jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components

Carsten Weinhold, Hermann Härtig
Technische Universität Dresden, Germany
{weinhold,haertig}@os.inf.tu-dresden.de

Abstract

The Virtual Private File System (VPFS) [1] was built to protect confidentiality and integrity of application data against strong attacks. To minimize the trusted computing base (i.e., the attack surface) it was built as a stacked file system, where a small isolated component in a microkernel-based system reuses a potentially large and complex untrusted file system; for example, as provided by a more vulnerable guest OS in a separate virtual machine. However, its design ignores robustness issues that come with sudden power loss or crashes of the untrusted file system.

This paper addresses these issues. To minimize damage caused by an unclean shutdown, jVPFS carefully splits a journaling mechanism between a trusted core and the untrusted file system. The journaling approach minimizes the number of writes needed to maintain consistent information in a Merkle hash tree, which is stored in the untrusted file system to detect attacks on integrity. The commonly very complex and error-prone recovery functionality of legacy file systems (in the order of thousands of lines of code) can be reused with little increase of complexity in the trusted core: less than 350 lines of code deal with the security-critical aspects of crash recovery. jVPFS shows acceptable performance better than its predecessor VPFS, while providing much better protection against data loss.

1 Introduction

Both VPFS and its successor jVPFS are built in response to the observation that the enormous code bases of monolithic OSes (hundreds of thousands to millions of lines of code) are likely to contain exploitable weaknesses that jeopardize platform security. Apparently, this observation is valid especially for mobile devices that currently have the highest speed of hardware technology innovation. Almost daily reports, for example on successful attacks on core system components such as drivers [2], USB stacks [3], passcode protection [4], common applications such as text messaging [5] or on "jailbreaks" [6], which consistitute successful attacks, too, substantiate that claim of significant vulnerability. On the other hand, as smartphones, tablets and similar appliances have evolved into powerful and versatile mobile computers, professional users are starting to use them for critical data. 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. Or a mobile device may store documents that are classified or contain trade secrets. Mobile payment systems on the other hand have strong integrity requirements to prevent tampering. Yet mobile devices are frequently connected to insecure networks (public WiFi, etc.) and in certain situations, users even must hand them over to untrusted third parties (e.g., leave them at the reception when visiting a company).

A general approach that so far seems mostly attractive for safety critical systems and to the miltitary is based on small isolation kernels or microkernels. Such kernels strongly separate applications, but also operating system components. Some of them [7], then called "hypervisors", contain the basic functionality to support virtual machine (VM) monitors and legacy OSes as guests. Based on such kernels, critical applications can run in their own compartments (built on microkernel services or in their own VMs) that are protected even against successful attacks on drivers or other parts of large, insufficently secure legacy OSes. Related work [8, 9] has also shown that applications can be split such that their security-critical cores run isolated, but reuse untrusted parts of the system for their non-critical functionality, thereby reducing the trusted computing base (TCB) of these applications by several orders of magnitude.

VPFS and jVPFS: File Systems for Microkernels. In previous work on VPFS [1] we built a file-system stack

that leverages a microkernel-based isolation architecture to achieve better confidentiality and integrity protection of application data. We achieved this by splitting the file-system stack into a small trusted and a larger untrusted part that reused Linux file-system infrastructure. Only the former is within the file-system TCB (see Figure 1 for an architectural overview). As VPFS uses untrusted components, which might be penetrated or otherwise corrupt data, integrity guarantees can only cover tamper evidence: manipulated data is detected through checksum mismatches and never delivered to applications. Unfortunately, better integrity protection also reduces availability of file-system data in the event of an unclean shutdown; for example, checksums may no longer match the corresponding file contents, if the battery of the mobile device failed unexpectedly or the system crashed.

With jVPFS, we address the problem of ensuring both robustness and integrity in a split file-system stack as described above. In monolithic file system stacks, the code for ensuring consistency of on-disk structures (e.g., journaling, soft updates [10]) is rather complex and difficult to get right [11, 12]. Recent research [13, 14, 15] has shown that even file system implementations that are widely used in production environments still have bugs, commonly found in code paths used for error handling and post-crash recovery. A subset of these bugs are security critical [2]. It is therefore a primary design goal for us to keep the inherent complexity of consistency mechanisms out of the file-system TCB in order to lower the risk of introducing exploitable design and implementation errors. Nevertheless, existing file system implementations are well tested and sufficiently reliable in common application scenarios (when not subject to sophisticated attacks). For practical reasons, it is therefore desirable to reuse this infrastructure in order to reduce engineering effort.

Contribution. The work presented in this paper makes the following contributions: We extend the file-system TCB for *confidentiality*, *integrity*, and *freshness* of all data and metadata such that these protection goals can be reached even after an unclean shutdown. To this end, we identify and isolate the security-critical functionality required to recover a consistent file system after a crash and discuss how existing, untrusted consistency infrastructure can be reused to complement the security-critical part. We devise a novel cooperation scheme that lets trusted and untrusted components cooperate and discuss precisely which metadata information must be revealed to untrusted code in order to facilitate this cooperation. We evaluate a prototype implementation.

Synopsis. On the following pages, we first provide required background on our security model and then present our design and optimizations in Sections 3 and 4, respectively. We evaluate our prototype in Section 5, before discussing related work in Section 6.

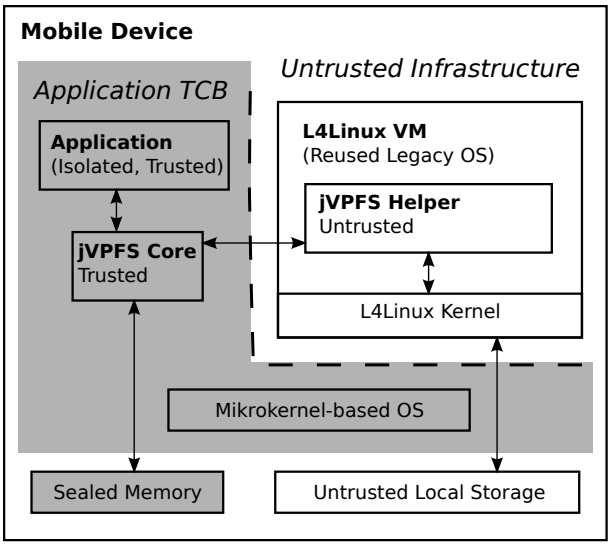

Figure 1: *jVPFS in a decomposed system architecture with strong isolation among components.*

2 Background

Figure 1 gives an overview of the system architecture into which the split jVPFS stack integrates. Strictly following the principle of least privilege, file-system contents are accessible only to the specific application that owns them. The small file-system kernel of jVPFS implements all security-critical functionality and reuses the file system stack provided by an untrusted, virtualized legacy OS. That is, Linux performs all non-critical tasks to manage persistent storage.

