3410	252212	61078	276	5316	6636
344218	313150	3626	1394	1444	22445	7909	3380	247846	61017	241	5472	6637
344689	312861	3803	1430	1330	23061	8047	3385	247512	61034	241	5492	6581
344186	312981	3611	1391	1347	22937	8107	3384	247735	61017	241	5174	6695
346127	314451	3641	1373	1348	23139	7856	3439	248989	61135	365	5377	6693
348570	317120	3676	1446	1287	22743	7973	3414	251750	61026	310	5579	6649
344807	313221	3606	1510	1429	22997	8147	3388	247764	61017	250	5478	6535
391309	316731	3620	1386	1339	22143	7978	3384	250890	61356	544	5981	6417
342858	313732	3819	1492	1505	20747	6095	3411	248229	60935	276	5086	6636
344759	313072	3578	1405	1312	22921	8384	3384	247886	61017	241	5673	6635
456076	317765	3647	1492	1410	129158	8004	3723	251904	61182	327	6139	6616
343442	313202	3782	1444	1380	21787	8288	3392	247836	61017	241	5223	6697
345707	314647	3677	1395	1435	22504	7881	3434	249089	61145	365	5363	6637
344154	312869	3676	1456	1480	22674	7968	3411	247301	61032	310	5471	6596
344732	313258	3644	1479	1356	22742	8141	3390	247835	61033	260	5544	6593
347684	316986	3621	1384	1377	22277	8077	3376	251712	61010	283	5283	6417
344313	313663	3595	1397	1341	22204	7862	3415	248314	61102	285	5343	6635
343964	313025	3666	1378	1383	22325	7913	3376	247757	61034	241	5447	6692
344936	313409	3806	1502	1460	22738	8043	3387	247886	61034	241	5475	6586
343301	312928	3694	1368	1357	22063	8111	3385	247698	61017	241	5135	6696
413261	321613	3667	1370	1342	22904	7874	3442	255355	61694	623	6283	6511
343711	313438	3841	1625	1498	21484	6056	3425	247792	60893	310	5529	6455
344906	313092	3740	1446	1395	23167	8494	3410	247727	61022	250	5432	6593
343884	313035	3622	1413	1276	22362	7981	3390	247803	61017	286	5349	6301
343971	313817	3640	1384	1515	21828	7835	3409	248314	61084	286	5167	6635
386427	319831	3578	1381	1358	22416	7898	3373	254110	61262	506	5823	6694
341824	313064	3995	1482	1407	20164	5909	3389	247739	60891	241	5250	6583
343768	312976	3655	1394	1359	22503	8493	3390	247691	61034	241	5160	6695
346101	314619	3641	1431	1388	22999	7866	3448	249087	61135	365	5325	6696
344852	312921	3750	1435	1346	23244	8014	3402	247502	61033	310	5454	6596
349645	317147	3732	1483	1445	23689	8156	3408	251636	61043	275	5597	6536
343542	313134	3617	1428	1334	22087	7970	3391	247817	61034	285	5183	6360
344227	313451	3605	1392	1269	22436	7883	3399	248144	61102	285	5261	6635
344448	313188	3578	1429	1341	22731	7910	3382	247930	61034	241	5434	6696
345290	313318	3754	1469	1490	23218	8021	3390	247799	61034	241	5493	6582
348151	317021	3647	1416	1292	22829	8110	3383	251761	61027	241	5205	6696
391264	318434	3628	1475	1454	23052	7866	3450	249115	64511	656	5701	6594
342111	313528	3783	1587	1504	19780	5997	3439	247873	60883	310	5623	6598
345773	313278	3749	1464	1346	23912	8455	3380	247864	61034	294	5381	6594
344258	313276	3622	1494	1412	22496	7994	3376	247817	61039	285	5350	6357
348259	317700	3639	1421	1426	22266	7854	3403	252249	61095	285	5141	6636
343460	313065	3603	1434	1316	21782	7938	3373	247786	61034	250	5523	6695
451409	314105	3766	1504	1456	128096	8016	3630	248311	61181	311	5961	6590
344179	313370	3790	1500	1508	22349	8250	3390	247816	61033	250	5186	6696
...	...	...	...	...	...	...	...	...	...	...	...	...

