Computers & Security 162 (2026) 104786

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full Length Article

The Reverse File System: Towards open cost-effective secure WORM storage devices for logging

Gorka Guardiola Múzquiz, Juan González-Gómez, Enrique Soriano-Salvador *

Universidad Rey Juan Carlos, Madrid, Spain

ARTICLE INFO

Keywords:
WORM
Logging
File system
Tamper-evident
Verification
Authentication
Forensics

ABSTRACT

Write Once Read Many (WORM) properties for storage devices are desirable to ensure data immutability for applications such as secure logging, regulatory compliance, archival storage, and other types of backup systems. WORM devices guarantee that data, once written, cannot be altered or deleted. However, implementing secure and compatible WORM storage remains a challenge. Traditional solutions often rely on specialized hardware, which is either costly, closed, or inaccessible to the general public. Distributed approaches, while promising, introduce additional risks such as denial-of-service vulnerabilities and operational complexity. We introduce Socarrat, a novel, cost-effective, and local WORM storage solution that leverages a simple external USB device-specifically, a single-board computer running Linux with USB On-The-Go (OTG) support. The resulting device can be connected via USB, appearing as an ordinary external disk formatted with an ext4 or exFAT file system, without requiring any specialized software or drivers. By isolating the WORM enforcement mechanism in a dedicated USB hardware module, Socarrat significantly reduces the attack surface and ensures that even privileged attackers cannot modify or erase stored data. In addition to the WORM capacity, the system is designed to be tamper-evident, becoming resilient against advanced attacks. This work describes a novel approach, the Reverse File System, based on inferring the file system operations occurring at higher layers in the host computer where Socarrat is mounted. The paper also describes the current Socarrat prototype, implemented in Go and available as free/libre software. Finally, it provides a complete evaluation of the logging performance on different single-board computers.

1. Introduction

Write Once Read Many (WORM) storage devices are a key component in systems that require immutable data retention. Once data is written to a WORM device, it cannot be modified or deleted, though it remains accessible for unlimited number of read operations. The WORM property is essential for a wide range of applications, including secure logging. Data regulations (see for example EU-GDPR, 2016; USSEC, 2003) increasingly require the assurance of long-term data retention, accountability, integrity, and verifiable data migration. WORM devices play a critical role in fulfilling these regulatory and operational requirements.

Despite the apparent simplicity of the WORM model, implementing compatible, secure, and reliable WORM storage is a complex issue. Traditional approaches depend on specialized hardware. These solutions are often expensive, inaccessible to the general public, or difficult to integrate with modern computing environments. Distributed systems offer alternative approaches but introduce their own challenges, including increased complexity and susceptibility to denial-of-service attacks. Moreover, they are not suitable for disconnected or intermittently connected systems. In this work, we focus on local solutions for logging purposes.

A critical threat arises when an attacker gains control over the host machine.[1] In such scenarios, conventional file system protections (e.g. the append-only and immutable attributes of the standard Linux file system) are insufficient: If the attacker elevates privileges, she can reconfigure the file system, modify or delete files, tamper with data blocks, or even reformat the storage device. These issues highlight the need for a WORM solution that is resilient to the complete compromise of the host and does not rely solely on the file system enforcement mechanisms (attributes, permissions, etc.).

To address these challenges, we propose Socarrat, a novel and cost-effective WORM storage solution implemented as a standalone

[1] From now on, we will refer to the machine to which the storage-providing device is connected as the host

* Corresponding author.
E-mail addresses: gorka.guardiola@urjc.es (G. Guardiola Múzquiz), juan.gonzalez.gomez@urjc.es (J. González-Gómez), enrique.soriano@urjc.es (E. Soriano-Salvador).

https://doi.org/10.1016/j.cose.2025.104786
Received 22 September 2025; Accepted 24 November 2025
Available online 5 December 2025

USB device. Socarrat is built on a single-board computer (e.g. a Raspberry Pi) running Linux with USB On-The-Go (OTG) support (USB-IF, 2012). This device is identified as a mass storage device that can be mounted on any conventional operating system (Windows, Linux or MacOSX) like any ordinary USB external disk formatted with a standard file system. At present, our system supports two different file systems: ext4 (Ext4 filesystem, 2025) (the standard file system for Linux systems) and exFAT (exfat filesystem, 2025). This way, for approximately 100 €,[2] Socarrat provides a practical and secure WORM solution accessible to a broad range of users.

At the core of Socarrat is a novel architectural approach we call the Reverse File System. It infers a superset of the high-level file system operations (those performed by the file system driver executing in the host). These operations are reconstructed by analyzing the low-level disk blocks written to the storage device (i.e. our single-board computer) through the USB link. This component focuses exclusively on append-only write operations of any size (from 1 byte to the maximum write limit) to designated log files (those with WORM capabilities), discarding all other modifications. It ensures that only valid, sequential log entries are retained, while unauthorized writes (i.e. modifications to the previously written data) are neutralized.[3]

Building such a system presents significant technical challenges. The research problem arises from the fact that a single partition cannot be mounted concurrently by two operating systems (the host and the Socarrat device), due to obvious synchronization and consistency issues. Our system exports a raw block volume to the host, and must accurately interpret the blocks it receives to determine whether they correspond to valid append operations on protected log files. This requires a deep understanding of the file system internals, including the structures (e.g. inodes), data blocks (e.g. extents), and journaling mechanisms. The system must also detect and handle partial writes and coalesce multiple block updates into coherent write operations.

In addition, Socarrat provides forward integrity to the data already written to the secured logs. It uses our previous system, SealFS (Soriano-Salvador and Guardiola-Múzquiz, 2021; Guardiola-Múzquiz and Soriano-Salvador, 2022), to enable tamper-evident, authenticated logs. If the attacker is able to circumvent the protections and attack our storage device over the USB link or by other means, any modification to committed data in the logs will still be detected by a verification tool.

