Reducing Size and Complexity of the Security-Critical Code Base of File Systems

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Carsten Weinhold

geboren am 30. September 1980 in Dresden

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig Technische Universität Dresden
Gutachter:	Prof. Dr. Scott Brandt University of California, Santa Cruz
Fachreferent:	Prof. Dr. Christof Fetzer Technische Universität Dresden

Korrigierte Version vom 2. Juli 2014

(eingereicht am 5. Juli 2013, verteidigt am 14. Januar 2014)

Acknowledgements

I want to thank my advisor Prof. Dr. rer. nat. Hermann Härtig for all the support and guidance over the years and for encouraging me in my work. I warmly thank Björn Döbel, Michael Roitzsch, Anna Krasnova, and Dr. Claude-Joachim Hamann for all the time and effort they put into proof reading draft versions of this thesis and for giving me so much valuable feedback for improving it. Thanks also go to Adam Lackorzynski and Alexander Warg for maintaining and continuously improving the platform of my work and the work of so many others. Everybody in the coffee area: thank you for all the interesting discussions (not only about work). Finally, my greatest thanks go to my family and to my friends for all their help, support, and patience, but especially for being so much fun to be with.

Thank you all!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Handheld devices such as smartphones and tablets have evolved into powerful and versatile mobile computers. They are often used for media consumption and entertainment, but professional users start to use them for highly critical data such as classified documents or trade secrets. More use cases are emerging; for example, a doctor making house calls may use such a device to store patient records, which are not only sensitive from the patient's point of view, but also subject to legal requirements. The data is stored in a file system and the user must trust this critical subsystem of the operating system (OS) to keep his files safe from corruption and secure with respect to various protection goals.

1.1 Motivation

Mobile and Lost Devices. Mobile devices are constantly at risk of getting lost or stolen. Unfortunately, those portables containing critical data are no exception, as media reports [65, 67, 72] on high-profile cases show. Confidential information on a lost device is highly valuable not only for the legitimate owner, but might also be of interest to the person who finds it. The problem of preventing unauthorized access to data at rest has been solved by mature and well-studied encryption algorithms such as AES [34]. If an attacker has access to an encrypted storage device but not the encryption key, the computational complexity of breaking a state-of-the-art encryption algorithm is considered to be too high even for exceptionally strong attackers. But in reality, the security of the user's data does not only depend on the strength of an encryption algorithm. Instead, the software through which the data can be accessed is most often the weakest link.

Untrusted and Insecure Software. Attempts to attack the software stack of mobile devices may occur at any time, for example, through malicious functionality hidden in seemingly innocuous applications. The software ecosystems for modern mobile platforms make it easy and highly convenient for the user to install software for virtually every use case. The distribution channels offer hundreds of thousands of applications, some developed with malicious intentions [73, 74, 144]. The ultra-portable computing devices on which these applications then run are connected to the Internet most of the time, often through untrusted networks (e.g., public WiFi). Permanent connectivity is convenient for the user, but also for attackers who seek to capitalize on confidential data stored on the device. The quality of the threats that mobile computing platforms face are in conflict with the level of security that current operating systems and system architectures can provide. The enormous code bases of both applications and the underlying OSes – consisting of hundreds of thousands to millions of lines of code – have been found to contain too many vulnerabilities. Attackers can, and do, exploit weaknesses in order to steal confidential user data, or to subvert other protection goals such as integrity and availability.

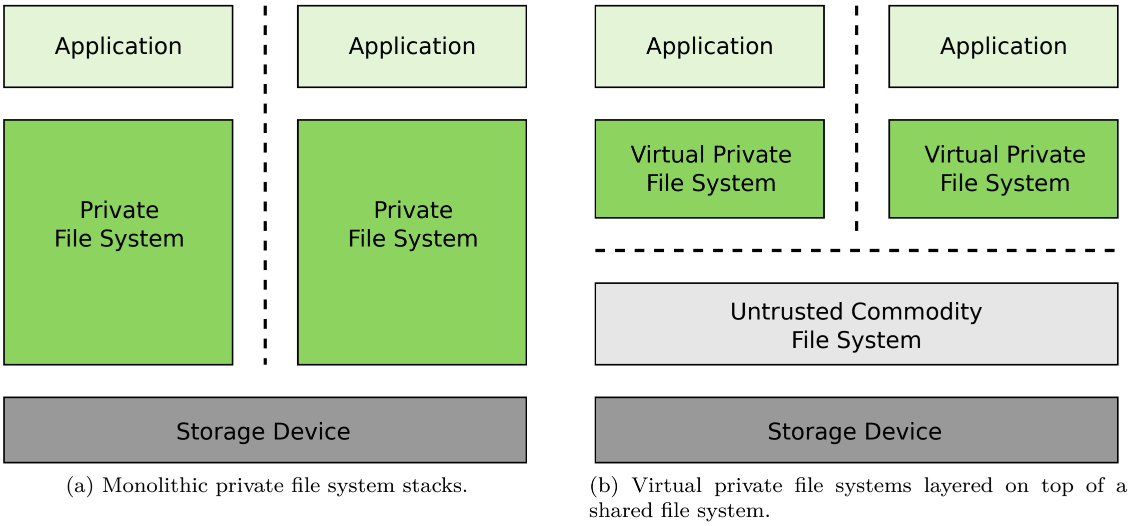
Threats and Exploits. Traditional personal computers have been plagued by malware attacks for decades. Many attack vectors that are still used to compromise these systems also work on mobile device platforms. For example, users run similarly powerful applications on them, including web browsers and document viewers with highly complex rendering engines. They even run full-featured office suites, many of which support file formats with a long history of being used to compromise desktop-class PCs. Many vulnerabilities in mobile applications [9, 25, 66] have been shown to be exploitable and some were in fact widely used as an attack vector to further compromise the security features of the underlying platform.

Probably the best examples for successful attacks on the security of mobile OSes are “jailbreaking” and “rooting” attacks. They give users the freedom to install software without any restrictions, but they necessarily break the protection mechanisms of the mobile OS kernel in order to do so [16, 19, 20, 22, 23, 24]. Like their desktop counterparts, popular smartphones and tablets run monolithic kernels that are highly complex. All interfaces and subsystems of the kernel reside in the same kernel address space, even though many of them are independent from each other. This architecture, as well as the use of unsafe languages such as C, are commonly justified by claims of higher performance. But this potential gain comes at a price: functionally independent subsystems and their data structures are not isolated from each other and faults in – or attacks on – any one component can compromise every other part of the monolithic kernel. As a result, any application running on top of it is compromised, too. An attacker who can subvert a kernel subsystem may therefore also be able to access any user data, regardless of cryptographic protection that was meant to defend against offline attacks on data at rest.

Strong Attackers and Defenseless Targets. Unfortunately, a jailbreaking user is not the only one who could exploit the security holes in order to break into a mobile OS. Security researchers [75] and public media [68, 69] report that highly sophisticated attacks on government agencies, NGOs, and industry are on the rise. Dedicated attackers try to infect their victims’ computing devices with malicious software in order to steal information. Anti-malware software or virus scanners that try to detect malicious code or suspicious behavior at runtime have proven largely ineffective against targeted attacks. In fact, highly sophisticated malware [90, 128] that has been identified in 2010 and 2011 infected only a small number of computers and thus remained undetected for several years.

Benefits of Security Architectures. Given the high probability of security-critical bugs in the large code bases of complex monolithic software, a more secure system architecture is needed. Component-based system architectures based on microkernels can significantly improve security by isolating individual subsystems of the OS in separate address spaces. The result of this decomposition are smaller components that are separated from each other, so as to provide better fault containment and to enforce the principle of least privilege [140]. This separation enables smaller trusted computing bases (TCBs) [134] for individual applications, as their security no longer depends on the entire OS, but only the specific OS components that affect operations on critical data. These application-specific TCBs can be several orders of magnitude smaller than in monolithic system architectures [109, 147]. The size of a code base correlates with the the number of defects and even well-tested software shows fault rates in the area of 6-8 errors per thousand lines of code [98, 111]. Therefore, a drastically smaller TCB can be assumed to be significantly harder to compromise.

The Case for a Secure File System Architecture. However, even though microkernel-based architectures decompose the OS in order to isolate subsystems, some components are still shared by all applications. The file system component is one of the most critical of these shared services, as it provides access to all non-volatile user data. At the same time, implementations of general purpose file systems are complex: their code bases comprise in the order of tens of thousands of lines of code. This complexity results from the requirement to operate efficiently under diverse, and often also concurrent workloads. Furthermore, file system stacks dedicate a significant amount



(a) Monolithic private file system stacks.

(b) Virtual private file systems layered on top of a shared file system.

Figure 1.1: Isolation of private file system instances for mutually distrustful applications: monolithic file system stacks (left figure) honor mutual distrust of applications in the storage architecture; split file system stacks (right figure) further reduce the size of the trusted computing base (TCB) for each application. (Block device drivers have been omitted for simplicity.)

of code and complexity to the handling of error situations, such as failing storage devices or for recovery of inconsistent on-disk state after a system crash.

Given the inherent complexity of file system stacks, it is not surprising that vulnerability databases [5, 7, 12, 17, 18, 61] and academic literature [106, 159, 160] document a large number of defects even in production quality file systems. Certain weaknesses have already been used to compromise kernels of widely used mobile platforms [19]. Two major attack vectors on file systems exist. First, the complex implementations expose themselves to potentially untrusted applications through a broad API [6, 8, 10, 11, 13, 14, 15]. Second, file system code needs to correctly interpret and process the bit patterns of possibly inconsistent on-disk data structures on the block storage device. While file system code can treat file contents as opaque data without further processing, the metadata structures encode information that is processed and used as input by large parts of the file system logic. The integrity of metadata is therefore critical for correct operation and parts of the metadata directly affect security mechanisms. For example, permission checks at the file and directory level rely on the correctness of access control information stored in inodes. The integrity of file system metadata is therefore essential for isolation of principals or other security policies.

1.2 A Virtual Private File System

Isolating File System Stacks. Mobile devices, which typically are single-user devices, treat each application as a principal for access control purposes. These applications do not need access to each other's files, yet some of the programs may turn out to be malicious. Or an application might have implementation bugs that an attacker can exploit to gain control over it. Consequently, applications that distrust each other should be provided *private storage* for their files so as to protect them from other applications. Due to the high complexity and potential vulnerability of large file system code bases, the security properties demanded by applications should be reflected in the architecture of the storage stack. That is, *mutual distrust* [143] among applications should be honored by isolating the respective instances of private file storages (see Figure 1.1a). If a private instances of the file system stack runs isolated in its own address space, an attacker who managed to compromise this component cannot easily gain access to files managed by other private instances. In addition to honoring mutual distrust, isolation of file system stacks is also a step towards instantiating the *principle of least privilege*, because unrelated instances are not part of the TCB.

Re-Architecting the File System Stack. Unfortunately, an application must still trust the tens of thousands of lines of code implementing the private file system instance with respect to its security goals, be it confidentiality, integrity, or availability. Being forced to trust so much code is undesirable, because isolating per-application file system stacks does not address the second attack vector described above: an attacker may still try to corrupt – or maliciously craft – on-disk metadata structures in order to exploit implementation defects in the file system code base.

A way out of this situation is to re-architect the file system stack such that major parts of its inherently complex code base do not need to be trusted. Figure 1.1b illustrates such a split architecture. Splitting the file system stack into differently trusted parts is feasible, because certain functionality does not require the full set of privileges. For example, file systems treat user data as opaque byte arrays, which can therefore be cryptographically protected. The key challenges of the splitting approach are (1) to find a sensible boundary between what must be fully trusted and what is not critical for enforcement of a specific security goal, and (2) to enable secure and efficient cooperation between the parts.

The architecture I strive for features a small security critical core that is trusted and part of an application's TCB. The larger and potentially more complex part of the file system runs with lower privileges: it is untrusted with respect to protection goals such as confidentiality and integrity. This splitting approach and its trust model are similar to the concept of a virtual private network (VPN), which layers a secure network on top of an untrusted network such as the Internet. Hence, I name the file system that is the subject of this thesis the *"Virtual Private File System"* or *VPFS*. Its trusted part is called *VPFS Core*.

1.3 Thesis Goals

The main objective for building VPFS is to improve file system security by making the code base of this OS service less vulnerable to attacks. This design goal aims at stronger enforcement of security guarantees than monolithic, shared file system stacks can offer. Nevertheless, the resulting system must also meet the efficiency and functional requirements that applications demand. In the following, I derive secondary design and protection goals that follow from the main goal of achieving lower vulnerability and from what the user expects in terms of security.

Confidentiality: The first protection goal is to ensure the confidentiality of the user's data in case the mobile device gets lost. An extreme approach to this goal could be to not use local storage at all, but instead store and process all data on a server the user considers secure. Access control through the mobile device would be enforced based on device-independent authentication factors (e.g., the user must enter a password). However, cellular networks still have insufficient coverage in areas where users wish to take their devices. Interactive access to a remote server is impossible in such locations and even if there is network coverage, it may be too expensive (e.g., when using a network in a foreign country). Therefore, it is essential that mobile devices provide local storage for at least parts of the user's data. VPFS must store this data in encrypted form to provide protection in case the device is lost or stolen.

Reduced Attack Surface: The high probability of security-related defects in commodity OSes and their file system stacks is well documented [95, 61, 106, 125, 159, 160]. The enormous size of their complex monolithic code bases – more than 10 million lines of code in the case of Linux 3.5 – is the main reason for their vulnerability, which puts the security of the user's critical data at a real and considerable risk. Microkernel-based OSes reduce the size of the attack surface significantly by isolating service components in address spaces [109, 147].

VPFS targets such a system architecture and aims to reduce the attack surface even further by minimizing the size of the file system TCB. To this end, VPFS shall isolate per-application instances of the file system, which instantiates the principles of *mutual distrust* and *least privilege* right in the architecture of the storage stack. VPFS shall further reduce the size and complexity of the TCB for individual instances of the file system stack. Ideally, the TCB should only contain functionality that is required to meet security goals.

Reuse of Untrusted Infrastructure: The goal of minimizing the file system TCB seems to be at odds with the requirement that VPFS must still be a sufficiently capable and efficient general-purpose file system. This apparent conflict can be resolved by isolating the security-critical functionality of the file system stack in a separate component, which then cooperates with those parts that implement non-critical functionality. The concept of a "trusted wrapper" [109] can be used to realize such a split architecture. For VPFS, I apply this approach to a file system stack by isolating the security-critical functionality in a layer added between the application and an existing untrusted file system implementation. Secure reuse of the untrusted file system is facilitated using cryptography and additional checks, while major parts of the inherently complex file system code base are not part of the TCB. An additional benefit of the trusted wrapper approach is that the composed system inherits properties and features of the reused infrastructure (e.g., robustness against certain failure conditions or optimizations made in an off-the-shelf commodity file system).

Integrity and Robustness: To ensure confidentiality, the security-critical core of VPFS must encrypt all data before it is stored in the untrusted file system. However, availability and integrity of the encrypted data cannot be guaranteed, if an attacker is able to compromise the untrusted file system. Apart from offline attacks on the untrusted storage medium, it must be assumed that the complex implementation of the reused infrastructure can be successfully attacked at run time. That is, deletion or unauthorized modification cannot be prevented under the assumption that parts of the storage stack are not trustworthy. However, a weaker definition of the term *integrity* – the ability to detect any unauthorized corruption after it occurred – can be enforced through cryptographic measures. For example, message authentication codes (MACs) enable efficient verification of data integrity. However, MACs are security-critical metadata that must be stored persistently, too, and the system must make sure that a MAC is always in sync with the piece of encrypted data that it protects; if it is not, it is impossible to verify the data as correct. Ensuring consistency of metadata and data after a crash or unexpected power loss is a non-trivial problem that VPFS must address. It requires secure cooperation between trusted and untrusted components, even if the reused file system stack already provides robustness against crashes.

Recoverability: I assume that the untrusted components of the storage stack cooperate as long as they are not being attacked. However, as the VPFS stack as a whole depends on them in order to make progress, the *availability* of data at the time of request cannot be guaranteed. This limitation is fundamental, because the untrusted components may malfunction or seize to cooperate. The user may also lose access to his data as a result of hardware failures or when he loses the mobile device. However, the weaker goal of *recoverability* can be achieved. It is possible to recover data, if an up-to-date and consistent backup is available (e.g., on a remote server). VPFS shall enable backup and restore in a secure and timely manner under the assumption that untrusted components of the VPFS stack misbehave temporarily.

1.4 Non-Goals

VPFS does not target protection goals other than confidentiality, integrity, and recoverability. The following two paragraphs summarize why certain security and design goals are out of scope of this thesis.

Limitations Imposed by Untrusted Infrastructure. First, VPFS does not aim to hide the presence of encrypted information like steganography-based solutions do. It is also not a goal of this work to allow the user to plausibly deny that he is in possession of specific information. For example, by presenting different file system content depending on which passphrase the user entered, like it is possible with TrueCrypt [53]. Second, as the trusted and untrusted components of the file system stack need to cooperate, a communication channel must exist between these two parts of VPFS. A misbehaving application could abuse this communication facility as a *covert channel* in order to leak information, despite transparent encryption. However, such an application could only harm itself, but not compromise the confidentiality of data handled by other VPFS instances it does not have access to.

No Redundant Mechanisms. Because mutually distrustful applications are assigned private file system instances, VPFS does not need to provide a file-level access control mechanism such as UNIX-style ACLs (with permission bits for 'owner', 'group', and 'other') or POSIX ACLs. Instead, the VPFS architecture relies on access control mechanisms of the underlying system platform, including hardware-protected address spaces and means to restrict inter-component communication. VPFS itself does not attempt to counter hardware-based attacks, except that it encrypts all data written to the untrusted storage medium. However, it can use platform support designed to protect against certain types of hardware attacks; these are discussed in Section 2.4.

1.5 Contribution

The VPFS architecture embraces the concept of application-specific TCBs as enabled by componentized system architectures with small kernels. It aims at minimizing the size and complexity that a file system implementation contributes to the TCB of an application. This minimization of TCBs is motivated by the goal to reduce the size of the attack surface of this essential OS subsystem. VPFS strives for this goal by assigning dedicated instances of the file system stack to applications and by isolating the security-critical core of each instance from non-critical functionality.

Splitting the code base into two isolated parts – one of them untrusted – allows users and applications to trust less code, compared to file system services of other componentized systems. For example, the microkernel-based OS QNX implements file system support as a separate user-level server, which includes all the complexity of a file system implementation in a single address space. The separation of differently trusted code bases in the VPFS architecture presents new challenges:

- The security-critical functionality must be identified. Chapter 3 discusses where to introduce a sensible boundary between the code that becomes part of the TCB and what can be provided by untrusted components of the file system stack.

- The trusted and untrusted components must be able to cooperate securely and efficiently. Chapters 3 through 6 discuss how to devise inter-component interfaces that make early input validation simple, such that the directly exposed attack surface becomes smaller.

- The reuse of existing storage infrastructure should be maximized in order to benefit from features and properties of a commodity file system implementation without having to trust it. Chapters 4 and 5 explore possibilities and limitations when reusing untrusted infrastructure for *naming and lookup* and *file system consistency*, respectively.

- Apart from functionality and efficiency considerations, the answer to the question "What belongs to a TCB?" depends on the specific security goals that should be achieved. Chapter 6, which covers *recoverability*, makes that case that several different, not necessarily overlapping, TCBs exist for different subproblems of backup and restore.

The prototype implementation, which is evaluated in Chapter 7, exploits the unique properties of the VPFS architecture and mobile device use cases. In particular, the implementation of the trusted core of a VPFS instance can be expected to have only a small number of clients (often just one application). Therefore, VPFS can favor simplicity of the implementation and reusability of generic code paths over maximizing scalability. Several design decisions following this principle are discussed in Chapter 3; they represent significant steps towards the goal of minimizing size and complexity of the file system TCB.

The split VPFS architecture reduces the TCB size by at least an order of magnitude compared to the size of the monolithic Linux file system stack, which in typical configurations contains in the order of 100,000 lines of code. A similar factor can also be achieved for specific subsystems of a file system code base. For example, the amount of code that must be trusted to provide robustness against crashes contributes less than 400 lines of code to VPFS [157], whereas a commodity file system such as as Ext4 dedicates approximately 4,000 lines of code to journaling and replay.

Chapter 2

Background and State of the Art

VPFS builds upon existing technologies from three areas of computer security:

1. Isolation architectures and secure reuse of untrusted infrastructure
2. Trusted computing techniques to assure software integrity
3. Protection of cryptographic secrets using hardware and software techniques

I first introduce microkernel-based architectures and summarize research results on how they integrate with existing commodity infrastructure. In particular, I discuss how monolithic commodity operating system (OS) kernels can be reused as part of a componentized system architecture to achieve broad compatibility and to offer the rich feature set that applications expect. This overview is followed by a brief description of trusted computing techniques and the state of the art in integrating them into system platforms. Based on these building blocks, I introduce the threat model and discuss realistic assumptions about platform security features that enable VPFS to deliver on its strong security guarantees. I then contrast the VPFS approach and its threat model with the diverse related work in the area of file system security.

2.1 Isolation Architectures and Legacy Reuse

In the introduction of this thesis, I discussed the reality of security vulnerabilities that are frequently found in the millions of lines of code that make up the complex monolithic OSes. These types of OSes run on the vast majority of end-user computing platforms, including mobile devices, but alternative system architectures promise better security properties through smaller code bases and strong isolation.

Microkernel-based Systems. Microkernels support the construction of highly componentized multi-server OSes. These systems move major parts of the complex functionality that a monolithic kernel provides into separate components: device drivers, storage and network stacks, and even memory management run as deprivileged user-level servers. Because the individual subsystems are isolated in their own address spaces, the multi-server system as a whole can provide better fault containment than a monolithic kernel. For example, a buggy networking stack cannot corrupt file system state that is cached in memory, because address-space separation limits the harmful effects of stray memory writes to the faulty component itself. Apart from enabling better fault isolation, address spaces also limit malicious code in its ability to compromise subsystems in other address spaces.

Nevertheless, the individual components must provide their functionality through interfaces to other components and applications. An attacker who controls one component could try to use the exposed APIs to his advantage. For example, a compromised networking stack may be able

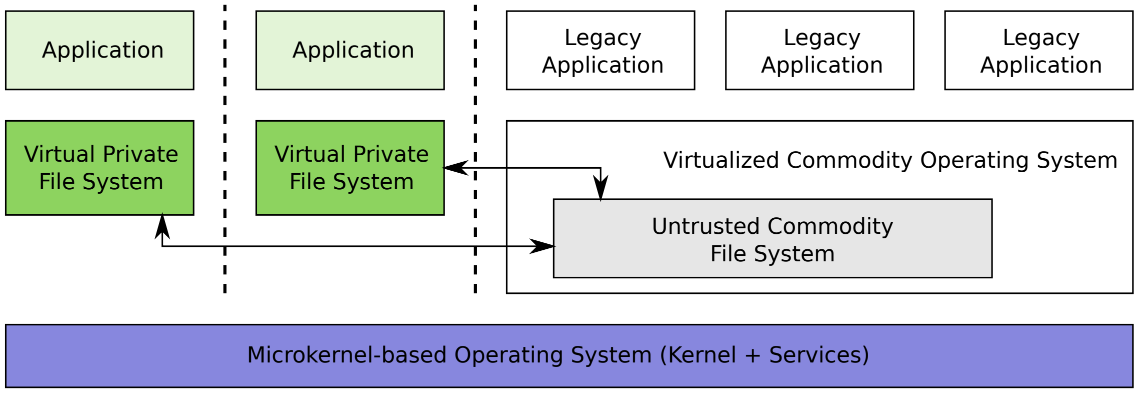

Application
Application
Legacy Application
Legacy Application
Legacy Application
Virtual Private File System
Virtual Private File System
Virtualized Commodity Operating System
Untrusted Commodity File System
Microkernel-based Operating System (Kernel + Services)

Figure 2.1: VPFS in the Nizza Secure System Architecture: VPFS implements a trusted wrapper to reuse an existing file system implementation, which is provided by a virtualized commodity operating system.

gain access to user data through the file system service's interface, unless access to it is restricted. Isolation kernels provide communication control mechanisms in order to prevent unwanted inter-process communication (IPC) between those components that do not need to cooperate. For example, EROS [146] and modern L4 microkernels [123] use kernel-protected capabilities to enforce fine-grained, mandatory access control on kernel and user-space objects, including IPC facilities. Formal verification [122] of the correctness of SeL4 microkernel relative to its specification suggest that highly-secure systems can be built on top of these kernels. The Nizza Secure System Architecture [109] builds on L4 and uses kernel-enforced access control mechanisms and address space protection to enable strong isolation of components, thereby instantiating the principle of least privilege. Applications benefit from this architecture, as they depend only on those isolated services whose functionality they require. Thus, componentization facilitates the construction of application-specific TCBs that do not include unrelated functionality. For example, if an application does not require the networking stack, it is not part of this application's TCB.

Application Cores. The Nizza architecture further improves on this reduction of TCB sizes by moving critical parts of applications (e.g., those dealing with cryptographic keys) into their own address spaces. An evaluation of three use cases [147] showed that splitting applications into a critical core and a larger non-critical part running on top of a microkernel-based OS can reduce the size of application-specific TCBs by several orders of magnitude when compared to monolithic architectures.

The isolation of critical application cores is a key technique for enabling small TCBs without losing functionality: the non-critical functionality is not removed nor is its feature set reduced, but it is merely isolated from code and data that must be trusted to meet security guarantees. The Nizza architecture also supports executing the non-critical parts of applications on a virtualized commodity OS, which is hosted on the microkernel platform. Virtualization greatly reduces development effort, as existing applications run in their legacy environment (e.g., on Linux [110]) and only their critical parts need to be ported to the microkernel platform.

Decomposing System Services. A virtualized commodity OS is of great value not only for porting existing applications, but also for providing non-critical OS functionality. The network stack is an example of a complex OS subsystem that often does not need to be trusted. Applications that need to communicate securely over an untrusted network use protocols such as Transport Layer Security (TLS) [64], which ensures confidentiality and integrity when transmitting data. TLS implementations are provisioned as a library and they provide end-to-end security. Thus, the security of the transmission does not depend on the functional correctness of the OS's network stack.

Virtual Private Networks (VPNs) enable secure networking with respect to the same set of protection goals as TLS does, but at a lower level, requiring tighter integration with the network stack. For example, IPSec support in Linux is part of the TCP/IP stack in the kernel. Thus, the same monolithic code base contains both security-critical functionality (e.g., key handling, encryption, policy enforcement) and non-critical functionality (generic packet processing, fragment handling, etc.). However, research on the applicability of the Nizza architecture principles found that the security-critical functionality of an IPSec gateway is small. It amounts to only about five percent of the Linux kernel code [113], which is more than an order of magnitude smaller than the monolithic code base it is part of. Assuming this key insight is valid for file system stacks as well, Härtig et al. proposed [109] a "trusted wrapper" for a file system that cryptographically protects confidentiality and integrity of stored data. Like a VPN reuses an untrusted network for sending and receiving packets, the file system wrapper reuses an untrusted commodity file system that provides non-critical functionality needed to store the data persistently. Figure 2.1 illustrates the resulting architecture.

2.2 Trusted Computing for Software Integrity

While isolation architectures and small TCBs drastically reduce the attack surface for software-only attacks targeting the running system, they provide no protection against offline manipulation of executable files or security-critical configuration. However, the integrity of system and application software that is part of the TCB is a necessary precondition for any security guarantees that must hold if an attacker gains physical access to a computing device. Defenses against offline tampering attacks on software must ensure that security and isolation mechanisms cannot not be removed and critical system configuration is not manipulated. This kind of integrity protection must be rooted in hardware and VPFS is assumed to run on a platform that supports it; the following paragraphs briefly describe possible implementations.

Secure Boot. The state-of-art approach to ensuring software integrity in mobile device platforms is to enforce a *secure boot* process. System-on-chip (SoC) packages as used in mobile devices typically include a hardware *root of trust*, which consists of non-modifiable boot code and a public key stored in read-only memory [80]. The boot code is the first piece of the boot software chain to be executed after power-on. It will only pass control to the next stage, the bootstrap loader of the OS, if the cryptographic signature of the bootstrap image containing the kernel is valid under the SoC's built-in public key. An unmodified OS kernel that passes this platform-enforced integrity check is trusted to provide a secure execution environment for applications. The kernel then checks the integrity of the user-space software that runs on top of it.

Authenticated Boot. Open platforms such as traditional PCs containing a Trusted Platform Module (TPM) [55] enable *authenticated booting* of any software configuration. An authenticated boot of a platform is similar to the previously outlined secure boot: a trusted piece of non-modifiable firmware is run first after powering on the device and computes a cryptographic hash of the executable code of the next stage in the boot chain. However, instead of checking this hash against a fixed signature, the initial boot code reports the hash to the TPM and then passes control to the second stage unconditionally. The following stages work analogously, first reporting the hash of the next stage's code and then starting it. The TPM records the hashes in so-called Platform Configuration Registers (PCRs) and enforces that previous values are not overwritten; it enforces that a PCR contains the aggregate hash of all individual hashes fed into it since the device has been powered on. Thus, any OS can be booted, even one that has been maliciously altered, but the startup process is logged: the PCR inside the TPM contains an unforgeable cryptographic record of all cooperating boot stages. If there are stages in the boot process that are not considered trustworthy, the first such stage will be logged in the PCR.

2.3 Trusted Computing for Secure Storage

Authenticated booting does not guarantee that the user interacts with a trustworthy software stack (i.e., one that is unmodified and known to be secure), whereas secure boot is designed to provide this guarantee. However, the TPM-augmented boot process can be used to restrict access to small pieces of critical data such as the cryptographic keys used by VPFS instances.

Sealed Memory. A mechanism called *sealing* binds TPM-protected user data to a specific software configuration: the unmodified version of the software stack will be granted access to the sealed data, but a maliciously or otherwise modified OS will be denied access and is therefore unable to compromise its confidentiality. This TPM-based access control thus limits the usefulness of tampering attacks, because the user can detect offline manipulation of his software as soon as he is no longer able to access his critical data. The hardware-based implementation of *sealing* in TPMs is based on asymmetric encryption. The private part of an asymmetric TPM storage key is required for decryption and therefore highly critical. TPM private keys exist in plaintext form only in volatile memory of the TPM, but never outside this hardware module [55]. User data encrypted under a TPM storage key is wrapped in cryptographically protected TPM metadata that includes a set of “target” PCR values. If, and only if, the current state of the PCRs recorded in the TPM match the “target” PCR configuration for the sealed data, the TPM will release the decrypted data. TPMs also provide *monotonic counters*, which can be used to ensure freshness of sealed data (i.e., data cannot be replaced by an old copy without detection). Monotonic counters are useful to detect replay attacks on untrusted storage [154].

The BitLocker [99] drive encryption feature of Microsoft Windows uses TPM-based sealed memory to bind the disk encryption key to specific versions of the Windows startup files. The TPM enforces that only an unmodified version of the Windows OS able to decrypt the system volume and user data stored on it.

Cryptographic Co-Processors. Storage encryption similar to BitLocker (i.e., with encryption keys protected in hardware) is also used in commercially available mobile devices. For example, Apple’s iOS platform features a security architecture that tightly integrates with cryptographic support built into the hardware [80]. Each device has a unique AES key that is fused into the system-on-chip (SoC) hardware during manufacturing. No software running on the device can directly access this key. Instead, it must use the cryptographic accelerator of the SoC to perform encryption or decryption operations with this key. Access control to this key is based on the secure boot process: once the vendor-approved software has booted, it can access the hardware accelerator and use the device encryption key. The underlying assumption is that no software other than an unmodified version of iOS can run on the device.

External Secrets for Multi-Factor Authentication. To improve resistance against physical attacks, the iOS security model treats the secure boot process as a first line of defense [80]. It uses additional encryption keys to protect certain data that need only be accessible to interactive applications (e.g., documents or web form passwords). Access to these keys depends on an external authentication factor: they are derived not only from the unique device key hidden in the hardware, but also from a user-provided passcode. The keys are not stored in persistent storage, but only exist temporarily in main memory when the device is in unlocked state. When the user locks an iOS device, the keys are wiped from memory and access to any stored data encrypted with them is impossible. The system can only regenerate the keys, once the user entered his secret passcode in order to unlock the user interface.

As the unique device key involved in this process is only accessible on the device, brute-force attacks on the unknown passcode can only be performed on the device itself. Public documentation [80] of the iOS security architecture states that the derivation function for generating these keys first uses the password derivation algorithm PBKDF2 to create an intermediate key from the salted passcode. A large number of iterations involving the unique device key is then required to obtain the final key from the intermediate PBKDF2 output. The cryptographic accelerator in the SoC enforces a rate limit to slow down the key generation process, thereby making brute-force attacks slow.

TPM-based sealed memory can make use of user-provided secrets as well. In addition to a storage key and PCR configuration, external “authdata” derived from a password can be used to encrypt and decrypt sealed data.

Component-based architectures and the security mechanisms discussed on the previous pages are the foundation of the system architecture into which VPFS integrates. The threat model of VPFS strongly depends on these building blocks. I therefore discuss it upfront, in the following section, before putting it into perspective with related work.

2.4 Threat Model and Platform Assumptions

In the VPFS architecture, applications are assigned their own private file systems. The underlying OS configures inter-process communication (IPC) capabilities to enforce access control on trusted VPFS Core instances (e.g., using mechanisms described in Section 2.1 and 2.2). It is conceivable that multiple applications share a file system, if they trust each other and IPC capabilities are configured accordingly.

Any authorized application as defined above is trivially part of its TCB. I therefore rule out that applications behave maliciously with regard to their own data. I assume that an attacker wants to compromise *confidentiality*, *integrity*, or *recoverability* of data stored in a VPFS file system. These protection goals and what constitutes a successful attack on them are defined as follows:

Confidentiality: All data must be accessible only by an authorized application. After a successful breach of security, the attacker obtained plaintext data or filenames of some or all of the files stored in the file system.

Integrity: Any data (file contents) and metadata (filenames, sizes, timestamps, etc.) that VPFS provides to an application are *correct, complete,* and *up to date.* Correctness includes that the information provided is what was requested. If data or metadata lacks any of the three properties (e.g., parts of requested data are missing), then VPFS must detect this violation and report an integrity error to the application. A successful attack causes the application to process information that does not have all three of the aforementioned properties.

Recoverability: A consistent and recent backup must exist to recover file system contents after a system failure or an attack on integrity. A successful attack on recoverability prevents VPFS from creating a new or retrieving a recent backup over extended periods of time, or it corrupts the integrity of the backup.

The VPFS threat model considers three attack vectors that can be used to compromise data security as defined above. I first discuss offline attacks on data, then more complex online attacks on software and, finally, attacks requiring physical access.

Attacks on Data at Rest. All cryptographically protected file system contents are stored in an untrusted commodity file system, which is backed by an untrusted storage medium. It has to be assumed that an attacker can access these files, possibly after gaining physical access to the device, but it is computationally infeasible to decrypt any information without knowing the encryption key. Given write access to the encrypted data, the attacker may also tamper with it in any way (e.g., corrupt it, delete it, or roll back encrypted files to an older version). The trusted part of VPFS can detect such manipulation through unforgeable message authentication codes (MACs), as soon as it attempts to verify the integrity of the data it received from the untrusted parts of the file system stack.

Attacks on Software. At run time, VPFS relies on hardware address spaces in order to isolate the trusted and untrusted components effectively. Since multiple VPFS instances on the same device are independent of each other and their trusted cores run isolated, it is sufficient to consider only one VPFS stack in this analysis.

VPFS may reuse the file system infrastructure of a virtualized commodity OS. In the common case, when not being attacked, this large untrusted component of the VPFS stack is expected to work correctly. However, an attacker may be able to fully compromise the untrusted part due to its high complexity and enormous code size, which potentially includes the virtualized monolithic OS kernel. Such an attack may occur at any time during operation. Thus, untrusted components may stop working or they might deliver corrupted data. Furthermore, they might interact with the trusted VPFS Core in a way that violates the interface contract between these differently trusted components (e.g., messages may be maliciously altered, replayed, or not sent at all).

The TCB of an application using VPFS consists of many small components, which are assumed to be significantly harder to compromise than a monolithic commodity OS. Also, the small kernel and the user-mode services that are part of the TCB present a smaller attack surface that is scattered across multiple isolated components. I therefore assume that these components always work correctly and that VPFS Core thoroughly checks the integrity of all input from untrusted sources. I further assume that the system platform enforces a hardware-assisted secure startup process as described in Section 2.2 to counters offline manipulation attacks on executable files and critical system configuration. Thus, the system platform prevents manipulated software from running as part of an application's TCB.

Attacks on Hardware. Physical access enables an attacker to target critical hardware components of the user's device. VPFS itself cannot defend against hardware-based attacks, but the underlying system platform may provide certain protection. I give a brief overview of conceivable attacks for which effective countermeasures exist that the system platform can implement.

I assume that VPFS runs on a highly integrated mobile device and that hardware debugging support disabled. I rule out cold-boot attacks [107], which allow an attacker to obtain confidential data preserved in memory modules. This assumption is acceptable for the following reasons: (1) memory chips in SoC-based mobile devices are not removable and may even be part of the SoC casing, and (2) the SoC firmware can easily wipe any confidential data left in non-removable memory before the attacker can boot custom software to create a memory dump (if not prevented by secure boot already). I further assume that an attacker is not capable of intercepting – or manipulating in a meaningful way – the bus communication between the processing unit in the SoC and the main memory.

Although an attacker can directly access cryptographically protected data in the storage device, I assume that he is unlikely to gain access to the secret decryption keys or integrity anchors hidden in hardware sealed memory (see Section 2.3). A dedicated attacker with significant resources (circuit analysis equipment costing in the order of hundreds of thousands of dollars) might be able to extract such secrets, but only using destructive procedures [70, 142]. Such a highly capable attacker can compromise confidentiality, but it may not be possible for him to return an undamaged device to the legitimate user in order to avoid detection. A sealed memory implementation that requires an external authentication factor may render such key extraction attacks useless: for this type of sealed memory, the ability to decrypt sealed data depends on an additional secret that is not stored in the hardware. For example, the scheme described in Section 2.3 uses a user-provided passcode for that purpose.

An attacker may gain indirect access to cryptographic keys hidden in hardware, if he can mount a hardware-based "jailbreaking" attack [77]. This class of attacks exploit defects in the boot process of the mobile device in order to run arbitrary software with full privileges. Given unrestricted privileges, the attacker may be able to access cryptographic co-processors and device-specific encryption keys [112, 118]. The effect of this type of attack is similar to the key extraction attack as described before; the same countermeasure can be used (i.e., sealed memory schemes with external authentication).

2.5 Related Work

In the following, I will compare the VPFS approach to research and production systems based on security properties and architectural similarity. Chapters 4, 5, and 6 discuss further related work on specific problem areas in file system construction (metadata protection, consistency, and remote backup, respectively).

2.5.1 Decomposition of Local File System Stacks

Splitting the file-system code base and assuming mutual distrust between multiple instances of the VPFS stack is a key difference in comparison to monolithic file-system architectures. It enables fine-grained isolation in multiple dimensions. Previous approaches to decomposing the file-system stack are often limited in the degree of isolation they can support.

Isolation of Subsystems. QNX Neutrino RTOS [46] is a commercially available microkernel-based OS that implements system services as server processes. File system access is provided by a "Filesystem Manager" server. It implements the complete storage stack including block a device driver, a buffer cache, a specific file-system driver, and the POSIX API layer. Such a manager process thus implements a monolithic file system stack that is shared among all applications. In the microkernel-based research OS MINIX [40], the file system is also implemented as a shared user-space service. However, the disk driver and file system implementation run in separate address spaces. Like on QNX, a successful attack on such a shared file server compromises files belonging to all applications. There is no further isolation.

Isolation at Process Level. In the Exokernel architecture, most parts of the file-system implementation are linked directly into the application. This approach allows for customized implementations, for example, with different cache replacement strategies optimized for each application. File system instances in different application address spaces can share the same disk through a low-level in-kernel storage system called XN. The XN module of the kernel enables safe disk access by enforcing consistency constraints and access restrictions at the block level, while major parts of the complex file system implementation run in user mode. However, the TCB of any application still contains a complete file system stack, whereas VPFS delegates the most complex parts to untrusted infrastructure. An existing file system implementation cannot be converted into a XN-based library file system without potentially intrusive modifications, so unmodified reuse of a mature commodity file system like in the VPFS approach is not possible.

Isolation at System-Call Level. Proxos [152] uses virtual machines (VMs) to isolate applications with security requirements that commodity OSes cannot meet due to their complexity. Such critical applications run in their own VM, which also hosts a small private OS. Proxos implements a system call routing mechanism ensuring that all security-critical operations are handled by this private OS, whereas the commodity OS is used to service non-critical system calls. One of the services provided by the library-based private OS is a simple file system that transparently encrypts and hashes files. The cryptographically protected files are stored in the commodity OS's file system, while hashes of their content are stored on a block device that only the VM running the private OS can access. This approach is similar to VPFS, but the Proxos implementation does not address two key challenges that arise in this split architecture: Although integrity of file contents is protected, it does not ensure integrity of important file system metadata such as the directory structure or the fact that a specific version of a file should exist (or, conversely, that it should no longer exist). Also, the Proxos file system does not address the robustness problem that arises when file system contents and hashes are stored separately: the hashes and the data they protect must always be in sync, otherwise the integrity of files cannot be verified. VPFS ensures that this property holds even after crashes.