2.1 Security Model

The jVPFS security model is identical to the one for the original version of VPFS [1]; we summarize it here. We consider a strong attacker who is trying to compromise *confidentiality*, *integrity*, or *freshness* of critical data stored in a VPFS file system. Confidentiality means that, after authorization, the user can access his data only through a specific application. Our notions of integrity and freshness apply to both user data (file contents) and metadata (filenames, sizes, timestamps, etc). The file system provides only complete and correct data and metadata to the application. We require that VPFS can detect any tampering and whether data and metadata are up-to-date, preventing an attacker from rolling back file-system contents to an older version without being no-

ticed. We assume that both software and certain hardware attacks are possible. At run time, VPFS relies on hardware address spaces and virtual-machine boundaries in order to isolate the trusted and untrusted components effectively. To counter offline attacks, VPFS requires the mobile device to enforce a secure startup process of the application TCB and a small amount of access-restricted, tamper-resistant memory to store cryprographic keys and a checksum for integrity checks.

Software-based Attacks. Both software and data stored in the mobile device may be tampered with. Given the high complexity and enormous code size of the virtualized legacy OS, we must assume that the attacker can fully compromise it; hence, untrusted components may stop working correctly at any time. Nevertheless, we assume that in the common case, when not being attacked, they function as expected and cooperate with the trusted part of VPFS. In the case of jVPFS, the untrusted infrastructure is expected to store cryptographically protected file-system contents persistently, taking any necessary consistency constraints into account.

Components within the TCB are considered to be significantly harder to attack, because their isolated codebases present a smaller attack surface. We assume that they either work correctly, or not at all, should the secure boot process of the mobile device detect that their executable files or configuration have been tampered with.

Hardware-based Attacks. We assume that an attacker is able to directly access or manipulate the device's mass storage (e.g., a flash memory card), but he cannot successfully read or manipulate the contents of the tamper-resistant memory or break the secure boot process. To ensure tamper resistance, the device could be equipped with a trusted platform module (TPM) [16] or a small amount of secure flash memory that is directly integrated into the system-on-chip (SoC) package. Access to the secure flash memory must be restricted to certain software stacks by means of secure boot, possibly augmented with hardware-based access control as enabled by the ARM TrustZone [17] technology.

Secret keys stored in a TPM can be extracted with equipment that costs in the order of hundreds of thousands of dollars, but the process is destructive. Similarly, we assume that gaining direct access to the secure flash in the SoC is too hard for the attacker. Making a user-provided secret such as a PIN code part of the storage encryption key limits benefits of such an attack further.

2.2 Cryptographic Protection

Earlier work on cryptographic storage systems (e.g., [18, 19]) shows how file-system contents can be protected against offline attacks by using encryption. In jVPFS, AES-CBC encryption ensures confidentiality at the block level, thereby enabling efficient partial updates.

The state of the art technique for efficiently ensuring integrity and freshness is to use a Merkle hash tree [20]. By construction, the tree provides all necessary information to verify correctness and completeness of file-system contents; one can guarantee freshness by storing the root hash of the tree in tamper-resistent, persistent memory. In our system, the entire file system including its metadata (names, etc.) is protected by the Merkle tree. Simpler approaches without using a tree structure do not meet our requirements: Single hashes for large regions or entire files make partial updates expensive; furthermore, storing one independent hash per region or file requires impracticably much tamper-resistant storage. Using keyed hashes (e.g., HMACs [21]) instead makes hashed data vulnerable to roll-back attacks, thereby defeating freshness.

3 Design

In jVPFS, the Merkle hash tree is essential to strong integrity and freshness guarantees. It must always be possible to restore it to a consistent state. The design of the jVPFS consistency mechanism is driven by the principle of least privilege, aiming at a minimal attack surface of the implementation in order to increase its trustworthiness. There are two key challenges to reaching these goals:

1. **Making the Cut.** Part of our integrity requirement is that data provided to applications is complete; freshness demands that the latest file-system state is available. The consistency mechanism thus has security-critical functionality that must be identified and isolated from uncritical, potentially untrusted components in an efficient and effective way.

2. **Secure Cooperation.** Despite isolation, the two parts of the file-system stack must be able to cooperate. Therefore, at least some information describing consistent sets of updates must be revealed to and processed by untrusted components whithout jeopardizing confidentiality, integrity, and freshness of file-system state.

3.1 Consistency Paradigm

After a crash, the on-disk structures in the untrusted storage must contain enough information to restore the Merkle hash tree spanning the file system data and metadata.

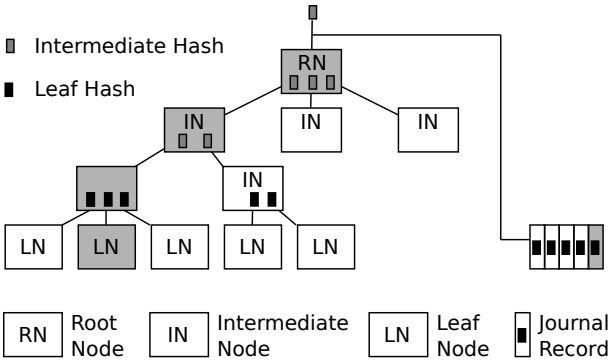

Figure 2: *Updating the Merkle hash tree: modification of one leaf node requires the complete chain of intermediate nodes up to and including the root node to be updated. Instead, it is more efficient to append leaf-node hash sums to a growing journal, which is cryptographically bound to the root hash.*

Naive Approach. The naive approach to meet this requirement is to ensure that the untrusted legacy file system always holds a consistent snapshot of the Merkle tree. In such a hash tree, changes are first made to the leaf nodes and then propagate to the root. Thus, modifications of the on-disk file system (e.g., writing new data to a file) must always include updates of all intermediate Merkle tree nodes up to the root node. This set of updates to the on-disk user data (i.e., file contents) and the Merkle tree nodes has to be applied *atomically*. In this context, *atomic* means that, after a system crash, either the complete set of updates is persistent, or the previous state is accessible—otherwise the hash sums become inconsistent, breaking the authentication chain. Figure 2 illustrates these update dependencies.

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 (i.e., disk blocks) as there are levels in the tree. For each of these blocks the trusted part of jVPFS has to perform cryptographic operations, further increasing run-time overhead and energy consumption. Also, modifying file contents might require updates of metadata such as file-size information, which is stored in an inode that must be protected by the Merkle tree, too. Thus, the costs of consistent, atomic updates rise even more.