1.1. Example scenario and operational procedure

To illustrate the use of our system, consider the following scenario: Alice connects a USB black box (i.e. the Socarrat device) to her server (i.e. the host). This device has been initialized previously by the Auditor (Alice or a third party). The operating system of the host detects a USB mass storage device formatted as an ext4 file system. Then, this drive is mounted in the system. At the mount point, there is a file named `log` (set up at initialization stage). The applications running in the server are able to use this volume as a regular one, by performing the traditional system calls to work with files. The only difference is that, when the applications access the `log` file, only read operations and append-only write operations are effective; the rest of the write operations over the `log` file (and its metadata) are discarded. Later, the Auditor can extract the `log` file from the black box, along with an additional file that authenticates all the entries added to the log for this period. Using a diagnostic tool, the Auditor can verify that the log data corresponds to the entries made during the operational period, ensuring that no records have been deleted or tampered with.

Therefore, there are three distinct stages:

1. Configuration: Configure which log files will be provided and generate the necessary data to initialize the tamper-evident mechanisms. This is done by the Auditor.
2. Logging: Alice connects the device and it works as an already formatted mass storage device.
3. Auditing: The Auditor extracts the logs from the device and verifies their integrity.

1.2. Contributions

The contributions of this paper are:

1. The introduction of a novel approach, the Reverse File System, to implement storage devices based on USB OTG. Such devices can be used as ordinary external disks, but permit us to control the operation performed over the data objects provided at higher levels in the host machine.
2. The design of an architecture, Socarrat, that enables cost-efficient tamper-evident WORM devices by integrating the Reverse File System approach and SealFS, as an out-of-the-box feature for any modern operating system.
3. The implementation details of our current research prototype, which supports ext4 and exFAT file systems.
4. The evaluation of this prototype running on different types of single-board computers, including the analysis of the results for the experiments conducted to measure its performance for logging purposes.

1.3. Organization

The rest of the manuscript is organized as follows: Section 2 discusses the related work; Section 3 briefly describes our threat model; Section 4 explains the general architecture of the system; Section 5 describes the indicators of compromise and possible responses; Section 6 provides the implementation details of the current prototype; Section 7 describes the experiments we conducted and the evaluation results; and Section 8 presents the conclusions.

2. Related work

2.1. Hardware

Old WORM systems used continuous feed printers (Bellare and Yee, 1997), tapes, and optical devices like the DEC RV20 laser drives with 2 GB RV02K-01 disks (1988), Sony WDD disks (1992) and Sony WDA Writable Disk Auto Changer jukebox (1992). Tape and optical media still exist, but it is now largely confined to niche or legacy applications. For example, Sony launched their third-generation optical disks (5 TB per unit) in 2020. These products were discontinued in 2024. IBM and Hewlett Packard also produce tapes for WORM storage (e.g. IBM LTO WORM Data Cartridge and HPE StoreEver LTO-8 Ultrium 30750). The price of the LTO-8 Ultrium 30750 tape drive is ≈ 4000 €. These devices have been falling into disuse for decades and are not as readily available as standard drives. We present a solution based on conventional storage devices and low-cost single-board computers: We are able to build a WORM device for ≈ 100 €.

Some lesser-known manufacturers offer WORM disks. For example, Greentec[4] offers a product called WORMdisk ZT Storage, compatible with file systems such as NTFS, exFAT, and ext4. Another example are the Flexxon's WORM SD Cards.[5] The underlying technology of these

[2] The cost of a Raspberry Pi 4, an SD Card, and an external 1 TB USB disk. No costs are incurred from software licensing, as our system is based on free/libre technology.

[3] Note that the idea of a Reverse File System is general and not limited to our application (i.e. building WORM devices). The approach is powerful and can be applied to other kind of applications (debugging, fuzzing, etc.)

[4] https://greentec-usa.com/products

[5] https://www.flexxon.com/write-once-read-many-worm-memory-card/

devices is closed and obtaining further details about the architecture and pricing appears to be challenging. Our motivation is to provide an open, low-cost, accessible, and flexible solution with a clear delineated set of guarantees.

There is significant ongoing research into new materials (e.g. organic components) for manufacturing new types of WORM memories, but no general-purpose products based on these technologies are commercially available yet (see for example Leppäniemi et al., 2012; Hsu et al., 2024). That corpus of work lies outside the scope of this study.

2.2. Drivers and file systems

Ordinary file systems like Linux's ext4, include mechanisms to force files to be append-only and immutable. Nevertheless, if the machine is compromised and the attacker has administrator privileges, those attributes can be disabled. We aim to provide a hardware device to ensure those properties (and provide extra guarantees, like forward integrity).

Finlayson et al. presented Clio (Finlayson and Cheriton, 1987) in 1987. Clio is an extension for the V-System file system, specifically designed for logging. It used a combination of optical drives and magnetic disks to provide append-only files.

In 1991, Quinlan developed a WORM system for the Plan 9 operating system (Quinlan, 1991). It was also based on an optical disk jukebox and a magnetic disk. The magnetic disk, which is used as a cache, provides the files for common use (read and write). The optical devices are used to provide read-only snapshots of the file system. Later, other file systems focused on snapshots emerged. For example, the WOWSnap file system (Suk and J, 2010) uses two partitions, one for ordinary files and other for WORM files to provide snapshots. Ext3cow (Peterson and Burns, 2005) was another open-source file versioning and snapshot system, derived from ext3. Those two systems differ in the approach. WOWSnap implements a ROW (Redirect On Write) policy, while Ext3cow implements a COW (Copy On Write) policy. These systems do not provide append-only files.