Combined Approaches. The Principled File System (PFS) [120] employs a combination of the previously discussed isolation approaches. Its design incorporates several security principles: (1) mutual distrust among isolated components, (2) least privilege for data access, (3) complete mediation to authorize every operation, and (4) minimization of trusted computing bases. The considerations taken into account to structure the PFS code base are similar to the design decisions that determined the architecture of VPFS. Certain subsystems in PFS can operate with fewer privileges than their equivalent in VPFS; for example, the buffer cache replacement algorithm in PFS cannot access the contents of the buffers, whereas the VPFS counterpart is part of the TCB. However, PFS assumes full trust in the storage device and does not attempt to reuse untrusted storage infrastructure. VPFS uses cryptography such that it does not need to trust the commodity file system that implements the most complex parts of the VPFS stack. As a result, VPFS has a smaller TCB.

2.5.2 Cryptographic Protection of Storage

Cryptographic protection of storage primarily aims to prevent leakage of confidential information in case a computing device or storage media gets lost. Consequently, the commonly used solutions only encrypt data, few also provide integrity guarantees. The systems can be categorized into two classes, depending on where in the storage stack the cryptographic operations take place.

Encrypted Block Devices. The most widely used approach is to encrypt the complete storage medium; examples are BitLocker, Truecrypt, dmcrypt in Linux, or FileVault 2. The file system that is layered on top does not need to be modified, as encryption and decryption is performed transparently at the level of disk blocks. Integrity is not ensured, unless the file system itself is using checksums to detect corruptions (like ZFS [57] does, for example). BitLocker [99], TrueCrypt [53], and FileVault 2 [96] use tweakable block ciphers. They cause even small changes (e.g., one bit) in the ciphertext to result in widespread corruption across the entire block's plaintext after decryption, thereby making meaningful changes to file system structures infeasible.

Nevertheless, the complete file system implementation is part of the TCB. As the storage medium is typically shared among applications, the file system is responsible for enforcing fine-grained access control (e.g., based on user accounts or other principals). Solutions that move encryption operations into the storage device itself have essentially the same trust model, although some of them [54] integrate with trusted computing support of the platform to restrict storage access to specific OS configurations.

Encrypted File Systems. The Cryptographic File System (CFS) [92] for UNIX adds encryption further up in the storage stack, on top of an existing file system. It is implemented as a NFS client that transparently encrypts and decrypts file contents and path elements, before they are sent to an NFS server running on the same machine or a remote host. Linux' eCryptfs [29], the Encrypting File System (EFS) [31] in Windows, and the user-space file system EncFS [30] are also layered on top of another file system. Both eCryptfs and EncFS offer limited integrity guarantees for individual files. EncFS can also detect if files are renamed or moved, but their contents can still be rolled back to older versions. It cannot detect deletion attacks on individual files.

In Apple's iOS, a hybrid approach is taken [80]: encryption is done at the block level, but different keys are used for the different types of blocks. Metadata blocks are encrypted under a partition key, but the system uses unique per-file keys for blocks containing user data. Per-file keys are stored in the metadata that is protected by the partition key. However, some of these keys can only be fully decrypted, if an additional user-provided secret (i.e., the passcode, as described in Section 2.3) has been entered. Once the per-file keys are unlocked, the complete storage stack becomes a shared part of the TCB of all applications.

2.5.3 Untrusted Storage

Remote Storage. Systems such as the Secure Untrusted Data Repository (SUNDR) [124] or SiRiUS [104] are specifically designed such that trust in remote storage can be minimized. They encrypt data and metadata and protect integrity through cryptographic hash chains and signatures. SUNDR stores content-addressed blocks on a special server, whereas SiRiUS can be layered on top of existing, unmodified network file systems (e.g., NFS [141]). Like VPFS, these two systems split the storage stack into a trusted and untrusted part. However, the untrusted storage is provided by remote servers and the client side is assumed to be fully trusted.

Local Storage. The Trusted Database System (TDB) [127] distrusts local storage and other software running on the same system. It assumes a secure execution environment and sealed memory to protect its encryption key; it also requires a small amount of tamper-resistant storage for the root node of a Merkle hash tree, which it uses to ensure integrity. TDB makes assumptions similar to VPFS about platform security. However, its strategy to reduce the size and complexity of its TCB is to remove functionality and to offer a less powerful interface, whereas VPFS isolates its security-critical core from the untrusted commodity infrastructure it reuses.

Operations on Encrypted Data. CryptDB [136], on the other hand, demonstrates secure reuse of untrusted infrastructure for databases in a way similar to the VPFS approach. The system implements a SQL proxy that stores encrypted records in a commodity database system that is hosted on another, untrusted machine. The proxy employs an onion-like encryption scheme, where each record is wrapped in multiple layers of encryption. The outer-most layer uses the strongest encryption and reveals the least amount of information to the untrusted database system. The proxy may remove one or more encryption layers for records in specific tables, so as to enable certain database operations to be performed on them. For example, deterministic encryption in the second layer enables the untrusted database software to compare records, whereas deeper layers preserve ordering properties or even allow for simple arithmetic operations to be performed on records that are protected by homomorphic encryption. Thus, CryptDB can delegate execution of complex SQL queries and updates to the untrusted database backend, but it trades this ability for strong integrity guarantees. Applying the trusted wrapper approach to a file system stack creates an equivalent conflict; I discuss the tradeoffs between metadata integrity protection and reuse of untrusted lookup functionality in the context of VPFS in Chapter 4.

2.5.4 Versioning and Tamper-Proof Storage

Tamper-proof versioning storage systems address the problem of data integrity and durability under the assumption that the OS and applications cannot be trusted (e.g., because they are not sufficiently secure due to their complexity). They aim to prevent permanent data loss due to file corruption or deletion originating from within the commodity software stack, including the OS kernel. Such data loss can occur either due to accidents or bugs in the software, or because an attacker compromised the complex code bases. In the latter case, the attacker may try cover up traces of the intrusion or he wants to destroy critical data.

Versioned Object Storage. S4 [150] implements a network-attached object storage device that autonomously keeps a history of each version of an object. As files are mapped to such objects, the system can retain each version of a file even after it has been deleted from the OSes point of view. S4 ensures durability of older versions using log-structured data storage and cryptographically protected metadata journaling. This protection goal is orthogonal to the security guarantees of VPFS.

Versioned Block Devices. The VDisk [158] system has a trust model that is similar to S4, but it uses virtualization to implement secure file-system versioning at the level of disk blocks. The core component of VDisk is the *VDisk secure kernel*, which runs in a dedicated, stripped down VM. A commodity OS running in another VM uses the virtual block device provided by the secure VDisk component. The VDisk secure kernel implements the security-critical functionality of block-level file system versioning, including logging of data blocks and enforcement of a version retention policy. Garbage collection of obsolete versions is implemented by an untrusted cleaner process running inside the commodity VM. The VDisk cleaner determines which older versions of file system blocks can be discarded and communicates this information to the VDisk secure kernel.

Like VPFS, the VDisk system has a split architecture that enables a small TCB by delegating non-critical work to untrusted components. Both VDisk and S4 address a different protection goal than VPFS: they ensure integrity and durability of previously written data within a certain timeframe. VPFS distrusts the lower layers of the storage stack with respect to confidentiality and integrity. It therefore arrives at different TCB boundaries than versioning storage.

Nevertheless, the use of logging and untrusted garbage collection turned out to be suitable approaches for solving the robustness problem in VPFS. Chapter 5 covers VPFS' solution to this problem in detail, but the following Chapters 3 and 4 first discuss the basic design and explore to which extent untrusted infrastructure can be reused.

Chapter 3

A Trusted File System Wrapper

In this chapter, I will discuss the design and implementation of the basic subsystems of the Virtual Private File System (VPFS). I start out by motivating the general architecture and how to reuse untrusted storage. I then discuss key strategies to minimize the trusted computing base (TCB) and the attack surface. Finally, I describe how the central data structure of VPFS, the block tree, contributes to this goal.

3.1 General Architecture

VPFS is influenced by an earlier effort to build a trustworthy file system called TrustedFS [155]. The primary design goal of this earlier file system was identical to that of VPFS: to make the file system code base less vulnerable to attacks in order to improve security. However, TrustedFS ignored several key problems that VPFS addresses, the most important being (1) robustness against crashes, and (2) secure integration of backup and restore with minimal impact on TCB sizes. To support these essential features, major design decisions with regard to caching, protection of metadata, and reuse of untrusted infrastructure have been rethought for VPFS.

VPFS implements a security wrapper that is layered on top of an existing file system. It uses encryption and ensures integrity based on a Merkle hash tree [132]. The leaves of the hash tree protect fixed-size blocks in an untrusted storage. VPFS maps each individual file in the virtual private file system to a file in an untrusted commodity file system. I will now discuss the design options for building a cryptographic file system to demonstrate that this design is the most suitable one for VPFS.

3.1.1 How to Reuse Untrusted Infrastructure

Splitting the file system stack into security-critical and non-critical components enables reuse of untrusted infrastructure. The separation can be made at various points between the storage device at the bottom of the stack and the virtual file system (VFS) at the top. Figure 3.1 shows the layers of a commodity file system stack and exemplifies where the storage stack could be split.

First Look at the Design Space. Both the level of abstraction and the complexity of the interface between the two parts vary, depending on where in the stack the split is made. The boundary also determines what functionality can be reused without having to trust it. When considering confidentiality and integrity as protection goals, three points in the design space deserve a closer look:

1. **Single Container:** Splitting above the block storage layer removes the storage device, the disk driver, and volume management (i.e., partition handling) from the TCB. All layers above this boundary, including a complete file system implementation, need to be be trusted; Figure 3.1a illustrates this splitting approach. The trusted part encrypts individual blocks

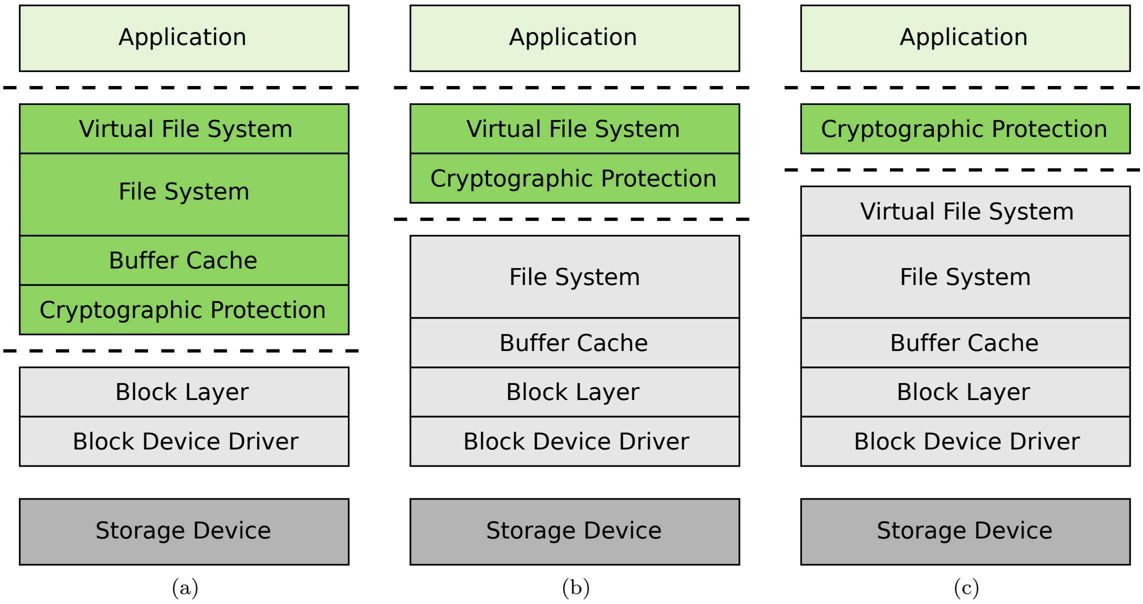


Figure 3.1: Conceptual view of three points in the design space for splitting a file system stack. Cryptography ensures protection of data and metadata. Subsystems below the cryptography layer are untrusted.

in a single container such as a disk partition. Integrity enforcement must be integrated with the trusted file system code base. Retroactively adding integrity support at the block level incurs significant overhead: there must be a hash or message authentication code (MAC) [89] for each block and they need to be stored persistently. The MACs need to be retrieved for checking and be atomically updated together with the respective blocks whenever data is written.
The single container approach delegates the least amount of functionality to untrusted code bases. If multiple instances of the file system stack shall be supported, the block layer must partition – or virtualize – the physical storage medium such that each instance has its own range of logical block addresses.

2. **Per-File Containers:** Splitting right above the file system implementation removes the bulk of the storage stack from the TCB, as shown in Figure 3.1b. The VFS layer remains in the trusted component, which can map files in a virtual private file system to "physical" files managed by the untrusted code base. The trusted component cryptographically protects all file contents stored in the untrusted file system and organizes the individual per-file containers in a flat namespace.
The difficulty of splitting between an existing file system code base (e.g., Ext4) and the generic VFS layered on top lies in functional dependencies between them: commodity file system implementations in monolithic kernels call many generic functions of the VFS and tightly integrate with the page cache infrastructure.

3. **Per-File Containers, Metadata, VFS:** To further reduce size and complexity of the TCB, the generic VFS could become part of the untrusted component as well (see Figure 3.1c). On Linux, the VFS implements metadata caches for inodes and directory entries.
This outsourcing strategy works with encryption layered on top of the VFS, but memory-mapped file access is impossible, if only an untrusted page cache exists in the split file system stack. The approach also limits integrity guarantees to individual files, because untrusted code manages certain metadata. For example, if the namespace (i.e., directory

tree) is untrusted, a maliciously removed file cannot be distinguished from a file that was never there or has been deleted by an authorized application.

Due to its simplicity, the first option is favored by many existing solutions such as dmcrypt, TrueCrypt, and others discussed in Section 2.5. The second and third option require a more complex interface between the trusted and the untrusted part of the split file system stack. But they delegate significantly more of the file system functionality to the untrusted component. Thus, they are more suitable for reaching the goal of a small TCB. The untrusted file system implementation can be shared among multiple instances of the trusted component without loss of isolation among application-specific TCBs. The two options represent the opposite ends of the design range where each file is mapped to its own backing store, rather than storing all file contents in a single container. Solutions within this range of the design space trade size and complexity of the TCB for the ability to enforce security guarantees for metadata.

Reusing Commodity File Systems. The possibility of reusing a commodity file system is very attractive for any of the splitting approaches I outlined, because it reduces engineering effort and increases utility. A significant portion of the feature set of mature file system implementations can be reused in a VPFS stack. In particular, VPFS can benefit from the following features and properties offered by existing implementations:

- **Compatible On-Disk Format:** Code for reading and writing on-disk data structures can be reused. It may even be possible to reuse maintenance tools (e.g., for resizing or defragmenting partitions).
- **Efficient Allocation:** Commodity file systems contain optimized algorithms for block and metadata allocation (e.g., delayed allocation in Ext4 in order to avoid fragmentation).
- **Free-Space Tracking:** File systems optimized for flash-based storage support mechanisms such as the TRIM command, which can be used to inform the flash translation layer (FTL) of the underlying storage device about blocks that no longer contain user data.
- **Request Ordering:** Consistency of on-disk data structures requires that updates to blocks be made persistent in a well-defined order. For example, file systems enforce write order using barrier operations or by flushing buffers on the device. Furthermore, read operations may be handled more efficiently on certain hardware, if the file system can submit multiple requests, which the storage device can serve in optimal order (e.g., as enabled by request queuing on rotating hard disks).
- **Bulk Transfers:** Optimizations such *read ahead* and *grouped writes* hide latencies when accessing the storage device.
- **Garbage collection:** Log-structured file systems, which are suitable for flash-based storage used in mobile devices, need to garbage collect log segments containing little or no live data. Journaling file systems need to free retired transactions. Both operations are non-trivial to implement.
- **Fault tolerance:** File systems such as ZFS [57] support redundancy through block-level replication, even for individual files. Such features may enhance reliability in case of failing storage media.
- **Scalability:** In most cases, commodity file systems are designed as shared file systems that efficiently support multiple applications and concurrently executing threads. They can also adapt to different resource availability (e.g., storage and memory size).

None of the features described above need to interpret or modify file contents; they operate on opaque data. Thus, the parts of a file system that implement these features can be untrusted with regard to confidentiality and integrity. However, the list is not exhaustive: certain functionality of file system stacks may not be effective when encryption is used. For example, deduplication of data is impossible, unless the same plaintext results always in the same ciphertext. Similarly, compression does not work for encrypted data, because strong encryption algorithms produce ciphertext with maximum entropy. Fortunately, deduplication is not relevant for the mobile application use cases that VPFS is targeted at and the largest pieces of data stored on mobile devices are commonly media files that are already compressed.

A Splitting Approach for VPFS. Size and complexity of a reused file system implementation are of concern for the single container approach to splitting, because the entire code base is part of the TCB. Thus, a simple implementation better suits the goal of TCB minimization. On the other hand, a simple or stripped-down file system code base must likely cut down on optimizations or have a reduced feature set. The choice of existing file system implementations is further limited, because the vast majority of existing implementations do not perform pervasive integrity checks for locally stored data. Unfortunately, integrated integrity support is desirable for good read and write performance; I hinted at inherent costs incurred by using MACs at the beginning of this subsection, a more thorough discussion of the problem follows in Chapter 5.

The compromises I just discussed do not have to be made when moving the commodity file system into the untrusted part of the storage stack. The TCB does not grow or get more complex beyond what is necessary to securely reuse the untrusted implementations. So, clearly, reusing an *untrusted* commodity file system is preferable to including the same – or a less powerful – code base in the TCB. VPFS therefore splits the file system stack *above* the commodity file system. This design decision enables VPFS to reuse files in the untrusted file system as per-file backing store.

I will focus on individual files and their content now in order to explain the fundamental design and implementation aspects of VPFS. Protection of metadata and naming will be covered in the next chapter.

3.1.2 Cryptographic Protection

General-purpose file systems implement files as linearly addressed containers that applications can read from and write to at the granularity of individual bytes. A file grows dynamically when an application writes beyond the end of the file, and it shrinks, if the application truncates it. To meet the expectations of application developers, the VPFS Core component must provide a similar abstraction of files to applications.

Expected Access Patterns. The abstraction of files as seen by the application matches the abstraction of the cryptographically protected *file containers* provided by the untrusted commodity file system. It is tempting to regard the trusted part of VPFS as a straight-forward cryptographic wrapper that transparently encrypts file contents before passing them to the untrusted component; during read operations, the wrapper would transparently decrypt the data. However, the solution is not quite so simple. The reason for that lies in the many ways that applications are allowed to access file contents: they may randomly read and write byte-aligned chunks of data with varying sizes. Applications are free to access just parts of a file – they do not necessarily read or write a file from begin to end. Furthermore, applications can map page-aligned regions of files into their address spaces, which requires the wrapper to manage memory holding a plaintext copy of the mapped regions.

Random access patterns must be supported by the encryption scheme as they are of critical importance for the mechanism that ensures data integrity.

Encryption. Ciphers can encrypt data only in certain block sizes (e.g., 16 bytes in the case of AES [34]). But for security reasons, it is desirable to encrypt larger regions of files (e.g., to hide the exact size of a file). A sensible size of an encryption unit in cryptographic file systems is the hardware page size. VPFS follows that path and encrypts file contents at the granularity of 4 KB blocks. The cryptographic community created a toolbox of well-studied encryption algorithms and best practices for using these ciphers. Well-understood modes of operation for block ciphers such as cipher-block-chaining (CBC) or XTS enable secure encryption of data larger than the block size of a cipher. VPFS uses AES-CBC encryption to ensure confidentiality of individual blocks stored in file containers. It thereby enables efficient access to parts of a file.

Hash Trees. Hash-based message-authentication codes (HMACs) [89] provide an efficient way to detect unauthorized modification of data. However, protecting the integrity by computing a single HMAC over an entire file is impracticable, because applications may randomly access parts of a file. Files can be arbitrarily large, depending on the use case. By calculating an HMAC for each block in a file container, the trusted VPFS Core could verify the integrity of individual blocks, or update them independently from each other. Unfortunately, independent HMACs are vulnerable to roll-back attacks: an attacker can replace a block B and its HMAC $h(B)$ with older versions B' and $h(B')$. Blocks and their HMACs could also be moved to a new location without the VPFS Core noticing. VPFS prevents these attacks by using a Merkle hash tree [132]. Such a tree chains integrity information of each leaf node (i.e., block in a file container) up to a single root hash, thereby enabling fast updates and verification of individual leafs. Merkle trees are commonly used in distributed file systems [104, 119, 124] as well as database systems [127]. However, the VPFS Core does not use plain hashing, but it computes an HMAC-like keyed hash for each node in the tree. I will use the acronym MAC in the remainder of this thesis.

Key Management and Sealed Memory. There are local [80] and distributed file systems [104, 124] that generate encryption and signing keys for individual files; these subkeys are protected by a small number of root keys. The primary reason for building up such key hierarchies is to enable fine-grained access control for individual files [80, 105, 119]. Because VPFS provides private storage to mutually distrusting applications, it does not need to enforce access control at the file level. Instead, it trusts the underlying operating system to enforce address space separation and restrictions for inter-process communication (IPC), such that only authorized applications can access files in a certain VPFS instance. Given this platform support, each instance of VPFS Core uses just two keys to protect all its files: one encryption key and one key for calculating MACs. The trusted core component is the only component in the file system stack that has access to these keys. VPFS relies on the platform's sealed memory service to enforce access control for each pair of keys.

Certain sealed memory implementations that I described in Section 2.3 can ensure freshness of sealed data. To ensure that VPFS file system contents cannot be rolled back to an older version, VPFS Core protects the root MAC of the Merkle tree with replay-resistant sealed memory.

3.1.3 Introducing the VPFS Subsystems

Figure 3.2 shows the internal structure of the trusted VPFS Core and the layers of the untrusted parts of the file system stack. An application accesses its VPFS Core instance via a private VFS module, which is therefore part of the TCBs. The VPFS Core component consists of two major subsystems: a memory file system (memfs) and a persistency layer. The latter implements cryptographic protection.

Trusted Memory File System. The memfs subsystem provides the basic abstraction of files to applications. Conceptually, it works like the in-memory temporary file systems (tmpfs) that are available on many UNIX systems (cf. Wikipedia: tmpfs). The VPFS memory file system stores file contents in page-sized buffers in the VPFS Core component's private memory. An application can request buffers to be mapped into its own address space, thereby enabling memory-mapped file access. Alternatively, it can use the `read()/write()` interface provided by its VFS layer.

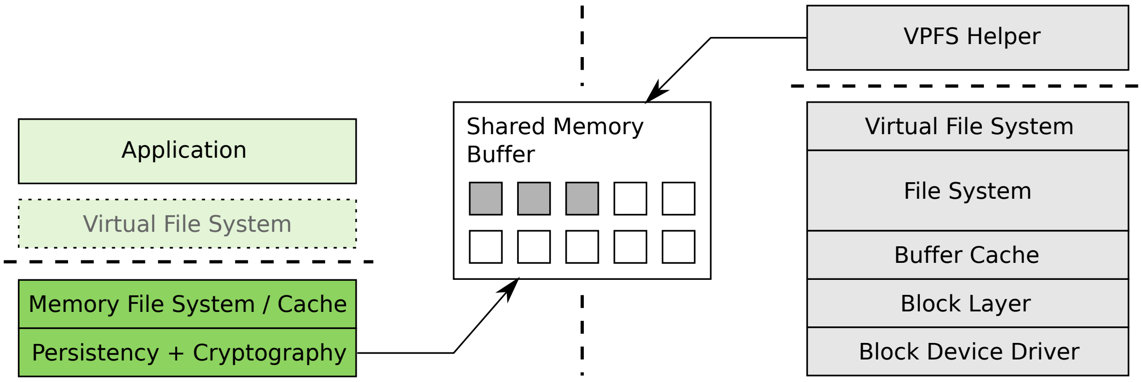


Figure 3.2: VPFS Architecture: The trusted VPFS Core implements a memory-based file system and persistency subsystem, which maps files to cryptographically protected file containers in the untrusted commodity file system. Cooperation across address spaces is based on signaling and a shared memory buffer that is used to transfer cryptographically protected data.

The buffers either hold file contents (i.e., user data) or they contain metadata. There are two types of metadata: (1) Merkle tree nodes that contain the chain of MACs required to verify the integrity of a block, and (2) blocks that contain traditional metadata structures such as inodes. The memfs implementation organizes all inodes in a table stored in a special file. A single root inode describing the inode file itself is stored in the superblock-like data structure called *file system root info*. Figure 3.3 shows the relation of the different block types to each other and how inodes link to per-file Merkle trees. Each inode contains the MAC of the corresponding file's root node, thereby forming one combined Merkle tree that is protected through a MAC computed over the root info.

Trusted Persistency Layer. The persistency and cryptography subsystem enables VPFS Core to function as a trusted wrapper for the commodity file system. This layer does not only provide access to untrusted storage, it also extends the memory-based file system such that it can operate as a trusted buffer cache. Ignoring robustness against crashes for the moment, the interaction between these two subsystems of VPFS can be summarized as follows: To transfer data and metadata blocks from volatile memory to stable storage, memfs requests the persistency layer to flush buffer contents to their respective location in file containers. Conversely, if a certain block of a file is not cached, the memfs subsystem instructs the persistency layer to retrieve the encrypted copy of the block and then, after checking integrity, put the plaintext into a free buffer. Both subsystems of VPFS Core are aware of the parent–child relations between blocks in the Merkle tree. If necessary, memfs loads parent nodes first, thereby ensuring that the MAC required for integrity checking is available. The persistency subsystem ensures that MACs in parent nodes get updated, before the system evicts a block.

Untrusted Helper. As far as file contents are concerned, the untrusted parts of the file system stack are merely responsible for persistently storing encrypted and integrity-protected data. The read/write strategies and optimizations they implement shall ensure reliable storage and provide high performance. In order to make use of the commodity file system, the VPFS Core component cooperates with an untrusted helper. The trusted core's persistency layer and the helper communicate via shared-memory and a signaling mechanism provided by the underlying platform. The helper component translates requests from the trusted persistency layer into regular API calls of the commodity file system. It is possible to reuse the commodity file system without modification. For example, in the VPFS prototype implementation, the helper is a Linux user-space program that uses the standard POSIX file system interface of a virtualized Linux kernel [110].

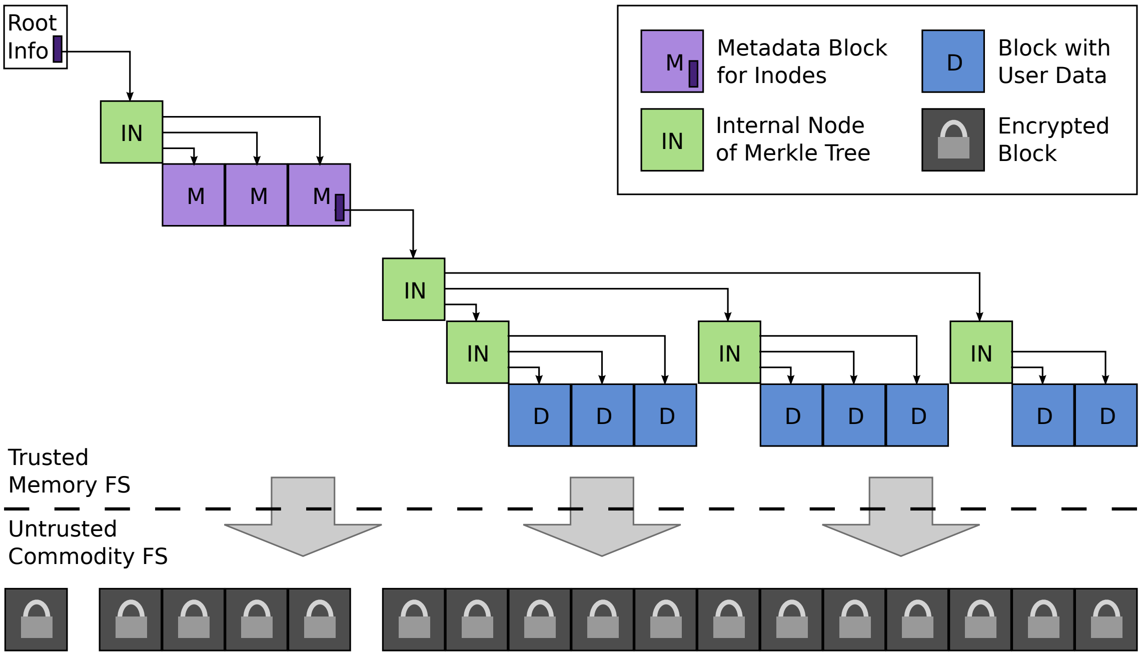


Figure 3.3: Mapping of file contents and metadata to untrusted storage: The memory file system in the VPFS Core component serves as a cache for encrypted blocks stored in file containers. A special metadata file contains the inodes of all regular files (inodes belonging to files not shown in the picture are omitted). Each inode contains the root MAC of the Merkle tree embedded in the corresponding file container. The file system root info contains a special root inode for the inode file itself, thus, all per-file trees are protected by a single MAC computed over the root info.

Duplication of Functionality. As shown in Figure 3.2 and explained in the previous paragraphs, certain functionality in the two parts of the file system stack is duplicated: (1) the application accesses VPFS Core via a private VFS layer, and (2) the memory file system in VPFS Core serves a file system buffer cache. Implementations of both subsystems also exist in the untrusted file system stack. A VFS implementation is part of the TCB, because applications may want to use convenience wrappers for the fundamental file system operations provided by VPFS. An example is the `stat()` API that the VFS maps to a combination of `lookup()`/`open()` and `fstat()`. Including a private buffer cache in the TCB is justified, because its ability to cache parent nodes of the Merkle tree is essential for efficient integrity verification.

Buffer Cache Size in VPFS Core	16 MB	2 MB	256 KB	32 KB
Total Runtime of Benchmark (PostMark1 Trace)	0.07 s	0.65 s	0.99 s	4.04 s

Table 3.1: Impact of buffer cache size on benchmark runtimes: More memory for the trusted buffer cache dramatically reduces the time needed to replay a trace of the PostMark benchmark.

Furthermore, caching plaintext versions of data and metadata blocks greatly improves performance. Table 3.1 shows how increasing the cache size dramatically speeds up workloads that repeatedly access the same files. Finally, the ability to cache file contents is required in order to support memory-mapped file access.

3.1.4 Alternative TCBs for Confidentiality and Integrity

In light of the description of VPFS I gave so far, it is reasonable to conclude that VPFS Core must have access to both the encryption key and the MAC key; VPFS Core is part of the TCB with respect to confidentiality and integrity. Alternative decompositions of the two TCBs are possible, but I rejected them as they do not substantially improve overall security. The following discussion of two obvious alternatives concludes the evaluation of design options for splitting VPFS.

Smaller TCB for Confidentiality. Encryption and decryption could be handled by an additional component layered between the persistency subsystem of VPFS Core and the untrusted helper. The TCB for confidentiality is minimal in this architecture, if (1) only the encryption component has access to the key, and (2) neither the application nor VFS and VPFS Core can communicate with any untrusted software on the system. However, the possibility of a covert channel through the encryption component still exists. Therefore, this finer-grained separation only improves security against bugs in VPFS Core that cause plaintext data to be written to the shared memory region accidentally; a compromised VPFS Core instance could still leak information over the covert channel.

Split TCB for Integrity. Another alternative is to single out "islands" of functionality in the file system stack that are critical for integrity. The components of the software stack starting at the persistency layer in VPFS Core up to the application are trivially part of the TCB. However, a second isolated component at the bottom of the storage stack could be introduced. This component could enforce that modifications to untrusted storage do not damage file containers, for example, through tamper-proof versioned storage [150, 158] or by verifying signatures of written data [133]. Such an architecture enables stronger integrity. However, VPFS still depends on untrusted components to make progress. As they can mount denial-of-service attacks that prevent access to the data, splitting the TCB for integrity any further yields no significant advantage.

3.2 Design Principles

Now that the general subsystems of VPFS and their responsibilities have been explained, I discuss design principles and give concrete examples how they help to reduce size and complexity of the TCB.

3.2.1 Metadata Integrity and Input Sanitization

The on-disk formats of commodity file systems are complex, and they have often been extended with new features and encoding variants over the course of several years or even decades [153, 32]. A major reason for reliability and security problems in file system stacks is that the code base needs to correctly interpret these complex metadata structures.

Threats Originating from Untrusted Input. For example, block pointers refer to opaque bit patterns on the storage medium; the correct interpretation of these bit patterns depends on the trustworthiness of the pointer. Similarly, information encoded in allocation bitmaps, free-block lists, and other metadata objects must be trustworthy, yet it may not be possible to verify the correctness of this information without running an expensive file system checker. Manipulation of on-disk data structures may enable an attacker to compromise security. Encryption based on tweakable block ciphers [99] prevents the attacker from changing individual metadata structures in a meaningful way, but modification of a block's ciphertext still causes random corruption after decryption. Such corruption may lead to undefined behavior, which can cause crashes or data loss. The manual page for the `mount` utility on Linux and BSD systems even states: *"It is possible for a corrupted file system to cause a crash"*.

Pervasive Integrity Checking. The VPFS Core's persistency layer performs pervasive integrity checking of every piece of data or metadata that is read from untrusted storage. The checks guarantee that every metadata block loaded into a buffer in VPFS Core's memfs subsystem is trustworthy: if, and only if, the MAC is correct, the trusted persistency layer wrote precisely this block to the specified location in a file container. Thus, cryptographic integrity checks not only protect file contents, they also reduce the attack surface of the file system TCB.

3.2.2 Simple Inter-Component Interfaces

The trusted and untrusted components of the VPFS stack run isolated in separate address spaces. However, this strong isolation must necessarily be broken to a certain degree in order to enable communication and transfer of data. The interface facilitating this cooperation is part of VPFS Core's attack surface and its implementation is security-critical: it must ensure that malformed input from the untrusted component does not compromise the integrity of the TCB.

Threats Originating from Complex Interfaces. The effectivity of any such sanitization depends on the complexity of the interface and the data items that travel through it. Complex interfaces are harder to secure against attacks. For example, the Google Chrome web browser features a multi-process architecture, which uses address-space separation and sandboxing to contain the damage that a successful attack on the browser's rendering and script engine can cause. Nevertheless, the browser could be fully compromised [76] through a chain of exploits for six different bugs. One of the exploits triggered an input sanitization error in a complex cross-process graphics rendering interface [81].

Persistency Interface in VPFS. In order to secure VPFS against similar attacks, the interface between VPFS Core's persistency subsystem and the untrusted helper is designed to be simple. It is based on shared memory that enables zero-copy data transfers and a signaling mechanism. Communication follows a request–response pattern, which is always initiated by the trusted component. VPFS Core writes requests to the shared memory and signals the untrusted helper; the untrusted helper processes the requests and signals the requestor after it put the response into the shared memory region. The structure of the shared-memory buffer is as follows:

- A fixed-size array of `int` variables, into which VPFS Core puts an opcode (`Root_info_read`, `Root_info_write`, `Block_read`, `Block_write`, `Shared_mem_init`) and additional parameters; the untrusted helper uses the first element to communicate the return value of the requested operation

- A fixed-size buffer for transferring the file system root info (cryptographically protected by the sealed memory service)

- Two fixed-size descriptor arrays, where each element is a `(file_container_id, block_no)` tuple; one array describes blocks to be written, the other describes blocks that the untrusted helper loaded into shared memory in response to a read request

- Two fixed-size block-buffer arrays, where each element holds the encrypted block described in descriptor arrays

To issue requests, the trusted persistency subsystem in VPFS Core performs only write requests to the write-descriptor and write-buffer arrays. After receiving the completion notification, it reads the return value variable. If the return code does not indicate an error, VPFS Core searches the fixed-size read-descriptor array for the requested `(file_container_id, block_no)` pair. The resulting index (computed locally and therefore trustworthy) specifies where in the read-buffer array the cryptographically protected block is located. No further sanitizing is required; the next step for VPFS Core is to decrypt the block into a buffer and verify integrity of the content based on the MAC as described in Subsections 3.1.3 and 3.2.1. Simple optimizations for this basic cooperation scheme will be discussed in Section 3.4.

3.2.3 Handling and Surviving Errors

Pervasive integrity checking guarantees that VPFS Core never processes corrupted metadata. If the trusted persistency subsystem cannot verify a decrypted block as correct, it immediately deallocates the cache buffer into which it put corrupted block contents. In such a situation, VPFS Core's memfs cannot finish the operation that initially caused that block to be read from untrusted storage. To handle the error gracefully, VPFS Core aborts the ongoing operation and reports an appropriate error code to the application (e.g., `EIO` or a custom code like `EINTEGRITY`).

Internal Consistency. Functions that need to modify multiple metadata blocks first retrieve all blocks and make sure they are pinned in their respective buffers. This precaution avoids late aborts due to integrity errors, but code paths that call multiple submodules to perform a more complex operation cannot always use this strategy. If a composite operation aborts, the memfs subsystem reverts any partial changes that have already been applied to metadata structures in the cache buffers. For example, creating a new file involves two steps: (1) allocating and initializing a new inode, and (2) creating the filename in the parent directory. If the second step fails, memfs deallocates the stale inode. This cleanup ensures that the internal state of the memfs subsystem is always consistent. Thus, the application may continue to use the VPFS instance and it may be able to access cached file system content. However, errors caused by untrusted parts of the file system stack may prevent VPFS from making further progress.

Language Support. In contrast to commodity file systems, which are commonly written in C, VPFS Core is implemented in C++. This programming language supports exceptions for error handling. While using C++ and exceptions is not a design principle, the features of this language simplify the implementation: in many cases, exception support frees the developer from the tedious task of manually writing error handling code around function calls. VPFS Core uses C++ exceptions extensively.

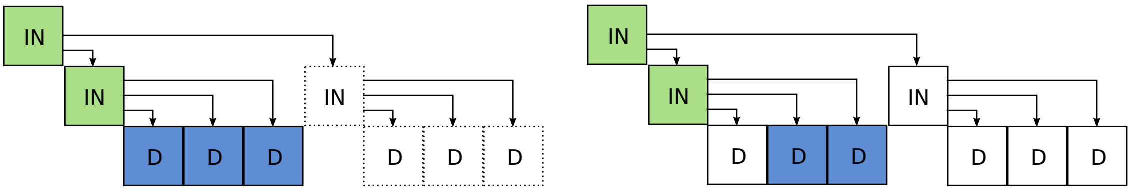
The language also supports the concept of "scope guards". Where possible, VPFS Core uses them for automatic cleanup of temporary resources (e.g., unpin buffers, release file handles). All other cases including rollback of aborted operations are handled within C++ `catch` blocks.

3.2.4 Reuse of Generic Implementations

VPFS does not only reuse untrusted infrastructure, but it is also designed to reuse functionality within TCB components. One example is that the memfs subsystem stores inodes as a simple table inside a special file, which is mapped to a file container in the same way as files containing user data. The consistency mechanism described in Chapter 5 reuses even more infrastructure of the memfs code base, including application-level APIs. The high degree of reusability within VPFS Core is largely enabled through careful design of its central data structure: the *block tree*. The next section covers this data structure and the algorithms tailored around it.

3.3 Central Data Structure: The Block Tree

One of the primary metadata structures in VPFS is the Merkle tree, which contains the MACs that protect the integrity of every data and metadata block in a VPFS file system. VPFS stores the nodes of this tree in encrypted file containers, which are organized in fixed-size blocks as illustrated in Figure 3.3 on page 37. A block containing an internal node of the Merkle tree can store more than a hundred MACs when using a state-of-the-art hashing algorithm and a block size of 4 KB. The maximum number of child nodes that a block can accommodate also depends on what other information, in addition to MACs, is stored in the block. VPFS maintains several metadata items in the block tree's internal nodes that support the construction of a smaller TCB, as more key functionality of VPFS Core's memfs subsystem can exploit the tree structure.



(a) Read dependency and block retrieval: Colored blocks are cached already, the subtree containing the white blocks shall be loaded now. Before leaf nodes in this subtree can be retrieved, VPFS Core must first load their parent node in order to obtain the MACs needed to check integrity of the leafs.

(b) Update dependency and dirty state propagation: Colored blocks are modified, the white blocks are unchanged. Modifications to non-root nodes cause dirty state to propagate up to the root. A clean internal node signifies that the entire subtree below it is clean and can henceforth be skipped when iterating over dirty blocks.

Figure 3.4: Two examples for inter-block dependencies in the block tree.

3.3.1 The Case for a Generic Block Tree

I discuss all relevant uses for the block tree upfront, because the combination of their requirements led to the TCB-efficient design of generic algorithms. I start with inter-block dependencies and how they affect block retrieval and write back, then I discuss iterating over blocks based on different criteria.

Read Dependencies and Block Retrieval. Assume that an application created files in its private file system and that all data and metadata have been flushed to the cryptographically protected file containers. Further assume that this VPFS instance has just been remounted. VPFS Core's memfs has no data or metadata blocks cached in its buffers, only the file system root info has been loaded. When the application starts to access existing files at this point, VPFS Core needs to load blocks back into the cache buffers.

Pervasive integrity checking imposes restrictions on the order in which VPFS Core can retrieve blocks from file containers. This order is determined by the parent–child relations of blocks in the tree structure, as illustrated in Figure 3.4a. A block can only be loaded, if its immediate parent is already cached in a buffer in VPFS Core's private memory. Otherwise, the trusted persistency layer in VPFS Core cannot verify the integrity of the block, because the block's MAC in the parent buffer is unavailable. By construction, each block has a transitive dependency on its direct and indirect parents up to the root node of the MAC tree (i.e., the file system root info). A request to retrieve a single block may therefore cause VPFS Core to load multiple parent nodes from file containers, potentially up to the root node of the inode file.