Split Journaling Approach. We therefore explored design alternatives in order to avoid the performance impact of the naive approach. Being the key reason for slow performance, the requirement always to have the on-disk Merkle hash tree in a consistent state needs to be relaxed. In fact, it is desirable to omit updates of Merkle tree nodes for short periods of time, for example, during high load or to minimize latency. In order to have the required hash sums available for post-crash integrity checking nonetheless, they need to be written to an alternative, more efficient data structure. Journaling file systems solve this problem: they allow for efficient and atomic updates of distributed file data and metadata, which in our case includes hash sums that enable integrity checking. A growing journal to which hash sums of updated leaf nodes are appended eliminates the need to immediately update disk blocks that store higher-level tree nodes; they may be flushed from the buffer cache later. This strategy also reduces cryptographic overhead, as only the leaf nodes of the tree are updated frequently.

Protecting the Journal. As the journal now contains information that is critical to ensuring integrity, it must be cryptographically protected, too. To keep the performance benefit, appending records to the journal must not require additional updates of other metadata (e.g., like the root node of the Merkle tree). We ensure journal integrity by continously hashing all appended records. New records are written to the end of the journal together with a new incremental hash sum that authenticates all preceding journal content, thereby enabling incremental integrity checking. To prevent forging of these journal hash sums, we first hash a random secret that is kept in tamper-resistent sealed memory and therefore unknown to an attacker who is trying compute new hashes. Additionally, we encrypt confidential metadata in the journal using AES in CBC mode. This incremental, keyed hashing and encryption scheme is well-understood and, for example, used by Maheshwari et al. in TDB [19].

We will discuss the frequency and granularity of intermediate journal hash sums in Section 3.5, following a detailed discussion of the structure and semantics of jVPFS journal records in Sections 3.3 and 3.4.

3.2 Architecture Overview

We are building on our previous work on VPFS [1] and integrate a consistency mechanism into its architecture. Figure 3 gives an overview of the jVPFS stack. The three top layers only deal with the concept of files, a namespace, and per-file security-critical metadata. They essentially implement a memory file system within the TCB. Another trusted layer called *Sync_manager*, which is located directly underneath this memory file system, implements support for making jVPFS state persistent. Sync_manager is called by the buffer cache whenever data needs to be read into cache buffers or evicted from them. Applications can influence write back through explicit operations such as fsync(), if required.

In order to avoid the complexity of managing a physical storage medium in its own codebase, Sync_manager

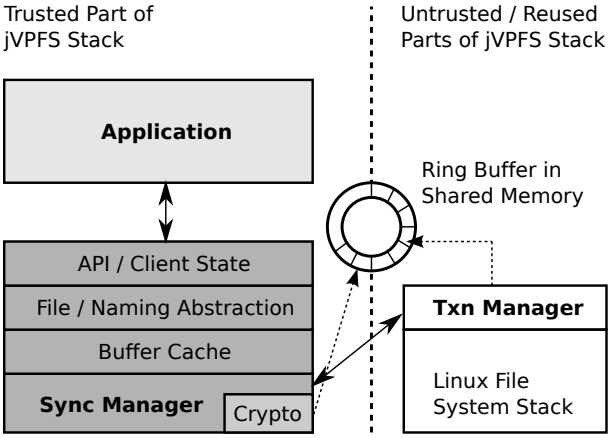

Figure 3: *Detailed view of the jVPFS file-system stack: jVPFS implements a memory-based low-complexity file system within the TCB of the application using it. Through Sync_manager, jVPFS reuses an untrusted Linux file system to make file system contents persistent.*

maps files in the virtual private file system (as seen by the trusted application) to files in an untrusted Linux file system. To distinguish between these two views of a file, we shall refer to the latter ones as *file containers* in the *untrusted storage*.

Cooperation. Like the trusted part of the original VPFS, Sync_manager transparently encrypts and decrypts all data and metadata it exchanges with untrusted components in the storage strack. Furthermore, it calculates and verifies cryptographic hash sums in order to ensure integrity of any data and metadata it receives from untrusted code. In jVPFS, it also performs a minimal amount of state tracking so as to ensure that only consistent changesets are written to persistent storage. Cooperation between Sync_manager and the Linux infrastructure it reuses is enabled by the untrusted *Txn_manager*. It receives from Sync_manager requests and consistency-related hints and translates them into Linux file-system calls. That is, Txn_manager writes cryptographically protected data to file containers and appends records to the journal, which is a file in the Linux file system, too. jVPFS makes extensive use of existing infrastructure, as it exploits any consistency guarantees the underlying Linux file system might provide (e.g., write ordering).

Communication Interface. Trusted and untrusted parts of the jVPFS stack cooperate using a narrow message passing interface and a ring buffer located in a shared memory area. Table 3.2 lists all message types. During normal operation, Sync_manager sends Read_block and Exec_ops messages to request uncached

Message type	Description
Read_block	Read a specific data block
Exec_ops	Execute buffered operations
Write_checkpoint	Create a new journal, which also marks a new checkpoint
Read_checkpoint	Read FS root info from last consistent checkpoint
Read_journal	Read set of complete transactions from journal
Init_shm	Set up shared memory once

Table 1: *Complete list of message types that Sync-manager uses for communication with Txn_manager.*

data blocks, or to flush operations and journal records queued in the ring buffer. Txn_manager handles these requests and writes back encrypted data blocks that are referenced by journal records. Messages of type Read_checkpoint and Read_journal are exchanged at mount time and, if necessary, during recovery. We shall explain the semantics of the Write_checkpoint message in the following section, which covers our approach to journaling and checkpointing of on-disk state.

3.3 Being Prepared for Crashes

Journaling in jVPFS is done at the level of metadata operations. Starting from a consistent set of file containers, which contain the latest *checkpoint* of all file system contents, data blocks are written to untrusted storage and any associated modifications to metadata are logged to the journal. We do not log full blocks of metadata, but only descriptions of specific operations such as updating hash sums or file sizes. Occasionally, Sync_manager flushes all cached blocks—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. The general idea for recovery after an unclean shutdown is to replay all journaled operations, iteratively updating metadata structures from the latest checkpoint.