Blutopia (Oliveira et al., 2007), a system implemented by an author of this manuscript in 2007, provided immutable file systems through stackable block device. It requires periodic snapshots and it is not suitable for append-only files.

Wang and Zheng (2003) proposed a preliminary approach to build append-only storage system with conventional disks. In this work, they categorize the WORM systems in three groups: physical WORM (e.g. optical drives), coded WORM (e.g. operating system drivers) and software WORM (e.g. applications). They proposed to modify (i) the operating system block device drivers for the disks to force append-only write operations and (ii) the disk's firmware. While (i) represents a realistic approach, (ii) poses a significant challenge to achieve, given that the disks' firmware is entirely closed source. In this work, the authors explain their approach and provide some pseudocode, but they do not describe a functional prototype. The approach for (i) is to manage two partitions, one for data and the other one for metadata that maps the regions already written (to detect rewrites). Although the authors state that the approach is independent of the type of file system, it is not clear how the file system metadata and internal structures (i.e. inodes, superblock, etc.) could be handled. We provide a complete and functional prototype: We have built our own "disk" with a single-board computer, to control its "firmware" (i.e. Socarrat), which is based on our Reverse File System idea. This way, (i) is not necessary. Debiez et al. (2003) patented a similar method, based on modifications of the device driver (i). Note that it is not secure in case of total compromise of the host system, because the attacker could replace the device drivers.

As far as we know, the concept of a Reverse File System is novel, and there are few comparable systems. The two most similar examples we have identified are closely related, with one having been directly inspired by the other. Vvfat (Vvfat, 2025) is a block driver served by Qemu. It serves a virtual VFAT file system in which the files represent the files of an underlying directory and follow them when read an written. Inspired on this, there is the `nbdkit` floppy plugin (Nbdkit, 2025). First of all, those systems are not used to provide WORM devices or secure logs. Second, they implement a kind of synthetic reverse file system. They are unconstrained by a liveness property: They only need to guarantee that the underlying files are completely synchronized when they are unmounted. In contrast, we must continuously extract valid operations to function properly. Last, our Reverse File System watches the data structures present in a block device image, while those two synthetic reverse file systems make up the data structures on demand.

2.3. Distributed approaches

There are numerous distributed approaches to provide logging systems, leveraging cloud computing (see for example Loggly, 2019; Stackdriver, 2019). In some cases, these systems offload the problem to another machine over the network. This just shuffles the problem elsewhere: Servers that store the logs can be attacked as well. In addition, the networked system architecture significantly increases the attack surface compared to our approach, which is confined to the USB mass storage interface. Furthermore, distributed systems may add extra latency, especially in the presence of lengthy consensus or proof of work algorithms. We aim to provide a local solution that can be used by systems operating offline by basing it on a small USB device.

There are some cloud services that offer distributed WORM archival capabilities. For example, AWS S3 Object Lock (Amazon, 2025) provides WORM properties with immutable files, but it does not support appendable files. IBM Cloud Object Storage also provides non-erasable and non-rewritable data objects. Files stored in such an immutable can be set to immutable or append-only (Ibm, 2025).

Samba, the open source implementation of the SMB/CIFS network file system, provides WORM functionality on the client side with a module named Samba vfs_worm (Samba, 2017). This module is an additional layer that forces WORM semantics. In case the host is compromised, the module can be disabled. Moreover, on the server side, the files are mutable.

NetApp SnapLock (Walter and Tulledge, 2024) is a commercial solution that offers WORM storage for network file systems, such as NFS and SMB. It supports WORM append-only files. It also stores cryptographic hashes to ensure the integrity of the immutable data in the storage servers. In this system, the data is appended in chunks of 256 KB. We provide append-only files without this limitation (the append-only write size is not fixed). In any case, as we stated before, we aim to provide a local solution without network dependencies.

Huawei also offers network storage devices (the OceanStor series) that include a system named HyperLock. This system can store regular files, append-only files, and immutable files (what they call WORM files) (Huawei, 2025).

GlusterFS (Davies and Orsaria, 2013; Selvaganesan and Liazudeen, 2016) is a storage system that exports a network file system from multiple existing volumes (e.g. ext4), hosted on different servers. It enables the creation of different types of compound volumes (distributed, replicated, dispersed, etc.). In this system it is possible to create a WORM volume that provides append-only files (Gluster, 2025). The Gluster native client is only available for Linux. Other systems can access the file system through NFS and SMB.

Quinlan et al. created Fossil (Quinlan et al., 2003) and Venti Quinlan and Dorward (2002). Fossil is a file system that can create archival snapshots, which are backed up to a Venti file server (Quinlan and Dorward, 2002). Venti is a content addressed block storage system that uses a Merkle tree of SHA1 hashes to index read-only blocks. Only snapshots are read-only. Although Fossil provides append-only files (through an attribute), if the corresponding blocks are not stored in Venti yet (i.e. they are not part of a snapshot), the data can be modified. While Fossil and Venti were created as part of a distributed system, Plan 9 (Pike et al., 1990), they can certainly be configured locally in the same machine. The problem is that, in that case, when there is a total compromise of the

system, the attacker can rewrite the Venti blocks, recreating the Merkle tree of hashes and the WORM property is compromised. In addition, Venti handles complete blocks, like other systems discussed before.

Other distributed file systems, like the Hadoop distributed File System (HDFS) (Shvachko et al., 2010) and Google FS (GFS) (Ghemawat et al., 2003), are also based on immutability. These file system have no concept of a change to a complete file. When a file is created, is append-only (Helland, 2015). Those systems are specifically designed to store large data objects for big data applications, its use just for storing secure logs may be overkill.