Update Dependencies and Block Write Back. Modifying blocks (or adding new ones) requires VPFS Core's persistency layer to update MACs in the parent nodes. Due to the properties of hash-based trees, VPFS Core can calculate MACs only in strict bottom-up order and each update propagates to the root. I defer the discussion of how to ensure consistency of the block tree during write back to Chapter 5. However, a consequence of the update dependencies is that the memfs subsystem should not select a block containing MACs for eviction, unless this block has no more child nodes cached.

Block Iterators. To serve the read or write requests of an application, memfs needs to locate the buffer that caches a block, or it must retrieve the block as described above. Read and write requests specify the exact range of blocks at the leaf level of the block tree (i.e., the blocks storing user data). However, certain operations need to be performed on a set of blocks without knowing in advance which blocks are part of the set, and which can be ignored. The following examples reflect key use cases:

1. **Invalidating Buffers:** When an application deletes a file or truncates parts of it, all cache buffers containing now obsolete blocks must be invalidated.

2. **Flushing Dirty Buffers:** In order to write back modifications or new data to file containers, the memfs subsystem needs to iterate over the respective dirty buffers.

3. **Retrieving Blocks for Backup:** File system backup as discussed in Chapter 6 needs to determine which blocks in file containers are newer than a previous backup. VPFS enables this comparison by maintaining per-block version numbers in the block tree; they are stored next to the MACs.

All three use cases for block iterators differ in subtle, but important ways. To find all buffers that need to be invalidated, the memfs subsystem needs to locate cached blocks within a certain range. An iterator for finding dirty buffers could use the range criterion as well, but it may not be efficient in all cases (e.g., if many blocks are cached, but only few have been modified). VPFS can exploit the order in which the MACs must be updated to achieve high efficiency: the dirty states could be tracked such that a "clean" parent node is guaranteed to have no dirty child blocks cached. Entire subtrees below clean internal nodes can be skipped this way. Figure 3.4b illustrates how VPFS Core tracks dirty states for buffers.

The third use case looks much like the second. Assume that, due to MAC update dependencies, the version number of an internal node is always the same or newer than the version of any child node below the parent. Then a VPFS backup routine can skip entire subtrees of blocks, if an internal node is older than the backup. The iterators for the second and third use case are structurally identical. However, there is key difference in the type of metadata that determines which blocks to return and which to ignore: the iterator for finding dirty buffers uses *buffer attributes* held entirely in memory, whereas the backup use case relies on *block properties* that may become accessible only after retrieving additional blocks from file containers.

The ability to iterate over all blocks that need backup is essential for reaching the protection goal of recoverability. Therefore, VPFS Core's memfs must provide or enable efficient retrieval of block tree nodes based on their properties. Other use cases benefit from property-driven block retrieval as well: in Section 3.3.3, I will describe an efficient inode allocator based on that mechanism.

From these considerations follows that finding blocks based on buffer attributes and based on block properties is equally important. In order to keep the TCB small, it should be possible to construct generic iterators and other algorithms that work on both types of metadata. This goal requires integration with the buffer registry.

3.3.2 Buffer Registry

File-system buffer caches commonly rely on tree-based lookup algorithms to enable efficient lookup of cached blocks. However, implementations of these algorithms are complex and comprise hundreds or even thousands of lines of code. For example, the Radix-tree implementation that Linux 3.5 uses to keep track of blocks in the page cache comprises 1,027 lines of C code. Predecessors [155] to and early versions [156] of VPFS relied on a similarly complex AVL tree for buffer lookup. The inclusion of the roughly 800 lines of code in the file system TCB was justified for that earlier work, because the AVL tree met the only two requirements: (1) efficient lookup of specific buffers, and (2) a lookup function suitable for iterating over buffers based on block keys with the possibility to skip block ranges. Furthermore, the same AVL tree implementation was already part of the TCB of any program running on the underlying research operating system (OS). However, this argument is based on assumptions about a specific execution environment.

The Case For an Integrated Approach. Unfortunately, the AVL-tree implementation – but also other off-the-shelf implementations of in-memory balanced search trees – is not suitable to meet the full set of requirements for VPFS. In particular, stock implementations do not map to search problems where the dataset does not fit entirely in memory, as is the case for searching and iterating

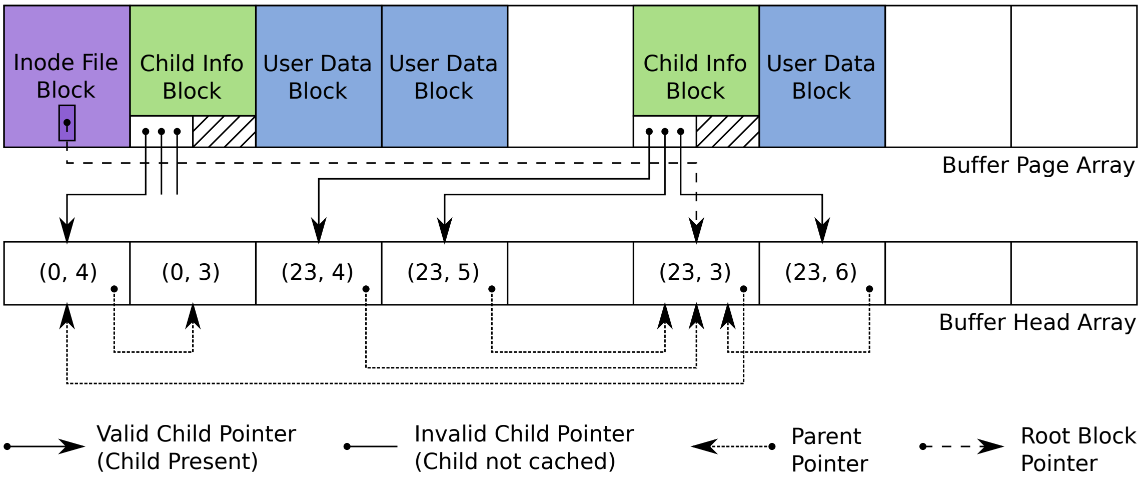


Figure 3.5: Pointer structure of the buffer registry: Internal nodes of the block tree (child info blocks) contain an array of child pointers specifying the buffer heads of all currently cached child blocks. Each buffer head contains a `(file_id, block_no)` tuple identifying the block and a back pointer to the buffer that holds the block's parent node. The root nodes of per-file block trees can be reached via a root node pointer in the file's inode.

based on block properties. I implemented the required functionality for the use cases I discussed in Section 3.3.1 next to buffer-based operations. However, the results were unsatisfactory with regard to TCB size and complexity: the routines for both buffer-based and block-based operations comprised several hundred lines of code each, yet they were solving similar problems. I therefore concluded that an integrated approach would enable a smaller and less complex TCB.

Child Pointers. VPFS Core's buffer registry exploits the existing tree structure of the block tree to find blocks in its cache buffers efficiently. For each buffer holding an internal node, the buffer registry maintains an array of *child pointers*. The number of pointers in such an array is equal to the maximum number of child nodes that a block tree node can have. Figure 3.5 shows how these pointers link buffers containing internal nodes to their child nodes. Each element in a child-pointer array will point only to one specific child block, as determined by the block tree structure. A valid pointer specifies the buffer that holds the corresponding child node, a pointer containing an invalid address indicates that the block is not cached. Child-pointer arrays are stored directly within cache buffers: when an internal node is cached, the array occupies a reserved area inside the block that is not used for MACs or backup version numbers. VPFS Core makes sure that the array in the reserved area is initialized with invalid pointers, whenever it put a non-leaf block in a newly allocated buffer. The initialization overhead is negligible compared to the costs of retrieving a block from a file container. Storing child pointers directly in internal-node buffers uses memory efficiently.

Parent Pointers. VPFS Core's buffer registry also maintains *parent pointers* for each buffer, as they enable the two main subsystems to find parent nodes quickly. For example, the persistency layer must know the parent's buffer in order to update MACs; the memfs subsystem frequently needs to propagate dirty states of buffers along branches of the block tree. In contrast to child pointers, there is only one parent pointer per buffer and this pointer is always required. Therefore, parent pointers are stored in the buffer head, which also contains metadata such as a `(file_id, block_no)` tuple describing the block's identity and dirty flags (see Figure 3.5).

Descending the Block Tree. Figure 3.5 also illustrates how the root node of a regular file's block tree is linked to the file's inode, which in turn resides in a leaf-level block of the inode file.

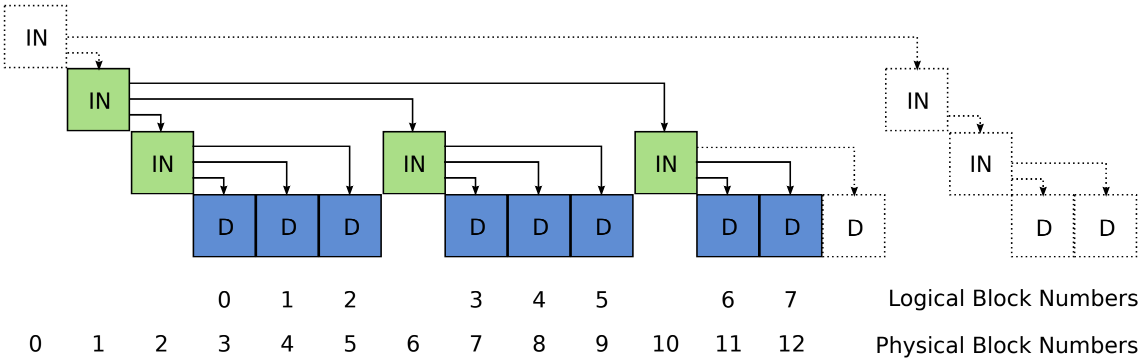


Figure 3.6: Addressing scheme for block tree nodes: Leaf nodes have logical block numbers derived from application-visible file offsets. Internal nodes do not have logical block numbers, but their structure determines how physical block numbers are assigned. Blocks drawn with dotted lines show how the block tree grows to its maximum height to accommodate more user data in leaf-level blocks. The physical block number of the root node depends on the height of the tree.

VPFS Core's memfs ensures that, if the root node of a per-file block tree is cached, it is always reachable via the *root node pointer* in the corresponding inode. Thus, starting at a file's inode, the lookup routine of the buffer registry can descend in the child-pointer tree to find any block of a file as long as that block is cached in a buffer.

Navigating in the block tree is straightforward. The following example explains the basic idea: assume that blocks are 4 KB in size. VPFS addresses leaf nodes using *logical block numbers* as shown in Figure 3.6. The application issues a read request for 1 KB at an offset of 21 KB. Thus, the application's offset at 21 KB translates into logical block number 5. At any level $l > 1$ in the tree, the lookup routine can determine the index of the correct child pointer i_l based on the logical block number b using the following integer computation:

$$i_l = \left\lfloor \frac{b \, mod \, N^l}{N^{l-1}} \right\rfloor$$

In the example, $N = 3$ is the maximum number of child nodes and the file's block tree has a depth of 3 (see colored nodes in the figure). The root node is then at level 2, whereas leaves are always at level 0. To lookup logical block number 5, the lookup routine computes index i_2 into the root node's child-pointer array and index i_1 for the first-level parent as follows:

$$i_2 = \left\lfloor \frac{5 \, mod \, 3^2}{3^1} \right\rfloor = 1; \qquad i_1 = \left\lfloor \frac{5 \, mod \, 3^1}{3^0} \right\rfloor = 2$$

Note that a table containing powers of N can be precomputed.

3.3.3 Generic Tree Walker and Specific Strategies

The buffer lookup algorithm I outlined in the previous subsection navigates through the block tree along the volatile root node pointer and the child pointers assigned to internal nodes. There are three key decision points in the routine that cause the algorithm to behave like a lookup routine for buffers containing user data:

1. **Index Selection:** In order to find the buffer containing a specific block, the algorithm choses the child pointer to follow based on a computation it performs using the logical block number and the current level in the tree.

2. **Handling of Uncached Blocks:** If a block is not cached, the lookup routine aborts. This behavior is intended for a cache lookups.

```
class Dirty_block_strategy  {
public:
    // never load uncached blocks, they cannot be dirty by definition
    bool fail_on_uncached() const { return true; }

    // only match dirty blocks that have no dirty child nodes cached
    bool is_match(Buffer *buf) const {
        return buf->is_dirty() && buf->dirty_refs() == 0;
    }

    // find first buffer with modified child node
    index_t determine_child_index(Buffer *buf, index_t level) const {
        for (index_t idx = 0; idx < Max_child_nodes; idx++) {
            Buffer *child = buf->child(idx);
            if (child && child->is_dirty())
                return idx;
        }
        return Invalid_index;
    }
};
```

Figure 3.7: C++ implementation of the dirty-block strategy that enables the construction of a buffer flush routine with O(N · log(N)) complexity.

3. **Node Types to Return:** In order to handle read and write requests from applications, VPFS Core must to retrieve leaf nodes. Internal nodes do not contain any data, so they are of no use for that specific use case.

It is easy to see that the algorithm can be extended to a block retrieval routine just by changing the behavior in the second decision point. If the root node pointer or a child pointer is invalid, VPFS Core can continue descending the tree immediately after it retrieved the missing node from the file container. Block retrieval includes setting the child pointer in the parent, which is either the inode or an internal node.

Note that the behavior of the retrieval operation could also be modified such that it creates a fresh block if no previously created version exists in the file container. These two extensions are all that is necessary to implement block retrieval with automatic resolution of read and write dependencies. VPFS Core implements exactly this solution, thereby unifying buffer lookup and node retrieval.

Strategies for Iterators. Additional use cases, including those discussed in Section 3.3.1, can also be implemented by changing the behavior in the aforementioned decision points. To enable TCB-efficient implementations of buffer lookup and block retrieval for different problems, I implemented several C++ strategy classes that a generic implementation of the tree-descent function calls into at the decision points. Figure 3.7 shows the C++ implementation of the dirty-block strategy, which the memfs subsystem uses to implement an iterator for finding all dirty buffers that belong to a file. The iterator repeatedly invokes the tree-descent function with the dirty-block strategy and flushes each returned buffer. It stops when the strategy object does not find any more dirty buffers; all blocks of the file are clean at this point.

VPFS Core implements four additional strategies for retrieving blocks and iterating over tree nodes. The strategy for the backup use case is by far the most complex: it comprises 30 lines of code. Algorithm 3.1 shows the C++ code for descending the tree in order to give an impression of the complexity of the solution's core functionality.

Algorithm 3.1 C++ code of the complete block tree descent function: To improve readability over the original C++ source code, computation of logical and physical block numbers has been omitted (two function calls in lines 31 and 32 act as placeholders).

```
 1  Buffer *descend_block_tree(Inode &inode, Strategy &strategy) {
 2
 3      // 1) prepare descent in tree, starting at root node
 4      Child_info *ci    = inode.child_info();
 5      index_t     level = inode.root_level();
 6      block_no_t  pbn   = phys_root_block_no(level);
 7      block_no_t  lbn   = 0;
 8
 9      // 2) locate root node of the file's block tree
10      Buffer *buf = buffer_ref(inode.root_node_pointer());
11      if (buf == Not_found) {
12          if (strategy.fail_on_uncached())
13              return Not_found;
14          buf = retrieve_phys_block(pbn, ci, level, 0);
15          inode.set_root_node_pointer(buf);
16      }
17
18      // 3) descend in block tree
19      while (level > 0) {
20          // 'buf' points to a parent node; check, if the strategy
21          // tells us to give this non-leaf node to the caller
22          if (strategy.is_match(buf, level))
23              return buf;
24
25          // find child node according to the caller's strategy
26          index_t idx = strategy.determine_child_index(buf, level);
27          if (idx == Invalid_index)
28              return Not_found;
29
30          // increment block numbers to positions of selected child
31          pbn += compute_physical_block_delta(idx, level);
32          lbn += compute_logical_block_delta(idx, level);
33
34          // switch to selected Child_info and get corresponding block
35          ci  = buf->child_info(idx);
36          buf = buf->child(idx);
37          if (buf == Not_found) {
38              if (strategy.fail_on_uncached())
39                  return Not_found;
40              buf = retrieve_phys_block(pbn, ci, level - 1, idx);
41          }
42
43          level--;
44      }
45
46      return strategy.is_match(buf, level) ? buf : Not_found;
47  }
```

A Strategy for an Inode Allocator. Another strategy enables the construction of a generic object allocator. A helper class in the memfs implementation uses this allocator to manage inodes in the inode file. A basic version of this helper comprises just 45 lines of C++ code and provides an array view on top of the generic file abstraction. Each array element contains one inode and the implementation automatically grows and shrinks the underlying inode file. An extended version of the helper class adds support for the object allocator, which has O(log(N)) complexity. The allocator strategy tracks the usage of array elements using a hierarchical bitmap stored in internal nodes of the inode file's block tree. The per-node bitmaps are persistent and reside in user-reserved areas of the internal nodes (32 bytes are set aside for user data). The total cost for allocating and deallocating objects in terms of code size is 102 lines of C++ code, including the allocator strategy and code to update and search the bitmap hierarchy.

3.4 Optimizations

I conclude the description of VPFS's basic design and implementation with a discussion of performance optimizations. These optimizations add little complexity to the TCB, but they improve performance significantly.

3.4.1 Caching Buffer Pointers

The lookup routine of the buffer registry implementation has O(log(N)) complexity and it always needs to descend the block tree, starting with the root node pointer. Unfortunately, the `get_next_object()` method of the previously described object-abstraction class, must perform a buffer lookup on every invocation. A filename lookup routine for a directory implementation I will describe in Chapter 4 needs to call this method in a loop.

Leaf-Node Cache. To maximize lookup performance for such access patterns, memfs maintains a *lookup cache* for each open file. The cache keeps a pointer to the buffer head of the least recently used leaf-level block. The quick-lookup routine provided by the cache can trivially determine the validity of the memorized pointer. If the pointer is valid, the lookup cache returns the pointer immediately without the buffer registry having to descend the block tree. Using the parent pointer in the memorized buffer head, the lookup cache also has immediate access to all child pointers in the parent node. The child pointers enable O(1) lookups of any sibling of the least recently used block. For workload traces captured from both synthetic benchmarks and multimedia applications, the lookup cache showed a hit rate close to one hundred percent. The implementation of this highly effective optimization contributes only 27 lines of C++ code to the code base of VPFS Core.

Iterator Acceleration. The iterator for invalidating obsolete buffers loops over child pointers in first-level internal nodes in a similar way. I extended the iterator with 5 lines of code in order to remove buffer lookups via the buffer registry from the tight loop.

3.4.2 Optimal Block Tree Depth

For each file container, there is an optimal depth for the block tree that correlates with the amount of the data stored in leaf nodes. A 10 GB file requires more tree levels than a file whose content fits into one block. As an application may cause a file to grow or shrink, VPFS must adapt the depth of the tree accordingly. VPFS Core uses the same block addressing scheme as earlier work [127, 155] to make sure that physical block numbers of existing blocks do not change when the height of the tree changes. In this scheme, the first leaf block containing user data (with logical block number 0) always has the same physical block number; its parent nodes are assigned block physical numbers in decreasing order, as shown in Figure 3.6 on page 44. The functionality to adapt the tree height to the current size of the file comprises approximately one hundred lines of code [156].

3.4.3 Write Batching and Read Ahead

The separation of the VPFS stack into trusted and untrusted components comes at a cost that is inherent to microkernel-based systems using hardware address spaces for isolation: in order to cooperate, the trusted VPFS Core and the untrusted helper must exchange messages. VPFS optimizes the overhead for data transfers by using a shared-memory region. However, both components still need to transmit control messages. VPFS transfers cryptographically protected blocks in batches, thereby reducing the relative overhead of inter-process communication (IPC) per block.

Write Batching. The write-buffer array in the shared-memory region is large enough to hold an internal node and all its child nodes. When flushing dirty blocks from large files, VPFS Core's persistency layer can buffer and then submit using one IPC an internal node and all its leaf blocks. The untrusted helper is then able to write back all blocks consecutively to the commodity file system using one `writev()` system call. Blocks belonging to multiple file containers require multiple `writev()` operations, but still benefit from batching.

Read Ahead. For each file container, the untrusted helper maintains a history of previous read accesses. If it detects a sequential access pattern, the helper exponentially increases the value of the size parameter for the `read()` system call. Thus, it can place multiple blocks into the read-buffer array of the shared-memory area. Reading 32 blocks in one operation is the sweet spot for the Linux-based prototype of the untrusted helper; this value amounts to half of the default read-ahead size of the Linux kernel. Impact on the TCB's complexity is minimal: before submitting a read request, VPFS Core's persistency layer optimistically searches the read-descriptor array in the shared-memory area to check if the block has already been loaded. If it finds the block, the trusted component does not perform any IPC, but consumes the pre-loaded block immediately.

3.5 Summary

By exploiting the static structure of the block tree, VPFS Core's buffer registry is simple to implement. The implementation is smaller and arguably less complex than the AVL-tree based solution it replaced: the buffer registry consists of approximately 230 lines of code, whereas the AVL tree and cache-specific routines amount to nearly 1,000 lines of code. The second advantage is that the core logic of the lookup routine is generic and enables the construction of at least six different iterators and block retrieval functions that can operate on buffer attributes or block properties. This functionality contributes about 150 lines of code. Exposing the child-pointer array introduced for the buffer registry proved beneficial to enable simple optimizations.

A key lesson I learned while building VPFS Core was that TCB minimization requires specialized solutions. Using off-the-shelf implementations is possible and can deliver results quickly, but the amount of code and complexity added to the TCB is comparatively large.

In the next chapter, I will discuss how VPFS protects its metadata structures, including filenames and the directory hierarchy.

Chapter 4

Metadata and Namespace Protection

In this chapter, I will describe approaches to protecting confidentiality and integrity of the file system namespace. I thoroughly discuss to which extent untrusted infrastructure can be reused in order to implement secure naming and lookup, and where security requirements impose limitations on reuse. I start out with related work in the area of metadata protection in file systems and then present three approaches to naming in VPFS.

4.1 Protection of Metadata in File Systems

File systems that operate on untrusted local storage must protect confidentiality and integrity of metadata through cryptographic means. They need to enforce these protection goals for per-file characteristics (e.g., the size or type of a file) and the file system namespace (e.g., filenames and directory structures). I focus the discussion on systems that need to protect individual files and are designed to ensure integrity beyond file contents.

4.1.1 Approaches to Namespace Protection and Freshness

Self-Certifying Pathnames. The Self-certifying File System (SFS) [129] introduced the concept of self-certifying pathnames for a globally distributed file namespace. In SFS, all pathnames include a collision-resistant hash called *HostID* that is computed over a location and a public key. The location can be an IP address or DNS name of a server; the public key is used to verify identity of the server. When the client resolves a self-certifying pathname (e.g., `/sfs/example.com:HostID/some/file`), it connects to the server and checks that the remote machine is the one specified in HostID. To do so, the client verifies that the server is in possession of the private key, whose public counterpart is specified in HostID. Unfortunately, the client must still trust each server to ensure integrity of all files and directories that it is hosting. Only a read-only variant called SFSRO [101] provides cryptographic integrity verification of file system contents.

Hierarchical Namespace Protection. EncFS [30] tightly couples namespace encryption with integrity enforcement. It layers on top of an unprotected file system and encrypts file and directory names using a symmetric cipher in cipher-block-chaining (CBC) mode. In its most secure configuration, the initialization vectors (IVs) required for encrypting and decrypting depend on the chain of unique IVs used to encrypt the names of all parent directories. This encryption scheme ensures that identical filenames in different directories are encrypted to different ciphertexts. EncFS can verify the correctness of pathname resolution based on the aforementioned IV chaining approach and message-authentication codes (MACs) that it computes over file contents. Assume that an attacker moved an encrypted file to another location in the directory tree. Then EncFS will not be able to compute the encrypted filename of the moved file and lookup fails. If the attacker replaces just the content of the file, but not encrypted name, EncFS will be able to open it, but the decrypted data fails the integrity check: because the IV used to decrypt the file does not match the IV used during encryption under the original pathname, the MACs do not match the content.

The principle behind EncFS's lookup verification is similar to host verification in SFS. However, when applied to a local file system, the weakness of the approach becomes apparent: an attacker can still compromise freshness by replacing an encrypted file in the EncFS backing store with an older version, or any other file that ever existed under the same pathname. The system cannot detect if files are up-to-date, nor can it determine whether a legitimate user or an attacker deleted a file.

File Lists and Freshness Information. To detect both rollback and deletion attacks on untrusted storage, the file system requires trustworthy information about the existence of a file – or a specific version of it. For example, the distributed file system SiRiUS-U, which is an extended variant of SiRiUS [104], keeps track of all files stored on untrusted servers by maintaining a Merkle hash tree over metadata files. The metadata files describe which files exist in which directory and they also include a reference to the latest version of each file in the corresponding directory entry. The system essentially implements a complete hierarchical namespace. An attacker cannot manipulate individual parts of it, because the root of the Merkle tree is signed by the owner of the file hierarchy. He can however replace the entire file system with a copy signed by himself. SiRiUS and SiRiUS-U do not encrypt file and directory names, but they could like EncFS does.

Byzantine Storage. Finally, there are distributed file systems that use Byzantine state replication to ensure namespace protection. For example, a secure directory service [97] built for Farsite [85] enforces access control for readers and writers. It further ensures that directories do not contain duplicate entries (despite case-insensitive naming and encryption) or entries with illegal characters. However, the security guarantees of the system are based on the concept of Byzantine fault-tolerance, which requires multiple servers.

Byzantine fault tolerance approaches [94, 126] are impracticable for securely using untrusted storage in the mobile devices that VPFS targets. However, the self-validating lookup schemes as used by SFS and EncFS are highly useful and VPFS can build upon them. Encryption of filenames seems straightforward, however, both EncFS and a hypothetical version of SiRiUS that encrypts names reveal the original directory structure. VPFS improves on them by organizing its cryptographically protected file containers in a flat namespace (e.g., stored in a single directory of the untrusted commodity file system). The necessity of trustworthy information about the namespace structure is fundamental, but leaves room for differently complex implementations.

4.1.2 Requirements for VPFS

As I explained in the previous chapter, VPFS stores traditional per-file metadata such as file sizes in inodes. VPFS Core protects inodes using MACs embedded in the block tree; a MAC describing the root of the block tree is protected by sealed memory. The block tree prevents successful rollback attacks by construction. The cryptographic protection also prevents an attacker from directly modifying per-file metadata without being detected. As VPFS Core encrypts its inode table, he cannot determine exact file sizes or gain access to other information stored in inodes.

Confidentiality vs Untrusted Storage. VPFS must not use plaintext names for file containers or reveal the directory structure. However, even then certain information may remain visible: an attacker who compromised the untrusted parts of the VPFS stack can determine the number file containers and how many blocks each of them contains. When a VPFS instance is active, he may also be able to observe access patterns.

Integrity of the Namespace. Integrity as defined in Section 2.4 includes that applications receive exactly the data or metadata they requested, not just any data or metadata. This requirement implies that an attacker must not be able to confuse the trusted component of VPFS such that it misdirects reads or writes, as this would compromise integrity. An intuitive interpretation of integrity for the namespace can be summarized as follows: the application shall always see and operate on the most recent state that resulted from its its own actions. An attacker must not be able to manipulate the file system namespace without VPFS Core noticing the modifcations. This intuitive definition of integrity translates into the following requirements, which are equivalent to definitions from earlier work [155, 156]; I use the term “file” to refer to any object in the file system namespace:

1. **No Silent Creation:** No file appears in a location in the namespace, unless the application created it under that specific name or renamed an existing file to that new location.

2. **No Silent Deletion:** No file disappears from its current location in the namespace, unless the application explicitly requested deletion or renamed the file. Operations that create a new file or rename an existing one must not cause silent overwrites, unless removal of an existing file with the same name is expected behavior (e.g., as specified for `rename()` on POSIX-compliant systems).

3. **Correct Lookup:** When the application accesses a file using a specific name, VPFS carries out the requested operation on the correct file (i.e., VPFS Core will not read data from any file other than the one named by the application).

4. **Correct Enumeration:** If the application enumerates the namespace or parts of it (e.g., a single directory), then the resulting list of files is correct, complete, and up to date.

Based on this list of requirements, I will now discuss different approaches to implement naming in VPFS. I start with a minimalist flat naming scheme and then discuss solutions suitable for general-purpose file systems.

4.2 Inode-Based Naming

The most simple solution to naming files is to refer to them by number. In VPFS, an index into the inode table is a unique identifier for any user-accessible file in the file system.

Design and Implementation. VPFS Core’s memfs implements inode-based naming and an extended persistency layer described in Chapter 5 uses this interface during file system recovery after an unclean shutdown. The implementation builds on the inode allocator described in Section 3.3.3. This allocator optionally accepts an inode number as input, so the aforementioned recovery code can chose a “name”. Attempts to allocate an existing inode location fail with an `EEXIST` error. The interface for inode-based naming is not exposed to applications, but doing so is straightforward.

Security Properties. The inode file is protected by the block tree. Therefore, inode-based naming trivially meets all of the integrity requirements defined in the previous section. As the scheme exposes a flat namespace, no directory structure is revealed in the untrusted storage. To conceal any correlation in the choice of inode indexes, VPFS Core’s persistency layer uses one-way encryption to generate names for the individual file containers. The persistency layer computes a collision-resistant hash over a random secret S hidden in the file system root info and a temporally unique file ID U stored in the inode:

$$name = H(S \,|\, U)$$

VPFS assigns a unique file ID U to new files simply by incrementing a counter and memorizing the value in the file's inode; the counter value is persistent across remounts. Thus, every file container gets a new unpredictable filename.

4.3 Untrusted Naming

A general-purpose file system needs to provide a more powerful naming abstraction than inode-based naming, which I expect to be suitable for only a small number of applications. The majority of applications expect a hierarchical namespace in which they can name files arbitrarily.

4.3.1 Potential of Reusing Untrusted Infrastructure

Efficiency vs Complexity. Efficient implementations of hierarchical directories in commodity file system stacks are large and complex. For example, the Ext4 implementation in Linux 3.5 dedicates almost 3,000 lines of code to directory handling and name lookup (figure determined by counting code lines in dir.c, namei.c, and hash.c). This code is complemented by more than 1,800 lines of code for the directory cache implementation in the Linux' VFS layer (figure based on lines of C code in dcache.c). I argue that this much code should not be part of the TCB of an application that uses VPFS. On the other hand, it would be interesting to see if VPFS Core can securely reuse a sophisticated directory implementation and the caching mechanisms of an untrusted commodity operating system (OS).

A Trusted Wrapper for Naming? I explored possibilities to reuse untrusted naming infrastructure through a trusted wrapper in an early version of VPFS [156].[1] The key requirement was that an attacker must not be able to compromise integrity and confidentiality of the directory tree under the VPFS threat model. Since the commodity file system stack cannot be trusted to function correctly, the wrapper in VPFS Core needs to verify the correctness of any operation that untrusted components perform on the namespace (e.g., file lookups).

The basic idea for securely reusing untrusted naming infrastructure is as follows: The trusted VPFS Core passes an encrypted pathname to the untrusted VPFS Helper. The helper then performs a lookup operation using the VFS-level APIs of the untrusted commodity file system and reports the results back to VPFS Core. The trusted component verifies that the lookup operation is correct with respect to the namespace integrity requirements defined in Section 4.1.2 on the preceding page: (1) no silent creation, (2) no silent deletion, (3) correct lookup, and (4) correct enumeration of directory contents.

Trustworthy Metadata for Verification. VPFS Core requires trustworthy information in order to assess whether the operations that untrusted components carried out are correct. Verification of the first three integrity properties requires only small extensions to existing metadata structures and the trusted code base. VPFS Core already maintains an up-to-date list of existing files that is encoded in the inode file; two additional metadata items in inodes enable VPFS Core to check file existence or non-existence efficiently. Unfortunately, the fourth integrity requirement means that VPFS Core would need to have an understanding of what a directory is and which files it contains. Keeping this kind of functionality and the associated data structures out of the TCB is exactly what untrusted naming is supposed to enable. A VPFS version that reuses untrusted naming infrastructure therefore cannot support enumeration of directory contents. This is a functional limitation that might be acceptable for many applications.

[1] In the original publication [156], the description of untrusted naming and its limitations used the terms "VPFS server" and "VPFS proxy" to refer to VPFS Core and VPFS Helper, respectively. It also referred to inodes as "master entries". Sections 4.3.2 and 4.3.3 of this thesis include an updated and streamlined version of the original description that uses the same terminology as the rest of this document.

The following section explains *untrusted lookup* and the concept of *proofs*. I discuss the limitations of the approach afterwards.

4.3.2 Untrusted Lookup and Proofs

The functionality that enables the namespace integrity checks is split between VPFS Core and the untrusted VPFS Helper. The trusted component maintains low-complexity metadata structures in inodes, which are protected by the block tree such that any information stored in them is trustworthy.

Path Hashes. The first element I added to each inode is the *path hash*. A path hash is a collision-resistant hash of a global secret S and the full pathname P of a file:

$$pathhash = H(S \mid P)$$

Path hashes enable VPFS Core to represent variable-length pathnames such as `/path/to/file` as short fixed-size byte strings. A file's path hash also serves as the encrypted name of the corresponding file container. Note that the hashed path completely hides the directory structure of the virtual private file system. To open an existing file, VPFS Core passes the path hash to the untrusted helper, which then uses a BASE64 representation of the hash to look up the file container in the untrusted file system.

Inode Hints. When it successfully opened the specified file container, VPFS Helper reports back to the trusted component the index of the inode that belongs to the requested file. The helper retrieves this index from a database that contains (`path_hash`, `inode_index`) tuples; it adds and removes tuples as part of file creation and deletion operations. VPFS Core uses the reported inode index as a *hint* to find the inode of the file that it wanted to open and compares the path hash it previously computed for lookup with the one stored in the inode; if they match, the untrusted components cooperated. A hint leading to successful verification is a *proof of correct lookup* for the requested file.

The simple authentication that VPFS Core performs is similar to the concept of self-certifying pathnames in SFS [129] and the mechanism that SiRiUS [104] uses to prevent file-swapping attacks. The implementation in VPFS Core is TCB efficient, as it largly depends on functionality that is already part of VPFS Core's TCB: the collision-resistant hash function is used to compute MACs of the block tree.

Next Pointers. The second field that I added to each inode is the *next pointer*. The next pointer in inode I_n points to the inode I_{n+1} that contains the lexicographically smallest path hash that is greater than the path hash stored I_n. Thus, next pointers link all inodes in a list sorted by their path hashes in ascending order. Note that this sort order does not reflect the order in which inodes are stored in the inode table. However, both inode hints as described in the preceding paragraph and next pointers are indexes into the inode table. VPFS Core ensures that the order of inodes in the linked list is total and it exploits this property in order to establish *proofs of existence* and *non-existence*. Such a proof is a hint to inspect inode I_n, which would sort directly before the inode of file F, if F had indeed been created. By evaluating the path hashes stored in both I_n and the inode I_{n+1} that I_n links to, VPFS Core can easily verify whether or not F exists.

Figure 4.1 visualizes the general idea (path hashes are represented as natural numbers for illustration purposes). Assume that an application wanted to open file F whose pathname is transformed into path hash 33; further assume that such a file does not exist and VPFS Helper needed to report a lookup failure. In this case the helper would give a hint to inspect inode I_n at index 5, which has path hash 23. Because I_n's successor in the sorted inode list contains path hash 42, there can be no file F that maps to path hash 33. VPFS Core can verify that the reported lookup failure is correct. If there is an inode I_{n+1} with path hash 33 linked from I_n, then the hint to I_n would proof that F is supposed to exist and the untrusted helper must have lied. The

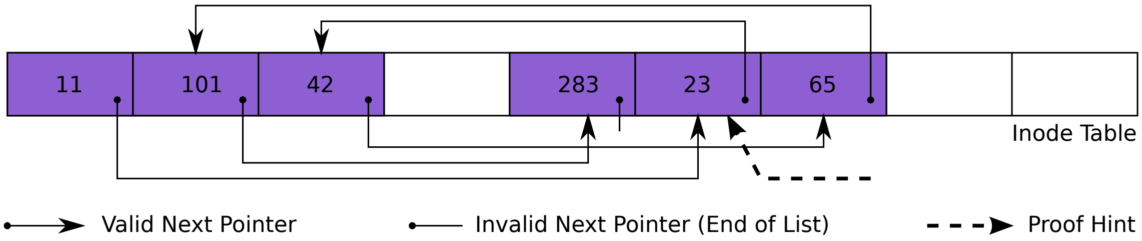


Figure 4.1: Inodes form a linked list sorted by their path hashes. The strict sort order allows the VPFS Core to determine the existence or non-existence of a file by examining only inodes. The sort order is independent of the order in which inodes are stored in the inode table. For readability, path hashes are represented as natural numbers.

untrusted helper is also identified as uncooperative, if it hints at two linked inodes whose path hashes do not bracket the path hash used for lookup; there are special cases for the first and last inode in the list, but they are trivial to handle.

Proofs for All Operations. The untrusted VPFS Helper provides proof hints for all operations that take an encrypted pathname as argument (i.e, `open()`, `create()`, `rename()`, and `delete()` operations). However, there are two special cases: first, in case of file creation, VPFS Core additionally tells the untrusted helper which newly allocated inode is linked to the path hash of the new file. Second, when opening a file as explained at the beginning of this section, VPFS Helper reports the inode belonging to the requested file container instead of the inode that would sort before. Updates to the list (i.e., insertions and deletions) are performed at the position that VPFS Helper hints at, after VPFS Core checked that the strict ordering is not violated and that path hashes in the two linked inodes bracket the path hash derived from the application-provided pathname. The VDisk [158] system used a similar approach also based on bracketing for validating correctness of untrusted garbage collection.

Database Support in VPFS Helper. The untrusted part of the prototype implementation [156] used the embeddable database Berkeley DB [43] to manage the `(path_hash, inode_index)` tuples. The tuples were sorted by path hashes, such that VPFS Helper could efficiently find the predecessor of a specific inode.

Security Properties. If VPFS Core keeps the inode table and all next pointers consistent, it is easy to see that the untrusted lookup approach meets the integrity requirements that I defined in Section 4.1.2: (1) silent creation is not possible, because there is no matching inode that VPFS Helper could hint at; (2) silent deletion is not possible, because the untrusted helper can only prove non-existence, or it lies and the wrong proof is detected; because VPFS Core checks the proof of non-existence for the target path during `create()` and `rename()` operations, it cannot be tricked into overwriting data unwillingly; finally, (3) the evaluation of path hashes stored in inodes guarantees that always the correct file is looked up.

4.3.3 Limitations of Untrusted Naming

At the end of Section 4.3.1, I pointed out that VPFS with untrusted lookup cannot provide the concept of directories. I shall now discuss this problem and related issues in greater detail and explain why I rejected possible solutions.

Hierarchical Namespace. Despite the lack of explicit directory support, path hashes and the way how they are computed enable applications to lay out files in a hierarchical name space: for example, an application can create a file under the pathname `/dir/subdir/file`. Applications that merely use the file system as a key-value store are compatible with the untrusted lookup approach; they do not enumerate directory contents. An example might be an e-mail application that stores many files in maildir folders, but maintains index files that provide a comprehensive list of all individual files in those folders.

Moving Directory Trees. General-purpose file systems usually offer the ability to move whole directory trees in a single operation by renaming the root directory of a sub tree. Unfortunately, VPFS with untrusted lookup cannot support this feature due to fundamental limitations:

- **Weaker Confidentiality:** To support an untrusted `rename()` operation on directories, VPFS Core would need to reveal the directory structure of the virtual private file system; it could no longer hide it in a flat name space. The reason for this requirement is that the untrusted VPFS Helper must be able to organize file containers in subdirectories, otherwise it could not move a directory and its contents. This design weakens confidentiality even further: each file or directory name must be encrypted individually (i.e., independent of the names of parent directories). Thus, two files with the same name, but stored in different directories, have the same encrypted name. If they had not, VPFS Helper could not look up any file container below a directory that has been moved. This problem could be solved by introducing per-directory salts. But then VPFS Core would need to be involved in the lookup of each path element, thereby neutralizing the benefits of untrusted lookup.

- **Increased Overhead:** Moving a directory invalidates the path hashes for all files stored below that directory. Revalidation of path hashes is possible, but it incurs additional run-time costs and adds more complexity to the TCB. To ensure that the integrity properties defined in Section 4.1.2 on page 51 hold under all corner cases, VPFS Core would have to revalidate the path hashes of all files in the file system, not just those that have been moved [155].

VPFS with untrusted naming therefore cannot rename directories, but applications can rename files individually.

Hard Links. Due to the use of path hashes that are associated with file containers, hard links cannot be supported. SiRiUS has the same limitation [104]. Ordinary applications use hard links rarely, or not at all, which might be a reason why Microsoft's new file system ReFS [151] does not support this feature. Soft or symbolic links are compatible with untrusted naming, but VPFS Core must issue another untrusted lookup for the pathname stored in the link.

4.3.4 Verdict on Untrusted Naming

The implementation of untrusted lookup and proof validation in VPFS Core required 180 lines of code [156]. This amount of code is more than an order of magnitude smaller than naming subsystems in commodity file system stacks. I therefore consider untrusted lookup a viable and secure option for providing naming in VPFS, if applications do not need to enumerate directories. Nevertheless, it cannot be denied that many applications need a more complete naming solution that includes the ability to read directory contents. I therefore implemented a simple directory-based naming scheme in the same prototype next to untrusted lookup. Integrated directory support in VPFS Core contributed 350 lines of code to the TCB. The implementation included a simple lookup cache for recently opened files. This lookup cache enabled the directory-based prototype to achieve performance comparable to the untrusted lookup solution in benchmarks with both hot and cold buffer caches. The performance numbers will be put into context with the final implementation of VPFS in Chapter 7.