Metadata Dependencies. Operation-level journaling allows for simple tracking of metadata dependencies in Sync_manager, which is important for our objective to minimizing complexity within the TCB. Figure 4 illustrates the dependencies of standard file-system metadata:

Inode: The inode of a file contains the file size in bytes that specifies how much data in the last data block is valid file content. In jVPFS, the inode also stores the root hash of the Merkle subtree that protects the file's contents. The inode itself is stored in the inode

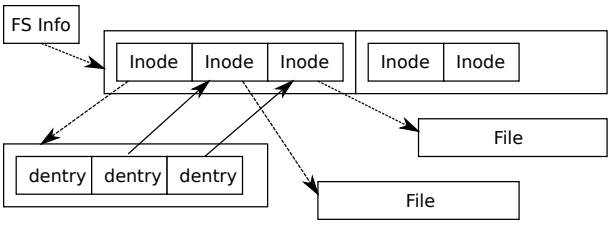

Figure 4: *Dependencies among standard file-system data structures: entries in directories point to inodes, which are associated with files. The superblock-like FS info in jVPFS contains the root hash of the Merkle tree.*

file, which is mapped to its own file container that contains the top levels of the Merkle tree.

Pathname: The inode of a file is referenced by a directory entry, which is stored in a directory file. Directory files form a hierarchical namespace, as directory entries can reference not only regular files, but also other directory files.

The dependencies described above are critical to the consistent representation of a file and must be obeyed by Sync_manager. They translate into the requirement that, for each newly created file, the corresponding inode and the directory entries along the file's pathname need to be written, too. The file and namespace abstraction in the upper layers of the jVPFS TCB already implements most of the required book keeping; a consistent checkpoint will have all this metadata stored in file containers. However, the consistency mechanism that Sync_manager contributes to jVPFS maintains additional state, such that it can log inode and namespace updates to the journal. For each newly created file (including directory files), it keeps the following information in memory:

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

Sync_manager stores this information in a simple table, which is essentially an extension of the file descriptor table. This volatile *new-file* state is complemented by a one-bit flag in the inode, indicating whether inode and filename of a file have already been written. Sync_manager executes Algorithm 1 to make sure that journal replay restores the inodes and the namespace for newly created files: for a new file and all directories along its pathname, it recursively appends to the journal in reverse path order a File_create record, unless this information has already been logged (or exists in an older checkpoint). Once a File_create record containing a copy of a new file's inode, a pointer to the parent directory's inode, and the filename has been journaled, data blocks can be written to the file container (see next paragraph).

Algorithm 1 Journaling metadata for new files.

```
function journal_file_create(File *file) {
  // check, if file creation has already been logged
  if (file.inode.is_logged == true)
    return;
  // not announced in journal, get "new file info"
  struct new_file *info = new_file_info_table[file.fd];
  // lookup of file descriptor of parent dir: if it is
  // still open, its creation may need to be logged, too
  int p_fd = info.parent_file_handle;
  int p_iptr = info.parent_inode_ptr;
  File *p_dir = get_file_descriptor(p_fd, p_iptr);
  if (p_dir != NULL) {
    // parent dir still open, make sure it is logged
    journal_create_file(p_dir);
  }
  // announce new file in journal
  file.inode.is_logged = true;
  File_create_rec rec(file.inode, p_iptr, info.name);
  append_to_journal(rec);
}
```

Metadata operations such as rename() or unlink() are logged analogously (i.e., with their parameters). Note that allocation bitmaps or other free-space information need not be considered, because the security-critical part of jVPFS delegates this functionality to the untrusted legacy file system. Also, no explicit write-barrier operations are required, as partial replay of the generated journal records may, at worst, recreate an empty directory or a zero-size file.

Writing User Blocks. The key requirement to be met when writing a block at the leaf level of the Merkle tree is that its hash sum needs to be written to the journal first. Sync_manager prepares write back of user-data blocks; it performs the following operations:

1. Calculate new hash sum over plaintext of updated block's contents.
2. Encrypt block, put ciphertext into free buffer space in shared memory area.
3. Put Block_update record containing updated hash sum and new file size into ring buffer.

Actually writing the data block to persistent storage is done by the untrusted parts of the jVPFS stack. Txn_manager ensures atomicity of block writes by enforcing the following three constraints (note that this scheme is conservative and potentially expensive in

terms of write barriers; we will discuss optimizations for the common case, including write batching, in Section 4):

1. Updated hash sums must reach the journal before the actual block is written to the file container. The underlying legacy file system must be made aware of this write-before relation, for example, by calling fsync() on the journal file.

2. Should a system crash interrupt the write back operations initiated by Txn_manager, the aforementioned order for journal and block writes ensures that either (a) the new hash sum 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 checksum still available in the corresponding on-disk Merkle tree node.

3. Before updating the same block a second time, Txn_manager must make sure that the first update reached stable storage, because of point 2.

It is assumed that the underlying legacy file system can guarantee that aligned writes with a size equal to its own block size are atomic. Most journaling file systems do meet this requirement in their standard configuration using ordered or data journaling [14].

Writing Metadata Blocks. Case 2(b) mentioned above implies that the previous version of a block's hash sum is guaranteed to be available during replay. To meet this requirement at all times, Txn_manager treats metadata blocks differently from blocks with user data. Metadata blocks contain either intermediate nodes of the Merkle tree, or any block from a directory or inode file. Whenever a metadata block is flushed and would overwrite the latest checkpointed version of itself, Txn_manager rescues a copy of the original version into the journal, thereby preserving it in case replay becomes necessary. Consequently, it is not necessary to log hash sum updates of metadata blocks. Sync_manager flags Block_update records as user or metadata, such that Txn_manager can handle the two block types correctly.

Checkpoints. Our split journaling scheme ensures that critical metadata can be restored after a crash. However, letting the journal grow indefinitely would effectively make jVPFS a log-structured file system [22]. This class of file systems requires complicated garbage collection, which in turn would add considerable complexity to the TCB (i.e., Sync_manager). Sync_manager therefore flushes all dirty user and metadata blocks occasionally. Once no more dirty state is in the trusted buffer cache, it signals Txn_manager with a Write_checkpoint message that a new consistent checkpoint can be established. The untrusted Txn_manager then execute the following steps:

1. Process all journal records still queued in ring buffer, submit all block updates to legacy FS.

2. When encountering a special Checkpoint record, instruct legacy FS to make all file containers persistent.

3. Atomically swap current journal file with newly created journal containing just the Checkpoint record.

Note that flushing the buffer cache must be part of the TCB to support any persistency scheme. However, the above checkpointing algorithm does not require any garbage collection in security-critical code, nor does it have to deal with the complexities of writing data to the storage medium safely. It is easy to see how a jVPFS instance can be fully reinstantiated from checkpointed file-system state (in fact, the umount() operation in jVPFS is identical to the checkpoint operation). In the following section, we shall discuss how to restore post-checkpoint state after an unclean shutdown.