2.4. Forward integrity

There are multiple distributed systems that provide forward integrity based on blockchain or distributed ledger technologies (see for example White et al., 2019; Rosa et al., 2019; Wang et al., 2018; Logsentinel, 2019). Again, those solutions are not suitable for disconnected or loosely connected scenarios. We achieve forward integrity locally, using a previous system called SealFS (Soriano-Salvador and Guardiola-Múzquiz, 2021; Guardiola-Múzquiz and Soriano-Salvador, 2022), that combines ratcheting (Bellare and Yee, 1997) with storage based tamper-evident logging. For a more in-depth study on tamper-evident logs, please refer to Soriano-Salvador and Guardiola-Múzquiz (2021), Guardiola-Múzquiz and Soriano-Salvador (2022).

3. Threat model

The asset to be secured consists of the logs generated by the processes running on the host system. To formalize the guarantees we aim to provide, we introduce the Continuous Printer Model (CPM). This model, inspired by the old approach to provide WORM logs (i.e. using a continuous printer to generate a hard copy of the logs), captures the essential properties of secure WORM storage in adversarial environments:

- Forward Immutability: Once data is committed to disk, it cannot be deleted or overwritten. There is no guarantee that:
  - Spurious or bad data will not be committed in the future
  - The system will not be stopped from printing (i.e. crashed).

  In any case, Immutability of already committed data is always preserved: What is printed cannot be unprinted.
- Liveness: Data is committed frequently enough to ensure timely persistence.

Additionally, with the SealFS component, Socarrat also provides:

- Forward Integrity: Even if the host or the Socarrat device are compromised, previously committed data remains verifiable.

Note that what we have called Forward Immutability is a storage property (data cannot be deleted or rewritten), whereas, Forward Integrity is a cryptographic property (data cannot be deleted or modified without detection).

3.1. Threats

We consider two distinct types of attacks:

- Logical attacks targeting the host. Upon exploitation, the attacker may control the host machine and execute code at any privilege level. Thus, she can write any information to the logs, change the metadata, modify the file system configuration, unmount the volumes, try to format it, write directly to the block devices, etc.
- Physical or logical attacks to the Socarrat device. Upon the attack, the adversary has access to the internal Socarrat system, its private local storage devices, etc.

3.2. Dependencies

The system has a single requirement: The Socarrat device must have a storage devices with enough capacity to store the logs, the authentication data, and the cryptographic keys used by SealFS, over extended operational periods.

The hardware platform for the Socarrat devices is standard (e.g. an ARM-based single-board computer equipped with conventional SSD, SD or NVME storage components).

3.3. Assumptions

- The host may operate either online or offline. Both hardware and software components are assumed to function correctly and remain trustworthy until compromised by an adversary through system exploitation, process corruption, or the execution of malicious code.
- The Socarrat device is connected to the host through a USB cable. The integrity of the USB link is preserved and it only provides the mass storage device within specifications. Both hardware and software components are assumed to function correctly and remain trustworthy until compromised by an adversary through system exploitation, process corruption, or the execution of malicious code.
- The attacks are carried out during Stage 2, that is, the normal operation of the system (the logging stage detailed in Section 1.1).
- The Socarrat device employs a secure deletion procedure, ensuring that once data is removed, it becomes irretrievable from all storage layers, including persistent memory, cache, and disk.
- The Auditor performs log verification on an independent, reliable, and trusted system connected to the Socarrat device.

3.4. Mitigation

Against remote or local logical attacks targeting the host, the first line of defense of the Socarrat device is the USB link. Therefore, the attack surface is limited solely to the USB interface. The block device interface, which we extend to the other machine via the USB mass storage device, acts as a choke point: The set of operations on it is limited.

Internally, the Reverse File System acts as a kind of data diode, by only honoring certain operations (i.e. writes at the end of the file) and ignoring the rest. The information finally stored in authenticated logs flows in only one direction (toward the logs themselves). If the USB link does not fail, we provide the CPM guarantees.

Upon attack detection, Socarrat can apply different policies (they will be discussed in Section 5).

Against logical attacks targeting the Socarrat device system (e.g. the attacker is able to exploit vulnerabilities of the USB link in order to execute malicious code in the Socarrat device), or in case of physical attacks to tamper with its storage components, the system just provides forward-integrity. If the data already stored in the log files is manipulated, the Auditor will be able to notice it and disregard the logs (i.e. the committed logging data cannot be deleted or counterfeit without being detected).

4. Architecture

An overview of the architecture of the system can be seen in Fig. 1. The host is an ordinary computer or another kind of system (e.g. a robot) running Linux (or any other operating system that supports ext4 or exFAT file systems), which is executing standard user-space applications. From now on, we will suppose that the selected file system is ext4 for the sake of simplicity.

The host has mounted the file system provided by the USB mass storage device, represented by an ordinary block device (`/dev/sda` in this example). In the figure, we suppose that the file system is mounted in `/var/log/blackbox`. This file system provides some preconfigured