B.2. Page-File Resolution (Plaintext File Contents)

Total	Total CPU	IPC	Find	Alloc	BS	IPC	IPC	Crypt	Hash		IPC to	Consume
	time FS	to FS	parent	buffer	read	to BS	to FS				client	full page
120233	84419	3785	1504	1656	21948	7912	3343	18608	61123	225	10583	6493
115362	84014	3707	1496	1514	23071	8011	3344	18470	61065	225	5053	6689
115225	83987	3735	1417	1422	22775	7713	3341	18464	61120	349	5192	6707
115608	84405	3695	1444	1575	22849	7728	3490	18604	61124	297	5260	6648
115938	84288	3726	1419	1372	23051	7928	3348	18898	61112	245	5341	6596
114665	84066	3662	1387	1309	22342	7767	3337	18862	61072	225	5059	6422
114759	84093	3680	1415	1416	22220	7720	3372	18726	61103	225	5182	6700
115298	84176	3643	1361	1408	22822	7895	3345	18867	61102	225	5122	6698
114765	84191	3809	1410	1271	22111	7871	3340	18891	61123	225	5139	6589
114949	83721	3635	1420	1340	23088	7893	3338	18432	61074	225	4983	6700
115504	84007	3653	1402	1354	23112	7728	3353	18547	61131	354	5190	6712
120327	84193	3686	1406	1274	27594	11806	3373	18816	61131	297	5350	6654
115049	84296	3652	1460	1386	22374	7922	3355	18889	61088	245	5210	6543
115020	84160	3611	1373	1346	22503	7734	3336	18946	61064	225	5203	6366
113664	84008	3622	1370	1371	21511	7747	3379	18734	61088	225	4996	6703
115343	84193	3587	1383	1285	22794	7859	3337	19000	61088	225	5227	6695
9147692	84804	3665	1434	1414	9053158	7850	3758	18854	61337	367	6606	6590
116398	84105	3888	1551	1491	23876	8857	3342	18443	61051	225	5069	6727
114241	84119	3680	1474	1419	21743	7746	3348	18559	61115	349	5162	6657
114762	84139	3704	1426	1456	22155	7795	3366	18590	61104	297	5254	6595
114882	84102	3628	1459	1349	22316	7937	3347	18654	61114	270	5323	6624
114280	83857	3638	1395	1371	22201	7787	3340	18593	61051	234	5056	6369
114033	83881	3640	1384	1367	21865	7753	3375	18601	61088	225	5109	6693
114473	83758	3667	1358	1276	22338	7901	3336	18598	61088	225	5173	6695
115135	84160	3835	1482	1423	22426	7924	3352	18621	61108	225	5220	6592
114885	83875	3735	1397	1375	22790	7879	3348	18571	61066	225	4966	11120
114344	83798	3664	1426	1307	22119	7719	3341	18377	61116	349	5222	6713
115268	84175	3711	1455	1400	22410	7789	3373	18633	61123	297	5472	6429
114610	83989	3736	1444	1390	22196	7927	3345	18586	61088	245	5167	6599
114399	83709	3632	1381	1221	22351	7757	3340	18617	61051	232	5172	6256
113761	84060	3640	1395	1478	21541	7745	3382	18646	61088	232	4994	6699
115028	83925	3603	1361	1352	22717	7870	3341	18672	61105	225	5242	6695
114441	83982	3902	1414	1360	22103	7894	3341	18580	61123	225	5032	6584
114989	83882	3721	1433	1340	22898	7952	3342	18493	61085	225	5042	6697
114482	83856	3684	1427	1378	22186	7719	3357	18384	61108	349	5190	6713
115296	84050	3725	1405	1328	22744	7767	3362	18636	61108	304	5267	6594
115338	84210	3767	1463	1390	22496	7933	3370	18730	61137	245	5334	6541
114396	83927	3629	1399	1326	22253	7744	3340	18689	61085	225	5042	6313
114325	83925	3639	1392	1238	22112	7748	3388	18732	61122	225	5117	6701
114543	84004	3602	1409	1308	22250	7858	3336	18743	61121	225	5136	6704
119041	88122	3797	1473	1350	22429	7863	3348	18711	65073	245	5160	6530
114664	83812	3637	1452	1258	22736	7853	3342	18516	61105	225	4980	6699
114721	83889	3707	1454	1395	22388	7701	3357	18377	61116	349	5183	6697
115770	84337	3714	1426	1442	22776	7742	3362	18709	61217	297	5424	6594
115126	84179	3736	1477	1346	22495	7949	3370	18732	61139	245	5206	6598
114598	83971	3629	1431	1365	22290	7763	3340	18667	61084	228	5161	6312
114592	84065	3639	1401	1410	22298	7755	3388	18697	61123	225	5042	6698
114569	84036	3602	1421	1290	22152	7880	3336	18750	61129	225	5248	6699
6915481	85099	3797	1502	1416	6820469	7851	3730	18618	61785	376	6664	6553
116769	84238	3852	1584	1568	24103	8911	3339	18423	61114	225	5054	6726
...	...	...	...	...	...	...	...	...	...	...	...	...