3.4 Recovering From Crashes

If the system did indeed crash, the untrusted commodity file system must recover first. Once the untrusted storage has been remounted and the virtualized Linux is booted up, jVPFS can start its own recovery process as we shall now explain.

Mounting a Checkpoint. At mount time, Sync-manager requests from Txn_manager the first record stored in the journal, which is the Checkpoint record containing the FS root info. Sync_manager decrypts the FS root info using the platform's sealed memory implementation and validates its integrity and freshness. Note that the write-back strategies explained in the previous section ensure that this operation succeeds even after an unclean shutdown. However, this assumption may not hold, if a successful attack (see attacker model in Section 2.1), a hardware failure, or a software issue outside the TCB damaged the first journal record. If the Checkpoint record could not be read or validation fails, an integrity error is reported to the application and the file system remains inaccessible. If the journal contains just the Checkpoint record, Txn_manager switches to normal operation mode and behaves as described in Section 3.3. Otherwise it prepares replay.

Preparing Replay. Our operation-level journaling scheme makes the following assumptions:

1. To ensure consistency and integrity, operations specified in journal records must be replayed in the precise order in which they were logged.

2. Journal records encode incremental updates to metadata blocks and modify the previous version of the block, starting with the version that was valid in the last checkpoint.

Requirement 2 dictates that, if there is a checkpointed 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 in-place locations just before replay starts.

Replaying Metadata Operations. Replay is cooperatively performed by both Sync_manager and Txn_manager: the former requests journal records by sending a Read_journal message. Txn_manager responds by filling the ring buffer with a set of records that end with a special record carrying an intermediate keyed hash, which authenticates all preceding journal records (see Section 3.1). Sync_manager then executes the following replay algorithm for each set of journal records provided by the untrusted part:

1. Decrypt all journal records in shared ring buffer, put decrypted versions into private memory buffer, which is inaccessible to untrusted code so as to prevent time-of-check-time-of-use (TOCTOU) attacks.

2. Check integrity of decrypted records using keyed hash sum from last record; in case of mismatch, abort and report integrity error.

3. Re-execute operations specified in all records.

To replay metadata operations such as creating, moving, or unlinking files, Sync_manager reuses existing jVPFS APIs and executes the same code paths that handle calls from an application; the required parameters are extracted from the respective journal records. Handling of filenames is slightly different, as those are recorded relative to their parent directories, which are referenced by their inode number rather than a full pathname.

Handling File Contents. Replaying update records for user data blocks is performed similarly as part of the above algorithm: updated file size information is written to the inode, the hash-sum update is applied to the direct parent node in the Merkle tree. Note that this parent node can always be retrieved and authenticated, because either it was never overwritten or, as a metadata block, it has been preserved in the journal—or an updated version has been generated earlier during replay.

However, since user-block contents are not journaled, file containers always contain the latest version of a block that reached stable storage. On the other hand, the journal may contain multiple Block_update records for the same block. Therefore, Sync_manager skips out-of-date hash sums until it finds the correct record for the user block. It eagerly requests each block during replay, checks its integrity, and applies the hash sum update if it matches the block's contents. A correctly behaving Txn_manager that obeys write-ordering constraints can always provide the latest matching version; misbehavior results in a stale hash sum that will be detected eventually.

We shall evaluate the complexity of jVPFS' consistency mechanism in Sections 5.1 and 6.

3.5 Journal Details

Now that we introduced the various types of journal records, we take a closer look at how the journal is protected in detail.

Confidentiality. The journal contains confidential metadata information such as filenames, so its contents must be encrypted. All payload data of the journal records, including parameters that are passed to internal jVPFS APIs during replay, are encrypted. However, as the untrusted Txn_manager must update file containers in a consistent way based on Sync_manager's constraints, some information cannot be concealed. In particular, we keep the location of data blocks unencrypted, such that untrusted code can write them to their correct locations. Furthermore, we reveal the type of blocks (user or metadata), so as to enable Txn_manager to preserve consistent checkpoints, which are essential for recovery. We consider this an acceptable tradeoff, because an attacker could also learn this information by observing access pattern in the Linux VM.

Integrity. The continuously calculated keyed hash that protects the journal is anchored in the FS root info of the last checkpoint through a random secret stored in it. Thus, a journal is bound to exactly one checkpoint. By embedding intermediate hash sums into the journal, Sync_manager can designate transactions; records between two intermediate hashes can only be authenticated all together, thereby preventing partial replay. We exploit this construction to ensure that security-critical metadata operations described by multiple records are replayed completely or not at all (replay stops in the latter case).

Freshness. Naturally, incrementally calculated hashes cannot reliably mark the end of a data stream (as the HMAC [21] scheme does). As a result, Sync_manager cannot determine from hash sums in the replayed journal, if untrusted components withhold any transactions from the end of the journal; doing so would constitute an attack on freshness. By storing the latest journal hash in tamper-resistent sealed memory before a crash occurs, Sync_manager could detect such an attack during replay: the hash marking the last replayed transaction must match the trustworthy copy preserved in sealed memory. For performance reasons, updates of sealed memory should be done only once for each checkpoint, or an application may request a freshness guarantee explicitely through an fsync()-like operation for transactions between checkpoints.

In our prototype implementation, sealed memory updates are currently dummy operations.

3.6 Managing File Containers

Removal of a file or—in the general case—truncation of it is a metadata operation that jVPFS must log in the journal. However, care must be taken when actually truncating the underlying file container. Assume that recovery becomes necessary after an unclean shutdown. Sync_manager can replay the truncate operation, however, as journal replay always starts relative to a checkpoint, other operations need to be re-executed before it. Some of these operations may depend on file contents that are to be removed, which must therefore still exist for replay to succeed. This is particularly important for metadata files (i.e., the inode file and directories), as logged operations may need to modify them during replay. As a consequence, we must not truncate file containers in the legacy FS right away—even if file truncation has already been logged. Txn_manager therefore builds a list of file truncation requests from truncation records it receives from Sync_manager; once a new checkpoint is persistent, truncated parts of files will finally be obsolete and Txn_manager will garbage collect them in idle time.

4 Optimizations

Intuitively, one would assume that ordered updates of the journal and file containers incur significant performance overhead. However, I/O costs can be reduced drastically by optimizing *untrusted* code.

Write Batching. New journal records and encrypted data blocks are buffered 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 legacy file system requires fewer I/O operations and at most one write barrier to update the journal file. Second, the legacy file system may write blocks to file containers according to its own optimized strategies, potentially achieving higher performance.