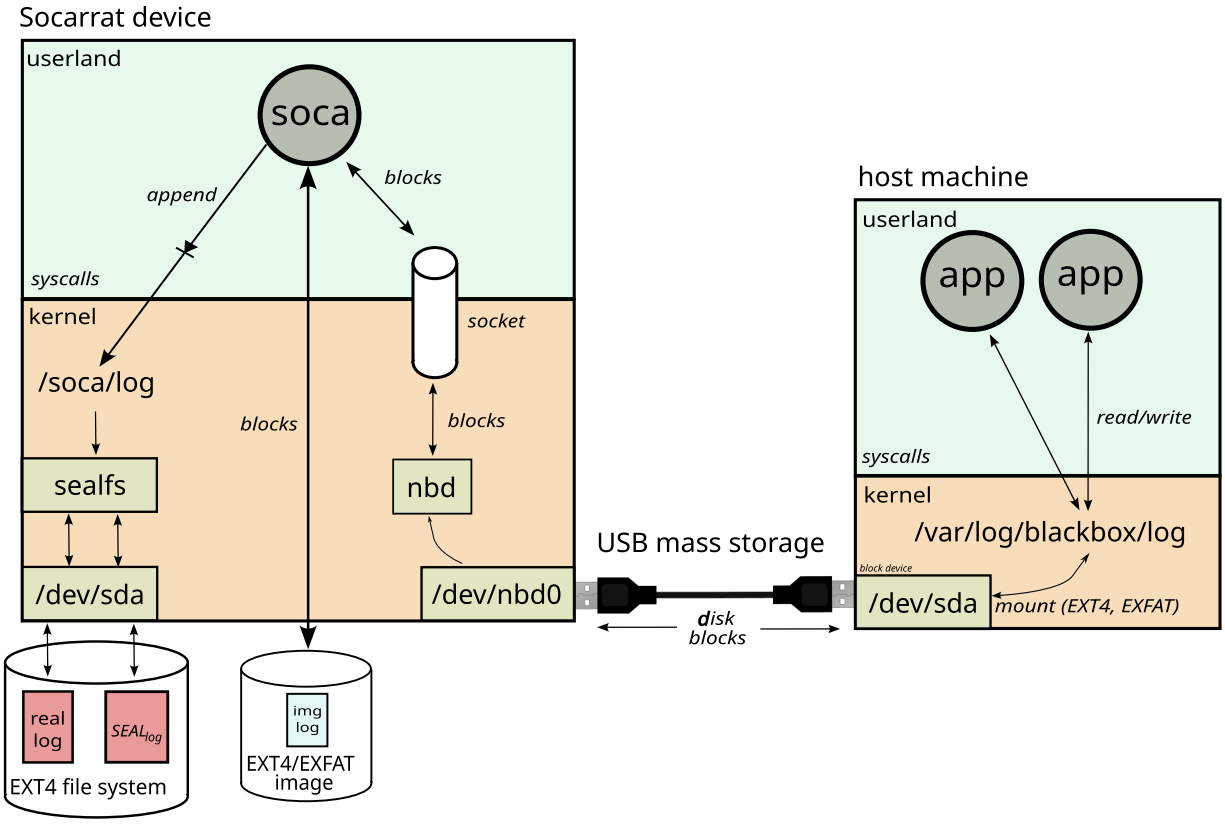

Fig. 1. General architecture of the Socarrat system.

log files (`log` in this example). The ext4 driver of the host will read and write blocks out of `/dev/sda`.

From the point of view of the user, the Socarrat device is an ordinary USB disk formatted as ext4. Those applications use the files normally. In the case of the logs, they use them as ordinary append-only files. These applications do not need to use any special library or framework to access the files.

In the Socarrat device, there is a user-space program named `soca` that implements the Reverse File System. `Soca` serves the blocks of the Socarrat device. It processes the read and write operations over blocks, and infers the operations that are being performed by the ext4 driver of the host. Note that it does not infer the exact original file system operations performed by the applications in the host (e.g. a write system call on a file), but a subset of the operations limited by the information available (i.e. the blocks it receives), possibly aggregated. It interacts with two storage elements provided by the local operating system:

- An ext4 image file or a disk partition that contains the blocks that it serves. From now on, we will assume that we are using an image file to simplify the explanations.[6] All the ext4 structures are stored in the image (the superblock, the inode vector, the data blocks, etc.). In the configuration stage (Stage 1), this image is formatted and the empty logs are created. Thus, the log files are stored within this image, but these are not the real logs. We will refer to the log files that are stored in the image as the img logs. The user can create, modify, and delete any other file in this image. Only the pre-configured log files are WORM append-only files.
- The files provided by SealFS for the real logs. These files are finally stored in a different, private block device of the Socarrat device, which is independent of the image file. SealFS authenticates the data appended to the real logs, enabling the tamper-evident properties by following a HMAC ratchet/storage-based hybrid scheme (Soriano-Salvador and Guardiola-Múzquiz, 2021; Guardiola-Múzquiz and Soriano-Salvador, 2022).

Finally, there are two kind of files in this private volume: (i) The real log files and (ii) the authenticated log file (also known as $SEAL_{\log}$, see Soriano-Salvador and Guardiola-Múzquiz, 2021) that contains the metadata of all the append operations performed over the `log` files. In the audit stage (Stage 3), the real logs can be verified with $SEAL_{\log}$.

`Soca` does not export any operation to read or modify (delete or overwrite) the committed data of the real logs, or access the $SEAL_{\log}$ in any way.

When `soca` detects an append operation to a log, it updates both the real log and the img log. When it detects a read operation from a log, `soca` responds with the data stored in the img log. This way, soca acts as a data diode for the real logs.

To redirect the blocks to `soca`, we use NBD (Nbd, 2024). While there are some alternatives available for Linux,[7] like ublk (Ublk, 2025), we decided to use NBD for various reasons. First, NDB has been available for a very long time and has been tested thoroughly. Second, the protocol between the NBD client (which talks to the Linux kernel, the Socarrat device's kernel in our case) and the NBD server (`soca` in our case) makes debugging simpler. NBD is composed of two different interfaces:

- The network interface, with a protocol to export block devices. It is a client-server protocol.
- The operating system interface. It permits the client program (a user-space program named `ndb-client`) to import the network NBD device as a regular block device.

[6] Note that the experiments described in Section 7 use both images and disks.