The small difference in code size – 350 instead of 180 lines of code in the early prototype – and the inherent limitations uncovered during the untrusted naming experiment lead to the conclusion that a general-purpose implementation of VPFS should include directory support in the TCB.

4.4 Trusted Directory Hierarchy

The current version of VPFS implements directories as an array of *dentries* layered on top of the same object array abstraction that VPFS Core's memfs uses for the inode table. There are two types of dentries: one for regular files and one for subdirectories. Thus, directory hierarchies are supported. Each directory resides in a special file in the virtual private file system; each such directory file is backed a file container in the untrusted commodity file system in the same way as regular files or the inode file. A dentry in a directory maps a filename to an inode number, which VPFS Core's memfs can use to retrieve file contents as described in Section 3.3.3.

Design and Implementation. VPFS Core's memfs stores dentries unsorted, name lookups therefore require a linear search. This minimalist implementation does not scale for large numbers of dentries in a single directory, but it contributes only 216 lines of code to the TCB. To speed up lookup for frequently accessed files, VPFS Core releases internal file descriptors lazily; a lookup cache enables quick resolution of full pathnames for any file whose file descriptor has not been released yet. This simple optimization performs surprisingly well for various workload traces discussed in Chapter 7. The implementation of the lookup cache adds 64 lines of code to the TCB and builds on a hash table implementation, which contributes another 36 lines of C++ code.

Security Properties. The directory structure is hidden from untrusted components: VPFS Core names file containers using the same encryption scheme used for inode-based naming described in Section 4.2. Because each directory file is part of the global block tree, VPFS trivially meets all integrity requirements for the namespace.

Chapter 5

Consistency and Robustness

VPFS layers its most important metadata structure – the block tree protecting all data and metadata – on top of an untrusted commodity file system. The metadata encoded in the block tree must always be consistent and in sync with the on-disk structures of the reused file system, including the encrypted blocks stored in file containers. An unclean shutdown (e.g., when the battery of the mobile device depletes unexpectedly) may result in inconsistencies that the underlying commodity file system cannot detect and repair on its own. A partially updated Merkle tree makes verifying integrity of severed subtrees impossible, which results in data loss if security guarantees shall not be weakened. To be robust against crashes, VPFS must ensure that the composition of trusted and untrusted layers of metadata is consistent even after a crash or power loss. However, file system consistency mechanisms are complex and difficult to get right [87, 88, 106, 160]. Therefore, it is desirable to keep them out of the TCB.

In this chapter, I first review existing approaches to solving the file system consistency problem. I then discuss how VPFS ensures robustness against crashes while adding minimal complexity to the TCB [157].[1]

5.1 Consistency in Local File Systems

File systems map high-level data structures such as files and namespaces onto a simpler block-level abstraction offered by the underlying persistent storage. As the size of both files and metadata structures demand much more storage than a single block can provide, file systems must distribute their on-disk data structures across multiple blocks. These blocks contain information with complex interdependencies, which require certain groups of blocks be updated either all together or not at all. In the event of a system crash or power loss, the on-disk state is likely to be only partially updated.

5.1.1 Approaches to Ensuring Consistency

There are four major approaches to solving the consistency problem in file systems:

Scan and Repair (fsck): FAT and Ext2 [93] are well-known examples for file systems that do not implement a dedicated consistency mechanism. The on-disk data structures of these file systems are updated in-place and are likely to become inconsistent, if the system crashed while write accesses were in progress. After such an unclean shutdown, a *file system checker (fsck)* scans the entire file system for inconsistencies, employing heuristics to bring the on-disk structures back into a consistent state. This recovery process may not always be successful and can result in files or directories being lost, or file contents can be restored but their location in the directory tree cannot be reconstructed. Fsck runs can take in the order of minutes to hours for large file systems.

[1] An earlier and shortened version of this chapter appeared in a separate publication [157].

Ordered Writes: The main idea behind using synchronous writes and the more efficient soft updates [102] approach is to update on-disk data structures like pointers and bitmaps in a safe order, such that only benign inconsistencies exist after an unclean shutdown. These inconsistencies can be fixed reliably by an fsck run. Copy-on-write file systems such as WAFL [114], ZFS [57], or BTRFS [2] also write blocks in a specific order: they create copies of all modified blocks of their tree-based on-disk data structures in unused space. Once all modified blocks are persistent, the entire set of changes is made active by atomically updating a single pointer. Copy-on-write file systems can recover from crashes without fsck runs.

Logging: Log-structured file systems [139] organize the storage medium as a single log, to which new data and metadata is appended. When modifying existing file system state, the data and metadata in the log is not overwritten, but instead the modifications are appended to the log. When mounting a log-structured file system, the log is scanned sequentially and complete updates are identified (e.g., based on transaction markers or checksums). By construction, the latest versions of all metadata information and data blocks represent the latest consistent file system state. Data and metadata that became obsolete because newer versions have been written are removed from the log. This garbage collection operation is performed by a cleaner process on demand or in background.

Journaling: Journaling file systems such as Ext3 [153], ReiserFS [63], or NTFS [42] combine unordered in-place updates with write-ahead logging. File system operations that require multiple blocks of the storage medium to be updated atomically first write all intended changes to a log (or *journal*) and mark them as a *transaction*. Once all intended modifications are persistent in the journal, the file system code can safely apply all updates in the transaction to their in-place locations. A transaction is *committed* once it is logged persistently; it is *checkpointed*, after all in-place updates reached persistent storage. Recovery after an unclean shutdown repeats the checkpoint phase for logged but not yet checkpointed transactions. A distinct advantage of journaling file systems over log-structured file systems is that garbage collection is simpler and limited to the journal: journal records describing checkpointed transactions are just removed from the beginning of the log. Journaling guarantees consistency of all metadata – and data, if logged as well – but each block appended to the journal needs to be written a second time.

Except for the scan-and-repair approach, all of the described solutions to the file system consistency problem must carefully write certain disk blocks in a specific order and they rely on *write barriers* to make sure the storage device does not violate per-block write-before constraints. More intelligent storage devices can relieve file system developers from this complex task.

Automatic Consistency Enforcement. Automatic consistency enforcing (ACE) disks [149] build on the concept of type-safe disks [148]. An ACE disk tracks pointer-based dependencies between blocks that the file system submits and makes consistent sets of updates durable. Thus, the storage device itself ensures consistency of on-disk structures, if the file system has been adapted to use the richer interface of ACE disks. Such a file system does no longer have to take care of the correct write order on its own. However, it may have to react to new types of "soft errors" by which an ACE disk demands that certain metadata blocks be flushed, such that the disk can commit a consistent set of changes where no pointers refer to yet unwritten blocks.

Explicit Dependencies. The authors of Featherstitch [100] went into the opposite direction by making dependencies between updates to on-disk structures explicit. Featherstitch offers the abstraction of *patches* that describe how specific portions of file system blocks shall be modified, and which other patches need to be applied to blocks beforehand. This declarative approach to file system construction frees file system developers from the tedious task of micromanaging write accesses. A particularly interesting feature of Featherstitch is that it offers applications the concept of *patch groups* to express write-before constraints at a much higher level: it enables applications to specify that writes to parts of one or more files shall not become persistent before other groups of write operations. The system enforces these constraints while exploiting opportunities for optimization.

VPFS can benefit from expressive consistency interfaces as offered by the underlying untrusted file system in order to describe update constraints for file containers; optimizations in that area will be discussed in Section 5.7. In the next section, I will evaluate which of the described approaches to file system consistency is suitable for VPFS.

5.1.2 A Consistency Paradigm for VPFS

Since the message authentication codes (MACs) have the strongest update dependencies, I focus the discussion of possible solutions on the Merkle tree that contains them. The general requirement is that the untrusted commodity file system must contain enough information to restore a consistent tree after a crash.

Incompatible Approaches. From the consistency mechanisms I described, those that require running an fsck routine after a crash are inherently incompatible with Merkle trees. This category includes systems based on unordered writes, synchronous writes, and soft updates. None of these approaches makes firm guarantees about the state of the recovered file system. For example, soft updates [131] allow in-place updates in such an order that only minor inconsistencies occur after a crash. Pointers are guaranteed to be valid, however, old and new metadata – or blocks with user data – may be mixed. This relaxation cannot work with Merkle trees, because an old parent node can never authenticate an updated child node.

Naive Approach. The naive approach to ensuring consistency of file containers is to submit consistent "patches" that contain all modified Merkle tree nodes. The update dependencies of the MACs in the tree require that any change made to leaf nodes (or internal nodes) propagate to the root in bottom-up order. Thus, when writing back modifications in a file (e.g., after appending new data), the system must always update internal nodes up to the root node. This set of updates must then be applied to file containers *atomically*. In this context, atomicity means that, after an unclean shutdown, either the complete set of updates is persistent, or the previous state is accessible – otherwise the chain of MACs in parts of the Merkle tree get severed and integrity verification is impossible. Unfortunately, atomic updates to Merkle trees are expensive, because small modifications such as writing a single block of user data involve writing as many tree nodes as there are levels in the tree. For each block that is part of the set of updates, VPFS Core must perform cryptographic operations that further increase run-time overhead and energy consumption.

The copy-on-write scheme described in the preceding section is an instantiation of the naive approach I just discussed. Also, using this scheme on top of a commodity file system API may not be efficient, if the reused file system implementation does not efficiently support atomic updates to multiple files (i.e., the file container of the inode file and one or more file containers containing user data or directory entries). Using data journaling [137] in the commodity file system could help, but this feature doubles the number of block writes to the storage device. Approaches similar to Featherstitch's patch groups or the transaction support in NTFS may be more efficient, but they are not widely available.

Journaling Tree Updates. The requirement that the on-disk Merkle tree must always be in a consistent state is the key reason for slow performance. Hence, consistency requirements for the tree need to be relaxed. In fact, it is desirable to omit updates of internal Merkle tree nodes for short periods of time, for example, during high load or to minimize latency. In order to have MACs available for post-crash integrity checking nonetheless, the system must keep them in a more efficient data structure. Write ahead logging as used in log-structured and journaling file systems solves this problem: they allow for efficient and atomic updates of distributed file data

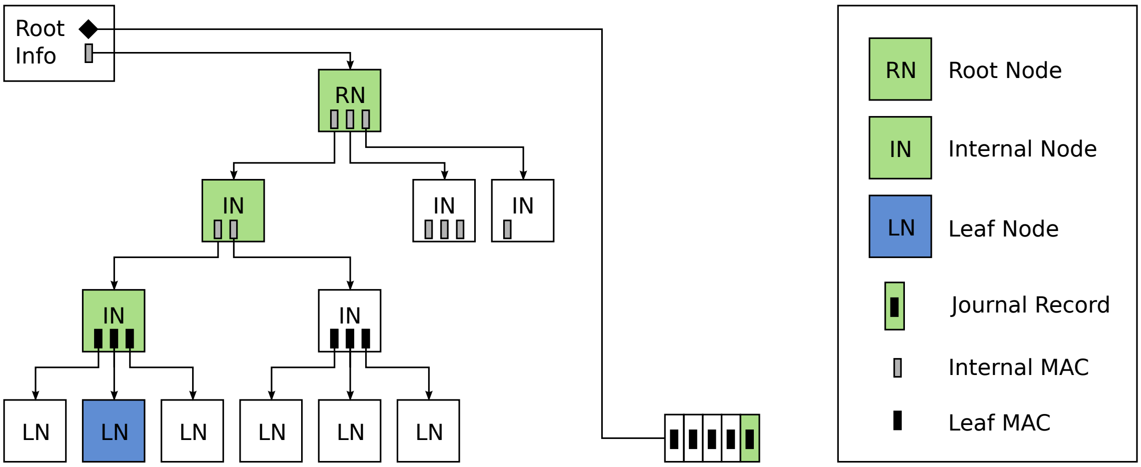


Figure 5.1: Lazy updates of the Merkle hash tree: Modification of one leaf node requires the complete chain of internal nodes up to and including the root node's hash to be updated (colored nodes are modified). A growing journal that contains updated leaf-node MACs enables lazy updates of the Merkle tree. Incremental MACs computed over the journal ensure the integrity of tree updates. (Journal records containing incremental journal MACs that authenticate preceding records are not shown in the figure.)

and metadata, which in the case of VPFS Core includes MACs that enable integrity checking. A growing journal to which MACs of updated leaf nodes are appended eliminates the need to update internal nodes of the block tree immediately. They can be flushed to file containers at a more convenient time; the respective parts of the journal may be garbage collected after the checkpointing phase. This strategy also reduces cryptographic overhead, as only the leaf nodes of the tree are updated frequently. The persistency layer of VPFS Core uses a journal to log updated per-block MACs first, then it writes the blocks themselves. Figure 5.1 illustrates the general idea.

Protecting the Journal. As the journal contains information that is critical to ensuring integrity, it must be cryptographically protected as well. To keep the performance benefit, appending records to the journal must not require additional updates of other metadata such as the root node of the Merkle tree. Therefore, VPFS Core ensures journal integrity by continuously computing a keyed hash or MAC over the journal. New records containing Merkle tree MACs are appended to the end of the journal together with a new *journal MAC* that authenticates all preceding journal content, thereby enabling incremental integrity checking. A journal MAC cannot be forged, because its computation depends on a secret key that VPFS Core stores in the file system root info, which is protected by sealed memory.

On the following pages of this chapter, I will introduce additional metadata that is logged in the journal as well; to ensure confidentiality of this metadata, VPFS Core encrypts journal records using AES-CBC. The incremental keyed hashing and encryption scheme that I just outlined is well-understood and, for example, used in the trusted database system (TDB) [127].

I will now briefly describe journaling-related extensions to the persistency interface between the trusted and untrusted VPFS components. I then discuss the details of journaling and replay in Sections 5.4 through 5.6.

5.2 Extended Persistency Interface

Compared to the basic persistency interface I described in Section 3.2.2, the journaling-enabled version of VPFS introduces two new buffers in the shared-memory area: (1) a log buffer for journal

Message type	Description
`Block_read`	Read a specific data block from a file container
`Txn_flush`	Execute buffered write operations after appending them to the journal
`Checkpoint_write`	Create new journal with root info as first record (marks new checkpoint)
`Checkpoint_read`	Read root info from latest consistent checkpoint
`Journal_read`	Read set of complete transactions from journal
`Shared_mem_init`	Set up shared memory once

Table 5.1: Complete list of message types that Sync Manager uses for communication with Txn Manager.

records to be written, and (2) a log buffer for reading journal records during mount and recovery. The interface further adds operations for handling journaling and replay. Both VPFS Core and the untrusted helper have been extended for the extended interface.

Sync Manager. Like in the basic version, VPFS Core's persistency layer transparently encrypts and decrypts all data and metadata blocks that it exchanges with the untrusted helper. Furthermore, it calculates and verifies MACs in order to ensure integrity of any encrypted data and metadata it receives from untrusted code. In journaled VPFS, the persistency subsystem also performs a minimal amount of state tracking so as to ensure that only consistent updates get written to persistent storage and it creates journal records as needed. For example, whenever the persistency layer puts a leaf-level block into the write buffer array in shared memory, it also creates a corresponding MAC-update record. I call this extended version of VPFS Core's persistency subsystem the *Sync Manager*.

Txn Manager. Cooperation between Sync Manager and the commodity file system that it reuses is enabled by the untrusted helper. To support journaling, the helper must not only provide access to blocks in file containers, but it must also write all records that Sync Manager provides to the journal. The VPFS journal is a file in the commodity file system and the helper appends records before it writes blocks to file containers that depend on those records (e.g., because a record contains an updated MAC for that block). Thus, the untrusted component enforces a simple transaction model for write accesses. I hence call this more complex helper *Txn Manager*. Txn Manager makes extensive use of existing infrastructure, as it exploits consistency guarantees the underlying commodity file system might provide (e.g., write ordering).

Message Types. Table 5.1 lists all message opcodes of the extended interface. During normal operation, Sync Manager sends `Block_read` and `Txn_flush` messages to request uncached data blocks, or to flush blocks and journal records queued in shared memory. `Txn_flush` replaces `Block_write` from the basic version; `Checkpoint_write` flushes the file system root info when all buffers are clean and file containers are in a consistent state. Messages of type `Checkpoint_read` and `Journal_read` are exchanged at mount time and, if necessary, during recovery.

5.3 Dependency Tracking in VPFS Core

Journaling updated MACs as described in Section 5.1.2 ensures that VPFS Core can verify the integrity of all blocks even after an unclean shutdown. However, consistency of traditional metadata such as inodes and directory entries is not addressed so far.

Metadata Dependencies. Many journaling file systems, including Ext3/4 and ReiserFS, append complete metadata blocks to their journal in order to log inode, directory, and allocation updates [137]. They implement *full-block journaling*. I considered this approach for VPFS Core's persistency subsystem, but rejected it as I found it to be too complex: journaling at the block level

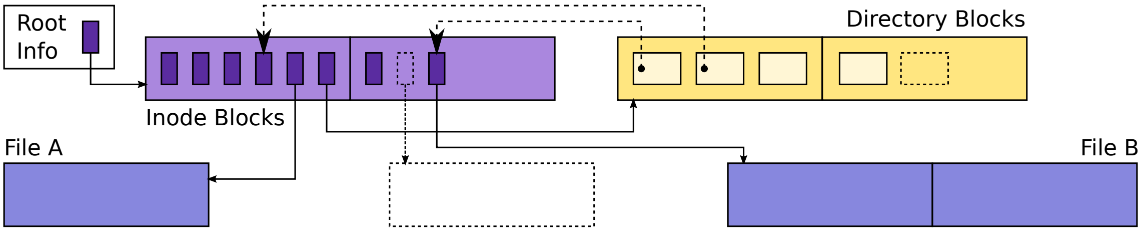


Figure 5.2: Dependencies among cached file system structures: Entries in directories point to inodes, which are associated with files. The superblock-like file system root info in VPFS contains the root hash of the block tree. Blocks and metadata objects with dotted outline have been deleted in cached file system state, but still exist in file containers.

requires fine-grained dependency tracking within the TCB. For each operation (e.g., creating a new file), VPFS Core would have to remember which metadata structures the memfs layer modified, in which buffers they reside, and how they relate to each other. Even worse, inode and directory entries of several independent files are co-located in the same metadata blocks. Co-location may cause false dependencies. Figure 5.2 illustrates the difficulties when writing back contents and metadata for a single file A: the directory block containing A's filename holds another entry for file B, which has recently been created, but whose data and inode have not been written yet. Thus, writing the directory block for A creates an inconsistency in the journal, because it makes persistent a directory entry for file B that references an incorrect or unallocated inode. In this example, writing the block containing B's inode as part of the transaction solves the problem for B, but may create a new one: because that particular inode block no longer contains the inode of a recently deleted file C, another inconsistency is the result, if the old directory entry for C still exists in a file container.

Breaking Cyclic Dependencies. To avoid increasing transaction sizes by including a potentially large number of unrelated metadata, file systems that use full-block journaling implement rollback mechanisms that temporarily remove incomplete updates from metadata blocks before they are written. An alternative to this rollback approach is to keep modified inodes and directory entries in separate caches, whose contents are flushed to metadata blocks selectively prior to write back. Certain journaling file systems do not log entire metadata blocks, but only small records that describe specific modifications to blocks (e.g., changes in an inode). Examples for file systems using this *record-based journaling* are NTFS and JFS [137]. Journaling small records is more space efficient than full-block journaling. However, the file system implementation still needs to track changes within individual metadata objects.

Operation-Level Journaling. It turned out that the least complex approach to journaling metadata updates in VPFS is to record the high-level operation that caused them (e.g., "create file X in directory Y"). This *operation-level journaling* is a form of record-based journaling, but it can be implemented with much simpler dependency tracking within the file system TCB.

5.4 Being Prepared for Crashes

The general idea for operation-level journaling is as follows: starting from a consistent set of file containers, which contain a persistent *checkpoint* of all file system state, Sync Manager issues write back operations for blocks as needed. The trusted component also creates journal records describing the high-level operations that caused metadata changes such as namespace updates or changes to specific inode fields. Upon receiving a `Txn_flush` message, Txn Manager appends the records to the journal and then writes all buffered blocks to file containers; for blocks that overwrite encrypted metadata (e.g., blocks with inodes or directory entries), the untrusted component logs undo information as well. The chain of journal records describes all operations that modified metadata since the last checkpoint. Occasionally, Sync Manager flushes all buffers – containing both data and metadata – in order to bring all file containers into a consistent state; this state marks a new checkpoint, at which all previously written journal records can be discarded and a new journal is created.

Recovery after an unclean shutdown then involves two steps: (1) Txn Manager restores all metadata blocks in file containers to the state from the last checkpoint; (2) Sync Manager replays all logged operations in their original sequence to update the checkpoint version of the metadata structures to the last consistent metadata state before the crash.

5.4.1 Journaling Metadata Operations

Figure 5.2 illustrates the dependencies of standard file system metadata. In VPFS, an individual file has the following dependencies to metadata objects:

- **Inode:** The inode of a file contains the file size in bytes, which specifies how much data in the last data block is valid file content. In VPFS, the inode also stores the root MAC of the per-file block tree that is embedded in the respective file container.
- **Pathname:** The inode of a file is referenced by a directory entry, which is stored in a directory file. As VPFS supports hierarchical subdirectories, a file's pathname can depend on directory entries stored in multiple directory files.

The dependencies described above are critical for a consistent representation of a file. For each newly created file that gets flushed from cache buffers, Sync Manager must make sure that the file's inode and all directory entries along the pathname are written to file containers as well. The file and namespace abstraction in VPFS Core's memfs layer already implements most of the required book keeping; a consistent checkpoint will have all required metadata correctly stored in file containers.

Creating New Files. In order to log the creation of files between checkpoints, Sync Manager maintains additional state. For each new file (including directory files), it keeps the following information for later use in a journal record:

- A copy of the filename
- A pointer to the inode of the parent directory

The filename and the inode number specify the file's location in the directory hierarchy; VPFS Core obtains them by resolving the pathname when the application calls `open()` to create the file. Sync Manager stores this information in a simple table that is essentially an extension of the file descriptor table in the memfs subsystem. This volatile `new_file` state is complemented by a one-bit flag in the inode that indicates whether the inode and pathname of a file have already been written. Based on this information, it is possible to re-execute the create operation.

Sync Manager executes Algorithm 5.1 to construct a `File_create` record for each newly created file or directory. The algorithm is recursive and takes care of all parent directories: records are created in the order of path elements, unless they have already been logged or they are part of the last checkpoint. In addition to the filename and the parent's inode number, a `File_create` record also contains a copy of the new file's inode. The `mode` field in the inode specifies whether the object is a regular file or a directory; thus, the `File_create` records can describe `mkdir()` operations as well. Once a file's creation has been logged, data blocks belonging to the file can be written back.

Renaming and Deleting Files. Namespace operations such as `rename()` or `unlink()` are logged analogously: A `File_rename` record contains the inode numbers of the source and destination directories as well as the old and new filename. A `File_unlink` record contains just the inode number of the parent directory and the filename. However, both operations differ in the way

Algorithm 5.1 C++-like pseudo code for journaling metadata of newly created files: The recursive algorithm ensures that the creation of all parent directories is logged before the `File_create` record for the target file is appended to the journal.

```
void journal_file_create(File *file) {

    // check, if file creation has already been logged
    if (file->inode()->is_logged())
        return;

    // not announced in journal, get new file info
    struct new_file *info = new_file_info_table[file->file_handle()];

    // lookup file descriptor of parent dir
    int p_fh    = info->parent_file_handle;
    int p_iptr  = info->parent_inode_ptr;
    File *p_dir = get_file_descriptor(p_fh, p_iptr);

    // if parent dir is still open, check if it must be logged, too
    if (p_dir)
        journal_create_file(p_dir);

    // announce new file in journal
    file->inode()->set_logged(true);
    File_create_rec rec(file->inode(), p_iptr, info->name);
    append_to_journal(rec);
}
```

they handle newly created files. A `rename()` operations always go to the journal. This heuristic is suitable for modern applications that primarily use the `rename()` operation to atomically update a file by first writing a new copy and then renaming it to the old pathname [108]. Before creating the `File_rename` record, Sync Manager executes Algorithm 5.1 on the source file and the parent directory of the target path to make sure their metadata is logged. No additional state tracking is required. An `unlink()` operation is logged in the journal only if the file's creation has been logged before. Thus, short-lived files that only existed in cache buffers do not show up in the journal.

Minimal Allocation Tracking. VPFS Core delegates block allocation and free-space tracking to the untrusted commodity file system. Therefore, it does not need to log updates to bitmaps, extent trees, or other allocation-related metadata. VPFS Core's memfs does maintain inode allocation information embedded in the block tree (see Section 3.3.3 for details), but it does not need to log updates to this metadata explicitely. It is sufficient to journal `File_create` and `File_unlink` records. There is one corner case related to unlinked files that may leave orphaned inodes in the inode table; I will present a simple solution to this problem when discussing post-crash garbage collection in Section 5.6.

5.4.2 Journaling Block Updates

Sync Manager distinguishes between user blocks and metadata blocks. User blocks are leaf-level blocks in the per-file block tree of every regular file. Metadata blocks are all blocks from inode or directory files (including leaf-level blocks) and all internal nodes of the block tree.

Writing User Blocks. The key requirement to be met when writing a user block is that its MAC needs to be appended to the journal first. Sync Manager prepares write back of user blocks by performing the following operations:

1. Calculate new MAC over plaintext content of the updated block.

2. Encrypt block, put ciphertext into free write buffer in shared-memory area.

3. Put `Block_update` record containing updated MAC and current file size into the write log buffer in the shared-memory area.

The untrusted parts of the VPFS stack are responsible for actually writing the block and its updated MAC to untrusted storage. Txn Manager enforces atomicity of block–MAC updates by enforcing the following constraints:

1. **Write Order:** Updated MACs must reach the journal before the actual block is written to the file container. The underlying commodity file system must be made aware of this write-before relation, for example, by calling `fsync()` on the journal file or by exploiting other write-order guarantees the underlying file system might offer.

2. **Keep Old MAC:** Should a system crash or power failure interrupt the write operations initiated by Txn Manager, the order of journal appends and block writes ensures that either (a) the new MAC is persistent in the journal and can be used to authenticate the updated block, or (b) the the old version of the block can be authenticated using the old MAC still available in the on-disk block tree node or in the journal.

Txn Manager may submit a newer version of a block before the previous version reached stable storage without violating point 2, because the MAC for the previously submitted update is logged in the journal.

The described block-update scheme is conservative and potentially expensive in terms of write barriers. I will discuss safe and secure relaxations for the common case in Section 5.7.

Writing Metadata Blocks. Case 2(b) described above implies that the previous version of a user block's MAC must always be available during replay. To meet this requirement at all times, Txn Manager treats metadata blocks that contain MACs differently from blocks containing user data. Sync Manager provides a hint in each `Block_update` record that informs the untrusted component about the block type. Assume a metadata block M is part of the latest checkpoint in untrusted storage. Before Txn Manager overwrites M with an updated version M', it rescues a copy of M into the journal, thereby preserving it in case replay becomes necessary. This operation incurs a write barrier between the journal append and the block update operation. If more updates to M arrive, Txn Manager writes them immediately. It does the same for other types of metadata blocks, including blocks containing directory entries. Thus, Txn Manager preserves the checkpoint version of all security-critical metadata in the journal. Consequently, it does not need to append MAC update records to the journal for any metadata block.

5.4.3 Checkpoints

The split journaling scheme in VPFS ensures that critical metadata can be restored after a crash and that all integrity guarantees hold. However, letting the journal grow indefinitely would cause unacceptably long recovery times. Sync Manager therefore flushes all dirty user and metadata blocks occasionally. Once VPFS Core's memfs determined that all cache buffers are clean, Sync Manager signals the untrusted Txn Manager with a `Checkpoint_write` message that a new consistent checkpoint can be established. Txn Manager then executes the following steps:

1. **Flush Log Buffer:** Process all journal records still queued in shared memory, submitting all block updates to the commodity file system.

2. **Sync File System:** When encountering a special `Checkpoint` record, instruct commodity file system to make all file containers persistent.

3. **Create Checkpoint:** Atomically swap current journal file with newly created journal containing just the `Checkpoint` record. Garbage collect the old journal.

A VPFS instance can be fully reinstantiated from checkpointed file system state. In fact, the `umount()` operation in VPFS is identical to the checkpoint operation.

5.4.4 Securing the Journal

VPFS Core leverages the following mechanisms to protect confidentiality and integrity of journal records.

Confidentiality. Journal records that encode high-level operations contain confidential metadata information such as filenames, inode pointers, or even complete inode contents. This information provides clues about the directory hierarchy and per-file metadata. Therefore, it must be encrypted. On the other hand, the untrusted Txn Manager must be able to infer consistency constraints from journal records in order to update file containers in a safe and consistent way. Sync Manager enables this kind of cooperation by encrypting the payload of journal records, but not their headers. Information that Txn Manager requires for correct operation is replicated in the header, but not necessarily with full precision. For example, the header of a `Block_update` record contains fields describing the type of a block and its location, such that untrusted code can correctly order block and journal writes. VPFS also uses a `File_trunc` record, which contains as encrypted payload the exact file size in bytes, while the header of such a record merely specifies the minimum number of blocks that the file container should have after the `ftruncate()` operation.

Integrity. A continuously updated journal MAC (i.e., keyed hash) protects the integrity of the journal. The initial input for computing the journal MAC is the content of the file system root info structure, which includes a random nonce and a checkpoint version number. Both these data items change after each checkpoint, therefore, a journal is cryptographically bound to exactly one checkpoint. Sync Manager embeds intermediate MACs in special `Txn_end` records, which serve to authenticate all preceding journal records. Sync Manager inserts a `Txn_end` record in the stream of journal records whenever it flushes the log buffer, which happens either when the buffer is full, or when an application issues a synchronous write operation such as `fsync()`.

In the following section, I will discuss how to restore the last consistent state after an unclean shutdown.

5.5 Recovering From Crashes

If the system did indeed crash, the commodity file system must recover first. VPFS benefits from the complex logic [87, 159] the untrusted infrastructure provides to recover file containers in the untrusted storage.

5.5.1 Replaying the Journal

Once the untrusted storage has been remounted, VPFS can start its own recovery process. It starts with a preparation step executed by the untrusted component.

Preparing Replay. Txn Manager first performs a rollback of metadata blocks in file containers. This step is necessary, because operation-level journaling can enable file system recovery after an unclean shutdown only under the following assumptions:

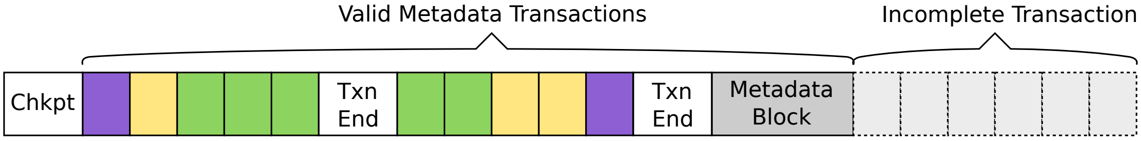


Figure 5.3: Transactions in the journal: Sync Manager can replay all operations that are logged in journal records followed by a `Txn_end` record, which contains a journal MAC that authenticates all preceding records. Incomplete transactions with a `Txn_end` record will not be replayed.

1. Journal records describe incremental updates to metadata blocks and each operation modifies the metadata state the preceding operation created. The initial metadata state is the state after the last checkpoint.

2. Operations specified in journal records are replayed in the precise order in which they were logged. Otherwise, consistency or integrity properties might be violated.

The first requirement dictates that, if there is a checkpoint version of a metadata block preserved in the journal, this version must initially be used during replay – even if a newer version reached its in-place location in the file container. To meet this requirement, Txn Manager copies all metadata blocks it finds in the journal (if any) back to their original locations in the respective file containers before replay starts.

Mounting a Checkpoint. For the trusted component, recovery starts by requesting from Txn Manager the first record stored in the journal. This `Checkpoint` record contains the cryptographically protected file system root info, which Sync Manager decrypts using the platform's sealed memory service; a sealed memory implementation as described in Section 2.3 can also provide freshness guarantees for the `Checkpoint` record.

The write-back and journaling scheme explained in the previous section guarantees that VPFS Helper's Txn Manager subsystem can always provide a valid `Checkpoint` record, even after an unclean shutdown. However, this assumption may not hold, if a successful attack according to the threat model defined in Section 2.4, a hardware failure, or a software issue outside the TCB damaged the first journal record. If the `Checkpoint` record could not be read or it failed validation, VPFS Core reports an integrity error to the application and the file system remains inaccessible. Otherwise it initiates the replay phase.

Replaying Metadata Operations. Replay is performed cooperatively by both Sync Manager and Txn Manager: the former requests journal records by sending a `Journal_read` message. Txn Manager responds by filling the read log buffer in the shared-memory area with a set of records that end with a `Txn_end` record, containing an intermediate journal MAC. Figure 5.3 shows the post-crash state of a VPFS journal; note that Txn Manager will restore all metadata blocks it finds in the journal, but it will not pass them to the trusted component. Sync Manager executes the following steps for each set of journal records to recover the file system state that was valid shortly before the unclean shutdown:

1. **Decrypt Records:** The decryption routine decrypts all record in the shared-memory buffer and places the decrypted versions in temporary log buffer in VPFS Core's private memory.

2. **Check Integrity:** Using the intermediate MAC in the `Txn_end` record, Sync Manager checks the integrity of the entire set of records. The check is performed on the records placed in the private temporary buffer so as to prevent time-of-check-time-of-use (TOCTOU) attacks. If integrity validation fails, Sync Manager aborts, reports an integrity error, and the file system is not mounted.

3. **Replay:** Re-execute all operations specified in all decrypted and verified records.

To replay metadata operations such as creating or renaming files, Sync Manager reuses existing APIs in VPFS Core's memfs: the replay routine extracts the type of the operation and the required parameters from a journal record and then simply calls the corresponding API function (e.g., `open()` or `rename()`). The implementation reuses the same code paths that handle file system calls from an application. The only difference lies in the handling of pathnames: while the application originally created a file by specifying a full pathname, the replay routine passes as arguments the inode number of the parent directory and a filename relative to this directory. The entry points for all high-level metadata operations are part of the internal inode-based naming interface that I described in Section 4.2.

Handling File Contents. Records describing metadata blocks are ignored; Txn Manager took care of them already. Replaying `Block_update` records for user blocks is performed similarly to re-executing high-level operations. However, metadata changes described in such a record must be applied only if they describe the version of the block that actually exists in the file container. To determine the validity of the update, Sync Manager requests the block specified in the record from Txn Manager and attempts to validate its integrity. Two situations can occur:

1. **Block–MAC Match:** The version of the block in the file container is the one described in the record. Sync Manager can safely apply the metadata changes described in the record: it updates the file size as specified in the update record and copies the MAC from the record into the block's parent node in the block tree. Note that the parent node can always be retrieved and authenticated, because either (a) it was never overwritten, (b) it had been preserved in the journal, or (c) an updated parent node has been allocated earlier during replay as part of a resize operation.

2. **Block–MAC Mismatch:** The block is either older or newer than the update described in the record. Sync Manager ignores the record and assumes that the MAC in the block tree matches the block or a matching `Block_update` record will be replayed later.

The indeterminism in the second case is caused by the journaling scheme: since Txn Manager does not log user-block contents in the journal (even though it could), there is only the block version in the file container itself. On the other hand, the journal may contain multiple `Block_update` records for the same block and finding the one that matches requires scanning the entire log.

The metadata of the file system is recovered to the last consistent and verifiably correct state, once Sync Manager processed the last `Txn_end` record provided by the untrusted component.

5.5.2 Recovery Results

A correctly behaving Txn Manager that obeys write-ordering constraints can always provide the last `Txn_end` record, such that Sync Manager recovers all MACs in the block tree correctly.

Integrity. Misbehavior on the part of Txn Manager can cause stale MACs to exist in the block tree without Sync Manager noticing immediately. However, such an attack is equivalent to any other type of block-level corruption and it will be detected as soon as an application requests the respective data. Iterating over all `Block_update` records a second time, after replay finished, could verify the correctness of all updated MACs immediately. However, doing so only proves that the untrusted parts operated correctly with respect to journaling and replay. It does not prove the absence of future attacks or the integrity of parts of the file system untouched during recovery.

Torn Writes. After an unclean shutdown while overwriting existing file contents, a file may contain both old and new user blocks. VPFS guarantees the integrity of both old and new data under the assumption that aligned and block-size writes to the underlying commodity file system are atomic. Log-structured and copy-on-write file systems meet this requirement. If *torn writes* due to a power failure cannot be ruled out, VPFS may lose user blocks during in-place updates of file containers: if a block is partially written, no matching MAC will exist to successfully verify its integrity. A modified version of VPFS Core could subdivide user blocks into multiple parts with individual MACs protecting sector-size regions. However, support for sub-block MACs adds complexity to the TCB and VPFS would still be susceptible to torn writes within a sector. The current implementation of VPFS ignores the possibility of torn block writes.

Atomic Updates. If an application requires a guarantee that the latest version of a file is persistent, then it should follow these steps: (1) write the data in question to a new file, (2) call `fsync()` on the file, (3) rename the new file to the old location, and (4) call `fsync()` on any file to flush the `File_rename` record to persistent storage. A similar sequence of operations is the recommended approach to atomic file updates for commodity file system as well.

Freshness. Unfortunately, incrementally calculated MACs cannot reliably mark the end of the journal. Therefore, VPFS Core cannot know for sure, whether it recovered the most recent consistent state before the crash occurred: there could be more records following the last `Txn_end` record that Sync Manager saw. If the untrusted components withhold parts of the journal, they successfully compromise freshness. To prevent this attack, Sync Manager must store in tamper-resistant sealed memory the journal MAC from the last `Txn_end` record that Txn Manager reports to be persistent. If the system crashed and Sync Manager replays, it can compare that last recovered journal MAC with the trusted reference MAC protected by sealed memory. For performance reasons, updates of sealed memory should be done only once for each checkpoint, or if an application requests a freshness guarantee explicitly through an `fsync()`-like operation.

5.6 Garbage Collection

The preceding two sections described the basic approach to journaling in VPFS. There are a two corner cases left that require special attention: (1) cleanup of orphaned inodes and (2) truncation of blocks from file containers.

Orphaned Inodes. VPFS Core's memfs implements POSIX-like semantics for `unlink()`, where an application may hold a file handle for a file that has been unlinked after the `open()` operation. The memfs subsystem removes the directory entry immediately and logs the `unlink()` operation in the journal. However, it cannot deallocate the inode until the application released its file handle. If the application keeps the file open across checkpoint boundaries, each of the checkpoints will include the *orphaned inode* of this file in the persistent copy of the inode table. If the system crashes after the first checkpoint, the original `File_unlink` record will have aged out of the journal and VPFS Core does not know that it can deallocate the orphaned inode.

The least complex approach to preventing this kind of resource leak is to keep pointers to all currently orphaned inodes in a table that Sync Manager flushes whenever requesting a new checkpoint. This table can have a fixed size equal to the maximum number of open files, which is small enough such that it fits into the file system root info structure. When remounting the file system, Sync Manager iterates over the table and deallocates all inodes mentioned in it. The implementation of this functionality contributes less than 20 lines of code to the TCB.

Delayed Truncation. Removal of a file or – in the general case – truncation of blocks is a metadata operation that VPFS Core must log in the journal. However, Txn Manager must be careful not to truncate file containers too early. Assume that VPFS Core's Sync Manager must replay the journal after a crash. As VPFS Core re-executes the logged operations, it may have to retrieve metadata blocks that `unlink()` or `ftruncate()` operations appearing later in the journal will discard. Thus, it is potentially unsafe to truncate or delete file containers that contain inodes or directory entries before the next checkpoint. Txn Manager could copy blocks to the journal in order to preserve them for replay like it does before overwriting metadata blocks. However, doing so requires additional read, write, and barrier operations as described in Section 5.4.2 on page 65.

As an optimization, Txn Manager therefore leaves all blocks that shall be be truncated untouched until after the next checkpoint: it buffers all truncation requests in a list and a background thread garbage collects all obsolete blocks after the next checkpoint. The implementation takes incoming write requests into account to make sure that no blocks written beyond a buffered truncation offset are cut accidentally. As the list of truncation requests is held in volatile memory, Txn Manager faces a problem similar to the issue of open file handles that link to orphaned inodes. To prevent the list from getting lost during a crash, Txn Manager flushes the current list to the new journal before atomically switching the old and new log to establish a checkpoint.

Delayed truncation is an optimization in the untrusted part of the VPFS stack. The following section concludes the discussion of journaled VPFS with an overview of other optimizations that require little or no support in the TCB.

5.7 Optimizations

Intuitively, one would assume that ordered writes to the journal and file containers incur significant performance overhead. Fortunately, I/O costs can be reduced drastically by optimizing write back.

Write Batching. VPFS Core buffers journal records and encrypted data blocks in the shared-memory area, until there is no more space or the application explicitly requests a synchronous write (e.g., by calling `fsync()`). Buffering reduces communication overhead and enables write batching. Batched writes require fewer synchronous writes, because Txn Manager can coalesce a large number of record appends into few journal updates. The benefit is twofold: first, the underlying commodity file system requires fewer I/O operations and at most one write barrier to update the journal file. Second, the commodity file system may write blocks to file containers according to its own optimized strategies, potentially achieving higher performance.