Relaxed Write Order. A write barrier after updating the journal ensures that user blocks can be updated safely. Synchronizing in-place updates of user data blocks that may still be in-flight allows Txn_manager to submit new updates to those same blocks again. However, many common write workloads do not perform any block updates at all (e.g., writing new files or growing them). For these types of workloads, where no old state is modified, the consistency-preserving write-order requirements of jVPFS can be dropped entirely: Txn_manager and the legacy file system may update untrusted storage without enforcing write order, because new data blocks can only be replayed once both their hash sums and the content are persistent; it does not matter which is written first, as long as both are present during replay. Incomplete writes of block–hash sum pairs are treated as if no write operation had been performed at all. In combination with write batching, jVPFS thus achieves I/O overheads close to that of the reused legacy file system, with only few addtional writes to the journal file and occasional checkpointing of Merkle tree nodes.

Note that the functionality for the just described relaxation is implemented almost entirely in untrusted components. Sync_manager only provides a hint indicating whether an older block exists; the hint is trivially computed by checking if the current hash sum in the parent node is null or not.

Out-of-Order Reads. Many write operations can be queued in the ring buffer and it may take a long time to process them. Txn_manager checks if block read requests it receives asynchronously are independent of pending writes; if they are, it handles the read requests immediately without the latency that flushing of pending writes first would cause.

Exploiting Existing Infrastructure. For our evaluation presented in Section 5, we used ReiserFS [23] and NILFS [24] as the underlying file system. With ReiserFS, 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 jVPFS requires: in the common case, we just require that certain I/O operations do not overtake each other (e.g., journal records with updated block hash sums are written before modifying the file container or not at all).

The log-structured file system NILFS can ensure a strict order of write operations without calling an explicit API such as fsync(). NILFS ensures that writes reach stable storage in the same order in which an application issued them. Our prototype implementation of Txn_manager exploits this behavior to eliminate I/O delays caused by fsync(), if possible. We extend reuse of existing consistency support even further by leveraging support for efficient checkpointing of file system state that is built into NILFS. Whenever Sync_manager wants to checkpoint its own file system state before starting a new journal, we create a checkpoint in the underlying NILFS file system.

5 Evaluation

We built jVPFS on a platform based on the Fiasco.OC [7] microkernel from the L4 family. The kernel ensures strong isolation of trusted and untrusted components and uses kernel-protected capabilities to enable secure resource access. The trusted part of jVPFS and test applications utilize libraries and services of the L4Re user-level environment [25]. jVPFS hooks into the generic, POSIX-like VFS interface of L4Re. We use L4Linux [26], a paravirtualized Linux 2.6.36 kernel, to run the untrusted parts of the jVPFS stack.

5.1 Complexity

Table 2 shows a breakdown of the source complexity of the jVPFS stack, which is written in C++ (cryptographic library routines are in C). All figures were generated using David A. Wheeler's 'SLOCCount' [27]. In addition to the subsystems listed in the table, jVPFS also reuses an AVL-tree implementation that is part of the TCB of any L4Re application. It comprises approximately 800 lines of C++ code. The L4Re VFS supports file-system plugins and is also linked to any L4Re application.

The main contribution of jVPFS compared to VPFS is its new persistency layer. In our prototype implementation, it comprises 729 source lines of code (SLOC). The functionality that implements journaling and replay of metadata operations requires 325 SLOC, including cryptographic protection as explained in Section 3.5. This is an order of magnitude smaller than in typical monolithic file systems; for example, the journal block device layer (JBD2) for Ext4 comprises almost 5,000 SLOC in Linux 2.6.36. We attribute this significant reduction of complexity to key design decisions in jVPFS: First, the logic to add operation-level journaling is a simple extension of the code that implements write batching using the shared ring buffer. We mainly added additional record types for different operations (e.g., File_create or File_unlink) and consistency state tracking. Second, Sync_manager reuses the same API entry points as the VFS layer to replay operations; parameters for API calls are retrieved from journal records. We implemented less than a dozen SLOC for replay of each type of operation in a switch statement. The remaining 404 SLOC of Sync_manager's current implementation would be required for persistency anyway (e.g., transfer of data blocks, shared memory setup, ring buffer logic).

Subsystem	**SLOC**
L4Re: VFS	2,303
jVPFS: memory file system	2,444
jVPFS: Sync_manager (persistency)	404
jVPFS: Sync_manager (journal/replay)	**325**
L4Re: libcrypto	667

Table 2: *Source complexity of jVPFS: Sync_manager contributes 729 lines of code to the TCB. Only 325 lines of code are related to journaling and replay.*

The functionality in the TCB could only be reduced this much, because Txn_manager (which is approximately 1,300 SLOC in size) makes extensive reuse of the complex untrusted Linux file system stack.

5.2 Write Performance

Due to space constraints, we focus our performance evaluation on write and metadata-intensive benchmarks and recovery. We did all benchmarks on the same hardware configuration we used for performance evaluation of the original version of VPFS [1]. The evaluation machine has two 2.0 GHz dual-core Opteron processors and 2 GB of DDR RAM. We restricted the hardware resources to one core and 256 MB of physical RAM in all benchmarks. We used two storage mediums, a 80 GB SATA hard disk (Samsung HD080HJ) and a USB flash disk (Buffalo Firestix, 1 GB).

Using strace, we recorded all file-system calls that benchmarking tools executed on Linux. Like we did for VPFS, C++ programs generated from these traces were compiled for L4Re and used to replay all file-system operations on a jVPFS stack; we also ran Linux versions of the trace players on native Linux without any encryption to establish a baseline. Native Linux could use the full 256 MB of RAM, whereas the jVPFS configuration allocated 64 MB of it to its trusted buffer cache that is isolated from L4Linux. Some traces were also used to

Trace	VPFS	jVPFS
PM-1	2.52 s	1.02 s *(0.16 s)*
bonnie++ (encrypted)	32.0 MB/s	38.4 MB/s
bonnie++ (plaintext)	42.0 MB/s	53.1 MB/s

Table 3: *Performance comparison between jVPFS and original version of VPFS using ReiserFS on the hard disk (VPFS figures taken from [1]).*

Trace	Storage	Base	w/o Jrnl	w/ Jrnl
PM-2	ReiserFS HDD	5.11 s *(0.10s)*	6.37 s *(0.14s)*	9.56 s *(0.27s)*
PM-2	NILFS Flash	27.12 s *(0.20s)*	12.36 s *(0.60s)*	13.49 s *(0.54s)*
untar	ReiserFS HDD	1.61 s *(0.06s)*	2.07 s *(0.02s)*	2.14 s *(0.03s)*
untar	NILFS Flash	7.09 s *(0.04s)*	9.65 s *(0.09s)*	9.83 s *(0.13s)*

Table 4: *Execution times and standard deviation for benchmarks of jVPFS with and without journaling enabled, compared against native Linux as a baseline.*