[7] We also use the 9P protocol for an alternative experimental implementation for the Plan 9 operating system (Múzquiz and Soriano-Salvador, 2025)

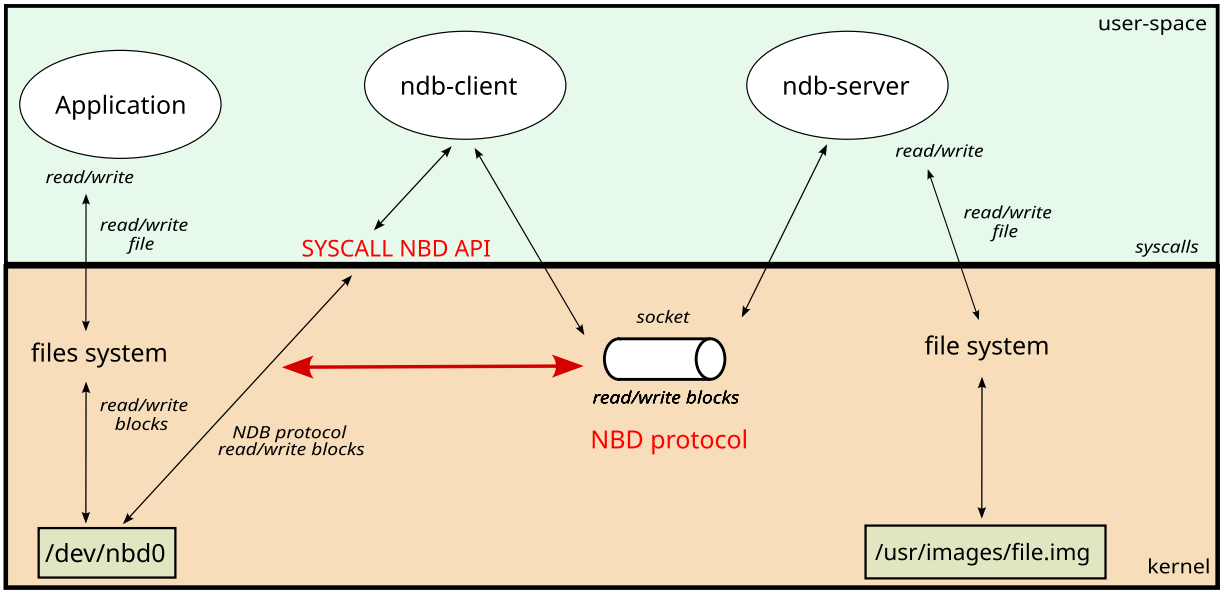

Fig. 2. NBD running in a single machine. The client and the server use a Unix domain socket to execute the NDB protocol.

The client and the server can be located on the same machine, using a Unix domain socket (UDS) to communicate, as illustrated in Fig. 2. This is the case of our architecture.

The NBD protocol has two distinct phases: configuration and data transmission. After the configuration phase, the client passes the socket to the kernel and waits for the session to finish (as depicted by the red arrow in Fig. 2). The transmission phase is performed by the kernel, that handles the operations performed on the device that represents the device block (`/dev/ndb0` in the figure). In our system, this device is exported by using USB On The Go (OTG). To do this, we configure a gadget[8] with `configfs` (Configfs, 2025).

Our current prototype has two modes of functioning:

- Remote mode: `Soca` acts as the NDB server, simply serving the Socarrat device. In this case, any external NDB client program (there are many versions available) can be used to connect `soca` to the Linux kernel. In this mode, `soca` is totally agnostic of the operating system and protocol it uses to connect to the NDB client.
- Local mode: `Soca` acts as both the NDB client and server. Fig. 3 depicts this mode. It uses the system call NDB API to interact with the kernel, skipping the NBD protocol negotiation phase for performance. Then, it starts the transmission phase with the kernel directly. This is the standard mode.

5. Compromise detection and response

There are different indicators of compromise of the host machine that can be detected by `soca`:

- Receiving a write operation for a log file that is not a data append. In this case, the host machine is violating the append-only attribute of the file system
- Receiving modifications of the log file metadata that are not compatible with append-only files (e.g. decrementing the size of the file).
- Detecting modifications of some file system structures (e.g. some parts of the superblock).
- Volume exhaustion. Note that we assume that the storage devices of the system are big enough to operate during Stage 2. If the file system within the image runs out of space, we can consider that the system is compromised.

When `soca` detects any indicator of compromise, it can follow two different approaches:

- Read-Only Remount: Forcing the file system to remount in a read-only mode prevents the attacker from writing to any file on the Socarrat device, although they may be aware of the detection.
- Honeypot Mode: Activating a honeypot mode seals the real logs, making them immutable. The attacker is able to modify any file on the image (including the img logs) without noticing, causing the real logs and img logs to diverge. Consequently, the attacker forges what they believe are the legitimate logs, remaining unaware of the detection.

  Later, during the audit phase, the differences between the real logs and the img logs can be analyzed to identify the data the attacker attempted to modify or delete from the logs.

6. Reverse File System: Implementation

To program a Reverse File System, we need access to the following resources:

- The block device where the file system being reversed is located (i.e. the image).
- The stream of block read/write operations.

Normally, this would be the job of a block device driver (e.g. a kernel-space component). For fast prototyping and ease of debugging, we wanted to be able to run this driver in user-space. Our current `soca` prototype is a user-space program written in Go. It has around 8394 lines of code at the time of writing this paper.

Socarrat caches heavily. It is a concurrent program written carefully to not block any operation unless it is strictly necessary. It is programmed following the CSP model (Hoare, 1978) (i.e. communicating threads via channels), sharing state by communicating. It also uses mutexes for some shared state.

As stated before, the prototype understands two popular file systems, ext4 (Ext4 filesystem, 2025) and exFAT (exfat filesystem, 2025). ext4 is the native Linux standard file system. ExFAT is widely used standard file system for removable devices and it is supported on all major operating systems (Windows, Linux, and macOS). Supporting both provides a good

[8] A gadget is essentially the profile of the device (the identifiers, kind of endpoints, etc.).