B.3. Performance Data for File-Open Operations (Cold Caches)

Total	Tot. CPU time FS	IPC to FS	Enc path	Open total	IPC to BS	open()	read()	IPC to FS	Crpyt meta	ME total	ME crypt	IPC to cl.
9665199	380204	21728	10831	9198880	15680	56981	9114686	5421	23524	354117	305782	6094
66958756	333675	3239	4919	611763	9241	18585	576237	4287	12361	66319037	306704	4683
754868	329683	3503	4654	395281	10027	18341	359733	3936	11782	332641	304938	4146
2237229	329332	3130	4509	1877817	8651	16860	1845141	3972	11628	333429	304928	3913
713377	329686	2927	4495	354890	8733	16990	322078	3886	11726	332623	305094	3919
21411080	331005	3112	4540	21050643	8918	24812	21009657	3982	11918	334216	306399	3964
827576	329159	3141	4522	468444	8894	19888	432566	3891	11683	333013	304845	3992
4637371	328963	3084	4568	4279182	8741	16467	4246896	3904	11396	332473	304713	3861
943045	329861	18822	5020	567792	8999	16543	535141	3920	11445	333433	305114	3766
4496692	330246	2777	4889	4137310	9126	17165	4103492	4239	12182	332881	305003	4104
1057598	335498	3287	4515	685448	8862	19661	649696	3973	11636	345622	310836	4071
6982634	329180	3019	4501	6623787	6712	17715	6592225	3961	11336	333137	305171	3937
1044422	329231	3170	4476	686732	8702	17450	653511	3894	11480	331828	305044	3781
23739048	332371	2991	4501	23377355	8861	17678	23343242	4295	13008	334583	306420	3968
783804	333318	3110	4505	421037	8688	17755	387435	3941	11466	336924	309307	3915
4669451	330522	3075	4502	4310034	8744	25228	4268439	4197	12461	332902	304971	3886
902159	334131	17119	4994	524441	9255	17418	490524	4039	11522	337481	309192	3871
4552998	330438	2775	4864	4192595	9025	17574	4158355	4339	12223	334043	305094	3874
994610	329075	3048	4475	636783	8862	17167	603624	3950	11542	332018	304845	3848
6817078	333347	3107	4477	6454363	8706	17520	6421003	3955	11421	337012	309336	3864
712410	330309	3090	4475	353542	8681	17383	319969	4235	12537	332328	305138	3842
4243699	330678	3035	4484	3884554	8813	17577	3850996	3986	11732	333297	306347	3785
793434	333692	3089	4478	429858	8718	17029	397019	3913	11450	337477	305075	4030
4673949	329718	3137	4469	4315127	8737	17243	4282048	3922	11516	332976	304980	3886
705095	334172	15013	5018	329767	9025	16904	296699	3956	11610	336893	309136	4029
39916462	331863	2854	4825	22029317	9005	24613	21987957	4382	12400	17859920	305713	4108
729858	329312	3015	4473	370978	10105	17915	335769	3923	11593	332726	304812	4097
11744585	333574	3101	4472	11381703	8915	17349	11348205	4038	11596	336756	309178	4120
829051	329442	3226	4478	469748	9145	17404	436071	3951	11505	333263	304942	4037
4638273	334956	3063	4484	4275256	8983	17330	4241757	4004	11420	337312	310597	3831
937992	329262	3020	4476	580052	8885	17447	546596	3940	11469	332091	304909	3982
4529780	333533	2999	4516	4167825	8694	17230	4134636	3961	15396	332010	304949	3871
1078789	330219	16950	4982	705741	8909	17746	671919	3966	11492	332901	304973	3917
6639517	330418	2845	4831	6280075	9012	17841	6245440	4496	12525	332830	304824	3953
1184881	333914	3058	4475	820979	8802	17573	787454	3953	11664	337664	309240	4029
4285212	329530	2953	4475	3926746	8855	27109	3883516	3983	11569	332637	305069	3931
1279721	329341	2913	4475	921007	9053	17307	887561	3887	11534	332883	305022	3959
24289001	330998	3024	4475	23929365	8970	17596	23895657	3963	11772	333537	306272	3995
1384797	329455	2933	4465	1025745	8968	17284	992414	3901	11461	333146	304943	4001
24178613	329503	2994	4463	23819260	8725	17174	23786232	3955	11659	333285	304781	4132
1495954	329963	15515	4963	1124591	8945	17018	1091479	3935	11589	332663	305010	3978
29555425	331457	2849	4822	26084724	8969	17033	26051195	4253	12246	3443909	305619	4085
705526	329525	3057	4466	346710	9615	17062	312897	3917	11570	332806	305063	3864
20732925	329238	2989	4452	20374529	8899	16635	20341804	4015	11711	332293	304739	4051
812412	329918	2920	4461	453024	9169	16504	420063	4118	11624	333388	305308	4006
4776900	330792	2986	4501	4416556	8921	24663	4375771	3928	11537	334581	306357	3965
915218	329515	3000	4460	556645	9049	16056	524453	3896	11480	332801	305083	3864
4406571	329092	3077	4460	4048085	8734	16850	4015244	3991	11517	332411	304729	4025
1030308	330458	16145	4999	657398	8947	16825	624464	3955	11809	333053	305076	4049
6686668	334502	2859	4864	6324529	9092	17221	6290700	4238	12161	335983	309097	3743