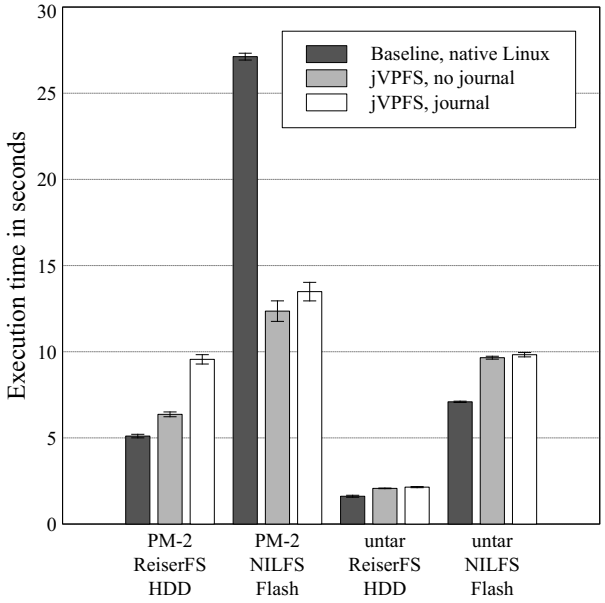

Figure 5: *Benchmark results for Postmark and untar traces; see Table 4 for exact values of execution times with standard deviation.*

benchmark VPFS [1], so we can roughly compare jVPFS against its predecessor, too. Unless stated otherwise, all benchmarks were run ten times and the results averaged; we give standard deviations for increased confidence.

Throughput. We first tested throughput performance by writing two 1 GB files using a bonnie++ trace (see Table 3). With metadata journaling, jVPFS achieves 38.4 and 53.1 MB/s for encrypted and for plaintext files, respectively, with ReiserFS on the hard disk. When we disabled jVPFS journal writes, the underlying legacy file system was eight percent faster to write the unencrypted, but integrity-protected, file containers: effective throughput was 57.1 MB/s, which is close the 58.1 MB/s we measured for native Linux (figures not in Table 3). jVPFS clearly outperforms VPFS (32 and 42 MB/s).

PostMark. PostMark is a synthetic benchmark that creates, modifies, and then deletes a large number of files. The PM-1 trace we used to measure the original VPFS configuration operated on 5,000 files with a size in the order of a few kilobytes [1]. We replayed this mostly-cache workload on jVPFS, which shows significantly better performance than the older VPFS. Another PostMark trace, PM-2 with ten times as many operations on 50.000 files, causes a large number of evictions from the trusted buffer cache and writes to the storage medium. We used this metadata-intensive trace to measure the journaling overhead (see Table 4, or Figure 5 for visual representation). With ReiserFS on the hard disk providing the untrusted storage, we see a 1.5x overhead when journaling is enabled, and about a factor of two compared to the baseline. We expected such a behavior, because calling fsync() on the journal file when required for consistency is expensive on magnetic disks.

With NILFS driving the flash disk, we found that jVPFS can actually perform better than running the benchmark natively in Linux. We determined the strict write ordering of NILFS to be the cause for this unintuitive result: PostMark frequently modifies the small files it created, causing a large number of serialized block updates (i.e., log writes) in NILFS. The jVPFS buffer cache absorbs many of these updates, such that less data actually reach the legacy file system. On the other hand, our system greatly benefits from the ordering guarantees of NILFS, as it does not require synchronous writes to update its journal. Nevertheless, journal writes do cause increased write traffic, as can be seen in the figures. We measured 3.5 MB of journal records, but they arrive in groups smaller than the NILFS block size, thus causing the 9 percent journal I/O overhead we measured.

Untar. The untar trace simulates unpacking a tar archive with thousands of small and large files (kilobytes to megabytes); when done, it flushes all data and metadata to stable storage. We measured 3 and 2 percent journaling overhead for the ReiserFS/HDD and NILFS/flash configuration, respectively. This overhead correlates well with the actual size of the journal file, which ac-

counted for 2.3 percent of all data written to the untrusted storage. These figures are lower than for the PostMark benchmark, as there is a number of very large files among the more than 3,000 files and directories that are created—those files dominate write traffic.

The total overhead of the jVPFS stack over the Linux baseline in this benchmark is 33 percent for ReiserFS and 39 percent for NILFS. Virtualization, increased communication costs, and cryptographic operations contribute to this overhead. While significant, but we believe this overhead is acceptable considering the security advantages jVPFS has over monolithic systems.

5.3 Recovery Performance

We tested jVPFS' recovery functionality using the untar trace both in simulation and on real hardware.

Simulations. In simulation mode, we let Txn_manager terminate itself after it logged a specific amount of data to the journal (about 70 percent of the files were written up to that point). We did not power-cycle the machine, but only restarted Txn_manager and the trusted component of jVPFS. Sync_manager successfully replayed all records from completed transactions as reported by Txn_manager. We then ran a test application that tried to open all files referenced in the journal and read their contents. In total, jVPFS recovered 13 directories containing 1,761 files, which could all be opened and read. Metadata for 26 files was not recovered, because the last transaction they were part of was incomplete; the application received an ENOENT error for these files. This test succeeded reliably and no integrity errors were found.

We repeated the tests with journal writes being disabled, such that jVPFS behaved like the original VPFS. Txn_manager was allowed to write Checkpoint records only. After the simulated crash, the file system could be remounted, but the application received an integrity error when the trusted file-server component of jVPFS tried to look up names in the root directory. The file system was inaccessible afterwards and all data was lost.

Real Hardware. We then tested our system on real hardware. We power-cycled the machine right in the middle of the benchmark and let Linux recover the partition containing the legacy file system. Due to its strict write ordering, NILFS quickly recovered file containers and a valid jVPFS journal, which could successfully be replayed. In multiple tests, hundreds to thousands of files were recovered, depending on the exact moment of the power loss. For example, in one particular instance our system restored 2,710 files consisting of 9,826 blocks of user data within 5.1 seconds; the journal contained 1.2 MB of valid metadata updates. All recovered files could be read; for all other files, the aforementioned test application received an ENOENT error. No corruptions (i.e., intergrity errors) were reported, as we expected.

In the configuration utilizing the hard disk, ReiserFS replayed varying numbers of transactions in its own journal and jVPFS recovered files with no errors other than ENOENT for missing files. We did however also use Ext4 in our experiments and got unexpected results: after recovering the Ext4 partition, jVPFS found a checkpoint record in the journal, but no transactions. We determined an Ext4 optimization called "delayed allocation" to be the reason for this behavior: it may produce zero-sized files after recovery, if the application did not call fsync() on the file descriptor. Due to our own optimization in jVPFS, which we explained in Section 4, Txn_manager did not use fsync() in the untar benchmark, except right after the file system had been created and the Checkpoint record was written. We are currently investigating ways to make jVPFS reliable on Ext4, too.