Relaxed Write Order. Txn Manager requests a write barrier after updating the journal to ensure that user or metadata blocks can be updated safely. However, many common write workloads do not perform any block updates at all: modern applications often do not overwrite existing data, but they write updates to new files and then swap the old and new file in a `rename()` operation [108]. For these types of workloads, Txn Manager can ignore the order of journal and block writes because there is no old state to be preserved. If MAC updates need to be replayed after a crash, VPFS Core will need both the block and the corresponding `Block_update` record, but it does not matter which was written first. Incomplete writes of block–MAC pairs are treated as if no block had been written at all. In combination with write batching, VPFS achieves I/O overheads close to that of the reused commodity file system, with only few additional writes to the journal file and occasional checkpointing of block tree nodes.

The logic for relaxed write order is implemented almost entirely in the untrusted part of the VPFS stack. Sync Manager only provides a hint indicating whether an older block exists; the hint is trivially computed by checking if – prior to flushing a buffer – the block's MAC in the parent node is null or not.

Exploiting Existing Infrastructure. For the performance evaluation presented in Chapter 7, I benchmarked VPFS with different types of commodity file systems, including ReiserFS [63] and NILFS2 [41]. When using ReiserFS or a similar commodity file system, Txn Manager can only use the POSIX function `fsync()` to order writes for consistency. This POSIX system call guarantees that all data and metadata of a file have reached stable storage upon return. However, this persistence guarantee is stricter than what VPFS requires: in the common case, Txn Manager merely requires that certain I/O operations do not overtake each other: journal records with updated block MACs must be written before modifying the file container. Thus, `fsync()` does serve as a write barrier, but it also flushes all data to the storage medium synchronously.

The log-structured file system NILFS2 can ensure a strict order of write operations without calling an explicit API: NILFS2 assures that writes reach stable storage in the same order in which an application issued them. Txn Manager exploits this behavior to eliminate unnecessary I/O delays when updating blocks. However, if an application explicitly calls `fsync()` on a file in the virtual private file system, then Txn Manager has no other choice but to force the corresponding file container and the journal to stable storage.

5.8 Summary

From the perspective of VPFS Core's Sync Manager, the approach to metadata journaling used in VPFS resembles the log-structured approach [139]. Like in a log-structured file system, the trusted component creates a continuous stream of records describing all changes to on-disk state and, occasionally, Sync Manager and Txn Manager checkpoint the file system state to get shorter mount and recovery times. The trusted VPFS Core avoids the key problem in log-structured file systems: it does not need to take care of complex garbage collection. VPFS Core delegates the complex cleanup work to the untrusted parts of the file system stack. Apart from its own simple dependency tracking, the trusted component does not need to deal with the complexities of writing data to the storage medium in the correct order. VPFS benefits from existing infrastructure in untrusted commodity file systems to enforce its own consistency guarantees. Finally, operation-level journaling avoids unnecessary complexity in VPFS Core by reusing APIs and code paths that are already part of the TCB. As a result, the support for journaling and replay in the security-critical VPFS Core comprises only about 380 lines of code, which is more than an order of magnitude smaller than the approximately 4,000 lines of code in Ext4.

The next chapter discusses how to support the protection goal of *recoverability* in VPFS.

Chapter 6

Backup and Restore

The VPFS architecture as discussed so far enables stronger security guarantees for locally stored data than monolithic system architectures do with respect to confidentiality and integrity. But it cannot guarantee availability, because VPFS relies on complex untrusted code bases. The threat model described in Section 2.4 acknowledges that attacks on untrusted components may occur and that cryptographically protected file system contents of VPFS instances may be damaged. If the user has a backup available in such a situation, he can recover the data after an attack. An up-to-date backup is also an essential safety net that prevents data loss due to other causes, including hardware and software failures and if the user's device gets lost or stolen, which is a real threat especially for mobile device owners.

This chapter describes an extended version of the VPFS architecture that enables *recoverability* through an integrated backup and restore mechanism. The key challenge of designing and implementing recoverability support in VPFS is that the attack surface must not become significantly larger, so as to not weaken the strong security guarantees for confidentiality and integrity.

6.1 Approaches to Backup and Restore

I already reviewed approaches to security in distributed file systems in Sections 2.5 and 4.1. In the following, I will discuss additional related work in the area of file system backup and cloud storage. I will then focus on how to ensure availability of data stored on remote servers.

6.1.1 Backup and Cloud Storage

Backup and Restore as an Operating System Service. The capability to back up and restore file system contents is a key feature of an operating system (OS). On mobile devices, this feature is well-integrated and easy to use. For example, Apple's iOS implements automatic backup of system configuration files and all user data managed by the installed applications. Backups are stored either on remote servers [78], or on a personal computer the user controls. If the need arises, the user can restore all data and system settings from such a backup on another device. The Android OS has integrated backup and restore functionality for personal data such as contacts and calendars as well as configuration settings. This data is stored on remote servers, which are operated either by Google or the user's device vendor or wireless carrier. Third-party developers who want their Android applications to utilize the system-wide "Backup Manager" need to implement an application-specific "Backup Agent" [26]. This opt-in approach gives developers explicit control over what data will be backed up, but it also forces them to invest additional work in order to ensure recoverability of stored data.

Large Trusted Computing Bases. The backup functionality in iOS is implemented as a system service, which is granted full access to all files that applications store in their private directories.

Thus, the backup service is a shared part of the trusted computing base (TCB) for all applications with respect to confidentiality, integrity, and recoverability. Additionally, when backing up to a remote server, the user must fully trust the server operators with respect to the same set of protection goals. Although the network connection between mobile device and server is cryptographically protected, the server receives a plaintext copy of all user data. Because of the limited security guarantees of this architecture, any passwords and other credentials stored in an iOS device are excluded from remote backup. The backup service integrated in Android works in a similar way; the Android developer documentation even warns: *"Because the cloud storage and transport service can differ from device to device, Android makes no guarantees about the security of your data while using backup."* [26]

There are alternatives to Android's built-in backup service. For example, Titanium Backup [52] is a third-party application that can create backups on remote servers. It provides stronger security guarantees for remote backups than the built-in solution, as it encrypts all configuration and user data locally before uploading it to remote servers. Unfortunately, this utility application and similar ones [47] require unrestricted access to the shared file system in order to create and restore backups (i.e., they must run with root privileges). Thus, they become part of the TCB for all other applications.

Cloud Storage Services. Many online storage services that are also popular on desktop-class computers, for example, Dropbox [28] or SkyDrive [39], require the user to place full trust in the servers. They often advertise the possibility to share data as a core feature, so they employ only on-the-wire protection for transmitting data between client and server. In contrast, Wuala [56] advertises a no-trust model for their server farm. The system uses client-side encryption and manages encryption keys using the Cryptree [105] scheme, which enables selective sharing of individual files or folders. The Wuala client utility also enables sharing through the service's website by encoding per-file or per-directory encryption keys directly in link URIs, which users can pass on to others; the Wuala web front-end can then decrypt the respective files transparently as users enter the URIs into their browsers. SpiderOak [50] offers a similar feature set and security model.

Weak Guarantees. None of the described backup and cloud storage systems can make verifiable guarantees about integrity or freshness. Service level agreements (SLAs) for cloud infrastructure such as Amazon S3 [62] or Windows Azure [83] do not assure these protection goals; they only cover service availability in the sense that "operations complete without error within certain time bounds for 99,9 percent of the time"; otherwise service fees are reduced. The user must trust the server cloud with respect to protection goals other than availability, unless he employs cryptographic measures locally to enforce confidentiality or integrity.

Provable Guarantees. CloudProof [135] improves on that by enabling cryptographic proofs for violation of SLAs that cover guarantees for integrity, order of write operations, and freshness. These guarantees are given at the level of *blocks*, which are accessed through a key–value interface. CloudProof checks correctness of the global write order and freshness of reads not immediately, when clients access the cloud storage, but when the *owner* of a block periodically audits previous operations. During such an audit, the owner compares *attestations* describing all read and write operations as reported by both clients and the server; if the attestations received from all sources match, the cloud did not violate write serializability or freshness of reads during the auditing period. To make auditing scalable, the owner assigns to each block a probability of being audited. Thus, not all blocks need to be checked every time, but the cloud must behave correctly for all blocks as it does not know which of them are going to be audited next.

6.1.2 Availability of Remotely Stored Data

Even if a storage provider can be held accountable for their misbehavior, data can still be lost if servers or disks fail and data becomes unavailable. Reliable storage systems often employ *replication* to distribute redundant copies of data across different storage nodes in order to compensate for failing disks or server downtime. For example, the Google file system [103] stores at least three replicas of each file on different servers. Data can be recovered if at least one replica is available, but each copy increases storage and bandwidth overhead for writes by one hundred percent.

Erasure Codes. Cloud storage systems such as Windows Azure Storage [116] and Wuala [56] use erasure coding schemes to distribute data across multiple disks. Erasure coding ensures availability up to a certain number of disk failures, but with less storage overhead than replication. Optimal erasure codes such as Reed-Solomon [138] split a data block B into m equally-sized source fragments and then transform these source fragments into n encoded fragments of the same size. There are more encoded fragments than source fragments ($m < n$) and a data block B can be recovered from any combination of m different encoded fragments. The storage overhead is $\frac{n}{m}$.

Multi Clouds. RACS [84] makes the case that single "clouds" can still fail or be temporarily unavailable. The authors of RACS propose to apply the RAID-like erasure coding techniques not only to storage devices operated by a single service provider, but to stripe data across multiple cloud storages that are independent from each other. This cloud-of-clouds or multi-cloud approach allows the user to access data even if a cloud storage provider suffers permanent failure or the data it stored fell prey to deletion attacks. Works such as NCCloud [115] improve on the idea by using network coding schemes to make repair operations after partial failures more efficient than with traditional codes such as Reed-Solomon-based RAID-6 (i.e., up to $n - m$ failed "clouds").

Byzantine Clouds. DepSky [91] showed that the cloud-of-clouds approach with four independent commercial storage providers can provide better availability than a single cloud at roughly twice the cost. The system uses erasure codes to distribute data across the cloud storages and handles arbitrary faults of individual clouds through byzantine quorum system protocols. DepSky further protects data confidentiality using a secret-sharing scheme [145], such that individual clouds cannot obtain plaintext.

6.1.3 Backup and Restore in VPFS

Integrating backup and restore into the operating system frees application developers from having to invent their own solutions against data loss. Also, users do not need to worry about which third-party backup tool to chose. I therefore extended VPFS to support backup and restore in a generic and application-transparent way.

VPFS does not try to innovate on the fundamental techniques for backup and restore that I outlined on the preceding pages, nor does it introduce new approaches to ensuring availability of backups stored on remote computers. Instead, I aim at integrating existing techniques into the multi-component architecture of VPFS, such that cryptographically protected backups can be created with minimal impact on size and complexity of TCBs, including those for confidentiality and integrity.

Backup on Remote Servers. To ensure recoverability, VPFS must create backups in a location that is physically decoupled from the user's mobile device, but reachable via network. I assume this location to be storage attached to a remote server. To improve availability of backups in the event of temporary failures, disasters, or attacks, the extended VPFS architecture supports the use of multiple independent backup servers as suggested by the multi-cloud works. Section 6.6 discusses backup replication and redundancy based on erasure codes; both techniques enable restore operations, when only a subset of servers can provide backup state.

TCB for Recoverability. Backup servers are trivially part of the TCB for recoverability. On the client side, parts of the VPFS functionality need to be included in this TCB as well in order to ensure verifiability of backup creation: a trusted component in the VPFS architecture must enforce that each newly created backup meets the integrity requirements and that backups are indeed created. I call this trusted component *VPFS BackupD* and I refer to the user's mobile device as the *client device* when discussing aspects related to the client–server architecture. The specific implementation of VPFS BackupD, its subsystems, and the structure of minimal TCBs that enable recoverability will be explained in Sections 6.4 through 6.6.

Reuse of Untrusted Infrastructure. VPFS BackupD implements a trusted wrapper for an untrusted commodity network stack running on the client device. Note that solutions that delegate backup and restore entirely to untrusted components without any TCB involvement are insecure. For example, a standard tool such as rsync [48] could backup cryptographically protected file containers on remote servers. However, the user would be left with no guarantee for recoverability, because the untrusted software may fail silently or send backups to non-trustworthy servers.

6.2 Authentication and Transport Security

Verifiable backup creation requires secure communication between the trusted VPFS BackupD on the client device and the remote backup servers. The communication channel must provide *mutual authentication* for the following reasons:

1. **Verifiable Backup Creation:** VPFS BackupD must be certain that it did indeed create a new backup. Therefore, it requires assurance that it sent the backup to a server entrusted to store backups securely and reliably.

2. **Backup Availability:** A server must be sure not to compromise the integrity of a backup unwillingly. Therefore, it requires assurance that any request to update a backup originates from the VPFS instance that initially created the backup.

Protocols for mutual authentication can provide both the client and the server the assurance that they are indeed communicating with the correct peer. In addition to authentication, each communication partner requires assurance that all messages and data it receives over the communication channel do indeed originate from the previously authenticated peer.

Transport Layer Security. One of the most widely-used protocols for authentication on the Internet is the Transport Layer Security (TLS) [64] protocol. It establishes authenticated communication channels over insecure networks. TLS is based on public-key cryptography and *certificates* that describe key owners. Each certificate contains a public signature key and is issued by a *certificate authority (CA)*. Each communication partner A can verify the identity of its peer B in a two-step process: (1) A checks that B is in possession of the private key whose public counterpart is encoded in B's certificate, and (2) A verifies that the certificate has been issued by a CA that it trusts to vouch for B's identity. TLS supports mutual authentication based on certificates, but in the most common usage scenarios, the protocol is used to authenticate only the server; the client authenticates itself using another mechanism (e.g., based on a username and a password). In addition to authentication, TLS also provides forward secrecy of a communication session by encrypting all data with a random session key negotiated by both peers. Furthermore, it ensures integrity of all transmitted data using hash-based message authentication codes (MACs). Integrity protection in TLS also guarantees that messages are received in the same order in which they have been sent.

Secure Shell Authentication. The remote terminal protocol Secure Shell (SSH) [58] also uses a public-key scheme for mutually authenticating [59] two communication partners: a user and the server the user wants log into. SSH guarantees confidentiality and integrity of the session. However, in contrast to TLS, the protocol does not use certificates issued by CAs, but it requires that users and administrators manually register public keys belonging to the server and the user, respectively.

Practical Security of TLS and SSH. The security of TLS strongly depends on the integrity of CAs. There can be more than one CA and they can vouch for each other, thereby establishing chains of trust. Many applications such as web browsers and email clients accept any certificate that chains up to any CA listed as a trusted *root CA*. The trust model collapses, if any of the root CAs is compromised such that an attacker can create fake certificates. This worst-case scenario already became reality multiple times [44, 27]. To reduce the risk of compromise, a custom application scenario such as VPFS backing up to specific servers should accept only a minimal set of trusted CAs for authentication. Even if TLS-based authentication is restricted to just one CA, the scheme is more convenient for the user than the manual key distribution method commonly used for SSH.

No Custom Implementation. It is widely accepted that developers should not try to invent their own cryptographic protocols, because the resulting schemes may contain weaknesses that are not immediately apparent. Instead, well-studied protocols shall be used. It would be hard to justify why VPFS should disregard this advise, as the carefully studied TLS and SSH protocols provide the required functionality for mutual authentication, as do similar protocol suites [60].

A plausible reason for a custom solution could be the size and complexity that TLS or SSH add to the security-critical code base: implementations of these protocols comprise in the order of tens of thousands of lines of code. For example, the TLS library MatrixSSL [38] consists of approximately 17,000 lines of code; libssh2 [37] would contribute at least 14,000 lines of code to a recoverability TCB. However, it is unrealistic to assume that a custom implementation of a state-of-the-art protocol would be significantly smaller than off-the-shelf libraries. There two major reasons for this assessment:

1. Splitting approaches that separate critical and non-critical functionality cannot be applied to such protocol implementations, because their entire implementations are security critical: one has to trust ciphers and hash functions, certificate and key parsers, mechanisms for certificate or key revocation, and even state machines like those implementing TLS or SSH handshakes. Hence, all code related to mutual authentication must be part of the recoverability TCB.

2. A review of the 17,000 lines of code in MatrixSSL showed that approximately 10,000 lines are required to implement essential cryptographic building blocks such as symmetric and asymmetric ciphers, collision-resistant hash functions, and key handling. Of these, several thousands lines of code contain unrolled inner loops and precomputed tables. The complex TLS state machine and the API add about 5,500 lines of code on top of the cryptographic foundation. A significant portion of this second part of the code base is related to deprecated protocol versions that need not be compiled into the binary. Stripping down the code base any further could result in non-obvious vulnerabilities or reduced compatibility.

Thus, no significant reduction in code size – like by a factor of five – can be expected that would result in a considerably smaller probability of implementation defects. I therefore decided to reuse off-the-shelf libraries to implement mutual authentication of VPFS instances and servers in VPFS.

Potential Vulnerability of Protocol Implementations. Unfortunately, weaknesses in the reused code cannot be ruled out. In particular, certificate and key handling in TLS relies on complex data formats. Commonly used implementations have a history of security vulnerabilities in the code that parses and evaluates information encoded in the certificates [3, 4, 21]. The OpenSSH implementation dropped complex parsing logic for its key handling in favor of a simpler RSA verification scheme. The developers of the software report [82] that this decision saved their software from eight security-critical bugs that were found in equivalent code of the OpenSSL protocol suite.

I do not assume a specific protocol or even implementation of authentication and transport protection. The extended VPFS architecture can support different solutions, which, for example, may be selected according to the specific protocol that the servers use. Diversity of implementations of a certain protocol can improve resilience against attacks. I will define the recoverability-related extensions to the VPFS threat model in the following section, before I present design and implementation of the client-side support in VPFS afterwards.

6.3 Extended Threat Model

The trusted parts of a VPFS instance encrypt all file system content before creating a backup and they can rely on the MACs in the block tree to detect any missing or corrupted data and metadata when a backup is restored. Therefore, VPFS does not need to trust backup servers with regard to *confidentiality* and *integrity* as defined in the basic threat model in Section 2.4. The basic threat model also defines the protection goal of *recoverability;* I repeat this definition here for reference:

Recoverability: A consistent and recent backup must exist to recover file system contents after a system failure or an attack on integrity. A successful attack on recoverability prevents VPFS from creating a new or retrieving a recent backup over extended periods of time, or it corrupts the integrity of the backup.

I consider three attack vectors on recoverability: (1) the user's device, (2) the network infrastructure used to transmit the backup, and (3) the servers entrusted to store the backup. Note that both the client and the server make similarly good targets with respect to the protection goal.

Attacks on Mobile Device Software. According to the basic threat model, I assume that the various TCBs in a VPFS instance cannot be successfully attacked. However, the attacker may be able to compromise any untrusted component such as the untrusted VPFS Helper or the commodity file system. In accordance with the principle of *mutual distrust*, any other VPFS instance is considered untrusted as well. For an attacker who seeks to compromise recoverability, the networking infrastructure in the user's device is a prime target. I assume the complex commodity network stack to be untrusted, so the attacker may be able to compromise it entirely.

Attacks on Network Infrastructure. In addition to compromising the client-side network stack, an attacker may be able to gain control of critical parts of the network that connects the user's device and a remote server. Such an attacker may be able to intercept or inject network packets. I assume that cryptographic communication protection as described in Section 6.2 prevents the attacker from corrupting control messages and backup payload. However, the attacker may block their transmission entirely or in part or cause secured connections to abort by injecting forged packets. The attacker may be able to divert network packets to servers other than those selected by the user. I assume that failure to authenticate the server as described in Section 6.2 reveals such an attack.

Attacks on Backup Servers. I distinguish two types of attackers that can target the server: (1) outsiders that can only communicate with the server via the network, and (2) insiders with privileged access to the server's software stack or the backup storage:

1. Outside attackers may send backup servers arbitrary requests. I assume that a correctly behaving server that implements authentication as discussed in Section 6.2 terminates unauthenticated connections and only accepts requests to update an existing backup, if the update originates from the same VPFS instance that initially created the backup.

2. A privileged attacker (e.g., a corrupt employee) may be able to compromise the server or its backup storage directly. Confidentiality and integrity of these backups are enforced cryptographically by VPFS, but the privileged attacker can take the server offline without notice or he can destroy backups. Furthermore, when a user tries to restore a VPFS instance from a backup, an attacker controlling a server could deny the user access to the backup. Any of these attacks make backed up file system state on that server unavailable.
   The privileged attacker could also provide the user an older version of a backup, although the

client device requested the most current one. Such a rollback attack defeats freshness guarantees, if the user's device no longer knows what the most current backup is (e.g., because sealed memory state is unavailable due to hardware failure or loss of the device).

If a sufficiently large number of servers are compromised by inside attackers (depending on the amount and type of redundancy), remote backup state is unavailable and the user cannot restore a backup. If enough inside attackers collaborate, they may present an older version of a backup as the newest. I assume that attackers do not collaborate, if servers are operated by independent service providers. Attacks on freshness of a backup at recovery time can also be prevented using hardware-protected monotonic counters [154] in the servers; a modified version of TLS that has been enhanced with trusted computing support [86] could be used to report freshness securely to the client. However, the prototype implementation of VPFS does not include such support.

Detection of Attacks. Since VPFS relies on untrusted infrastructure to transmit and store backups, a temporary inability to create or restore backups cannot be ruled out. Unavailability during a recovery attempt is evident to the user. However, backup creation should best be performed automatically and in the background such that the user is not bothered with maintenance tasks. Attacks on client-side untrusted software components or the network may prevent timely backup creation. Unfortunately, VPFS BackupD cannot distinguish a temporary network or server unavailability from a malicious attack aiming to prevent creation of a new backup. VPFS could warn the user, if data could not be backed up for a certain time. A simple implementation of denial-of-service detection in VPFS BackupD could notify the user in the following cases: (1) if a backup could not be created for three days, and (2) when a backup could be finally be created after a reported failure period.

Now that the complete threat model has been defined, I explain the extended architecture of VPFS in greater detail. I start out with a discussion of the structure of the individual TCBs and then explain how backup works, first for a single and then for multiple servers.

6.4 Client-Side TCBs for Backup and Restore

The VPFS architecture that I presented in the last three chapters only deals with local file system operation. In Section 3.1.4, I argued that the TCBs for confidentiality and integrity can be co-located in the same components, as splitting them further does not improve security in a practical way. The situation is different for the security-critical functionality that ensures the protection goal of recoverability.

Privileges and Requirements. In the extended architecture, the trusted components of VPFS play two major roles: (1) serving ordinary applications, and (2) creating backups. The backup use case differs from VPFS Core serving an ordinary application. Nevertheless, VPFS BackupD can be regarded as a special type of application that just requires read-only access to file system state in a VPFS instance. The following functionality and privileges are required for each of the two use cases:

1. **Serving Applications:** VPFS Core must perform read and write operations on behalf of an application. The TCB has to ensure confidentiality and integrity of all data and metadata.

2. **Creating Backups:** VPFS Core must be able to traverse the block tree in order to determine which blocks are newer than the backup. This functionality performs read-only accesses and is part of the TCBs for confidentiality and recoverability. The VPFS BackupD component must compile the update from all blocks that VPFS Core found to be newer than the backup and then send the update to the remote backup servers. VPFS BackupD is part of the recoverability TCB, which also includes the functionality for mutual authentication and transport security.

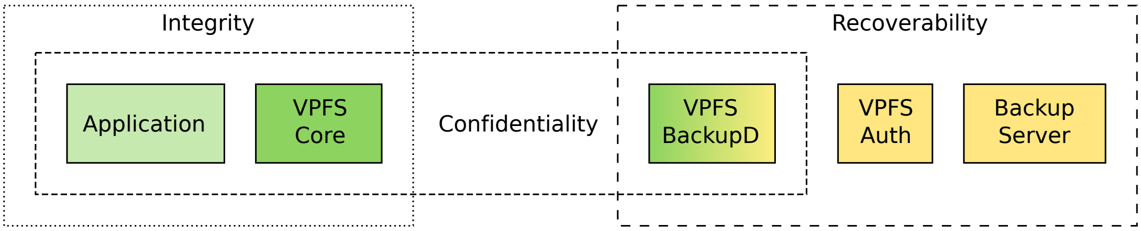


Figure 6.1: TCBs for confidentiality, integrity, and recoverability: The client-side TCB for recoverability is split into VPFS BackupD, which creates cryptographically protected "patches" for remote backup state, and VPFS Auth, which ensures secure communication with backup servers.

Interestingly, the TCB for integrity does not need to include the functionality for creating a backup, because modification of file system contents is not necessary. This insight is based on the definition of integrity from the basic threat model defined in Section 2.4, which specifies that no unauthorized *modifications* must be possible *without being noticed*. However, the backup-related components need read-only access to the block tree – including the Merkle tree nodes in it – in order to create a consistent update that will not fail integrity checks after a backup is restored.

Furthermore, the functionality that secures the communication between client and servers need not be included in the TCBs for confidentiality and integrity. This functionality can be assigned the same level of trust as the backup servers that are merely trusted to ensure availability of a backup, which is equivalent to recoverability.

Split TCB for Recoverability. Considering the non-negligible code size and potential vulnerability of an off-the-shelf implementation for TLS or a similar protocol, I split the recoverability TCB on the client device into a part that overlaps with the confidentiality TCB, and a part that belongs to the domain of trust of backup servers. Figure 6.1 illustrates the idea, showing both VPFS BackupD and an isolated *VPFS Auth* component. This architecture has two main advantages over a monolithic TCB for recoverability: (1) the TCB for confidentiality does not grow beyond what is necessary to build the BackupD component, and (2) it supports modularity: the implementation of the Auth component can be exchanged depending on which protocol or protocol implementation shall be used to secure communication between client and servers.

Removing Backup from the Integrity TCB. VPFS ensures integrity by enforcing that only VPFS Core can access the MAC key required for Merkle tree updates. No other component must be allowed access to this key, as it can be used to compute new root MACs. On the other hand, VPFS must ensure that it will send only consistent and integrity-preserving backups to the remote servers. Therefore, VPFS BackupD requires access to the MAC key in order to check the integrity of each block that shall be sent to the servers. This conflict of privileges can be solved by replacing the root MAC of the block tree with an asymmetric key signature: VPFS Core is in possession of the private signature key, VPFS BackupD can only access the public verification key. However, this approach would require additional cryptographic algorithms to be included in the integrity TCB. A more efficient solution is to compute two root MACs with two different keys: one with a *verification key* V and another with a *signature key* S. Both VPFS Core and VPFS BackupD have access to the verification key, which is stored in the file system root info. The signature key is stored in sealed memory and only VPFS Core can access it: VPFS Core can update and check both root MACs, but VPFS BackupD could change only the MAC generated using the verify key. The prototype implementation uses only one root MAC, but adding a second one as described is straightforward.

Auth Credential and Server Identity. VPFS Auth is responsible for establishing the secure communication channel with the backup server. In the split architecture outlined in the previous paragraphs, the VPFS Auth component is also the only part of the client-side recoverability TCB

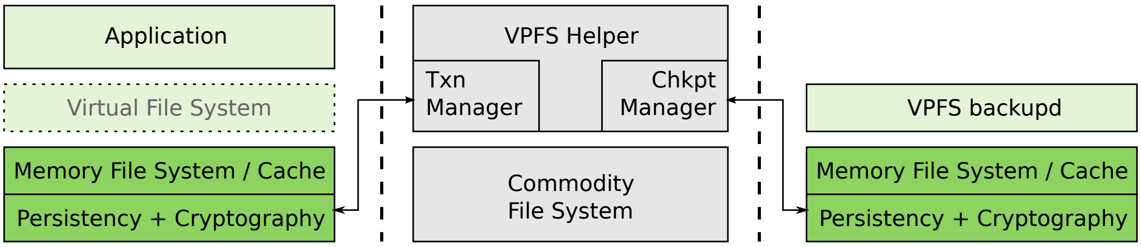

Application
Virtual File System
Memory File System / Cache
Persistency + Cryptography
VPFS Helper
Txn Manager
Chkpt Manager
Commodity File System
VPFS backupd
Memory File System / Cache
Persistency + Cryptography

Figure 6.2: Architecture for creating backups in the background: The untrusted VPFS Helper serves two instances of VPFS Core, one for ordinary applications, and one with read-only access to the latest checkpoint state that is exclusively used by VPFS BackupD.

that requires access to authentication data: (1) the credential used to authenticate the client to the server, and (2) information in order to establish the identity of the server (e.g., a public key or CA certificate). The VPFS Auth component stores both the credential and the identity anchor in sealed memory.

6.5 Backup Creation

VPFS implements incremental updates to reduce the amount of data that the client device needs to send to remote servers. In this section, I describe the mechanisms for creating a stream of block and file container updates that bring a backup stored on a remote server to the same state as the local file system. I will explain how to generalize the approach to updating multiple servers in Section 6.6.

6.5.1 Enabling Backups in Background

It is important that applications can still use the file system while backup is in progress. Clearly, it is not acceptable to block writes from applications, just because VPFS BackupD is not done yet sending a consistent update to a remote server. To ensure both liveness for applications and progress for VPFS BackupD, the system must allow block updates to be written to file containers and it must preserve those blocks that still need to be backed up. A copy-on-write scheme for block accesses efficiently meets these requirements.

Avoiding Additional Complexity in TCBs. Initially, I intended to build VPFS BackupD as a special application that shares a VPFS Core instance with ordinary applications. I quickly dropped this idea, as I found that it would add unnecessary complexity to the TCB and it is cumbersome to implement. First, the TCB would have to include copy-on-write support as described above. Second, VPFS Core would have to take special care of cached copies of internal nodes of the block tree. During normal operation, VPFS Core's persistency layer updates MACs and backup version numbers in parent nodes lazily as child blocks are written back to untrusted storage. To support backup, it would need to compute MACs and version numbers eagerly, otherwise the parent node could not be backed up.

Backing up Checkpoints. From an architecture point of view, the cleanest and simplest approach is to create backups from a checkpoint. I introduced checkpoints in Section 5.4.3 as a fully consistent representation of local file system state. To enable backups in background, VPFS instantiates a second, read-only instance of VPFS Core that mounts the checkpointed file system in untrusted storage. I extended the untrusted helper such that it can handle requests from two instances of VPFS Core in parallel: an *application instance*, which serves ordinary applications, and a *backup instance*. VPFS BackupD is the only "application" that holds access rights to the latter instance of VPFS Core. VPFS BackupD uses the internal inode-based naming interface of VPFS Core's memfs subsystem.

Coordinating Access to Untrusted Storage. The untrusted helper's Txn Manager subsystem handles the application instance. It cooperates with a closely related subsystem called *Chkpt Manager,* which provides read-only access to checkpointed versions of file containers. The most important aspect of this cooperation is that Txn Manager gives its read-only counterpart a chance to rescue any checkpoint version of a block before it gets overwritten. The checkpoint version of each such block is copied to a temporary *backup container* and Chkpt Manager redirects future read accesses for any rescued block to this container; the approach is similar to rescuing metadata blocks in the journal to enable crash recovery (see Section 5.4) except that no synchronous writes are required. The backup container is deleted after backup or, if the system crashed during backup, as soon as the file system is remounted.

Figure 6.2 on the previous page shows the components involved when updating a remote backup in background. Note that the architecture delegates most of the backup-related complexity to untrusted parts of the VPFS stack: the untrusted helper implements copy-on-write for blocks to ensure progress and a consistent view of different versions of the local file system state for both VPFS Core instances.

6.5.2 Updating a Remote Backup

Incremental updates of remote backups require that the client-side TCB of VPFS knows what is contained in the latest backup on a server. To update remote backups, VPFS BackupD sends "patches" to the server containing all changes that accumulated in the local file system after the last backup. VPFS computes this delta based on version numbers assigned to both local file system state and the remote backup.

Finding Out What Changed. To determine what changed in the local file system, VPFS BackupD iterates over all inodes and checks the version numbers stored in them. Each of these version numbers describes the root node of the corresponding per-file block tree. If the version number is greater than the backup version, the file contains changes that need to be backed up. In this case, VPFS BackupD instructs its instance of VPFS Core to crawl the corresponding file's block tree in order to find all blocks that are newer than the backup. The crawling operation utilizes the generic tree walker described in Section 3.3.3 in conjunction with a backup-specific strategy: the `Modified_blocks_strategy` class facilitates retrieval of blocks by comparing the version numbers stored in internal nodes of the block tree against the reference version of the remote backup. Figure 6.3 shows an example of which blocks would need to be backed up to update an existing remote backup.

Note that retrieval of inodes and file contents is subject to the same integrity checks that VPFS Core performs when serving an application.

Encoding an Incremental Backup. VPFS BackupD creates a patch for updating the remote backup on the fly, as it iterates over the file system as described above. All necessary changes to data and metdata are encoded in a stream of records. There are four different types of records: (1) a single `Backup_journal` record at the beginning of the stream contains the new version of the checkpointed file system root info, (2) `Backup_file` records indicate the new size of a file container, (3) `Backup_block` records contain a single block's content and a block number, and (4) a `Backup_commit` record contains the current checkpoint version number and is the last in the stream. Records of type `Backup_block` are grouped by file container and all records belonging to a file F appear in the stream after F's `Backup_file` record. VPFS BackupD does not need to take special care for enforcing such an order, the tree walker and the modified-blocks strategy automatically provide blocks exactly as needed.

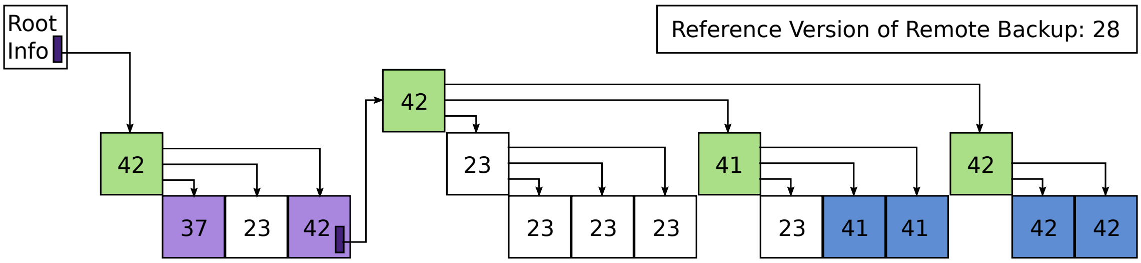


Figure 6.3: Identification of blocks that require backup: Based on version numbers stored in inodes and internal nodes of per-file block trees, VPFS BackupD can determine which blocks are newer than a remote backup. All blocks that have a version greater than reported by the remote backup server need to be backed up. (The figure shows only the inode file and one regular file to which the application appended additional data after checkpoint version 28 has been backed up.)

Cryptographic Protection. VPFS BackupD encrypts the payload of the journal and block records before appending them to the backup stream. To this end, VPFS BackupD reuses APIs of VPFS Core's Sync Manager subsystem, which includes the functionality for encrypting blocks and "sealing" the file system root info using the platform's sealed memory service (see Section 2.3). Note that the block number in a `Backup_block` record is not encrypted, such that the backup server can apply block updates to file containers correctly. VPFS BackupD also computes an incremental hash of the entire backup stream up to the `Backup_commit` record. This record includes the final hash, which the server uses as an additional safeguard to determine whether the client sent the complete stream over the cryptographically protected communication channel.

Forwarding the Backup. The backup stream reaches the remote server through a chain of differently trusted components. In the following, I summarize the roles and TCB assignment of all involved components:

- *VPFS BackupD* ensures confidentiality of the data and metadata. This component further ensures that the "patch" encoded in the backup stream updates the remote backup to a consistent copy of the local file system. Thus, VPFS BackupD is part of the TCBs for confidentiality and recoverability.

- *VPFS Auth* implements the client-side endpoint of the mutually authenticated and integrity-protected communication channel to the server. The Auth component forwards the backup stream received from VPFS BackupD through the channel. The code base of VPFS Auth is not part of the TCB for confidentiality anymore; it must only ensure recoverability by verifying that the incremental backup reaches the correct server.

- *VPFS Helper* is fully untrusted. It transmits cryptographically protected messages of the authentication and transport protocol implemented by VPFS Auth. VPFS Helper establishes the underlying network connection via the insecure network between client and server.

The remote backup server is again part of the recoverability TCB and it implements the other endpoint of the secured connection.

Telling the Server What Is Obsolete. The stream of records encoded by VPFS BackupD contains only information about files that changed since the last backup. In particular, `Backup_file` records specify the size of each file container and the server can truncate them to save space if necessary. Unfortunately, VPFS BackupD does not know which files have been removed since the last backup, so it cannot tell the server directly which file containers are no longer needed. However, the local backup component possesses trustworthy information about all existing files in the form of the inode table. To assist the server with garbage collection of obsolete file containers, VPFS

BackupD occasionally sends `Backup_file` records for all files, and not just those that changed. Using the complete list of live file containers in the encoded backup stream, the server can identify file containers that became obsolete.

The client side encodes in the `Backup_commit` record, whether the list of `Backup_file` records is complete or partial. It sends the full list when VPFS Helper indicates that a fast connection has been established with the server (e.g., when using WiFi), otherwise it minimizes the size of the backup stream by sending a partial list.

Committing a Backup. The server receives the backup stream and logs it in its file system. Only when the stream matches the hash included in the `Backup_commit` record at the end, will the server apply the changes to the file containers and increase the server-side backup version to the version stated in the `Backup_commit` record. The server can signal a successful backup as soon it received the stream. Note that all changes described in the backup stream are idempotent: if the server fails during the update, it can safely re-apply the logged stream after it recovered from the crash. To ensure atomicity, the server must always fully apply a logged update before serving the then most-recent backup to the client. The backup is temporarily not available while the server updates all file containers and the clean VPFS journal containing the checkpointed file system root info.

6.5.3 Impact on TCB Complexity

The TCBs for both confidentiality and recoverability include VPFS BackupD, which contributes approximately 400 lines of code to implement the functionality described above. The implementation of a block iterator based on the `Modified_blocks_strategy` class in VPFS Core accounts for an additional 40 lines of code in these TCBs. The backup-related enlargement of the trusted code base is comparable to the additions that enable the untrusted VPFS Helper to support concurrent execution of two VPFS Core instances: checkpoint coordination and copy-on-write support in VPFS Helper comprise about 250 lines.

6.6 Backup to Multiple Servers

When backing up to only one server, the user must trust the party that operates this server to ensure recoverability. This protection goal depends on the availability of the backup. The server operators can use fault-tolerance techniques such as storage replication to ensure durability and improve availability, but an inside attacker can still compromise the backups as discussed in Section 6.3. To reduce the trust in a single server, VPFS can distribute backups across multiple backup providers that operate independent servers. This multi-cloud approach [84, 115] has been shown to increase availability of remote storage in practice [91].

6.6.1 Redundancy Schemes

I consider two approaches to distributing the data: (1) replication, which distributes identical copies of backups, and (2) erasure-coded backups striped across the remote storages.

Replication. The advantage of replication is its simplicity for both backup and recoverability. To backup, VPFS BackupD sends the backup stream to all servers. Since each server stores a full copy of the backup, one available server is sufficient in order to recover a lost file system. The disadvantages of replication are high storage overhead on the servers and increased network traffic, because VPFS must send copies of an incremental backup to each server. If the user choses to back up to n servers, the aggregated storage requirement and data volume to be sent increases by a factor of n as well.

The amount of data sent from the client device and the latency can be reduced using pipelining: just one server receives the backup stream directly from the client and as soon as it obtained the

Rate m/n	**1/1**	**1/2**	**1/3**	**3/4**	**2/3**	**2/4**
Storage overhead	1	2	3	1.33	1.5	2
Availability (critical+)	0.95	0.9975	0.9999	0.9860	0.9928	0.9995
Availability (all)	0.95	0.9025	0.8574	0.8145	0.8574	0.8145

Table 6.1: Availability of groups of unreliable servers: The availability of each individual server is assumed to be 0.95. The probability that at least a critical subset of m servers are available is calculated for different coding rates and varying numbers of independent servers. Coding variants with $m = 1$ are replication-based, the others require an optimal erasure code.

first records, it forwards them to the other servers. This approach is, for example, used in the Google File System [103]. To verify that the backup has indeed been updated on all servers, VPFS BackupD establishes connections to each server via the VPFS Auth component, such that the servers can securely acknowledge receipt of the backup stream to the client. Servers that offload network traffic from the client must cooperate. However, they may still be operated by independent parties, thereby satisfying the threat model.

Erasure Codes. Erasure codes improve on replication by reducing the amount of storage required on each server and the amount of data that needs to be transmitted. Minimizing network traffic is important for mobile devices, which often rely on slow or expensive cellular networks. Optimal erasure codes transform a piece of data into n fragments, of which any subset of m different fragments (i.e., $m < n$) is sufficient to recover the data. The rate $\frac{m}{n}$ determines the coding efficiency. The inverse of the rate is equal to the aggregated storage overhead across all locations; hence, rates greater than $\frac{1}{2}$ require less bandwidth and storage than replication-based schemes. The fragment size $\frac{s}{m}$, with s being the size of the original data, determines how much data the client sends to each of the independent servers in order to update an erasure-coded backup.

Erasure coding is compatible with the incremental backup scheme introduced in Section 6.5. The difference compared to sending a replica of changed file system state is that `Backup_journal` and `Backup_block` records in an erasure-coded backup stream contain fragments of clean journals and blocks, respectively. I call a backup stream containing encoded fragments of the original payload a *fragment stream*.

Availability. Table 6.1 lists the probabilities that at least a critical number of m servers are available (figures labeled "critical+"). Probabilities are given for both replication ($m = 1$) and erasure-coding schemes ($m > 1$) for different configurations of m and n. The model underlying these example calculations is a binomial experiment consisting of a series of n Bernoulli trials with $p = P(A)$ and $P(A)$ being the availability of each independent server. For illustration purposes, $P(A) = 0.95$ is chosen, which is conservative and significantly lower than the 99.9 percent advertised for commercial cloud storages [62, 83]. The calculations show that replication achieves the best availability, but at a higher storage and bandwidth overhead.