B.4. Performance Data for File-Open Operations (Warm Caches)

Total	Tot. CPU time FS	IPC to FS	Enc path	Open total	IPC to BS	open()	read()	IPC to FS	Crpyt meta	ME total	ME crypt	IPC to cl.
57989	21207	3002	5306	31977	9464	9251	6669	3488	10379	1355	0	3110
51292	18861	2581	4373	27995	8742	7453	5480	3460	10041	916	0	2941
50511	18403	2548	4358	27467	8623	7380	5200	3424	10168	774	0	3150
50652	18550	2554	4330	27557	8652	7364	5290	3426	10199	909	0	3033
50567	18485	2426	4341	27685	8580	7450	5342	3430	10153	804	0	3024
50514	18577	2495	4368	27457	8552	7357	5297	3426	10143	853	0	3038
62019	18980	12554	4872	28658	8881	7944	5443	3510	10278	739	0	3082
51874	19118	2661	4727	28267	8929	7702	5275	3473	10096	1143	0	2999
50378	18160	2488	4341	27717	8650	7331	5500	3417	10002	737	0	3061
51104	18867	2505	4330	27685	8582	7402	5359	3467	10099	896	0	3106
54503	18552	2549	4348	31393	8586	11243	5250	3469	10065	745	0	3086
50467	18580	2508	4325	27448	8596	7305	5296	3426	10036	924	0	3013
50571	18486	2510	4337	27653	8607	7368	5388	3426	10036	916	0	3027
50648	18452	2521	4325	27728	8672	7410	5314	3417	10065	836	0	3037
61335	19184	11911	4928	28380	8833	7740	5429	3516	10146	752	0	3081
51794	19119	2616	4715	28201	8940	7451	5415	3517	10164	1007	0	2985
50575	18385	2509	4335	27794	8659	7450	5419	3459	10077	836	0	3019
50537	18358	2512	4327	27725	8627	7178	5509	3548	9981	794	0	3028
50276	18298	2451	4337	27568	8624	7323	5349	3463	10151	710	0	3041
50516	18488	2508	4327	27591	8577	7345	5376	3478	10076	994	0	3006
50492	18263	2508	4337	27788	8656	7477	5276	3526	10124	708	0	3014
50610	18509	2507	4326	27645	8591	7464	5325	3459	10116	887	0	3035
60704	18855	11780	4836	28199	8766	7643	5432	3501	10194	665	0	3091
51695	19307	2616	4715	27940	8951	7270	5324	3517	10117	1122	0	2989
50033	18262	2410	4349	27416	8589	7282	5271	3459	10065	741	0	3023
50859	18633	2500	4326	27743	8618	7287	5400	3565	10038	792	0	3081
50492	18328	2543	4349	27624	8644	7518	5184	3469	9981	736	0	3074
54204	18453	2502	4325	31346	8567	11144	5335	3478	9996	930	0	2978
50656	18519	2510	4345	27666	8598	7366	5281	3567	10022	922	0	3050
50680	18352	2523	4325	27886	8579	7416	5468	3505	9964	827	0	3004
60464	18979	11787	4837	27828	8768	7300	5375	3528	10060	725	0	3091
51644	19000	2560	4714	28149	8902	7529	5323	3517	10073	808	0	3075
50460	18579	2446	4347	27429	8574	7206	5383	3459	10007	889	0	3085
50790	18615	2450	4365	27620	8605	7212	5397	3542	10109	805	0	3199
50608	18418	2443	4356	27754	8597	7484	5403	3460	10170	701	0	3070
50500	18651	2373	4342	27353	8569	7012	5514	3444	10171	942	0	3205
50735	18473	2457	4322	27840	8558	7407	5531	3464	10164	838	0	3053
50482	18659	2472	4311	27348	8643	7092	5372	3435	10148	1005	0	3082
60654	18965	11786	4849	28047	8822	7479	5458	3409	10239	678	0	3081
51604	19052	2564	4688	28189	9072	7194	5612	3432	10295	873	0	3061
50048	18250	2436	4322	27609	8685	7245	5409	3465	10103	782	0	2958
50384	18868	2442	4300	27147	8726	6854	5256	3449	10203	848	0	3146
50242	18534	2572	4405	27386	8634	7128	5346	3469	10116	762	0	3081
50509	18554	2414	4287	27513	8513	7105	5636	3444	9995	852	0	3107
50282	18678	2427	4382	27220	8676	7006	5195	3485	10082	886	0	3061
50593	18603	2413	4359	27701	8752	7160	5424	3446	10206	826	0	3107
60248	18978	11857	4848	27559	8772	7200	5308	3409	10116	800	0	3086
50868	18827	2621	4714	27668	8992	7183	5181	3423	10217	810	0	2920
49728	18432	2497	4335	26830	8553	6882	5124	3465	9996	801	0	3053
49899	18585	2502	4326	26954	8581	6895	5171	3443	9990	978	0	3071