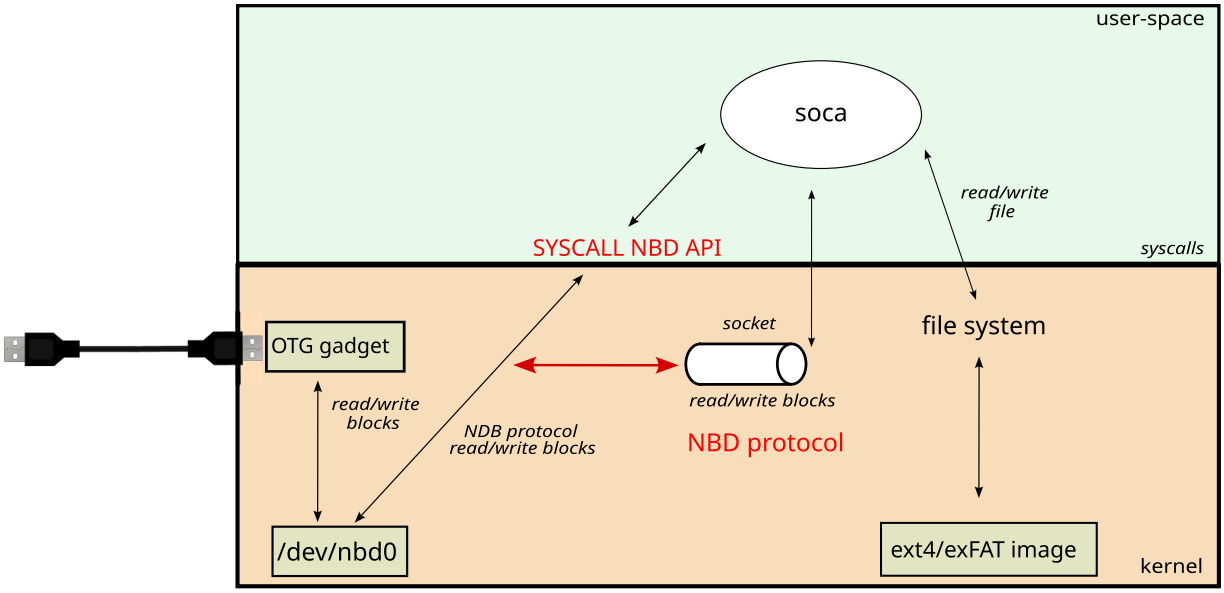

Fig. 3. `Soca` in local mode.

balance between portability (exFAT) and being able to experiment with advanced features (ext4). Note that, in order to develop the Reverse File System, we had to implement libraries to access, serve, and modify the file systems (i.e. we have implemented simplified user-space versions of the ext4 and exFAT drivers).

6.1. Ext4

Ext4 is based on inodes (like the traditional Unix file system). Simplifying, the partition in which the file system resides is composed of two parts: (i) the metadata describing the file system (superblock and group descriptors) and (ii) the data of the file system itself (the inode vector, which is composed of inodes and an allocation bitmap, and the data blocks and its allocation bitmap). Inodes represent file system objects, directories and files, and contain all the metadata except for the name. Data blocks contain data for files and directory entries for directories. A directory entry contains the name and the inode number.[9]

An inode contains pointers to a subtree whose leaves are the data blocks. There are different ways to configure how this subtree behaves. `Soca` configures the ext4 file system to use extents, which are simply contiguous regions of blocks. In this configuration, the tree is a tree of extents. This approach increases the performance by enabling the reading of contiguous blocks from the disk.

`Soca` watches the read/write operations on the blocks of the img log's inode. When it grows, the new data added to the last blocks referenced by the inode is appended to the real log (which is not served to the user's machine).

The prototype supports ext4 with and without journaling. A file system with journaling has a special data structure on disk (the journal) where it records updates to the file system transactionally, before committing them to disk. In case of an unexpected reboot (for example from a loss of power) the file system can transition to a consistent state by completing or undoing a partial update. The file system can provide different guarantees depending on what is recorded in the journal. If the journal only records metadata, the file system will be guaranteed to be coherent (the tree will not contain broken links, loops, orphan subtrees, etc), but some of the file's data may be lost or made up in the case of a loss of power. If the journal records data and metadata, the data contained in a file may be lost but not only the file system will be coherent, but spurious data will not appear in the file system.

6.1.1. Coherency

The main problem is that the blocks written to the file system go through the cache and may be appear out of order. The implementation of any Reverse File System needs to answer the question: How does one guarantee that the current file system state (on the block device) provides a consistent view of the file (img log) to be able to update the real log? The answer depends on the guarantees the file system itself provides. We have followed two approaches:

- The first approach is to not assume anything about the file system except that, after a small wait subsequent to the first update of the metadata (i.e. the size), the data being written will be in its final position in the disk. To make sure the update is completed, we also wait for the file system to be quiescent for a small interval. The rationale is that a well designed file system server should, at some point, write the data and metadata to its final position and then not move it. The time interval during which a well designed and implemented file system stays incoherent should not be too wide. Otherwise, the file system will end up inconsistent very often.

  Some special programs (like defragmenters or file system checkers) can rearrange data later. Usually, these programs run at special times (i.e. not during Stage 2). Just in case, as an extra precaution, because we control the initial image, we initialize it to a known value (zero), to detect and delay any incoherency due to a partial update.

  We have tested this approach and it works well without exceptions. Nevertheless, there are no strong guarantees. Even with a reasonably wide coherency window, something may happen that breaks our assumptions. In that case, the log would be updated with zeroes and data would be lost. We have not seen this case in our experiments, but it is possible.
- To provide better guarantees, we can use the journal. For the purpose of writing a Reverse File System, an important property of a journaling file system is that it clearly defines at what point in time the data and/or the metadata is known to be coherent on disk.