To recover a backup from a set of at least m out of n servers, each of the servers must store a replica or a fragment of that backup. Consequently, the client must distribute a backup across *all* servers in order to guarantee the recoverability probabilities as modeled above. Unfortunately, the last row of Table 6.1 indicates there is a significant chance that only a subset of the machines is available at a certain time: the probability that at least one server is unavailable during a backup attempt already exceeds ten percent for $n > 2$. Even with more reliable servers that are available 99 percent of the time, there is a four percent chance that not all servers can be reached (assuming four independent servers).

6.6.2 Inconsistent Updates

Clearly, it is not in the interest of the user to abort backup attempts, if a non-critical number of servers (i.e., at most $n - m$ machines) are unavailable. Instead, VPFS backs up the most recent local file-system state, if at least m servers are available (i.e., the minimum number of servers that are needed to retrieve a backup).

Inconsistent Updates. For replication, it is safe to update just one server, as each server stores a complete copy of the backup. Such an *inconsistent update* can only improve recoverability of the most recent backup in the future. It cannot destroy an existing backup, if updates are applied atomically using the logging approach described in Section 6.5.2. When using erasure coding, an inconsistent update applied to a critical subset of the servers (i.e., just m out of n servers) may prevent a future restore operation, if one of the updated servers is not available at the time of restore. If each server stores just one version of its backup state, a permanent failure of one of the inconsistently updated servers will leave the user without any backup.

Versioned Backups. To prevent the harmful effects of inconsistent updates with erasure coding, the servers must keep a history of previous versions of the backup, such that the client device can retrieve fragments from the most recent backup version that is available on m or more servers.VPFS backup servers keep older versions of the backup; the client can easily provide a hint to the servers that old versions can be removed, when it commits a *consistent update* on all servers.

Maintenance and Repairability. Once a failed server is available again, the client's VPFS BackupD can establish a consistent state on all servers by sending an incremental update that includes all changes since the oldest backup on any server. For both replication and erasure codes, the client can minimize outbound traffic to servers that already hold a recent version of the backup: the backup streams for these servers need to include only those `Backup_file` and `Backup_block` records that describe changes that are newer than the respective server's backup version. The correctness of this optimization is obvious for replicas, but it is also safe for erasure-coded backups, if the coding scheme deterministically produces the same fragments for the same input data (e.g., as done by Reed-Solomon codes [138]).

6.6.3 Recoverability Model

At least m out of n servers must be available in order to restore any remote backup. The probability that the *most recent* backup can be recovered depends on the availability of each independent server during both backup and restore operations. I assume that, if a server is available, it will successfully complete the operation requested by the client.

Probability of Successful Backup. The client will only update a remote backup, if at least the critical subset of servers is available. This event C_U occurs, if m or more of said servers complete the update operation. Each independent server S_i is updated with probability $P(U_i)$. Since I assume that available servers successfully complete the requested operation, one can set $P(U_i) = P(A_i)$. Thus, the probability $P(C_U)$ can be determined using the previously described binomial experiment based on a series of n Bernoulli trials with $p = P(A_i) = P(A)$ with $P(A)$ being the availability of an independent server. Results for selected configurations of m and n are shown in Table 6.1 on the preceding page (row labelled "critical+").

Probability of Recoverability. The set of servers that are available at recovery time may not be the same as the set of machines that participated in the previous update operation. However, to retrieve the most recent backup, at least m of the up-to-date servers must be available. In order to compute the probability of this event C_R, one must know the probability $P(R_i)$ that an independent server S_i is both up to date and available during recovery:

$$P(R_i) = P(U_i \cap A_i)$$

Rate m/n	**1/1**	**1/2**	**1/3**	**3/4**	**2/3**	**2/4**
Storage overhead	1	2	3	1.33	1.5	2
Recoverability,	0.95	0.9905	0.9991	0.9501	0.9733	0.9966
$P(A) = 0.95$		(0.9975)	(0.9999)	(0.9860)	(0.9928)	(0.9995)
Recoverability,	0.99	0.9996	0.9999	0.9977	0.9988	0.9999
$P(A) = 0.99$		(0.9999)	(0.9999)	(0.9994)	(0.9997)	(0.9999)

Table 6.2: Probability of successfully restoring the most recent or a consistent backup with an older version: The user can recover the latest backup, if at least m of the servers that have last been updated are available. The figures are based on a per-server availability of either 0.95 or 0.99. Values in parenthesis show the probability of retrieving a potentially older backup version that has been retained on all servers.

I assume that temporary failures last shorter than the time that passed between the last backup and the moment the user wants to restore a file system. Therefore, the availability of S_i at the time of backup is independent from its availability at the time of the recovery operation:

$$P(R_i) = P(U_i) \cdot P(A_i) = P(A_i) \cdot P(A_i) = P(A) \cdot P(A)$$

Table 6.2 shows the probability $P(C_R)$ that the user can restore the most recent backup, if inconsistent updates are allowed. The figures are computed under the same model as is used for Table 6.1, except that the probability of each Bernoulli trial is set to $p = P(A) \cdot P(A)$. The figures in parenthesis show recoverability of any (potentially older version of) a consistently stored backup with $p = P(A)$; these numbers are identical to the ones in Table 6.1. Different configurations of unreliable – or likely to be compromised – servers as well as sets of reliable machines are considered: unreliable servers have an assumed availability of $P(A) = 0.95$, whereas reliable servers are assumed to successfully handle update and restore requests in 99 percent of all cases ($P(A) = 0.99$).

Consequently, using erasure codes to distribute backups across multiple servers is more efficient than replicating full backups. Based on the computations shown in Table 6.2, erasure-coded backups with the 3/4 or 2/3 configuration provide high availability of the most recent backup when using reliable servers. The 2/3 and 2/4 configurations of the erasure-coded backup variant are suitable for unreliable servers. Even if coding efficiency is the same for replication and erasure coding, the latter provides better recoverability as shown for the 1/2 and 2/4 configurations, respectively.

Erasure coding increases the complexity of the recoverability TCB. The next section discusses how this complexity can be minimized.

6.6.4 Self-Attesting Erasure Coding

Each server must acknowledge the receipt of its fragment stream through the secure communication channel, such that VPFS BackupD can verify that a backup has indeed been created. In the case of erasure-coded backup streams, this verification must also include a guarantee that a consistent backup can be restored from the coded fragments.

Naive Approach. Intuitively, one would assume that the coding functionality must be part of the TCB. If the erasure code's implementation is trusted, VPFS BackupD can be certain that the fragments are correctly coded and distributing them across all independent servers ensures recoverability as expected. Unfortunately, erasure coding libraries contribute significant complexity. For example, the open-source library Jerasure [117] contributes about 2,000 lines of code for Reed-Soloman coding variants. Considering that the TCB for recoverability already includes in the order of ten thousand lines of code for authentication and transport security, it seems acceptable to add an erasure coding implementation to this TCB as well. However, the bulk of encoding and decoding complexity can also be moved to untrusted code – even if the specific coding algorithm itself is unknown to the TCB.

Verifying Correct Encoding. The key insight that makes untrusted encoding possible is the following: assume that an untrusted encoder E transforms a data block d into a set of fragments f. If a decoder D can produce the original data d from no input other than f, then E must have behaved correctly. A trusted component verifies the correctness simply by comparing the original data d with the the output d' of the decoder. Correctness is established, if d and d' are identical. Note that the decoder need not be trusted, if its only input is the set of fragments. If however a side channel exists, then E could use this channel to provide the correct answer (i.e., data block d) to the decoder, which is then no longer required to produce d from f.

Further note that the trusted component can treat both encoding and decoding operations as black boxes. However, it must know the parameters m and n of the coding scheme in order to test redundancy of the encoded fragments.

Verifying Redundancy. Knowing the total number of fragments n, VPFS BackupD can be assured that all servers will be part of the distributed backup storage. Knowing the critical number of fragments m enables probabilistic verification of recoverability from a random subset of fragments: assume a data block d has been transformed into n fragments. By randomly erasing up to $n - m$ fragments from the set of fragments provided to the decoder, the trusted component forces the decoder to prove that the encoded fragments contain sufficient redundancy for correctly recovering the original d. As the untrusted encoder cannot know which fragments will be erased, it has no choice but to operate correctly for all blocks d. Otherwise, its misbehavior will be detected with probability $\frac{m}{n}$.

Integration in VPFS. The generic verification scheme I just described can be integrated into the VPFS architecture, such that the recoverability TCB no longer contains the routines for erasure coding and decoding. The untrusted encoder and decoder run in isolated address spaces. The communication capabilities of the decoder process are configured such that it can only receive input from VPFS BackupD, which includes the trusted verification logic described in the preceding two paragraphs. The algorithm for *self-attesting erasure coding* of backup streams works as follows:

1. VPFS BackupD creates the original backup data stream and forwards it to the untrusted encoder.

2. The untrusted encoder generates fragment streams by creating copies of the stream where the payload of `Backup_journal` and `Backup_block` records has been replaced with the respective fragments.

3. The comparator logic in VPFS BackupD receives the fragment streams, recomputes the hashes in the trailing `Backup_commit` records, and then forwards the streams to the isolated decoder. Before doing so, it randomly introduces erasures in order to test redundancy of the black-box coding scheme.

4. The isolated decoder restores the original backup stream from the partially erased fragment streams and returns the decoded backup stream to VPFS BackupD.

5. VPFS BackupD checks if the decoded backup stream matches the hash of the original backup stream it calculated itself. If they match, it is assured that the fragment streams encode a backup and can therefore be sent to the independent backup servers.

The prototype implementation executes steps 1 through 4 for groups of backup records in a loop and optimistically forwards the newly generated parts of the fragment streams to the servers. This pipelining approach minimizes latency and the amount of buffer space required on the client device. VPFS BackupD executes step 5 when it creates the `Backup_commit` record of the original

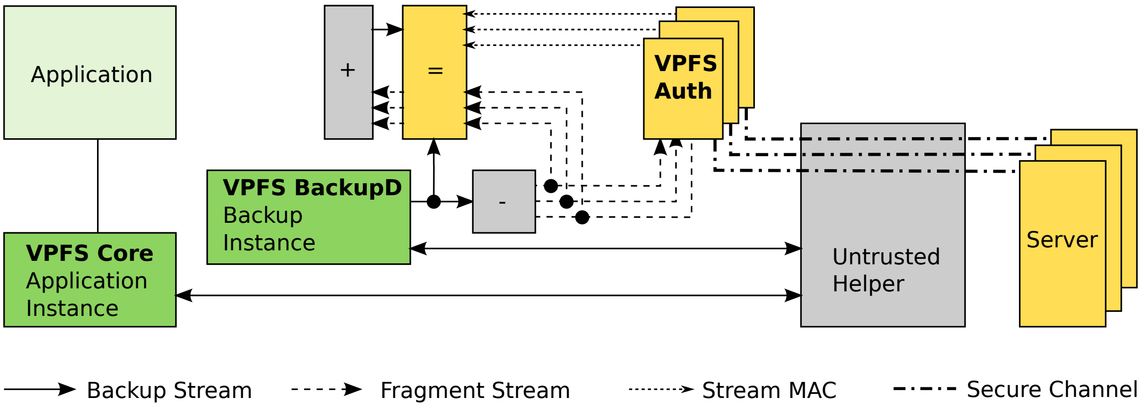


Figure 6.4: Functional components of VPFS backup architecture and TCB assignment: The black-box encoder (-) and the corresponding decoder (+) of the erasure-coding scheme do not need to be trusted. The comparator (=) component must be trusted for recoverability; due to its low complexity, it may be co-located with VPFS BackupD.

backup stream. Only if the last step succeeds, it sends the `Backup_commit` records for the fragment streams to the servers. As explained in Section 6.5.2, servers will commit an incremental backup only when they receive their commit records via the secure communication channel.

Figure 6.4 illustrates the integration of self-attesting erasure coding into the VPFS architecture.

6.6.5 Isolation of Independent Auth Components

The threat and recoverability models (Sections 6.3 and 6.6.3, respectively) assume servers are operated by independent parties; therefore they are expected to fail independently. This assumption holds, if servers are unavailable due to hardware or software-related issues outside the client device. I further ruled out that attackers who managed to compromise individual servers collaborate or conspire against the user. However, if an attacker is in full control of a server, he can use it as an attack vector to attack the potentially vulnerable implementation of the authentication and transport-security library in the VPFS Auth component.

To prevent an attacker from compromising backups on other servers via security-critical bugs in this component, VPFS uses a dedicated and isolated VPFS Auth instance for each independent server. Each such instance stores in sealed memory the credentials needed for mutual authentication of the client and the respective server. A compromised VPFS Auth instance cannot gain access to the authentication secrets for other servers. Figure 6.4 shows the fully decomposed architecture and TCB assignment for all involved components on the client side and the independent servers.

6.7 Restore from Backup

The recovery mechanism in VPFS extensively reuses the code bases involved in creating backups, including Auth components connecting to each of the backup servers and the previously described black-box decoder. This section discusses the recovery procedure for a single VPFS instance, assuming that cryptographic secrets associated with this instance are available in sealed memory. The problem of backing up and restoring these secrets is discussed in Section 6.8.

Mutual Authentication. In principle, a VPFS backup could also be restored using only untrusted software running on the user's device. However, in order to determine securely which backup on the available servers is the most recent, the recovery functionality running on the client device must communicate with the servers over an authenticated channel. Therefore, VPFS relies

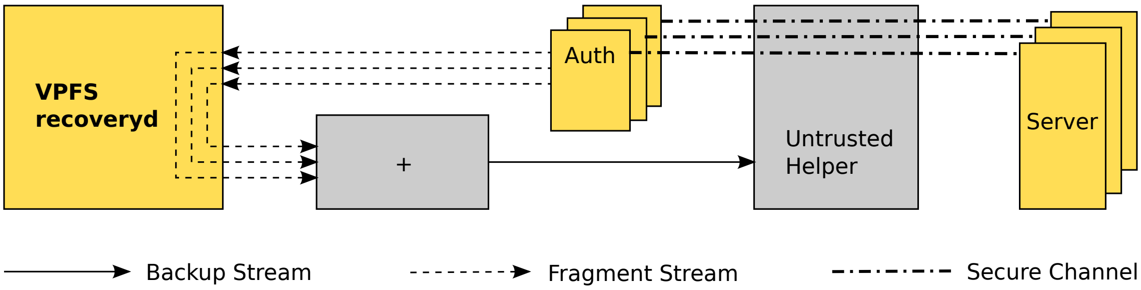
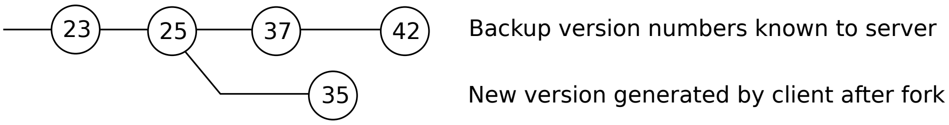


Figure 6.5: VPFS components involved in recovering a file system from backup.

on the Auth components involved in transmitting incremental backups also when restoring file system contents from backup. The requirement to use VPFS Auth also enables the servers to enforce read-access control on backups, thereby reducing the potential for denial-of-service attacks (i.e., only the recoverability TCB can generate the substantial load involved in downloading backups, but untrusted code or computers other than the user's mobile device cannot).

Retrieving a Backup. Figure 6.5 shows the VPFS components involved in recovery. VPFS RecoveryD determines the version to restore and requests via the Auth components the backup from the respective servers. The servers encode backup contents using the scheme described in Section 6.5.2 for sending incremental updates (i.e., as a stream of `Backup_journal`, `Backup_file`, and `Backup_block` records). The RecoveryD component forwards the received streams to the black-box decoder, which generates as result a backup stream containing the backup copy of the file system. The untrusted VPFS Helper extracts this backup stream into the untrusted file system, thereby recreating file containers and a clean journal, which contains only the file system root info. Note that, in contrast to the use case of self-attesting erasure coding, it is not necessary to isolate the black-box decoder from other untrusted components during recovery.

Continuing a Backup History. The application instance of VPFS Core can readily mount the recovered file system and provide authorized applications both read and write access to it. Any changes that the application does to the file system after recovery can be backed up incrementally as described earlier in this chapter. However, care must be taken, if an older version of the backup had been restored instead of the most recent one (e.g., because the newest version was not available on all reachable servers). Creating new backups from a recovered file system that is not the most recent one would fork the version timeline, as shown in the following example:


23
25
37
42
35
Backup version numbers known to server
New version generated by client after fork

If the client-side file system has been recovered from backup version 25 and a new backup with version 35 shall be created, this new version is no longer comparable to backup versions 37 and 42 created before the restore operation took place. To prevent inconsistent updates due to forks, each server increments an *epoch field* in the version numbers of all backups that are newer than the one the client requested in a restore operation. When the client initiates an incremental backup in the future, it can reliably reliably identify the newest compatible version on the servers in order to compute a consistent delta backup. Servers can garbage collect versions in dead-end branches on their own in order to save space.

TCB Complexity and Component Separation. The functional complexity of RecoveryD is similar to that of VPFS BackupD, but without block retrieval and the code base comprising VPFS

Core. The recovery-related code need not be part of the TCBs for confidentiality or integrity. However, there is potential for significant code sharing between the backup and recovery functionality (e.g., version negotiation and stream forwarding). Thus, one can also realize RecoveryD as a second mode of operation within the code base of VPFS BackupD; a command-line parameter specifies whether a new backup shall be created or an existing one is to be retrieved from the servers.

6.8 Key and Credential Backup

The backup mechanism described in Sections 6.5 and 6.6 ensures recoverability of VPFS file systems under the following two conditions: (1) cryptographically protected file containers or the VPFS journal on the user's device are lost or damaged, and (2) the cryptographic secrets needed to retrieve, decrypt, and verify the backup copy of the file system contents are still available.

Sealed memory implementations as assumed in Section 2.3 enforce cryptography-based access control on these secrets, but they require persistent storage for the secrets in their cryptographically sealed form. If the sealed version of a VPFS instance's secrets still exist, the user can perform the restore operation on the same device without any further requirements. However, the backup and restore functionality described so far will fail, if the storage backing the original device's sealed memory is damaged, too. Also, if the user must restore VPFS file system state on another device, the sealed memory service on the new device will not be able to provide the required keys, or even the credentials needed for retrieving the backup from the backup servers.

Design and Implementation of a complete sealed memory service are out of scope for this thesis. However, for completeness, I present a design draft for a sealed-memory backup solution that uses a special VPFS instance in order to reduce the problem to one-time backup of a small set of per-device secrets.

VPFS Key Bags. In order to restore a VPFS instance, the following cryptographic keys, credentials, and identity information are necessary:

- Encryption key E required to decrypt file system root info
- MAC key V required to verify file system integrity
- MAC key S required to sign file system updates
- Auth credentials A_i and server identities I_i $(i = 1...n)$ required to access backups

VPFS Core stores the integrity verification key V in the cryptographically protected file system root info, which is stored on the remote servers and therefore needs no additional backup. The keys E and S for each VPFS instance must be stored in sealed memory to protect confidentiality and integrity. As explained in Section 6.4, auth credentials and server identities are protected by this platform service as well. The collection of E, S, A_i, and I_i $(i = 1...n)$ is called the *key bag* of a VPFS instance; a backup copy of this key bag must be stored in a safe and secure place.

In contrast to file system contents, the key bag need not be updated during normal operation. Therefore, it is sufficient to back up the key bag once after initializing a VPFS file system. For example, the user could be asked to create a backup copy of the key bag on a flash drive, which he keeps in a safe and secure place such as a vault. However, each time the user installs a new application on his device, a new VPFS instance must be created and its key bag be backed up. It would be impractical to rely on the user for backing up key bags of each newly created VPFS file system.

Sealed Memory and Launch Service. To ensure timely backup of new VPFS key bags, I propose to store the contents of each such key bag in a special VPFS file system that is assigned to a key–value (KV) store service. VPFS Core and VPFS Auth components use the KV storage's interface to persistently store store keys in their sealed from. The KV store's underlying file system is backed up to the remote servers in the same way as ordinary VPFS instances, thereby ensuring that all VPFS key bags are backed up along with it.

In a practical system architecture, the KV store could be part of the platform service that launches applications along with their private VPFS instances. This launch service can easily restrict access to objects in the KV store based on information stored in an access control database, which it needs to maintain in order to correctly set up communication capabilities between the started application and VPFS instance. This access control database could reside in the same file system as the KV data objects.

A Minimal Recovery Key Bag. As the KV store described above is built on top of a VPFS file system, the problem of key and credential backup is reduced to one-time backup of a single VPFS key bag: the *recovery key bag* needed to restore the private file system of the KV store and launch service. Based on the two keys E and S and the server identities I_i, it is possible to securely derive the auth credentials A_1, ... A_n using a key derivation function (KDF) based on a collision-resistant one-way hash algorithm:

$$A_i = KDF(E||S||I_i)$$

The server identities I_i could be the URIs of the backup locations picked by the user (e.g., `www.backup.org/user` or `www.vpfs-backup.net`) in conjunction with expected fingerprints of the public keys or TLS certificates that identify the respective servers. In the prototype implementation, VPFS Core requires E and S to be symmetric keys. These keys are short enough such that the user can even write down textual representations on a piece of paper that can be deposited in a vault. URIs and key or certificate fingerprints are short as well. Thus, in the simplest case, the user can with moderate effort write down the reduced recovery key-bag once when he sets up his device. To restore all VPFS-based file systems from remote backups, he just needs to type them into a user interface of the device's recovery or setup service. Devices shipping with pre-configured server identities require even less effort, as only human-readable versions of E and S need to be backed up or re-entered.

6.9 Summary

In this chapter, I extended the attacker model to also consider recoverability as a primary protection goal. The backup and restore functionality to ensure recoverability is integrated into the VPFS architecture such that unrelated TCBs grow only moderately: in the order of 400 lines of code in the case of confidentiality and much less for integrity, if the privilege separation as outlined in Section 6.4 is used. VPFS distributes backups to multiple, independently operated servers in order to limit the damage that both malicious insiders and temporary server failures can cause to remotely stored backups. Self-attesting erasure coding enables VPFS to create backups with low space and bandwidth overhead, while keeping most of the complexity of coding schemes out of the recoverability TCB: only 45 lines within VPFS BackupD are required to verify that incremental backups encode a correct and complete delta backup with sufficient redundancy.

The following chapter evaluates the overall complexity as well as the security and performance properties of the VPFS prototype that I described in Chapters 3 through 6.

Chapter 7

Evaluation

This chapter evaluates the complexity and performance of a VPFS prototype implementation that I built on top of a microkernel-based operating system. As improving the security of file-based storage is the main motivation for VPFS, I review the gains in security that VPFS offers for the protection goals of confidentiality, integrity, and recoverability. This part of the evaluation, presented in Sections 7.2 and 7.3, focusses on complexity metrics for individual components of VPFS and the attack surface exposed by these components. In the second part of the evaluation in Section 7.4, I discuss run-time performance based on microbenchmarks and application workloads. This section also covers robustness against crashes.

In order to put the complexity discussion and performance measurements into perspective, I first give an overview of the experimentation platform.

7.1 Experimentation Platform

The general architecture of VPFS is agnostic to the underlying system platform. However, a specific implementation must be tailored around the primitives and mechanisms that the operating system (OS) provides. In the following, I give an overview of the research OS platform and briefly describe the platform-specific parts of the VPFS prototype.

Operating System Platform. I implemented VPFS on top of a multi-server OS based on Fiasco.OC [51], which is a member of the L4 family of microkernels. System services of the L4 run-time environment (L4Re) [123] run in user mode, as do applications. The kernel ensures strong isolation between user-level programs using address spaces and a capability-based mechanism for restricting inter-process communication (IPC) [123]. Programs executing in different address spaces (e.g., an application and a system service such as VPFS Core) can only communicate with each other, if they are given *capabilities* for kernel-provided communication facilities: *IPC gates* for synchronous message passing, or *IRQ objects* for asynchronous notification. Two programs can establish shared-memory regions by transferring capabilities for page mappings *(flexpages)* through an IPC gate. The receiving program must agree to receive mappings in a designated region of its address space. Therefore, even mutually distrusting programs may use the mapping mechanism for securely enabling zero-copy communication via shared memory.

Commodity File System and VPFS Helper. Fiasco.OC and L4Re support hosting of virtual machines (VMs), including paravirtualized Linux guests. VPFS reuses the file system stack provided by a paravirtualized L^{4}Linux [36] kernel that runs on top of the microkernel platform. Instances of the untrusted VPFS Helper run as Linux user-mode processes and access the commodity file system through Linux' VFS interface. An L^{4}Linux-specific kernel module provides a shared memory area that is accessible by both a VPFS Helper process and the corresponding VPFS Core instance. The module also enables two-way signaling by linking `trigger()` and `wait()` operations on Fiasco.OC IRQ objects (directly accessible to VPFS Core) to `read()` and `write()` system calls on a Linux character device that VPFS Helper can access.

Note that this configuration was chosen for the prototype implementation, but does not limit the applicability of the VPFS architecture to virtualization enabled platforms. The untrusted commodity file system could also be a standalone component such as a user-level server that is part of the microkernel-based OS.

VPFS Core and Virtual File System. The implementation of VPFS Core is provided as a library; a server program that links against this library provides VPFS Core's functionality to an application through an IPC gate and shared memory. Any application that has a capability for invoking the server's IPC interface can use the VPFS instance. Such an application performs file and directory operations via L4Re's generic virtual file system (VFS) infrastructure, for which I wrote a plugin that calls the VPFS Core service via IPC. Both stream-oriented and memory-mapped file I/O are supported.

VPFS BackupD and Backup Codecs. It is also possible to link VPFS Core directly into an application. I used this approach for VPFS BackupD in order to use VPFS Core's internal, inode-based naming interface. VPFS BackupD also implements a version-based block retrieval strategy using low-level APIs of VPFS Core (see Section 6.5). VPFS BackupD can create cryptographically protected, incremental backups with the minimal amount of data needed to update the oldest backup on any of the servers. Redundancy is supported using either replication or erasure coding based on the Cauchy Reed Solomon [138] algorithm. The backup codecs needed for self-attesting erasure coding run as separate L4Re processes and can only communicate with VPFS BackupD.

Prototype Limitations. VPFS Auth, the component that forwards backup streams and ensures cryptographic transport security, and VPFS RecoveryD are currently not implemented. The complexity analysis for the former is based on estimates of expected code size assuming that the TLS implementation from MatrixSSL [38] is used; for VPFS RecoveryD, estimates are based on functionality already implemented in VPFS BackupD.

User interfaces required for initial setup or recovery from backup are not part of the prototype (e.g., for choosing a specific backup version to restore or for entering credentials as proposed in Section 6.8). The complexity of user-interface components depends on libraries and services from parts of the system platform that are out of scope for this thesis. However, I assume that user-facing parts of VPFS BackupD and VPFS RecoveryD are isolated from untrusted applications and services in the same way as the threat model assumes for ordinary applications (see Section 2.4).

Platform Limitations. Support for sealed memory is currently not available on Fiasco.OC and L4Re and is therefore not considered in this evaluation. The VPFS prototype includes dummy code at those points in VPFS Core where sealed-memory APIs would be called in a complete implementation; I assume a platform service as described in Section 2.3. VPFS also requires high-quality random numbers for generating cryptographic keys and for checking redundancy of self-attesting erasure coding. There is currently no such platform service available for L4Re; the prototype uses a pseudo random number generator that is insecure, but suffices for proof-of-concept testing.

Apart from the limitations described above, the evaluation prototype of VPFS has been implemented as described in Chapters 3 through 6.

7.2 Code Size and Complexity

This section evaluates how complex the trusted parts of VPFS are in comparison to a monolithic commodity file-system stack. I first focus on the TCBs for confidentiality and integrity; complexity of functionality required for recoverability is discussed in Section 7.3.

Untrusted Reuse. The key strategy to avoid complexity in the TCB is to reuse untrusted infrastructure for non-critical operations. The commodity file system and the storage layers underneath perform a substantial amount of work that is essential to persistently storing data. These untrusted parts of the storage stack may also implement optimizations and strategies to improve fault tolerance. In particular, the VPFS architecture removes the following functionality from the TCB of applications that depend on file-system support:

- Provision of persistent containers for cryptographically-protected data and metadata
- Efficient and safe block allocation, free-space tracking, and garbage collection for logs
- Optimization of data transfers (e.g., request batching or reordering of read and write requests)
- Enforcement of correct write order to meet consistency requirements
- Scalable concurrent access to shared storage devices for multiple applications
- Fault tolerance in the event of failing storage devices (e.g., replication, bad-block lists)
- Network transport for storing backups on remote servers

Moving a large body of complicated functionality into the untrusted part of a split code base reduces the size of the security-critical part and makes it less complex. However, to meet security requirements despite splitting, it might be necessary to (1) use cryptography, (2) perform additional checks, or (3) duplicate functionality that cannot be reused securely. The trusted VPFS components include logic for all three of these security measures. The motivation for building VPFS using the trusted-wrapper approach (Sections 1.2 and 1.3) was that the security-critical part of the split implementation would be smaller and less complex than a monolithic file-system stack. To validate this assumption, complexity metrics can be used.

Quantification of Complexity. A simple and widely used metric for assessing the complexity of software is code size. The size of a program can be determined based on compiler output (i.e., object code), which is either measured in bytes or number of instructions. However, programmers usually understand this metric more easily, when it is derived from source code. This representation of software is what they are most familiar with. The metric is commonly given as number of lines of code or number of statements. When computed for source code, the size metric also includes declarations of data types and other non-functional pieces of code. These parts of a program's source may be easier to write and understand than the logic of a complicated algorithm, yet both have the same weight in the metric. Cyclomatic complexity [130] is a metric that attempts to quantify algorithmic complexity. The cyclomatic number is derived from the number of decision points that can divert the control flow through a program.

Both code size and cyclomatic complexity are well-established metrics that programmers can easily understand. Therefore, I will use them to evaluate the complexity of VPFS and its dependencies.

7.2.1 Code Size

I start out with a presentation of SLOC numbers for all security-critical VPFS components; the figures quantify how much code they contribute to the TCB. This presentation includes a breakdown of components into their individual subsystems and, in some cases, even specific functionality within a subsystem (e.g., code within Sync Manager that ensures robustness against crashes). SLOC numbers for individual parts of the VPFS code base are directly comparable, because they use the same coding style. I also present figures for library dependencies caused by VPFS and for the Linux storage stack. The size of third-party libraries and Linux subsystems can be compared to VPFS only roughly due to different programming styles and languages.

Note that the evaluation of code size only covers file-system related code or library dependencies that are caused by VPFS in order to implement a file-based data store. The C library, system-call

Subsystem	SLOC
L4Re VFS	**3,130**
VPFS Core	**3,102**
Basic MemFS	1,738
Naming + directories	305
Lookup cache	100
Persistency	438
Robustness	384
IPC layer	137
Cryptography library	**1,208**

(a) Code sizes for security-critical parts of VPFS stack required to serve application requests

Subsystem	SLOC
VPFS Helper	**3,119**
Linux storage stack	**100,000+**
VFS + generic routines	41,621
Specific file system, e.g.:	
Ext4 + JBD2	32,081
XFS	61,740
Block layer + CFQ + EFI	14,220
SATA support (libata)	12,064
Generic AHCI driver	2,953

(b) Code sizes for untrusted parts of VPFS storage stack, consisting of VPFS Helper and Linux 3.5

Table 7.1: Code size of trusted and untrusted components in a VPFS stack.

bindings, or generic frameworks that any application depends on are not counted, because they are already part of the TCB. Furthermore, I removed any debugging code and instrumentation for benchmarks from the VPFS code base.

Table 7.1a shows the amount of code that VPFS adds to an application's TCB for confidentiality and integrity. The code-size figures were generated using David A. Wheeler's 'SLOCCount' [49]. This tool considers all content of a source file that is actual source code, but not blank lines or lines that contain only comments. The following paragraphs discuss the code size of all parts of the VPFS stack in top–down order.

Virtual File System Layer. Applications access the file system via a file system API provided by a generic virtual file system. L4Re VFS enables applications to use various file-system implementations, including, but not limited to VPFS. L4Re VFS and the IPC code that forwards requests to VPFS Core add approximately 3,100 lines of code to an application's TCB. The benefit is that applications can use a familiar POSIX-like API via a standard C library; L4Re VFS maps POSIX-level operations to the basic file-system operations offered by VPFS Core. However, applications that are tailored towards the minimal VPFS interface may also access their instance of VPFS Core with a stripped down VFS [156] implemented in the order of few hundred lines of code.

Memory-Based File System. More than half of the code lines in VPFS Core (1,738 SLOC) are required to implement a memory-backed file system. It provides an abstraction of dynamically growing and shrinking files in a flat namespace. Another 400 lines add hierarchical directories and a simple lookup cache, which improves efficiency of namespace operations. As explained in Chapters 3 and 4, it is necessary to include abstractions of files and naming into the TCB. Otherwise, confidentiality and integrity of file contents or the directory hierarchy cannot be guaranteed. Fortunately, this functionality adds only slightly less than 2,200 lines of code to an application's TCB.

Persistency and Robustness. The persistency and IPC layers of VPFS Core extend the in-memory file system into a trusted wrapper that can reuse an existing, untrusted storage stack. In total, the persistency subsystem comprises about 800 SLOC. Within this subsystem, 438 lines enable VPFS Core to store cryptographically protected data and metadata in the commodity file system. The code that enables fault tolerance in addition to basic persistency contributes 384 SLOC, which include cryptographic protection of the journal. This number compares favorably against similar subsystems in a monolithic Linux storage stack, where crash-recovery features typically require an order of magnitude more code. For example, the journal block device layer (JBD2) for Ext4 comprises almost 5,000 SLOC in Linux 3.5; the journaling functionality in ReiserFS comprises more than 3,000 SLOC. In VPFS, these complex subsystems can be reused without trusting them.

Cryptography Libraries. VPFS Core depends on cryptographic library routines to deliver on its confidentiality and integrity guarantees. The implementations of AES [34] and SHA-1 [33] chosen for the prototype comprise approximately 1,200 lines of C and x86 assembly code. However, the amount of code that developers write to implement these algorithms varies considerably. For example, certain implementations include hundreds of lines of pre-computed tables in the source code, while others generate them at initialization time; some developers also chose to manually unroll loops so as to improve performance. Earlier versions of VPFS [156] used more compact implementations, which contribute about half the number of lines to the TCB. For the current version of VPFS, I used off-the-shelf code from the Linux kernel tree that performs more efficiently on the evaluation hardware described in Section 7.4.

Reused and Untrusted Infrastructure. The use of cryptography serves a second purpose in addition to preventing attacks on stored file-system contents: encryption and MAC-based integrity checking are essential to delegating work to untrusted parts of the storage stack. Table 7.1b shows that these untrusted parts make up the majority of the code base. While the untrusted VPFS Helper comprises only about 3,100 SLOC, the size of a typical Linux file-system stack underneath is in the order of a hundred thousand lines of code.[1] Thus, the TCB for confidentiality and integrity of VPFS is more than an order of magnitude smaller than a monolithic commodity file-system stack.

A significant reduction of code size is also possible for specific features. For example, TCB enlargement caused by crash robustness has been achieved through two complexity-avoiding design decisions (details in Chapter 5). First, operation-level journaling in VPFS Core simplifies both state tracking for consistency enforcement and replay: Sync Manager can reuse VFS-level APIs to roll forward incomplete metadata updates after a crash or power loss; for each type of journal record, between six and 21 SLOC are needed for replaying the corresponding operation. Second, VPFS Core delegates work to untrusted infrastructure in order to keep necessary, but not security-critical complexity out of the TCB. Complicated tasks such as garbage collection and ordered writes can be handled by the untrusted VPFS Helper and a robust, but untrusted commodity file system stack.

The SLOC figures show that the trusted-wrapper approach used for VPFS can drastically reduce the amount of code that a file system adds to an application's TCB – the trusted parts of the VPFS stack are more than one order of magnitude smaller than Linux' monolithic file system stack. Such a significant reduction of size suggests that a smaller number of implementation and design defects exist in the security-critical code base. Assuming that the defect rate [98, 111] for the TCB is similar to the defect rate of the larger untrusted code base, one can expect that the VPFS architecture does indeed improve security by reducing the number of potential security vulnerabilities.

7.2.2 Functional Complexity

The cyclomatic complexity metric (CCM) was introduced by McCabe [130] as a measure of control-flow complexity. It can also be understood as *conditional complexity*, as this metric is derived from the number of conditional branches that can influence program execution. A program with no

[1] The Linux figures are based on Linux version 3.5. Code sizes for the VFS layer and generic implementations of file-system functionality were generated from all C source and header files in the directory 'fs' of the kernel source tree. All subdirectories containing specific file systems implementations were excluded from this measurement, but the tables gives numbers for select file systems. The block layer is assumed to use the CFQ I/O scheduler and EFI partition table support.

conditional statements will execute sequentially and thus has a cyclomatic complexity of 1; such a program is considered the simplest under CCM. Adding, for example, one `if–else` statement to the program increases its cyclomatic complexity to 2, because the control flow can take either the `if` branch or the `else` branch. Since more conditions require the programmer to consider more potential control flows, programs with larger CCM values are regarded more complex.

'The tool 'pmccabe' [45] determines the cyclomatic complexity for all subroutines of a program written in C or C++. Thus, in the case of the VPFS file-system stack, subroutines are either C functions or methods of C++ classes, including constructors and destructors; I will use the term *functions* to refer to these parts of the code in the following.

Average Complexity. The second column of Table 7.2 lists the average cyclomatic number computed over all functions of each subsystem. The data shows that functions within trusted VPFS components are less complex than functions in a Linux commodity file-system stack: VPFS Core and L4Re VFS have a low average CCM value of 2.0, whereas functions within Linux are up to three times as complex under the metric. Functions from the erasure-coding library Jerasure and the TLS implementation in MatrixSSL are, on average, even more complex: they have CCM values of 8.0 and 9.2, respectively. A review of the Linux code base and the libraries [35, 38] reveals that there are two main reasons for their higher per-function complexity:

1. Functions are typically longer than in trusted VPFS components and they contain a larger number of conditions. The Ext4 and XFS file system code bases even contain functions with CCM values in the hundreds, which is considered highly problematic [130]. The most complex function in VPFS Core is, by far, the journal replay loop with a cyclomatic number of 47.

2. The trusted VPFS components consist of a larger number of functions that execute sequentially. In the case of VPFS Core, 64 percent of all functions have a CCM of 1 (many of them are trivial accessor methods for private class members).

Arguably, breaking up functionality into smaller subroutines improves readability and maintainability of code, but a lower complexity per function does not allow drawing conclusions about the overall complexity of the complete code base. However, McCabe's metric is defined to be *adding* for interconnected subroutines or modules of a program [130]. The total complexity of each part of the VPFS stack can therefore be computed by summing up CCM values of all functions.

Overall Complexity. The third column of Table 7.2 lists the aggregated cyclomatic complexity for trusted VPFS components, third-party libraries, and Linux subsystems. The figures show that the security-critical parts of the VPFS stack are significantly less complex than the reused Linux subsystems that need not be trusted: L4Re VFS and the prototypes of VPFS Core and VPFS BackupD increase cyclomatic complexity of an application's TCB by no more than $1,600$. In contrast, the untrusted code bases are an order of magnitude more complex according to the metric: a configuration of the Linux file-system stack based on NILFS2 has an aggregate CCM value of about $15,000$. If XFS were used, the cyclomatic complexity would exceed $22,000$.

Note that encryption routines and a collision-resistant hash function must be included in the TCB for confidentiality and integrity as well. The table does not list CCM figures for these algorithms. The reason is that the cryptography library used for the prototype (as described in Section 7.2.1 on page 97) contains an implementation of AES that is written in assembly language. Unfortunately, complexity measurement tools like 'pmccabe' can operate only on code written in high-level languages. To provide an estimate of the complexity caused by the cipher and hash function, I measured C-only implementations of AES and SHA-1. Their code bases increase cyclomatic complexity by 74, which is only a fraction of the overall complexity added by VPFS. It is reasonable to assume that the complexity of different implementations is comparable under CCM (i.e., in the same order of magnitude). Under this assumption, the isolated and trusted parts of the

Subsystem	**CCM** **(avg/func)**	**CCM** **(all func)**	**STM** **(all func)**	**STM$_{\text{cond}}$** **(avg/kSTM)**
L4Re VFS	2.0	624	1,469	214
VPFS Core	2.0	863	1,790	244
VPFS BackupD	2.6	101	286	217
Jerasure	8.0	491	1,641	262
MatrixSSL	9.2	2,677	7,850	304
Linux				
VFS + generic routines	4.6	8,868	26,507	261
Ext4 + JBD2	6.0	6,080	20,018	253
NILFS2	3.8	2,411	8,710	204
XFS	6.2	10,033	33,297	253
Block layer + CFQ + EFI	3.6	2,993	8,022	269
Generic AHCI driver	4.5	435	1,401	241

Table 7.2: Complexity of all code within function bodies for VPFS components, their library dependencies, and Linux storage subsystems: the number of statements (STM) is given, as well as cyclomatic complexity (CCM) within all functions, per 1000 statements, and on average per function.

VPFS stack – including trusted-wrapper logic and cryptography – are still an order of magnitude less complex than a monolithic commodity file-system stack.

Complexity Density. A third way evaluate to code complexity under CCM is to look at how often conditional statements appear in the code base. I first determined the total number of statements (STM) that appear in function bodies (see fourth column of Table 7.2). The STM figures do not include code for global declarations of datatypes or syntactic artifacts such as `#include` statements. Thus, they describe the size of the program logic within individual subsystems, whereas SLOC is computed over all lines of the source code. The figures show that the number of function-related statements in trusted and untrusted parts of the VPFS stack differ by more than an order of magnitude. This observation is in line with the code-size measurements discussed in Section 7.2.1.