Bibliography

[1] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry M. Levy. A Crawler-Based Study of Spyware on the Web. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS 2006)*, February 2006. 1

[2] U.S. CERT United States Computer Emergency Readiness Team. Cyber Security Bulletins. Available from:
`http://www.us-cert.gov/cas/bulletins/`. 1

[3] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001. 1

[4] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza Secure-System Architecture. In *Proceedings of CollaborateCom*, 2005. Available from:
`http://os.inf.tu-dresden.de/papers_ps/nizza.pdf`. 2, 14

[5] The Fiasco Microkernel. Located at:
`http://os.inf.tu-dresden.de/fiasco/`. 5

[6] L4Linux. Located at:
`http://os.inf.tu-dresden.de/L4/LinuxOnL4/`, `http://l4linux.org/`. 5

[7] Trusted Computing Group: TPM. Located at:
`https://www.trustedcomputinggroup.org/groups/tpm/`. 7

[8] Trusted Computing Group: Home. Located at:
`https://www.trustedcomputinggroup.org/home/`. 7, 15

[9] Bernhard Kauer. Authenticated Booting for L4, November 2004. Available from:
`http://os.inf.tu-dresden.de/papers_ps/kauer-beleg.pdf`. 8, 9

[10] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis 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, December 2004. 9, 14

[11] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the 10th Network and Distributed Systems Security (NDSS) Symposium*, pages 131–145, February 2003. 9, 14

[12] Hans Marcus Krüger. Zufallszahlen unter L4/DROPS, May 2005. Available from:
`http://os.inf.tu-dresden.de/papers_ps/krueger-beleg.pdf`. 11

[13] The Linux Kernel Archives. Located at:
http://www.kernel.org/. 13, 22

[14] Jetico Website. Located at:
http://www.jetico.com/. 13

[15] Matt Blaze. A Cryptographic File System for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993. 13

[16] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In *Proceedings of USENIX Technical Conference, FREENIX Track*, June 2001. 13

[17] 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. 13

[18] Valient Gough. EncFS Encrypted Filesystem. Located at:
http://arg0.net/wiki/encfs. 13, 22

[19] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Enhancing File System Integrity Through Checksums. Technical report, Computer Science Department, Stony Brook University, May 2004. Available from:
http://www.fsl.cs.sunysb.edu/docs/nc-checksum-tr/nc-checksum.pdf. 13

[20] Microsoft Corporation. Secure Startup – Full Volume Encryption: Technical Overview. Available from:
http://www.microsoft.com/whdc/system/platform/pcdesign/ secure-start_-tech.mspx. 13

[21] C. Stein, J. Howard, and M. Seltzer. Unifying file system protection. In *Proceedings of the USENIX Technical Conference*, pages 79–90, 2001. 14

[22] Umesh Maheshwari, Radek Vingralek, and Bill 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, October 2000. 15, 44