6 Related Work

We shall now discuss other work that relates to the improvements we made to VPFS [1] in order to securely add robustness against unclean shutdowns.

File System Consistency. jVPFS implements journaling and replay of high-level metadata operations. A similar approach is used by journaling file systems such as Windows NTFS. Others, including Ext3/4 and ReiserFS, instead append complete metadata blocks to their journal in order to log inode, directory, and allocation updates [11]. They implement *full-block journaling*. We considered this approach for jVPFS, but rejected it as we found it to be more complex in our architecture. Journaling metadata blocks requires much more fine-grained dependency tracking within the TCB. Operations such as creating a file modify many metadata structures, which are distributed across multiple blocks in the buffer cache. Furthermore, the fact that more than one inode (or directory entry) is stored in a single metadata block causes additional false dependencies. For example, when writing back metadata for one file, the directory block containing the filename might contain an entry for another file that has recently been created, but whose data or inode has not been written yet. Thus, writing such a directory block actually creates an inconsistency in the on-disk state. To avoid having to increase transaction sizes by including a potentially large number of unrelated files, systems that use full-block journaling implement roll-back mechanisms that temporarily remove incomplete updates from metadata blocks before they are written. We tried to integrate such mechansims into jVPFS, but found them

to add more complexity to the TCB than operation-level journaling as described in Section 3.

Transactional file systems such as ZFS [28] share the problem of false metadata dependencies. They use a copy-on-write approach to prevent inconsistent on-disk state in the first place. Instead of updating data and metadata in-place, they write all modified blocks to free space and then adjust pointers to reference those updated blocks. In conjunction with a hash tree, updates must always propagate to the root. As explained in Section 3.1, the overhead incurred by this approach is significant.

The soft update [10] approach makes sure that a consistent file system can always be restored. The key idea is to apply 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 is inherently incompatible with Merkle tree updates. We therefore did not further consider the soft update approach for solving the robustness problems of VPFS.

The journaling scheme in jVPFS is related to the log-structured approach [22]. What sets our system apart from this type of file systems, is that its consistency mechanism is split into two isolated parts, with complex garbage collection not being part of the TCB.

Untrusted Storage. The logging approach the Trusted Database System (TDB) [19] uses to protect its transaction log is similar to that of jVPFS. It also uses a Merkle tree to ensure integrity. However, jVPFS splits the implementation of journaling and replay into two isolated components using a novel cooperation scheme. jVPFS also reuses existing consistency primitives of an untrusted file system, whereas TDB implements a complete, new database in the TCB.

The protected file system (PFS) [29] unifies journaling and hash logging in a way similar to jVPFS in order to securely use untrusted storage. However, it operates at the level of file-system blocks rather than metadata operations and has a monolithic codebase.

SiRiUS [30] is an example for a network file system that uses untrusted servers. It also stacks onto existing network file systems such as Sun NFS [31] and delegates management of persistent file storage to untrusted infrastructure. However, to the best of our knowledge, SiRiUS does not have an integrated recovery mechanism to ensure consistency of its *metadata freshness files*.

Non-standard Consistency Primitives. Systems such as Featherstitch [32] offer efficient means to applications to specify write-before constraints. The untrusted part of jVPFS can benefit from such expressive consistency primitives in the same way as it benefits from write-order guarantees in NILFS.

7 Conclusions

We built jVPFS, a secure stacked file system that implements post-crash recovery with a minimal trusted computing base (TCB): it requires only 325 lines of C++ code for the security-critical functionality of metadata journaling and recovery, which is an order of magnitude less than widely-used Linux file systems require to provide crash resistence. It reuses an untrusted Linux file system, from which it is strictly isolated through address spaces and virtual-machine boundaries. jVPFS delegates most of the work for managing a physical storage medium to the Linux file system stack, while making extensive use of existing consistency primitives. For example, it can exploit strict write-order guarantees offered by NILFS. Thus, the trusted core of jVPFS can operate at a high abstraction level of metadata operations, greatly reducing the complexity that file-system consistency mechanisms usually contribute to the TCB.

jVPFS outperforms its predecessor VPFS in all benchmarks we did and was shown to be much more robust against unclean shutdowns. It successfully and reliably recovered from temporary damage after power loss. Its strong integrity checks did not detect any corruptions in the recovered secure file system, which was layered on top of ReiserFS on a hard disk, or NILFS, a log-structured Linux file system optimized for flash storage.

8 Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Bryan Ford for their valuable feedback and suggestions for improvement of this paper. Thanks also go to the members of the Operating System Research group at Technische Universität Dresden for helpful discussions and feedback. This work has been supported by the German Research Foundation (DFG-Geschäftszeichen HA 2461/9-1).

References

[1] Carsten Weinhold and Hermann 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.

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

[3] Apple Inc. About the security content of iPhone OS 3.1.3 and iPhone OS 3.1.3 for iPod touch. `http://support.apple.com/kb/HT4013`, February 2010.

[4] Removing iPhone 3G[s] Passcode and Encryption. http://www.youtube.com/watch?v=5wS3AMbXRLs, July 2009.

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

[6] iPad Jailbreak - Jailbreak Your iPad. http://www.ipadjailbreak.com/p/jailbreak-your-ipad.html.

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

[8] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *Proceedings of the Eleventh ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[9] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing tcb complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, 2006.

[10] Marshall Kirk McKusick and Gregory 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.

[11] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[12] Valerie Aurora. Soft updates, hard problems. http://lwn.net/Articles/339337, July 2009.

[13] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.

[14] Andrea 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.

[15] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben 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.

[16] Trusted Computing Group. Trusted Platform Module. http://www.trustedcomputinggroup.org/developers/trusted_platform_module.

[17] ARM Limited. ARM Security Technology - Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

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

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

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

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

[22] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 1–15, New York, NY, USA, 1991. ACM.

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

[24] NILFS - Continous Snapshotting Filesystem for Linux. http://www.nilfs.org/en/.

[25] Adam Lackorzynski and Alexander 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.

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

[27] D. Wheeler. *SLOCCount*. available at: http://www.dwheeler.com/sloccount/.

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

[29] C. Stein, J. Howard, and M. Seltzer. Unifying File System Protection. In *Proceedings of the USENIX Technical Conference*, pages 79–90, 2001.

[30] 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.

[31] 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.

[32] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei 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.