Based on the STM measurements, I computed the density of statements that can influence control flow. This number is a lower bound for cyclomatic complexity, which is additionally increased by 1 for each subroutine of a program. The fifth column of the table lists the results as average number of conditional statements per 1,000 statements. Across all code bases – both trusted and untrusted ones – the density of potential branching points is roughly the same, ranging from about 200 to 300 and centered around 250 STM$_{\text{cond}}$ per 1,000 STM. From these figures, one can draw two conclusions: (1) there is no significant difference in complexity for code bases of similar size, and (2) code size can be used as a rough estimator for the overall complexity of the differently trusted parts of the VPFS stack.

The second conclusion above is clearly valid when the difference in code size is huge. Thus, the VPFS architecture does indeed reduce complexity. However, the implicit working theory for the analyses in the preceding three paragraphs was that CCM figures for code bases written in C++ (L4Re and VPFS) and code written in C (Linux and libraries) can be compared in a meaningful way. A factor that potentially limits comparability is the use of C++ exceptions for error handling. Unfortunately, this programming concept was not considered when CCM was originally defined and the 'pmccabe' tool ignores exception-related statements as well. This weakness can be addressed by preprocessing the C++ sources. For completeness, I outline below the methodology I used to measure cyclomatic complexity of C++ code with exception handling.

C++ Exceptions and Cyclomatic Complexity. To capture cyclomatic complexity hidden in exception handling, I ran a preprocessor tool on the C++ source. It transformed exception-related statements into dummy statements that increase the cyclomatic number of a function according to the control-flow semantics of exception handling. Two types of transformations were done:

1. Each `catch` clause in a function increases the cyclomatic number of this function by 1, as do `if` statements or `while` loops. All `catch` statements are therefore replaced by a `while` statement, which includes a condition to adjust the computed CCM value as required.
2. Each point in the code from which an exception can originate increases the complexity of the surrounding function by 1. Explicit `throw` statements in `if` or `else` branches require no transformation. However, both L4Re and VPFS make extensive use of special inline functions, to which they pass results of computations and error codes; these `check_*()` functions throw an exception, if they are passed an invalid result (e.g., a `NULL` pointer or a negative system-call return code). To make such hidden control-flow decisions visible to 'pmccabe', each invocation of the `check_*()` function is modified to include a dummy conditional expression encoded as an additional parameter.

The preprocessed code no longer compiles, but 'pmccabe' is able to process it as intended. All figures discussed earlier were computed on code preprocessed as described above. Note that invocation of a function that merely passes on an exception (i.e., thrown further down in the call stack) is not counted as a branching point. This behavior is justified for both L4Re and VPFS code bases, because the general programming style explained in Section 3.2.3 demands that (1) temporarily used resources are always deallocated by destructors of guard objects, or (2) `catch` clauses handle any additional cleanup of state outside the current stack frame. Guard objects increases code size, but not cyclomatic complexity.

7.3 Separation of TCBs and Attack Surface

The extended VPFS architecture adds support for recoverability in the form of five additional components: (1) VPFS BackupD for creating backups, (2) VPFS RecoveryD for restore operations, (3) VPFS Auth for ensuring secure communication with backup servers, and (4) the backup encoder and (5) decoder needed for self-attesting erasure coding.

7.3.1 Size and Complexity of the Recoverability TCB

VPFS BackupD is most critical to minimizing the impact that the added functionality has on the TCBs for confidentiality and integrity. I fully implemented this component and the backup codecs such that the recoverability-related impact on size and complexity of previously discussed TCBs can be determined precisely. For VPFS RecoveryD and VPFS Auth, I give code-size estimates; their functionality is only security-critical with regard to backup and restore.

VPFS BackupD. The implementation of VPFS BackupD comprises 434 source lines of code; see Table 7.3a for a breakdown by functionality. This component provides the functionality that is required for creating a backup stream that encodes all file-system modifications since the latest reported backup on any of the reachable backup servers. The code base also includes 41 lines of IPC-client code needed for cooperation with both VPFS Auth and the backup codecs. Approximately 40 SLOC of the code base implement the `Modified_blocks_strategy` class in VPFS Core, which is co-located with VPFS BackupD.

Only 45 lines of C++ code are dedicated to the comparator and stream-forwarding logic of self-attesting erasure coding, which I described in Section 6.6.4. This small amount of code is all that is needed to verify that (1) the erasure coded backups distributed to n servers encode a correct and complete backup, and (2) that this backup includes sufficient redundancy that it can be recovered, if at least m of the n servers ($m < n$) are available at recovery-time. The erasure-coding

Subsystem	SLOC
VPFS BackupD	**434**
Backup Creation + IPC	389
Codec Verification	45
Jerasure	**2094**

(a) Code sizes for VPFS BackupD (included in at least two TCBs) and the erasure-coding library (untrusted).

Subsystem	SLOC
VPFS Auth	**14,000+**
Backup Forwarding	1,000+
Transport Security (TLS)	14,198
VPFS RecoveryD	**300+**

(b) Estimates for code sizes of VPFS Auth and VPFS RecoveryD (both part of recoverability TCB).

Table 7.3: Code sizes of trusted and untrusted components and library dependencies of the extended VPFS architecture supporting recoverability.

library Jerasure, which consists of about 2,100 lines of C code need not be part of the recoverability TCB, nor has its complexity any impact on the TCBs for confidentiality and integrity. Thus, self-attesting erasure coding reduces the size of code needed for space and bandwidth-efficient backup to multiple servers by almost two orders of magnitude. The implementation of the untrusted backup codecs utilizing Jerasure consists of about 400 lines of code.

VPFS Auth. The impact that VPFS Auth has on code size is much larger: using the implementation of Transport Layer Security (TLS) [64] in MatrixSSL, the recoverability TCB is enlarged by more than 14,000 lines of C code. In Section 6.2, I explained in detail why implementations of cryptographic protocols such as TLS cannot be split. Furthermore, the complexity figures listed in Table 7.2 on page 99 indicate that implementations of such protocols are highly complex, in this case more than three times as much as VPFS under the CCM metric. Beyond the ability to secure communication between VPFS BackupD and the servers, the Auth component must also include functionality for three other tasks:

- Forward backup version numbers between VPFS BackupD (or RecoveryD) and one server
- Forward new backup streams to one server or receive a stream during recovery
- Instruct one server to either commit or drop a newly sent backup or request a specific backup

I estimate that adding this functionality to VPFS Auth – for example, in the form of a minimalist HTTP client – contributes an order of magnitude less code than the cryptographic protocol. I therefore conclude that size and complexity of a fully functional version of VPFS Auth will be dominated by TLS or a similar protocol suite. Access to network sockets can be provided by the untrusted VPFS Helper in a way similar to previous work [147, 113].

VPFS RecoveryD. I estimate that a working implementation of VPFS RecoveryD will have a size and complexity that is similar, possibly smaller, than that of VPFS BackupD. This estimate is well-founded, because most of the functionality needed to receive a backup via the VPFS Auth instances is a subset of what is currently implemented in VPFS BackupD (see Section 6.7).

The code-size estimates for VPFS Auth indicate that the recoverability TCB should be separated from the TCBs for confidentiality and integrity. The following discussion of different types of attack surfaces supports this conclusion.

7.3.2 Size and Complexity of the Attack Surface

The fundamental assumption of the VPFS threat model is that the underlying system platform isolates the TCB using address spaces and other protection mechanisms. The application is part of the TCB. However, as VPFS depends on untrusted code bases, it exposes itself to parts of the system that may be under control of an attacker (see Section 2.4). The size of a security-critical code base marks an upper bound for the size of the attack surface. Unfortunately, those several thousand lines of code that VPFS contributes to an application's TCB are still significant. However, design principles such as pervasive integrity checking (Section 3.2.1) and simple inter-component interfaces (Sections 3.2.2) help to shrink down the attack surface considerably, or limit the kind of attacks that remain possible. The two principles limit the abilities of the attacker as follows:

1. **Pervasive Integrity Checking:** Validation of message authenticate codes (MACs) by trusted VPFS components reveals any corruption of data or metadata that VPFS Core reads back from persistent storage. As a result, an attacker cannot manipulate input to file-system code layered above VPFS Core's persistency layer.

2. **Simple Inter-Component Interfaces:** An attacker in control of untrusted parts of the system can send arbitrary information to the security-critical side of a VPFS interface. The strategy to defend against successful chosen-input attacks is to keep the interface implementation small and simple, such that it can thoroughly validate the correctness of all replies, before any data is passed on to program logic behind the interface.

The security-critical part of the interface implementation and the input validation logic represent the *fully-exposed attack surface* of a VPFS component. Only through abuse of this interface can the attacker chose input freely. Beyond that, his ability to influence computations and control flow in a TCB is limited to aborting operations (e.g. reads of uncached data) by sending either forged replies or no message at all. Therefore, those parts of the trusted code base behind the line of defense marked by interface and input validation represent the denial-of-service attack surface.

Size and complexity of the combined attack surface (i.e., complete components) have already been evaluated on the preceding pages of this chapter. In the following, I discuss the fully-exposed attack surface of VPFS components.

VPFS Core. In the basic architecture without recoverability support, VPFS' fully-exposed attack surface is restricted to those parts of the persistency layer, through which VPFS Core cooperates with the untrusted VPFS Helper. This part of the attack surface includes IPC-related code and all functions that check integrity of data blocks, scan block-descriptor tables in shared memory, or validate and decrypt journal records. Table 7.4 shows the size and complexity measurements for these parts of the code in the first row (labeled "VPFS Core IF"). Apart from previously discussed decryption and hashing algorithms, the interface implementation and input validation comprise about 600 lines of code. Up to 330 statements in function bodies potentially process input that was chosen by the attacker.

VPFS BackupD. In the extended architecture, VPFS BackupD is added. It exposes an interface to three other components: (1) the black-box backup encoder (for either replicated or erasure-coded backups), (2) the decoder that extracts the original backup stream from the encoded backup streams (for self-attesting erasure coding), and (3) an instance of VPFS Auth for each backup server. All three components receive backup (or fragment) streams; the decoder and the VPFS Auth instances also send back a MAC of the streams they received. VPFS BackupD uses the same interface implementation for all communication with these components. In addition to the codebase of this interface, the fully-exposed attack surface also includes parts of the comparator logic for self-attesting erasure coding.

The second row of Table 7.4 shows SLOC and CCM complexity numbers for the directly exposed functionality (as determined by code review). With 235 lines of code and an aggregated cyclomatic complexity of 58, the directly exposed attack surface of this components is less then half as complex as VPFS Core's under both SLOC and CCM metrics (about 40 percent each). On the other hand, the size of the entire codebase of VPFS BackupD is only about 14 percent that of VPFS Core.

VPFS Auth. As for the protection goal of recoverability, the attack surface is significantly larger. As a rough estimate – and without intimate knowledge of the specific implementation of the

Subsystem	SLOC (all code)	CCM (avg/func)	CCM (all func)	STM (all func)	$\mathbf{STM_{cond}}$ (avg/kSTM)
VPFS Core IF	576	2.3	142	330	239
VPFS BackupD IF	235	2.6	58	158	228
VPFS Auth IF (est.)	14,198+	9.2	2,677	7,850	304

Table 7.4: Complexity of fully-exposed parts of security-critical interfaces and input validation code in trusted VPFS components.

transport-security protocol being used – one must assume that all code related to cryptographic protection of client–server communication is directly exposed. The figures in the last row of Table 7.4 labelled "VPFS Auth IF" indicate that the TLS state machine in MatrixSSL is much more complex than the interface logic in the other two trusted VPFS components; cryptographic algorithms included in the library account for only about 8,200 SLOC and have an aggregated CCM of approximately 1,500. I expect that the additional complexity added by backup forwarding and a minimalist HTTP client library (only GET and PUT requests with a small set of agreed-upon headers) is dwarfed by the TLS implementation.

Two conclusions can be drawn from the results I presented in the preceding three paragraphs:

1. Adding support for recoverability has a disproportionate impact – relative to code size – on the amount of code that an attacker can feed arbitrary input. This observation is of importance for defending against attacks on confidentiality, because neither the backup codecs nor VPFS Auth are part of the TCB for this protection goal. A potentially negative effect on integrity depends on the privileges of VPFS BackupD itself (see Section 6.1).
2. Adding support for recoverability increases the size of the fully-exposed attack surface dramatically. It is therefore desirable from a security point-of-view that TCBs for confidentiality and integrity and the security-critical code base for recoverability are separated.

The design and implementation of VPFS BackupD that I described in Chapter 6 enable strong isolation of these different types of TCBs with small overlap: only VPFS BackupD must be included in both security domains. The architecture allows to remove most of the complexity related to efficient multi-server backup and transport security from confidentiality and integrity TCBs by treating VPFS Auth and the backup codecs as *untrusted* components with regard to these protection goals.

This discussion of the size and complexity of the attack surface concludes the evaluation of VPFS's security properties. I will now evaluate the efficiency of the VPFS prototype implementation.

7.4 Efficiency

The VPFS architecture improves security through two key measures: (1) it isolates trusted from untrusted code via address spaces, and (2) it uses cryptography in order to enable untrusted reuse. Separation of code bases adds run-time overhead due to context and address-space switching, both of which are unnecessary in monolithic architectures. The trusted-wrapper approach also requires additional checks and cryptographic operations can degrade performance even more. While widely-deployed mobile-device platforms such as Android already use encryption to protect confidentiality of data at rest, VPFS also uses MACs in order to guarantee file-system integrity. Therefore, VPFS may perform slower than file systems that offer fewer and weaker security guarantees.

This section evaluates the cost of strong security in VPFS. The performance evaluation is based on a set of microbenchmarks and application workloads. Before discussing the results, I first describe the test hardware and software configuration.

7.4.1 Hardware and Software Configuration

The evaluation hardware is a system with an Intel Atom D2550 CPU. This processor has two cores clocked at 1.86 GHz and two logical CPUs per core. Atom CPUs have long been used in cheap laptop computers known as "netbooks" and are now appearing in mobile devices such as smartphones and tablets. The system is equipped with one 4 GB module of DDR3 memory. I used two solid-state drives (SSDs) for the benchmarks:

- **Fast SSD:** An Intel 320 Series SATA solid-state drive (SSDSA2BW120G3) with 120 GB capacity is used as a representative of fast, flash-based storage with both high sequential and random I/O throughput.
- **Slow SSD:** A Corsair Voyager GT thumb drive connected via USB 2.0 is used as a representative of slow flash-based storage. This storage device achieves moderate throughput, but shows slow performance for small random writes.

Note that the fast SSD can transfer data at higher rates than the evaluation system's CPU can perform cryptographic operations on the blocks being read or written. I include this configuration in order to give an estimation of how VPFS performs in a device with high-performance storage. The slow SSD resembles both the type of storage that manufacturers build into mobile devices today and the memory cards that users can use to add more space for their data [121].

Choice of Commodity File Systems. I benchmarked VPFS configurations with two different commodity file systems enabled in L^{4}Linux: Ext4 [32] is used on the fast SSD, while NILFS2 [41] was chosen for the slow flash-storage device. I found that using Ext4 on this device would increase run times for some of the workloads discussed in Section 7.4.3 by several hundred precent. A log-structured file system such as NILFS2 performs significantly better, because it translates logically distributed write accesses into sequential writes to the storage device. This observation is in line with a recent study on storage performance in smartphones [121].

Baseline and Benchmark Setup. All benchmarks were also done on monolithic file-system configurations: in this case, the workloads were executed directly on the commodity file system provided by the L^{4}Linux kernel. The commodity file-system stack was configured to use dmcrypt for transparent block-level encryption, representing a system that offers guarantees for file-system confidentiality, but not integrity. VPFS and dmcrypt use the same AES implementation in order to make benchmark results comparable.

Benchmark applications and the trusted parts of VPFS ran on the first processor core. VPFS was assigned 512 MB of memory for its trusted buffer cache. The untrusted VPFS Helper ran inside the L^{4}Linux VM, which was given the second CPU core. The VM had 200 MB of free memory available for the commodity file system's inode, directory, and page caches. In the L^{4}Linux/dmcrypt configuration, the commodity OS could use both CPU cores and additional 512 MB of memory. Thus, both configurations have about 700 MB of memory for caches; this amount is arbitrarily chosen, but could be found in a real mobile device. Hyper threading was disabled in all configurations.

Benchmark Methodology. Unless stated otherwise, all results presented on the following pages are the average of ten iterations of the respective benchmark. Error bars in the diagrams give the standard deviation across all runs of a particular benchmark. All of the four configurations started from a fresh file system after boot. A script then executed each benchmark ten times, before continuing with ten runs of the next workload. Before each iteration of a benchmark, an *init program* crawled the directory tree and reverted it to the initial state expected by the respective workload. Once the init program recreated all missing files and directories and deleted any leftovers from the previous benchmark, it instructed the file-system stack to flush all dirty buffers and drop all caches afterwards. Thus, each benchmark iteration started with cold caches, including VPFS Core's buffer cache and the inode, dentry, and page caches of the commodity file system.

Running all benchmarks without reboot put load on the file system for several hours. Consequently, the total amount of data written by all benchmarks exceeded the capacity of the slow storage device. Thus, the garbage collector of NILFS2 needed to run several times. The benchmark script ensured that garbage collection was active only during clean-up phases between benchmark iterations. Once the NILFS2 cleaner freed enough disk space for the next benchmark to complete, the script paused garbage collection again. This approach is acceptable, because continuously running benchmarks is an unusual workload. A production system that uses a log-structured file system would garbage collect (partially) unused segments during idle times.

Workload Traces. I benchmarked the four configurations of storage device and file-system stack by measuring the time needed to replay traces of file-system operations. These traces were captured from benchmarking tools and complex applications that ran on two commodity OSes (Linux and Mac OS X). I use traces in this evaluation for two reasons:

1. Porting large applications to the Fiasco.OC/L4Re research platform is difficult, and in certain cases impossible, because their source code is unavailable.
2. Trace-based file-system benchmarks do not suffer from measurement noise caused by the application, whose performance may vary when run on different OSes.

The *VPFS trace player* used in the experiments relies on the POSIX file-system interface and can be compiled either as a Linux program or as a native L4Re application. All operations of a particular trace are encoded in a static data structure within the binary's data segment; thus, one binary is generated for each trace. Both on Linux and on L4, trace-player binaries reside in a RAM disk in order to eliminate any I/O operations except those initiated by trace playback on the target file system. Additionally, the trace data is encoded in a custom data structure that can be parsed without any library code, such that different implementations of standard containers or memory management do not influence benchmark results. The trace player executes all operations as quickly as possible and without delay.

I present benchmark results for microbenchmarks in the next section. Traces of application workloads are covered in Section 7.4.3.

7.4.2 Microbenchmarks

The microbenchmarks test homogeneous workloads: (1) sequential reads and writes, (2) creating and reading a large number of files, (3) bursts of operations on many files that fit into the buffer cache. These benchmarks[2] were originally run on a Linux system. All file-system calls performed by the programs were recorded using 'strace'.

Read Throughput. Bonnie++ [1] is a standard benchmark. I recorded two traces, which measure throughput when sequentially reading or writing two 1 GB files. Figure 7.1 shows measured read and write bandwidth in MB/s. As expected, VPFS reads large files slower than the monolithic file-system stack: I measured 22 and 18 percent overhead for VPFS on the fast and the slow SSD, respectively. Read performance is limited by CPU speed, as VPFS Core's Sync Manager must decrypt all blocks and compute MACs. Cryptographic operations are also the bottleneck for the dmcrypt-based configurations, where the L^{4}Linux kernel must decrypt blocks. The measured overhead can be explained entirely with the cost of calculating MACs, which are required for pervasive integrity checking. When MAC computation is disabled, VPFS reads the 2 GB of data even slightly faster than the dmcrypt configuration that uses the same AES implementation: VPFS read throughput is 29.2 MB/s versus 28.2 MB/s for the monolithic configuration on the fast SSD.

[2] The traces discussed in this subsection were used to benchmark earlier versions of VPFS on desktop-class hardware. Results for these versions have been published in [156, 157], some of the figures and discussions are included in this subsection.

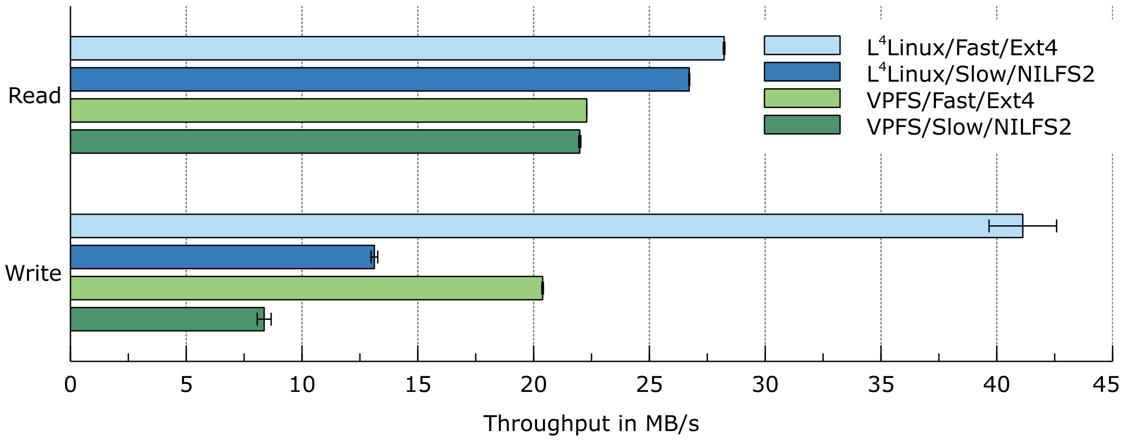


Figure 7.1: Measured throughput for sequential reads and writes using 'bonnie++' traces.

Write Throughput. The benchmarks of write throughput include the time needed to flush all dirty buffers to stable storage. The results vary greatly among the different configurations. On the fast SSD, the L^{4}Linux/dmcrypt configuration performs best, because dmcrypt distributes encryption work to both CPU cores. The VPFS prototype is single-threaded and can use only one CPU. VPFS still achieves approximately 50 percent of the monolithic system's throughput, even though it performs more cryptographic operations on a single CPU (i.e., both encryption and computation of MACs). This result shows that VPFS can effectively offload all work related to writing back encrypted blocks to VPFS Helper and the commodity file system, which execute in parallel on the second CPU. When writing to the slow SSD, the difference in performance between VPFS and the monolithic file-system stack is smaller (37 percent lower throughput for VPFS). One would expect the write rate to be roughly the same, because the cryptographic operations are clearly not the limiting factor. The performance drop is not specific to NILFS2: using Ext4 on the slow SSD gives higher write throughput for both VPFS and the dmcrypt configuration, but the ratio is about the same at 14.0 MB/s and 19,6 MB/s, respectively. However, NILFS2 is still the better choice for most of the application workloads discussed in Section 7.4.3, where non-sequential writes are common.

Creating Many Files. The 'untar1' trace is a write-only benchmark that creates a large number of files. It simulates unpacking a tar file containing an email archive in maildir format (i.e., text files containing email headers and message bodies). Most files are in the order of a few kilobytes, about 1.5 percent are larger than 1 MB. In total, the trace writes approximately 70 MB of data to more than 3,200 files, which are distributed across 24 directories. Like in the bonnie++ write benchmark, the trace player called `sync()` in order to flush all data and metadata to stable storage. Figure 7.2 shows the run times of this benchmark. Variations among the four configurations are similar to the previously discussed throughput benchmarks, except that L^{4}Linux/dmcrypt cannot draw any significant benefit from using both CPUs for encryption.

Reading Many Files. The 'grep' trace simulates a keyword search within all files written by the 'untar1' trace. Each file is opened and read sequentially until the end of the file is reached. I ran this trace with both cold caches (results labelled 'grep1' in Figure 7.2), and again with all data already present in VPFS Core's trusted buffer cache.

In the cold-caches variant, VPFS Core must request each block from the untrusted commodity file system, which in turn must issue read requests to the storage device. VPFS causes between 27 and 29 percent overhead compared to the monolithic configuration. These figures are larger than the 18 to 22 percent overheads measured for sequential reads in the 'bonnie++' benchmark. The difference can be explained with larger communication overhead and I/O delay: since most of

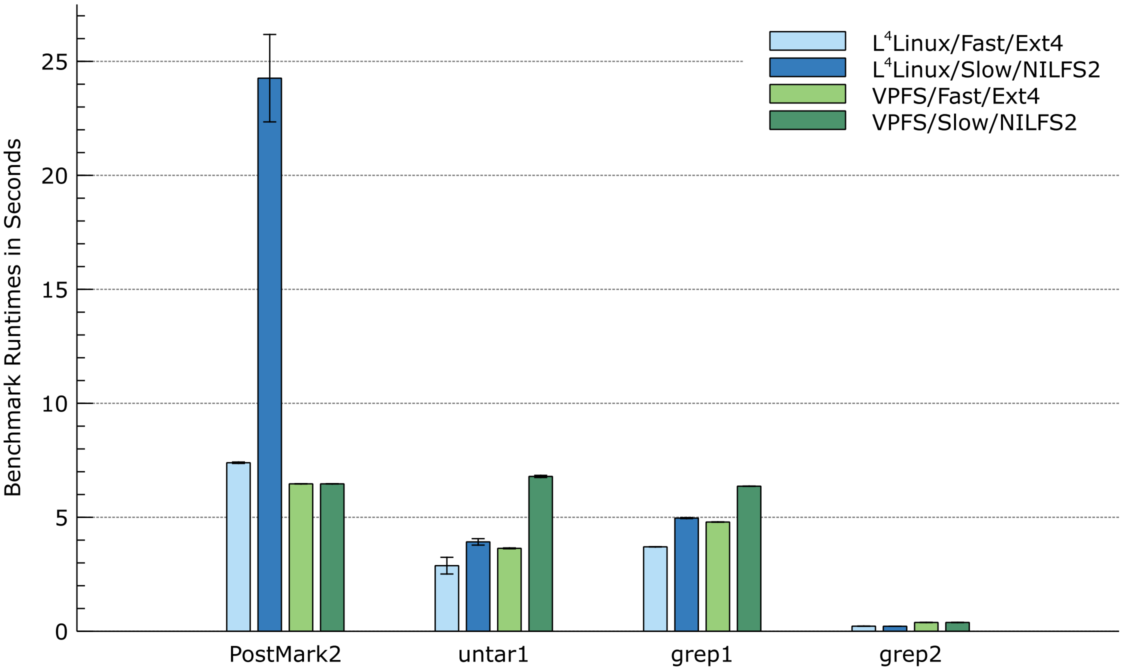


Figure 7.2: Runtimes for microbenchmarks (PostMark, untar, and grep traces).

the files consist of only one 4 KB block and only the trace player knows which file is going to be read next, VPFS Core cannot benefit from read-ahead optimizations implemented in the untrusted VPFS Helper.

Run times for cached reads (see low bars labelled 'grep2' in the figure) are in the order of a few hundred milliseconds. VPFS requires 0.39 seconds to complete the benchmark, whereas L^{4}Linux delivers all data within 0.22 seconds (standard deviations across ten runs are below 10 ms).

Operations on Short-Lived Files. PostMark is a synthetic benchmark that creates, modifies, and then deletes a large number of files. The PostMark2 trace operates on 10 batches of 5,000 files with a size in the order of a few kilobytes. In the VPFS configuration used in this evaluation, this benchmark is cache-only, because write-back thresholds are not reached. When run directly on Ext4 or NILFS2, Linux does write short-lived data to the storage medium. NILFS2 is almost four times slower, because it does not coalesce repeated writes to the same block and it operates on the slow SSD. However, overwrites are common in this workload, because PostMark frequently appends small amounts of data to files it created earlier. While strict serialization of all write operations in NILFS2 causes performance issues with PostMark, this behavior is beneficial for journaling support in VPFS.

Journaling Overhead. Table 7.5 shows benchmark results of an earlier version of VPFS [157]. On this version, I ran the 'untar1' and 'PostMark2' traces, once with operation-level journaling disabled, and once with robustness support as described in Chapter 5. For these experiments, I used both a rotating disk and a slow flash-based storage medium. Due to smaller cache sizes with lower write-back thresholds, the PostMark2 trace caused a large number of evictions; VPFS Helper also received blocks containing metadata multiple times. Under such write patterns, the helper must instruct the commodity file system to serialize writes to the VPFS journal and file containers.

With strictly-ordering NILFS2 operating on flash storage, VPFS did not require synchronous writes in order to update its journal. Nevertheless, journal writes did cause increased write traffic, as can be seen from the figures in Table 7.5. VPFS Core generated 3.5 MB of journal records, but

Trace Name	Storage Device	Without Journal	With Journal
PostMark2	HDD/ReiserFS	6.37 s *(0.14 s)*	9.56 s *(0.27 s)*
PostMark2	SSD-1G/NILFS2	12.36 s *(0.60 s)*	13.49 s *(0.54 s)*
untar1	HDD/ReiserFS	2.07 s *(0.02 s)*	2.14 s *(0.03 s)*
untar1	SSD-1G/NILFS2	9.65 s *(0.09 s)*	9.83 s *(0.13 s)*

Table 7.5: Run times (with standard deviation) for PostMark2 and untar1 traces on an earlier version of VPFS with and without operation-level journaling enabled. Measurements were taken on a desktop-class machine equipped with a rotating disk (HDD) and slow flash storage (SSD-1G); figures originally published in [157].

Trace Name	iBench Name	Description of Workload
iPhoto1	Dup	Duplicate 400 photos in photo library
iPhoto2	Edit	Edit 400 photos in photo library
iPhoto3	Del	Delete 400 photos from photo library and empty trash
iTunes1	ImpS	Import a small album of 10 songs
iMovie1	Exp	Export a short video clip
Keynote1	PPTP	Export 20 slides with JPG images in PPT format
Keynote2	PlayP	Play back presentation of 20 slides with JPG images
Pages2	DOC	Export a 15 page document in DOC format
Pages3	NewP	Create document with 15 JPG images and save in Pages format
Numbers1	XLS	Export document with 5 sheets and diagrams in XLS format

Table 7.6: Description of workload traces from productivity and multimedia applications (descriptions and original iBench traces [71] published by Harter et. al. [108]).

they arrived in groups smaller than the NILFS2 block size. I measured 9 percent journaling-related run-time overhead. With ReiserFS being used as the commodity file system on the hard disk, the overhead increased to about 50 percent. I expected such a behavior, because VPFS Helper's Txn Manager must call `fsync()` on the journal file in order to enforce the correct write order for consistency. This operation is expensive on rotating disks.

For the 'untar1' trace, I measured 3 and 2 percent journaling overhead for the HDD/ReiserFS and SSD-1G/NILFS2 configuration, respectively. This overhead correlates with the actual size of the journal file, which accounted for 2.3 percent of all data written to the commodity file system. These figures are lower than for the PostMark benchmark for two reasons: first, there is a number of large files among the more than 3,200 files and directories that are created—those files dominate write traffic. Second, the 'untar1' workload causes fewer metadata blocks to be overwritten (if any). Relaxations implemented in VPFS Helper (see Section 5.7) then enable fast performance on rotating disks as well.

After having discussed microbenchmarks, I will now evaluate how VPFS performs under real-world workloads.

7.4.3 Application Workloads

Table 7.6 lists file-system traces used to evaluate the performance of VPFS for application workloads. These *iBench traces* [71] were originally captured by Harter et. al. [108] using 'dtrace' on Mac OS X. They include file-system related activity generated by six widely-used productivity and multimedia applications, which were executing scripted tasks.

Trace Playback and API Completeness. Not all file-system APIs and features used by the iBench traces are supported on VPFS. Therefore, I had to use modified versions of the traces in

POSIX operation in VFS	**Basic Operation in VPFS Core**
`open()`, `opendir()`	`File_system::open()`
`close()`, `closedir()`	`File_descriptor::close()`
`readdir()`	`File_descriptor::pgetdents()`, `File::block()`
`read()`, `readv()`, `pread()`	`File_descriptor::pread()`, `File::block()`
`write()`, `writev()`, `pwrite()`	`File_descriptor::pwrite()`, `File::block()`
`lseek()`	`N/A` (file pointer handled by L4Re VFS in client)
`fsync()`, `sync()`	`File_descriptor::fsync()`
`ftruncate()`	`File_descriptor::set_size()`
`fstat()`	`File_descriptor::stat()`
`stat()`, `lstat()`	`File_system::open()`, `File_decriptor::stat()`
`mkdir()`	`File_system::open()`
`unlink()`, `rmdir()`	`File_systen::unlink()`
`rename()`	`File_system::rename()`
`mmap()`, `munmap()`	`File_descriptor::get_dataspace()`, `Dataspace::map()`, `Dataspace::info()`

Table 7.7: POSIX file operations supported by L4Re VFS when using VPFS as a file system.

this evaluation; the following changes applied by a conversion tool ensure compatibility with the VPFS trace player:

- **Dropped Operations:** API calls related to file-based access control such as `chown()` or `access()` cannot be replayed, because these concepts have no counterpart in per-application VPFS instances. Furthermore, L4Re VFS and the VPFS prototype do not support extended file attributes, which the Mac OS X applications access frequently. These operations were omitted from playback.

- **Alternative APIs:** Asynchronous read and write operations were mapped to synchronous APIs. Symbolic and hard links are not supported (but could be). Pathnames containing symbolic links could be resolved statically by the conversion tool for all affected operations in the trace files; hard links are not used in the traces listed in Table 7.6. The tool also converted calls to `dup()` and an equivalent use of the `fcntl()` function into additional invocations of `open()`. The system call `exchangedata()` only exists on Mac OS X; the conversion tool replaced this operation with a sequence of `rename()` operations without causing incorrect behavior for the traces that make use of it.

Table 7.7 lists the operations that the VPFS trace player exercises after conversion of the iBench traces. An exception is memory-mapped file I/O (`mmap()`, `munmap()`, and page-fault resolution). These operations do appear in the iBench traces, but they operate only on executable files and shared libraries that would not be located in a VPFS-based file system storing user data and configuration files of an application.

Despite the modifications I just described, the workload traces encode file-system operations that realistic applications would execute in order to perform the tasks listed in Table 7.6. All operations included in these modified traces completed successfully or failed as specified by the return codes that were recorded when the real applications executed.

Short and Bursty Workload Traces. The diagram in Figure 7.3 shows benchmark results for replaying six traces of file-system activity that were captured from multimedia and productivity applications. The 'iTunes1' trace was recorded when iTunes imported an album of ten songs. The 'iMovie1' trace simulates exporting a 3-minute movie file. During these tasks, the applications read and write files containing several megabytes of data and access hundreds of files, in the case of iMovie more than 1,000. The traces captured from the three office applications Keynote, Pages,

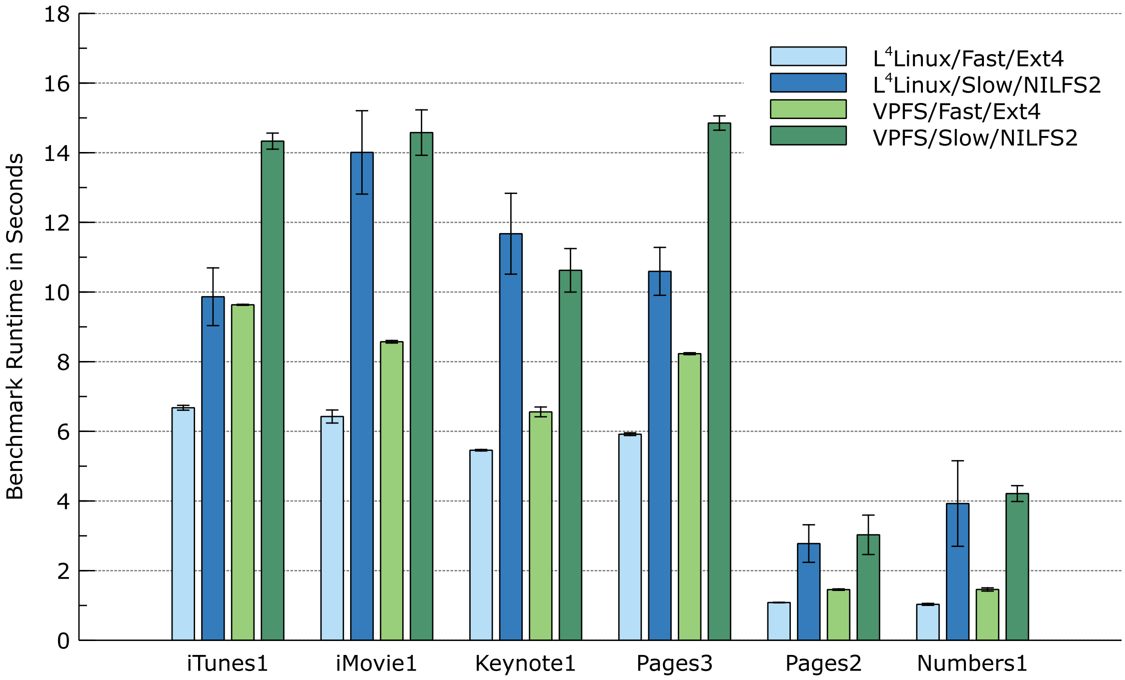


Figure 7.3: Benchmarks of short-running workload traces

and Numbers simulate creating documents. These activities also include reading files containing images that are written into the exported document. Although these traces were captured from desktop applications, there exist mobile versions or similar applications for hand-held devices that allow users to perform the same tasks.

The overhead that VPFS causes for most of these workloads is mixed. The 'iTunes1' and 'Pages3' traces show the highest overhead with up to 50 percent; these traces include `fsync()` calls for the music files and the document that are written. On the other hand, when VPFS runs on the slow SSD, the run times of 'iMovie1', 'Pages2', and 'Numbers1' increase by less than 10 percent. For the 'Keynote1' trace, it shows negative overhead. The amount of data written by these traces is small enough such that VPFS Core's buffer cache can absorb it, thereby hiding the slow write performance of the VPFS/Slow/NILFS2 configuration that became apparent in the microbenchmarks. These traces force synchronous writes only for small configuration files.

Data-Intensive Workloads. Figure 7.4 shows measurements for long-running traces of the iPhoto application. Each of the traces operates on 400 photos in the image library. They read and write between 3 and 10 GB of data and access between 3,000 and 10,000 unique files. Run-times are in the order of minutes, so it is unlikely that users would put such a workload onto their mobile devices. These traces give an impression of how the VPFS prototype performs under massive file-system load.

The VPFS configurations achieve about 60 percent of the performance of the monolithic file-system stack. The overhead is higher than for the previously discussed application workloads. For VPFS using NILFS2 on the slow SSD, the observed performance impact is similar to the results of the write-only throughput benchmark 'bonnie++' and the 'untar1' trace. Indeed, all three traces from Figure 7.4 are write-intensive. Also, run times for long-running traces are unstable: the time required to complete all operations in the 'iPhoto2' trace varied by as much as 20 percent or almost 200 seconds. In isolated experiments, VPFS completed the workload much faster in under 700 seconds. I found that shorter run times require the underlying NILFS2 file system to be fresh. Even though sufficient free space was garbage collected between benchmark runs, writes to the

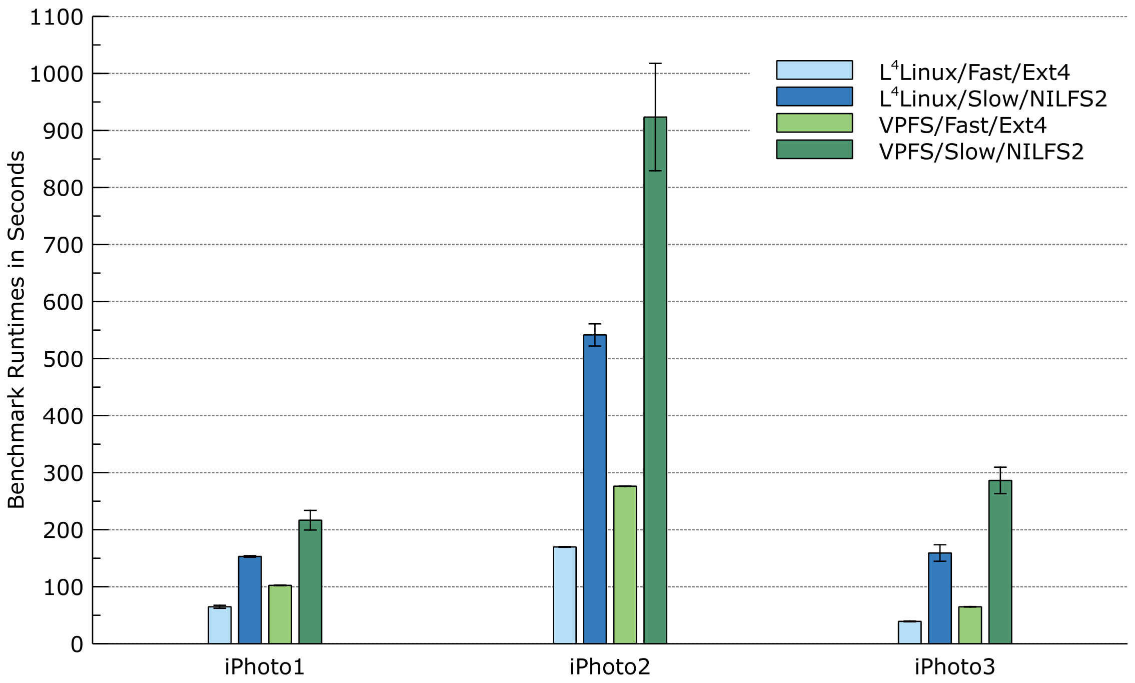


Figure 7.4: Benchmarks of long-running workload traces

storage device are not entirely sequential and thus cause worse performance on the slow SSD. I did not try to optimize the benchmarks for aged file-systems.