[23] R. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–134, 1980. 15, 24

[24] Trusted Computing Group: Storage Work Group. Located at:
https://www.trustedcomputinggroup.org/group/storage. 15

[25] Christopher Batten, Kenneth Barr, Arvind Saraf, and Stanley Trepetin. pStore: A Secure Peer-to-Peer Backup System. Technical Memo MIT-LCS-TM-632, Massachusetts Institute of Technology Laboratory for Computer Science, October 2002. 15

[26] Emin Martinian. Distributed Internet Backup System (DIBS). Located at:
http://www.csua.berkeley.edu/~emin/source_code/dibs/. 16

[27] Wikipedia, the Free Encyclopedia. Reed–Solomon Error Correction. Available from:
http://en.wikipedia.org/wiki/Reed-Solomon_error_correction. 16

[28] Trusted Computing Group. TCG Interoperability Specification for Backup and Migration Services. Available from:
https://www.trustedcomputinggroup.org/groups/infrastructure/. 16

[29] DROPS The Dresden Real-Time Operating System Project. Located at:
http://os.inf.tu-dresden.de/drops/download.html. 16

[30] Federal Information Processing Standards Publication 197: Announcing the Advanced Encryption Standard. Available from:
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. 17

[31] Markku-Juhani Olavi Saarinen. Encrypted watermarks and linux laptop security. In Chae Hoon Lim and Moti Yung, editors, *WISA*, volume 3325 of *Lecture Notes in Computer Science*, pages 27–38. Springer, 2004. 17

[32] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation on Key Management – Draft, Special Publication 800-57, August 2005. Available from:
http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf. 18

[33] Shafi Goldwasser, Silvio Micali, and Ron L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988. 18

[34] Federal Information Processing Standards Publication 180-1: Secure Hash Standard. Available from:
http://www.itl.nist.gov/fipspubs/fip180-1.htm. 18

[35] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Collision Search Attacks on SHA1, February 2005. Available from:
http://theory.csail.mit.edu/~yiqun/shanote.pdf. 18

[36] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Message Authentication Using Hash Functions: the HMAC Construction. *CryptoBytes*, 2(1):12–15, 1996. 19

[37] OpenSSL Website. Located at:
http://www.openssl.org/. 35

[38] SSL 3.0 Specification. Available from:
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. 35

[39] rsync Website. Located at:
http://www.samba.org/rsync/. 36

[40] D. R. Morrison. PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968. 37

[41] G. M. Adelson-Velskii and E. M. Landis. An Algorithm for the Organization of Information. In *Soviet Mathematics Doklady*, pages 1259–1262, 1962. 37

[42] The Single UNIX Specification. Available from:
http://www.unix.org/what_is_unix/single_unix_specification.html. 39

[43] Bernhard Kauer and Marcus Völp. L4.Sec Preliminary Microkernel Reference Manual. Technical report, 2005. Available from: http://os.inf.tu-dresden.de/L4/L4.Sec/l4_sec_20051019.pdf. 46

[44] Bernhard Kauer. L4.sec Implementation - Kernel Memory Managment. Master's thesis, TU Dresden, May 2005. 46

[45] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 experimental kernel reference manual, version x.2. Technical report, 2004. Latest version available from: http://l4hq.org/docs/manuals/. 46

[46] Christian Helmuth. Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur. Master's thesis, TU Dresden, 2001. Available from: http://os.inf.tu-dresden.de/project/finished/finished.xml.de#helmuth-diplom. 51

[47] Frank Mehnert. *Kapselung von Standard-Betriebssytemen.* PhD thesis, TU Dresden, July 2005. 51

[48] Christian Böhme. PCI-to-PCI-Bridge mit sicherheitsrelevanten Eigenschaften, September 2005. Available from: http://os.inf.tu-dresden.de/papers_ps/boehme-beleg.pdf. 51

[49] Advanced Micro Devices, Inc. *AMD Secure Virtual Machine Architecture Reference Manual*, May 2005. 51

[50] Russell Coker. Bonnie++. Located at: http://www.coker.com.au/bonnie++/. 53

[51] Namesys Website. Located at: http://www.namesys.com/. 53

[52] Thomas J. McCabe. A complexity measure. *In IEEE Transactions on Software Engineering*, SE2(4):308–320, December 1976. 61

[53] pmccabe – McCabe-style function complexity and line counting for C and C++. Located at: http://www.parisc-linux.org/~bame/pmccabe/overview.html. 61