7.4.4 CPU Load

Replaying traces as fast as possible is a test for the performance of a file system. For the microbenchmarks and application workloads discussed on the preceding pages, the trace player executed all operations without any delay. However, in common usage scenarios on desktop and mobile platforms, the file system is idle most of the time while applications do computation or wait for the user. The file-system stack becomes active only when applications start, when the user opens a document, or when changes need to be made persistent. For example, if the user is flipping through photos, the application needs to read data in bursts, but not continuously. Only when the file system is active, it consumes CPU cycles.

Timed Trace Playback. To measure the impact of VPFS' increased CPU load in such interactive usage scenarios, I replayed two traces from the iBench suite at the same speed at which they were recorded. In this *timed replay* mode, the trace player evaluates timestamps recorded for each operation in the trace. If the timestamp of an operation lies in the future, the player sleeps until it is time to execute the operation; otherwise, it executes the operation immediately. The wall-clock time at the start of playback and the timestamp of the first operation are synchronized once.

For these experiments, I used NILFS2 as the commodity file system on the slow SSD, which resembles the type of storage devices built into today's mobile devices. The following measurements for CPU load include all system activity while the trace player ran on either VPFS or the dmcrypt-based file-system stack. No other applications or services were active during the benchmark; sampling resolution of load measurements is one second.

Heavy Load. The diagram in Figure 7.5 on the following page shows plots of the combined load on both CPUs during playback of the 'iPhoto4' trace. The green graph represents VPFS, the

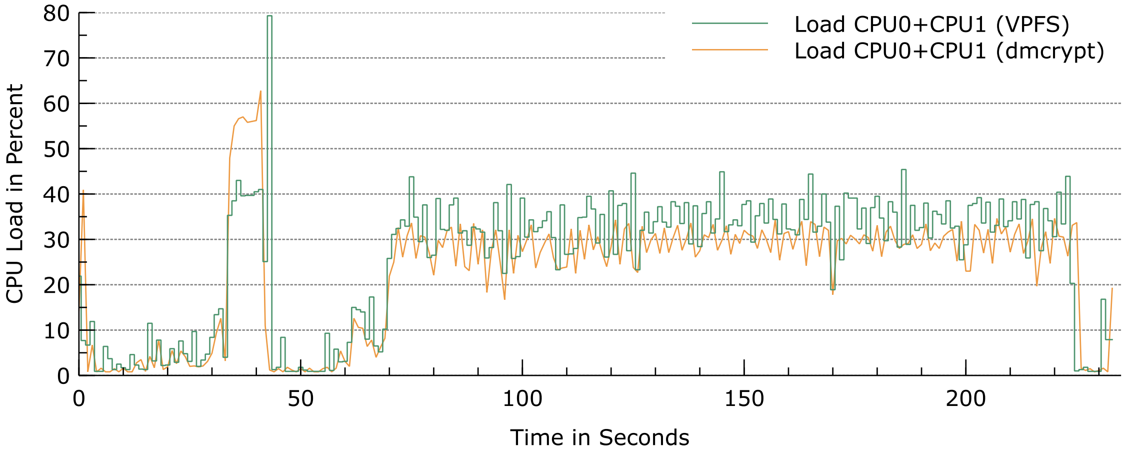


Figure 7.5: CPU load under timed playback of 'iPhoto4' trace for VPFS and dmcrypt

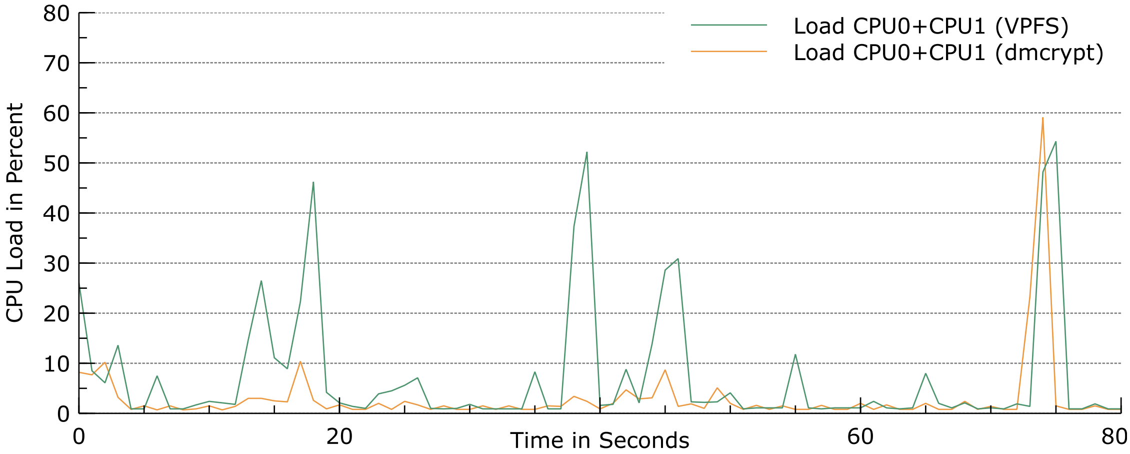


Figure 7.6: CPU load under timed playback of 'Keynote2' trace for VPFS and dmcrypt

orange lines visualize the load when the trace player ran in the L4Linux VM using dmcrypt. This trace runs for 235 seconds, including operations performed during startup of the application; some data is written at the beginning of the trace. About 70 s into playback, the workload simulates loading 400 photos from the file system in order to be displayed in a slideshow. This trace uses the file system heavily; approximately 1 GB of data is read.

As expected, VPFS consumes more CPU cycles than the dmcrypt configuration, which performs fewer cryptographic operations (i.e., no MAC computation). With VPFS, the load is about 35 percent when the trace player loads the 400 "photos". On the monolithic file-system stack based on dmcrypt, trace playback causes approximately 30 percent load. Since there are two CPUs in the system that can handle up to 200 percent load, only between 15 and 17.5 percent of all available CPU cycles are used for file-system work in this usage scenario.

Light Load. Figure 7.6 visualizes system load for the 'Keynote2' trace. This workload uses the file system lightly. Following application startup, a presentation containing 20 slides with images is played back. For most of the 81 seconds of run time, either file system is completely idle. The workload contains short spikes of high activity that cause loads of up to 50 percent when using VPFS. During the first two periods of increased activity at approximately 15 and 40 seconds into playback, the workload performs a large number of accesses to small files distributed across hundreds of subdirectories. I found that in these phases of trace playback, VPFS loads, decrypt, and authenticates hundreds of blocks from file containers that store directory contents. Apparently, NILFS2 requires fewer accesses to the underlying block storage, thereby spending less CPU cycles decrypting data. The spike at end of the trace is caused by write operations.

Even though VPFS uses slightly more CPU resources, it is able to handle both workloads at about the same speed as the monolithic file-system stack. This leads to the following conclusion: a user who stores his files in a VPFS instance will most likely experience much less overhead for realistic workloads than the worst-case slowdown that was measured when benchmarks raced through workload traces as discussed in Sections 7.4.2 and 7.4.3

7.4.5 Crash Recovery

To survive unclean shutdowns, VPFS implements operation-level journaling and replay. I tested the efficacy of this mechanism in two ways. First, I simulated crashes by forcefully terminating VPFS Helper while it wrote to the VPFS journal and the file containers in the commodity file system. Second, I tested robustness on real hardware by power cycling the test machine introduced in Section 7.4.1.

Simulations. The first set of experiments were done in a virtualized testbed on a development machine. In these experiments, a "crash" does not involve power-cycling the machine, but only the application and the VPFS-specific parts of the file-system stack are terminated. To test recovery, VPFS Helper and VPFS Core are restarted. As explained in Section 5.5, the replay loop of the trusted Sync Manager then replays all records from completed transactions as reported by Txn Manager. The prototype successfully recovered snapshots of various points in trace playback from the crashed file system instances. For debugging purposes, VPFS Core also contains a routine that iterates over all inodes and – for each inode not marked invalid – reads all blocks from the corresponding file. This low-level file-system checker verifies the integrity of all data and metadata in the file system. It reliably succeeded without errors when recovery completed.

Real Hardware. For the tests on real hardware, I used both storage devices and commodity file systems. For these experiments, I ran workload traces from the the iBench suite as well. During trace playback, I manually cut power to the machine at random points in the benchmark. After powering up the device again, the commodity file system in the L^{4}Linux kernel recovered the partition containing all file containers and the VPFS journal. The prototype implementation is sufficiently[3] stable such that a previously recovered VPFS instance can be used as a starting point for subsequent robustness tests. Once recovery completed in the experiments, the VPFS trace player started playback of the next workload trace after the corresponding init program (see Section 7.4.1) crawled the file system and restored the directory tree to the state expected by the workload.

Recovery Performance. Table 7.8 lists representative recovery results for different combinations of storage devices and workload traces ('Init_iPhoto2' is the init program that populates the file system for the 'iPhoto2' trace). Power was cut during playback of iBench traces. The data shows that the amount of metadata operations that need to be replayed (fourth column in the table) is small compared to the actual user data whose integrity is validated (last column). Journal sizes are in the order of megabytes, while up to 160 MB of user data are validated. Thus, recovery times are in the order of seconds, but always shorter than the time that passed since the last checkpoint before the crash. This result is expected for systems following a log-structured approach: they must roll forward all operations that have been logged after the last checkpoint.

[3] Due to an implementation bug in the version of VPFS used for this evaluation, replay sometimes aborts with an integrity error, because a metadata block that is part of the last checkpoint was not preserved in the journal before being overwritten.

Workload Name	Storage Config	Recovery Time	Replayed Log Size	Restored Metadata Blks	Checked Data Blks
iTunes1	Fast/Ext4	4.4 s	1.13 MB	2	23,126
Keynote1	Fast/Ext4	1.1 s	0.33 MB	3	3,060
Pages3	Fast/Ext4	2.0 s	0.58 MB	3	8,524
iPhoto1	Fast/Ext4	21.3 s	2.38 MB	11	42,871
iPhoto2	Slow/NILFS2	1.9 s	0.30 MB	5	1,981
Init_iPhoto2	Slow/NILFS2	6.8 s	1.73 MB	0	23,394
iMovie1	Slow/NILFS2	4.1 s	0.90 MB	0	17,636

Table 7.8: Recovery results for VPFS instances after an unclean shutdown

In the case of VPFS, the replay loop in Sync Manager must also reconstruct MACs in the Merkle-tree structure embedded in the block tree. Multiple MAC updates for the same block may appear in the journal. In order to determine which of them correctly authenticates the version of the block that was last written to stable storage, Sync Manager must read the designated block every time (see Section 5.5 for details). For example, during recovery following the crash of the 'iPhoto1' trace listed in Table 7.8, Sync Manager found 236 outdated `Block_update` records as a total of 42,871 updates could be reintegrated into the MAC tree. Note that the latency of file-system recovery as observed by the user can be much smaller than the time needed to recover all inconsistent VPFS instances. Since VPFS is designed such that there is a private file-system instance for each of the mutually distrusting applications, post-crash recovery can be done in background for most of them. The OS can prioritize recovery of those VPFS instances whose application the user starts first. Private file systems that were not mounted require no recovery.

Both the simulations and the tests on real hardware show that low-complexity operation-level journaling is an effective consistency mechanism that enables VPFS to recover from an unclean shutdown. For completeness, I also briefly present the result of trying to run VPFS with journaling disabled. In this experiment, the Txn Manager subsystem of VPFS Helper was allowed to write `Checkpoint` records only, but none of the metadata records that Sync Manager submitted afterwards. After a simulated crash of the 'iPhoto1' trace, VPFS Core started up successfully together with its application. However, the L4Re VFS plugin linked into the application was unable to mount the file system, because the first block of the inode file failed the integrity check. The file system was inaccessible afterwards and all data was lost.

The next chapter summarizes the results of this thesis and gives an outlook on future work.

Chapter 8

Conclusions and Future Work

In my thesis, I described design and implementation of the Virtual Private File System (VPFS). VPFS targets three protection goals: First, it ensures *confidentiality* by encrypting all user data. Second, VPFS uses message authentication codes to ensure *integrity* of all data and metadata in the file system. The third protection goal guaranteed by VPFS is *recoverability* of data by integrating backup and restore into the storage stack.

VPFS Architecture. The main design goal when building VPFS was to re-architect the file-system stack such that the aforementioned security guarantees also hold when strong attackers attempt to exploit defects in the software stack of end-user platforms. The VPFS architecture achieves this goal through two key measures:

1. **Private File-System Instances:** Mutually distrusting applications are provided with their own private file system, which is managed by a dedicated instance of a system service called VPFS Core that runs in its own address space. If an attacker managed to compromise one file-system instance, he can still not access data stored in other unrelated VPFS instances.

2. **Isolation of Security-Critical Code:** VPFS isolates code and data structures that are critical to securing file-system contents from functionality that need not be trusted.

Both measures help to drastically reduce the size of the attack surface. Exploitable bugs in untrusted file-system infrastructure or in applications that are not part of the trusted computing base (TCB) can no longer be used to gain access to user data.

Minimal TCB Complexity. In addition to isolating differently trusted code bases, I also strived for low-complexity implementations of security-critical functionality. A key lesson I learned while building VPFS was that minimizing size and complexity of the TCB requires specialized solutions. In Section 3.3, I discussed how to tailor a whole class of algorithms around a single central data structure: the block tree, which unifies a Merkle tree required for integrity checking with all data and metadata stored in the file system. The routines for cache management and block retrieval built into VPFS Core are two thirds smaller than size-optimized solutions that utilized an off-the-shelf implementation of a balanced search tree in previous versions of VPFS.

Secure Reuse of Untrusted Infrastructure. By far the biggest reduction in code size and complexity can be achieved by delegating non-critical tasks to untrusted infrastructure. To this end, VPFS reuses a Linux file-system stack for persistently storing files and directories in cryptographically protected file containers. File containers are ordinary files in the untrusted commodity file system, so the security-critical VPFS Core does not need to implement low-level functionality such as block allocation and free-space tracking. Thanks to secure reuse, VPFS contributes an order of magnitude less code to the TCB of an application needing file-based storage than a monolithic file-system stack: between 4,500 and 7,500 lines of code are required to protect confidentiality and integrity of user data and metadata, whereas a Linux file-system stack contributes in the order of 100,000 lines of code. VPFS reuses the Linux code base without having to trust it.

Separate TCBs for Different Protection Goals. The functionality that must be part of the TCB depends on the specific protection goal. I found that the TCBs for confidentiality and integrity have significant overlap and can therefore be co-located. However, the functionality that is necessary for recoverability is significantly larger (in the order of 15,000 lines of code). Fortunately, most of the code needed to implement backup and restore can operate on cryptographically protected data. In order to enforce the principle of least privilege, I split the VPFS code base even further. VPFS BackupD, which need not be trusted for integrity, runs in its own address space; multiple instances of VPFS Auth, which communicate with backup servers, are only part of the recoverability TCB.

Finally, self-attesting erasure coding (Section 6.6.4) optimizes both network utilization and storage overhead on backup servers with minimal impact on any of the TCBs: 45 lines of code ensure that (1) erasure-coded backups are distributed to multiple servers and that (2) the encoded backup includes sufficient redundancy such that it can be recovered from a pre-defined subset of servers at recovery-time. The erasure-coding library, which is almost two orders of magnitude larger than the security-critical checking code, need not be trusted.

Applicability of the VPFS Architecture. The VPFS architecture and its feature set are highly suitable for improving security on mobile devices. Application ecosystems for mobile platforms already embrace the concept of sandboxing, which maps well to the private file systems that VPFS provides to mutually distrusting applications. Note that an “application” in this context does not necessarily have to be user-facing, but it could also be a service component of the operating system (OS). For example, a service for managing certificates, a “password wallet” for storing credentials, or other services such as the system’s VPN component need to store critical data persistently. If such a service application already has a small code base itself, VPFS contributes a smaller percentage to code size and complexity of this service’s TCB than a monolithic file system stack.

The VPFS architecture can also improve file system security on traditional desktops as well as on servers. For example, server-side applications in cloud computing scenarios in which ownership of data can be clearly attributed to individuals or groups of users (e.g., web-based productivity software) can benefit from private storage. In such cloud usage scenarios, it is feasible and desirable to strictly isolate data; sharing is not required beyond closed groups. In contrast, social networks such as Facebook, where sharing of data is the core functionality demanded by users, most likely cannot benefit from storage level isolation.

The results of the complexity analysis of the VPFS prototype presented in Sections 7.2 and 7.3 quantify the gains in security and indicate a significant improvement over monolithic file system architectures. The efficiency evaluation in Section 7.4 revealed non-negligible overhead. For realistic interactive workloads, the overhead amounts to a single-digit increase in CPU utilization due to cryptographic protection. However, I also measured overheads in the range of 20 to 50 percent for workloads that use the file system excessively. Whether such overheads are acceptable or if fast performance is needed depends on the use case. There is a tradeoff between maximum performance and the security that is gained.

Fortunately, there are ways to improve the efficiency of VPFS beyond what the initial prototype delivers. Some options are discussed as part of future work on the following page.

Future Work

I conclude my thesis with an outline of how to improve the performance of VPFS and its utility:

Acceleration of Cryptographic Operations: A promising way to improve the efficiency of VPFS is to add support for cryptographic accelerators. Mobile devices already ship with cryptographic co-processors or they have encryption engines integrated into the data path between main memory and flash storage [80]. Support for any of these accelerators requires driver support in the Fiasco.OC/L4Re research OS. An alternative approach is to use implementations of AES and hashing algorithms that can exploit instruction-set extensions to speed up cryptographic operations. For example, future generations of Atom CPUs will support special instructions [79] for the Advanced Encryption Standard (AES). The second option is straight-forward and requires no additional support from the OS.

Exploiting Multicores: Multiple VPFS instances can run in parallel to make use of multiple CPU cores. However, the prototype implementation of VPFS Core is single threaded in order to avoid the complexities of parallel programming. Microbenchmarks discussed in Section 7.4 showed that executing the commodity file-system stack on another CPU can improve performance. The untrusted VPFS Helper handles requests from VPFS Core in a single thread, but it offloads writes to the commodity file system as well as garbage collection to two additional background threads. A fourth thread is spawned, when the backup instance of VPFS Core reads checkpointed file-system state. New programming paradigms based on lambdas and concurrent queues promise to simplify parallel programming. It would be interesting to see how these programming techniques can be used for parallelizing VPFS Core without increasing its complexity too much.

Complete Backup and Restore: The current state of recoverability support in VPFS allows for an accurate evaluation of the complexity that must be added to the confidentiality and integrity TCBs in order to support backup and restore. I also gave well-founded code-size estimates for the client-side of the recoverability TCB. However, in order to evaluate the performance overhead caused by creating backups in background, the implementation of VPFS Auth must be finished and a proof-of-concept server implementation is needed.

Cross-Device File-System Synchronization: If users own multiple devices (e.g., different form factors), they expect that it is possible to start work on a document on one device and then continue working on an up-to-date copy on another. Synchronizing copies of the same file system across multiple devices is a challenging task, especially when servers involved in data exchange must not be able to access files in plaintext or perform modifying operations on them. To keep the security properties and small TCB footprint of VPFS, it is necessary to integrate synchronization primitives into the file system stack in a way similar to the recoverability extension. Such synchronization functionality will most likely offer low-level APIs and act in concert with application-level mechanisms for conflict resolution and merging.

Bibliography

[1] Bonnie++.
http://www.coker.com.au/bonnie++/. 105

[2] BTRFS Documentation Wiki.
http://btrfs.wiki.kernel.org/index.php/Main_Page. 58

[3] CVE-2003-0545 - Incorrect memory management in ASN.1 certificate handling in OpenSSL 0.9.7 that may result in denial of service or arbitrary code execution.
http://www.cvedetails.com/cve/CVE-2003-0545. 77

[4] CVE-2003-0818 - Multiple integer overflows in Microsoft ASN.1 library that allow arbitrary code execution.
http://www.cvedetails.com/cve/CVE-2003-0818. 77

[5] CVE-2007-5904 - Denial of service and possibly arbitrary code execution through multiple buffer overflows in CIFS VFS code in Linux 2.6.33 and earlier.
http://www.cvedetails.com/cve/CVE-2007-5904/. 15

[6] CVE-2008-0001 - Incorrect permission check may allow local user to remove directories in various Linux 2.6 versions.
http://www.cvedetails.com/cve/CVE-2008-0001/. 15

[7] CVE-2009-2908 - Denial of service and possibly arbitrary code execution via bug in eCryptfs on Linux 2.6.31.
http://www.cvedetails.com/cve/CVE-2009-2908/. 15

[8] CVE-2010-1146 - Insufficient permission checks allow local user to modify metadata and access control information on mounted Reiserfs file systems in Linux 2.6.33.2 and earlier.
http://www.cvedetails.com/cve/CVE-2010-1146/. 15

[9] CVE-2010-1797 - Multiple buffer overflows in FreeType before 2.4.2.
http://www.cvedetails.com/cve/CVE-2010-1797/. 14

[10] CVE-2010-2066 - Incorrect arguments check allows local user to overwrite append-only files in ext4 file system.
http://www.cvedetails.com/cve/CVE-2010-2066/. 15

[11] CVE-2010-2226 - Incorrect permission check in XFS online degramentation code allows local user to gain access to data without read permission.
http://www.cvedetails.com/cve/CVE-2010-2226/. 15

[12] CVE-2010-2521 - Denial of service and possibly arbitrary code execution through crafted messages sent to NFS file server on Linux 2.6.
http://www.cvedetails.com/cve/CVE-2010-2521/. 15

[13] CVE-2010-3015 - Integer overflow in ext4 allows denial of service through sequence of file system operations.
http://www.cvedetails.com/cve/CVE-2010-3015/. 15

[14] CVE-2010-4210 - Denial of service and possibly arbitrary code execution through sequence of operations on a pseudofs file system with FreeBSD kernel older than 7.3-RELEASE and 8.0-RC1.
`http://www.cvedetails.com/cve/CVE-2010-4210/`. 15

[15] CVE-2011-0180 - Integer overflow in HFS allows read access to arbitrary files on Mac OS X before 10.6.7.
`http://www.cvedetails.com/cve/CVE-2011-0180/`. 15

[16] CVE-2011-1823 - Privilege escalation and arbitrary code execution in Android volume management service.
`http://www.cvedetails.com/cve/CVE-2011-1823/`. 14

[17] CVE-2011-4077 - Denial of service and possibly arbitrary code execution via crafted XFS file system image on Linux 2.6.
`http://www.cvedetails.com/cve/CVE-2011-4077/`. 15

[18] CVE-2011-4330 - Denial of service and possibly arbitrary code execution via crafted HFS file system image on Linux 2.6.
`http://www.cvedetails.com/cve/CVE-2011-4330/`. 15

[19] CVE-2012-0642 - Arbitrary code execution via crafted HFS file system image in iOS kernel before 5.1.
`http://www.cvedetails.com/cve/CVE-2012-0642/`. 14, 15

[20] CVE-2012-0643 - Sandbox escape through debugging system calls to iOS kernel before version 5.1.
`http://www.cvedetails.com/cve/CVE-2012-0643/`. 14

[21] CVE-2012-2110 - Memory corruption in OpenSSL due to incorrect integer interpretation.
`http://www.cvedetails.com/cve/CVE-2012-2110`. 77

[22] CVE-2012-3524 - Vulnerability of libdbus 1.5.x and earlier, if used in setuid programs.
`http://www.cvedetails.com/cve/CVE-2012-3524/`. 14

[23] CVE-2012-3727 - Privilege escalation and code execution in Racoon IPSec service.
`http://www.cvedetails.com/cve/CVE-2012-3727/`. 14

[24] CVE-2012-3728 - Privilege escalation in iOS kernel.
`http://www.cvedetails.com/cve/CVE-2012-3728/`. 14

[25] CVE-2012-3979 - Arbitrary code execution through malicious Javascript Firefox on Android.
`http://www.cvedetails.com/cve/CVE-2012-3979/`. 14

[26] Data Backup - Android Developer Documentation.
`http://developer.android.com/guide/topics/data/backup.html`. 73, 74

[27] DigiNotar public report version 1.
`http://www.rijksoverheid.nl/bestanden/documenten-en-publicaties/rapporten/2011/09/05/diginotar-public-report-version-1/rapport-fox-it-operation-black-tulip-v1-0.pdf`. 77

[28] Dropbox - Simplify your life.
`https://www.dropbox.com`. 74

[29] eCryptfs - Enterprise Cryptographic Filesystem.
`http://ecryptfs.org`. 28

[30] EncFS Encrypted Filesystem.
`http://www.arg0.net/encfs`. 28, 49

[31] Encrypting File System Technical Reference.
http://technet.microsoft.com/de-de/library/cc780166(WS.10).aspx. 28

[32] Ext4 website. https://ext4.wiki.kernel.org/index.php. 38, 104

[33] Federal Information Processing Standards Publication 180-1: Secure Hash Standard. Available from:
http://www.itl.nist.gov/fipspubs/fip180-1.htm. 97

[34] Federal Information Processing Standards Publication 197: Announcing the Advanced Encryption Standard. Available from:
http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. 13, 34, 97

[35] Github repo for Jerasure Library.
https://github.com/tsuraan/Jerasure. 98

[36] L4Linux Website.
http://os.inf.tu-dresden.de/L4/LinuxOnL4/. 93

[37] LibSSH2 - The SSH Library.
http://www.libssh2.org. 77

[38] MatrixSSL - Open Source Embedded SSL and TLS.
http://www.matrixssl.org. 77, 94, 98

[39] Microsoft's cloud storage service 'SkyDrive'.
https://skydrive.live.com. 74

[40] MINIX 3 Website.
http://www.minix3.org. 27

[41] NILFS - Continous Snapshotting Filesystem for Linux.
http://www.nilfs.org/en/. 70, 104

[42] NTFS Technical Reference.
http://msdn.microsoft.com/de-de/library/cc758691(WS.10).aspx. 58

[43] Oracle Berkeley DB Products - Overview.
http://www.oracle.com/us/products/database/berkeley-db/overview/index.html. 54

[44] Phony SSL Certificates issued for Google, Yahoo, Skype, Others - threatpost.
http://threatpost.com/en_us/blogs/phony-web-certificates-issued-google-yahoo-skype-others-032311. 77

[45] pmccabe – McCabe-style Function Complexity and Line Counting for C and C++.
http://www.parisc-linux.org/~{}bame/pmccabe/overview.html. 98

[46] Qnx neutrino rtos.
http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html. 27

[47] RerWare - MyBackup Pro.
http://www.rerware.com. 74

[48] rsync Website.
http://www.samba.org/rsync/. 76

[49] SLOCCount.
http://www.dwheeler.com/sloccount/. 96

[50] SpiderOak.com - Free Online Backup.
https://spideroak.com/. 74

[51] The Fiasco Microkernel.
http://os.inf.tu-dresden.de/fiasco/. 93

[52] Titanium Backup (root) for Android.
http://matrixrewriter.com/android/. 74

[53] TrueCrypt - Free Open-Source Disk Encryption - Documentation.
http://www.truecrypt.org/docs/. 18, 28

[54] Trusted Computing Group: Storage Work Group.
https://www.trustedcomputinggroup.org/group/storage. 28

[55] Trusted Platform Module.
http://www.trustedcomputinggroup.org/developers/trusted_platform_module. 23, 24

[56] wuala - Secure Online Storage.
http://www.wuala.com/. 74, 75

[57] ZFS Website.
http://hub.opensolaris.org/bin/view/Community+Group+zfs/WebHome. 28, 33, 58

[58] RFC 4251 - The Secure Shell (SSH) Protocol Architecture.
http://tools.ietf.org/html/rfc4251, January 2006. 76

[59] RFC 4252 - The Secure Shell (SSH) Authentication Protocol.
http://tools.ietf.org/html/rfc4252, January 2006. 76

[60] RFC 4422 - Simple Authentication and Security Layer (SASL).
http://tools.ietf.org/html/rfc4422, June 2006. 77

[61] The Month of Kernel Bugs (MoKB) Archive.
http://projects.info-pull.com/mokb/, November 2006. 15, 16

[62] Amazon s3 service level agreement.
http://aws.amazon.com/de/s3-sla/, October 2007. 74, 85

[63] ReiserFS on Namesys website (archived 2007).
http://web.archive.org/web/20071023172417/www.namesys.com/, 2007. 58, 70

[64] RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2.
http://tools.ietf.org/html/rfc5246, August 2008. 22, 76, 101

[65] Secret terror files left on train - BBC news report.
http://news.bbc.co.uk/2/hi/uk/7449255.stm, June 2008. 13

[66] About the security content of iPhone OS 3.0.1.
http://support.apple.com/kb/HT3754, August 2009. 14

[67] Previous cases of missing data - BBC media report on loss of confidential information and hardware by UK officials between 2007 and 2009.
http://news.bbc.co.uk/2/hi/uk_news/7449927.stm, May 2009. 13

[68] Wide Cyber Attack is Linked to China - The Wall Street Journal.
http://online.wsj.com/article/SB123834671171466791.html, March 2009. 14

[69] Google China cyberattack part of vast espionage campaign, experts say - The Washington Post.
http://www.washingtonpost.com/wp-dyn/content/article/2010/01/13/AR2010011300359.html, January 2010. 14

[70] Hacker extracts crypto key from TPM chip.
http://www.h-online.com/security/news/item/Hacker-extracts-crypto-key-from-TPM-chip-927077.html, February 2010. 26

[71] iBench - Productivity and multimedia application workload traces.
http://research.cs.wisc.edu/adsl/Traces/ibench/, November 2011. 108

[72] New security scare as senior civil servant leaves laptop on commuter train - Mirror Online.
http://www.mirror.co.uk/news/uk-news/new-security-scare-as-senior-civil-89968, November 2011. 13

[73] The Official Lookout Blog - Security Alert: DroidDream Malware Found in Official Android Market.
https://blog.lookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/, March 2011. 13

[74] Trend Micro - Malware Blog - Trojanized Apps Root Android Devices.
http://blog.trendmicro.com/trendlabs-security-intelligence/trojanized-apps-root-android-devices/, March 2011. 13

[75] Trends in Targeted Attacks - Trend Micro White Paper.
http://www.trendmicro.co.uk/media/wp/trends-in-targeted-attacks-whitepaper-en.pdf, October 2011. 14

[76] A Tale of Two Pwnies (Part 1).
http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html, May 2012. 39

[77] History of iOS jailbraking (in Wikipedia).
http://en.wikipedia.org/w/index.php?title=History_of_iOS_jailbreaking&oldid=514204219, September 2012. 26

[78] iCloud: Backup and restore overview.
http://support.apple.com/kb/HT4859, November 2012. 73

[79] Intek Advanced Encryption Standard Instructions (AES-NI).
http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni, 2012. 117

[80] iOS Security (Whitepaper).
http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf, May 2012. 23, 24, 28, 35, 117

[81] Issue 117656: Pwnium bug: GPU memory corruption - Chromium Bug Tracker.
http://code.google.com/p/chromium/issues/detail?id=117656, March 2012. 39

[82] OpenSSL ASN.1 vulnerability: sshd not affected - OpenSSH developer maling list openssh-unix-dev.
http://lists.mindrot.org/pipermail/openssh-unix-dev/2012-April/030386.html, April 2012. 77

[83] Windows Azure Support - Service Level Agreement.
http://www.windowsazure.com/en-us/support/legal/sla/, June 2012. 74, 85

[84] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A Case for Cloud Storage Diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 229–240, New York, NY, USA, 2010. ACM. 75, 84

[85] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 1–14, New York, NY, USA, 2002. ACM. 50

[86] F. Armknecht, Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, G. Ramunno, and D. Vernizzi. An Efficient Implementation of Trusted Channels Based on OpenSSL. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable Trusted Computing*, pages 41–50, New York, NY, USA, 2008. ACM. 79

[87] A. C. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 802–811, Washington, DC, USA, 2005. IEEE Computer Society. 57, 66

[88] V. Aurora. Soft updates, hard problems.
`http://lwn.net/Articles/339337`, July 2009. 57

[89] M. Bellare, R. Canetti, and H. Krawczyk. Message Authentication Using Hash Functions: the HMAC Construction. *CryptoBytes*, 2(1):12–15, 1996. 32, 35

[90] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi. Duqu: A Stuxnet-like malware found in the wild. Technical report, Laboratory of Cryptography and System Security (CrySyS), Budapest University of Technology and Economics, Department of Telecommunications, October 2011. 14

[91] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. In *Proceedings of the EuroSys 2011 Conference*, EuroSys '11, pages 31–46, New York, NY, USA, 2011. ACM. 75, 84

[92] M. Blaze. A Cryptographic File System for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993. 28

[93] R. Card, T. Ts'o, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. Proceedings of the First Dutch International Symposium on Linux, 1994. 57

[94] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002. 50

[95] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An Empirical Study of Operating System Errors. In *In Proceedings of the 18th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 73–88, 2001. 16

[96] O. Choudary, F. Gröbert, and J. Metz. Infiltrate the Vault: Security Analysis and Decryption of Lion Full Disk Encryption. *IACR Cryptology ePrint Archive*, 2012:374, 2012. informal publication. 28

[97] J. R. Douceur, A. Adya, J. Benaloh, W. J. Bolosky, and G. Yuval. A Secure Directory Service based on Exclusive Encryption. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC '02, Washington, DC, USA, 2002. IEEE Computer Society. 50

[98] N. E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000. 14, 97

[99] N. Ferguson. AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista.
http://download.microsoft.com/download/0/2/3/0238acaf-d3bf-4a6d-b3d6-0a0be4bbb36e/bitlockercipher200608.pdf, 2006. 24, 28, 38

[100] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized File System Dependencies. In *SOSP '07: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 307–320, New York, NY, USA, 2007. ACM. 58

[101] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and Secure Distributed Read-Only File System. *ACM Transactions on Computer Systems*, 20(1):1–24, 2002. 49

[102] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems*, 18(2):127–153, May 2000. 58

[103] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. 75, 85

[104] 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. 29, 35, 50, 53, 55

[105] D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer. Cryptree: A folder tree structure for cryptographic file systems. In *Reliable Distributed Systems, 2006. SRDS '06. 25th IEEE Symposium on*, pages 189–198, Oct. 2006. 35, 74

[106] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: Error Handling Is Occasionally Correct. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–16, Berkeley, CA, USA, 2008. USENIX Association. 15, 16, 57

[107] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold-Boot Attacks On Encryption Keys. *Commun. ACM*, 52(5):91–98, 2009. 26

[108] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '11, pages 71–83, New York, NY, USA, 2011. ACM. 64, 70, 108

[109] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *Proceedings of CollaborateCom*, 2005. 14, 16, 17, 22, 23

[110] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of μ-Kernel-based Systems. In *In Proceedings of the 16th ACM SIGOPS Symposium on Operating System Principles (SOSP)*, pages 66–77, 1997. 22, 36

[111] L. Hatton. Reexamining the Fault Density-Component Size Connection. *IEEE Software*, 14(2):89–97, Mar. 1997. 14, 97

[112] J. Heider and R. E. Khayari. iOS Keychain Weakness FAQ - Further Information on iOS Password Protection.
http://sit.sit.fraunhofer.de/studies/en/sc-iphone-passwords-faq.pdf, July 2012. 26

[113] C. Helmuth, A. Warg, and N. Feske. Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture. In *Proceedings of the D.A.CH Security 2005*, March 2005. 23, 101

[114] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association. 58

[115] Y. Hu, H. C. H. Chen, P. P. C. Lee, and Y. Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, Berkeley, CA, USA, 2012. USENIX Association. 75, 84

[116] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, Berkeley, CA, USA, 2012. USENIX Association. 75

[117] C. D. S. James S. Plank, Scott Simmerman. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications. Technical Report UT-CS-08-627, Department of Computer Science, University of Tennessee, August 2008. 87

[118] M. B. Jens Heider. Lost iPhone? Lost Passwords! - Practical Consideration of iOS Device Encryption Security.
`http://sit.sit.fraunhofer.de/studies/en/sc-iphone-passwords.pdf`, February 2012. 26

[119] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 29–42, Berkeley, CA, USA, 2003. USENIX Association. 35

[120] Kevin Walsh and Fred Schneider. Costs of Security in the PFS File System. Technical report, Cornell University, 2012. 28

[121] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. *ACM Transactions on Storage*, 8(4):14:1–14:25, Dec. 2012. 104

[122] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In J. N. Matthews and T. E. Anderson, editors, *In Proceedings of the 22th ACM SIGOPS Symposium on Operating System Principles (SOSP)*, pages 207–220. ACM, 2009. 22

[123] A. Lackorzynski and A. Warg. Taming subsystems: capabilities as universal resource access control in L4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pages 25–30, New York, NY, USA, 2009. ACM. 22, 93

[124] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation OSDI'04*, December 2004. 29, 35

[125] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, New York, NY, USA, 2006. ACM. 16

[126] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud Storage With Minimal Trust. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–12, Berkeley, CA, USA, 2010. USENIX Association. 50

[127] U. Maheshwari, R. Vingralek, and B. Shapiro. How to Build a Trusted Database System on Untrusted Storage. pages 135–150. 29, 35, 47, 60

[128] A. Matrosov, E. Rodionov, D. Harley, and J. Malcho. Stuxnet Under the Microscope. `http://go.eset.com/us/resources/white-papers/Stuxnet_Under_the_Microscope.pdf`, 2010. 14

[129] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating Key Management From File System Security. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '99, pages 124–139, New York, NY, USA, 1999. ACM. 49, 53

[130] T. J. McCabe. A Complexity Measure. *In IEEE Transactions on Software Engineering*, SE2(4):308–320, December 1976. 95, 97, 98

[131] M. K. McKusick and G. R. Ganger. Soft Updates: a Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 24–24, Berkeley, CA, USA, 1999. USENIX Association. 59

[132] R. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–134, 1980. 31, 35

[133] E. Miller, D. Long, W. Freeman, and B. Reed. Strong Security for Distributed File Systems. In *IEEE International Conference on Performance, Computing, and Communications*, pages 34 –40, apr 2001. 38

[134] G. H. Nibaldi. Specification of a Trusted Computing Base (TCB). Technical Report A138801, MITRE Corp., Bedford MA, November 1979. 14

[135] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, USENIXATC'11, pages 31–31, Berkeley, CA, USA, 2011. USENIX Association. 74

[136] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, October 2011. 29

[137] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference*, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association. 59, 61, 62

[138] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960. 75, 86, 94

[139] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. In *SOSP '91: Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, New York, NY, USA, 1991. ACM. 58, 71

[140] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, pages 1278 – 1308, September 1975. 14

[141] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. *Innovations in Internetworking*, pages 379–390, 1988. 29

[142] A. Schlösser, D. Nedospasov, J. Krämer, S. Orlic, and J.-P. Seifert. Simple Photonic Emission Analysis of AES. In *Lecture Notes in Computer Science*, volume 7428 of *Cryptographic Hardware and Embedded Systems (CHES2012)*, pages 41–47, 2012. 26

[143] M. D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility.* PhD thesis, Massachusetts Institute of Technology Laboratory for Computer Science, September 1972. 15

[144] SecurityWeek.com. Resilient 'SMSZombie' Infects 500,000 Android Users in China. `http://www.securityweek.com/resilient-smszombie-infects-500000-android-users-china`, August 2012. 13

[145] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979. 75

[146] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A Fast Capability System. In *In Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 170–185, 1999. 22

[147] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB Complexity for Security-sensitive Applications: Three Case Studies. *SIGOPS Operating Systems Review*, 40(4):161–174, 2006. 14, 16, 22, 101

[148] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-Safe Disks. In *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 15–28, 2006. 58

[149] G. Sivathanu, S. Sundararaman, and E. Zadok. Automatic Consistency for Disk Storage. Technical report, Stony Brook University, 2007. 58

[150] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised System. In *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation (OSDI2000)*, volume 4, 2000. 29, 38

[151] S. S. Surendra Verma. Building the next generation file system for Windows: ReFS. `http://blogs.msdn.com/b/b8/archive/2012/01/16/building-the-next-generation-file-system-for-windows-refs.aspx`, January 2012. 55

[152] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, November 2006. 27

[153] S. C. Tweedie. Journaling the Linux ext2fs Filesystem. In *Proceedings of the 4th Annual LinuxExpo*, May 1998. 38, 58

[154] M. van Dijk, J. Rhodes, L. F. G. Sarmenta, and S. Devades. Offline Untrusted Storage with Immediate Detection of Forking and Replay Attacks∗. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing STC'07*, pages 41–48, 2007. 24, 79

[155] C. Weinhold. Design and Implementation of a Trustworthy File System for L4. Master's thesis, Technische Universität Dresden, 2006. `http://os.inf.tu-dresden.de/papers_ps/weinhold-diplom.pdf`. 31, 42, 47, 51, 55

[156] C. Weinhold and H. Härtig. VPFS: Building a Virtual Private File System With a Small Trusted Computing Base. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 81–93, New York, NY, USA, 2008. ACM. 42, 47, 51, 52, 54, 55, 96, 97, 105

[157] C. Weinhold and H. Härtig. jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIXATC'11, Berkeley, CA, USA, 2011. USENIX Association. 19, 57, 105, 107, 108

[158] J. Wires and M. J. Feeley. Secure File System Versioning at the Block Level. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 203–215, New York, NY, USA, 2007. ACM Press. 29, 38, 54

[159] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 131–146, Berkeley, CA, USA, 2006. USENIX Association. 15, 16, 66

[160] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. *ACM Transactions on Computer Systems (TOCS)*, 24(4):393–423, 2006. 15, 16, 57