  The journal provides a transactional view of the updates of the file system. What parts of the file system are journaled (i.e. metadata or data) and when are they guaranteed to be written to the disk, depends on the journaling mode, as we will see later. The idea is to

[9] It also contains the inode type, but this is an optimization and the information is also on the inode itself.

pick an instant when the journal guarantees that the updates to the img log are coherent and then update the real log.

6.1.2. Ext4 without journaling

When ext4 is configured without journaling, `soca` waits for the following events:

1. A change of the file size resulting from a write to the block pertaining the inode for an img log.
2. A quiescent state of the file system for a short time window (λ).
3. The end of a small fixed time window (ω).

Both λ and ω are configurable. We will refer to the sum of both as $\tau = \lambda + \omega$.

After τ, we update the real log, traversing only the subtree needed to access the correct offsets, which should have been updated on disk. τ must:

- Be large enough to provide coherency.
- Be as small as possible, because it affects the time when it is possible to modify the Socarrat device's memory[10] to delete uncommitted entries for the logs that may content indicators of compromise (i.e. traces of exploitation or privilege elevation, etc.).

  In other words, it is the maximum time an attacker has to exploit the host, compromise the USB link, and execute privileged malicious code in the Socarrat device to remove the evidence. After this time, once the real log is written, it is tamper-evident.

Note that the value of τ does not affect performance. Applications running in the host do not have to wait at all, because the data is already on the Socarrat device and the block operation request (i.e. the USB operation) has already been processed.

6.1.3. Ext4 with journaling

Ext4 has support for journaling, in a format called jbd2 (Jbd2, 2025), which permits various levels of transactional guarantees. There are three possible modes of journaling for ext4 other than not having journaling:

- journal_data: Data and metadata are committed to the journal prior to being written to the file system.
- journal_data_ordered: Only metadata is committed to the journal. Data is flushed to disk before the data is committed to the journal.
- journal_data_writeback: Only metadata is committed to the journal. Metadata is journaled but there is no guarantee with respect to data.

Additionally, ext4 has support for journaling some special operations which work at a higher level of abstraction and are kept on a different part of the journal (i.e. fast commits). We disabled this feature to simplify the journal access. `Soca` supports the following modes:

- journal_data_ordered. This mode would be preferred, because it theoretically provides all the guarantees we need. In this case, at the point of time when the metadata of a img log is completely updated, data should already written to the disk. Therefore, `soca` can update the corresponding real log. Unfortunately, empirically, if the data is being updated continuously, it may persist on cache and this guarantee is violated. From the documentation, it is a challenge to ascertain if this is a implementation error or this guarantee is weak.

  For this reason, in this mode, `soca` takes advantage of the journal, but at the point where it would update the real log, it follows an approach similar to the one described for the non journaling case (i.e. it waits for the file system to be quiescent, etc.). This works well with the ext4 driver implementation and provides weak guarantees (those of ext4).
- journal_data. In this case, `soca` provides strong guarantees. It just watches the journal and updates the real log whenever changes are completely committed to the journal for the corresponding img log. Note that, in this case, `soca` does not wait τ.

The prototype processes operations in strict order, so that dependencies in the journal data blocks do not cause an order inversion.

6.2. ExFAT

ExFAT is a hybrid between a FAT (i.e. a table-based linked list allocated file system) and a contiguous allocation file system. The partition is roughly comprised of the boot sector with the metadata of the file system, the FAT regions containing the FAT table (the list of file system blocks, which they call clusters), and the blocks region (called cluster heap). Directory entries contain all the metadata of the file/directory and are in the data clusters of the directories. Directory entries for files point to their data clusters. There are essentially (again, simplifying) two type of directory entries: contiguously allocated and FAT allocated. Whenever posible, as an optimization, files are contiguously allocated.

ExFAT does not provide journaling. It supports an extension called TexFAT which adds transaction-safe operational semantics, but it is only implemented on some operating systems (like Windows CE). `Soca` does not support it.

Thus, `soca` simply acts like in the case of ext4 when there is no journal. Whenever the directory entry for a img log is modified and the file size changes, it waits τ and updates the real log. To read the data written to the img log, it traverses the prefix of the subtree (which may mean accessing the disk contiguously or using the FAT table depending on the contents of the directory entry).

7. Evaluation

7.1. Testing

We have taken a four level testing strategy:

- Go unit tests for each file system type. Some tests use our user-space implementation of the drivers used by the Reverse File System. Other tests exercise the file systems by using a block device which injects a zero block every other time. This way, we check that the implementation correctly detects uninitialized blocks.
- Go integration tests. Those tests run the whole system as an integrated NBD client an server (local mode), attacking it concurrently. The objective of these tests is to stress the implementation.
- Shell integration tests. They use the standard Linux NBD client and the kernel to serve the block device, mounting it locally, with the kernel acting as a client. This way, we check that our implementation of the Reverse File Systems is compatible with the kernel file system drivers.
- Manual integration tests through the USB link. Those tests check the system when in normal operation.

We tested different values for τ. Empirically, in our hardware, several hundreds of milliseconds are enough to pass the tests. For regular operation, we set a value of $1\ s$ to provide a security margin.

7.2. Measurements

In all the experiments, `Soca` has been configured in local mode. It sets $\lambda = 10\,ms$ and $\omega = 1\ s$.

To evaluate our prototype, we conducted experiments using a slightly modified version of the standard Filebench benchmark (Tarasov et al., 2016). Filebench was modified to work with append-only files, following the same approach used to evaluate SealFS (Soriano-Salvador and Guardiola-Múzquiz, 2021; Guardiola-Múzquiz

[10] Note that this does not happen in the host, but on the other side of the USB link.

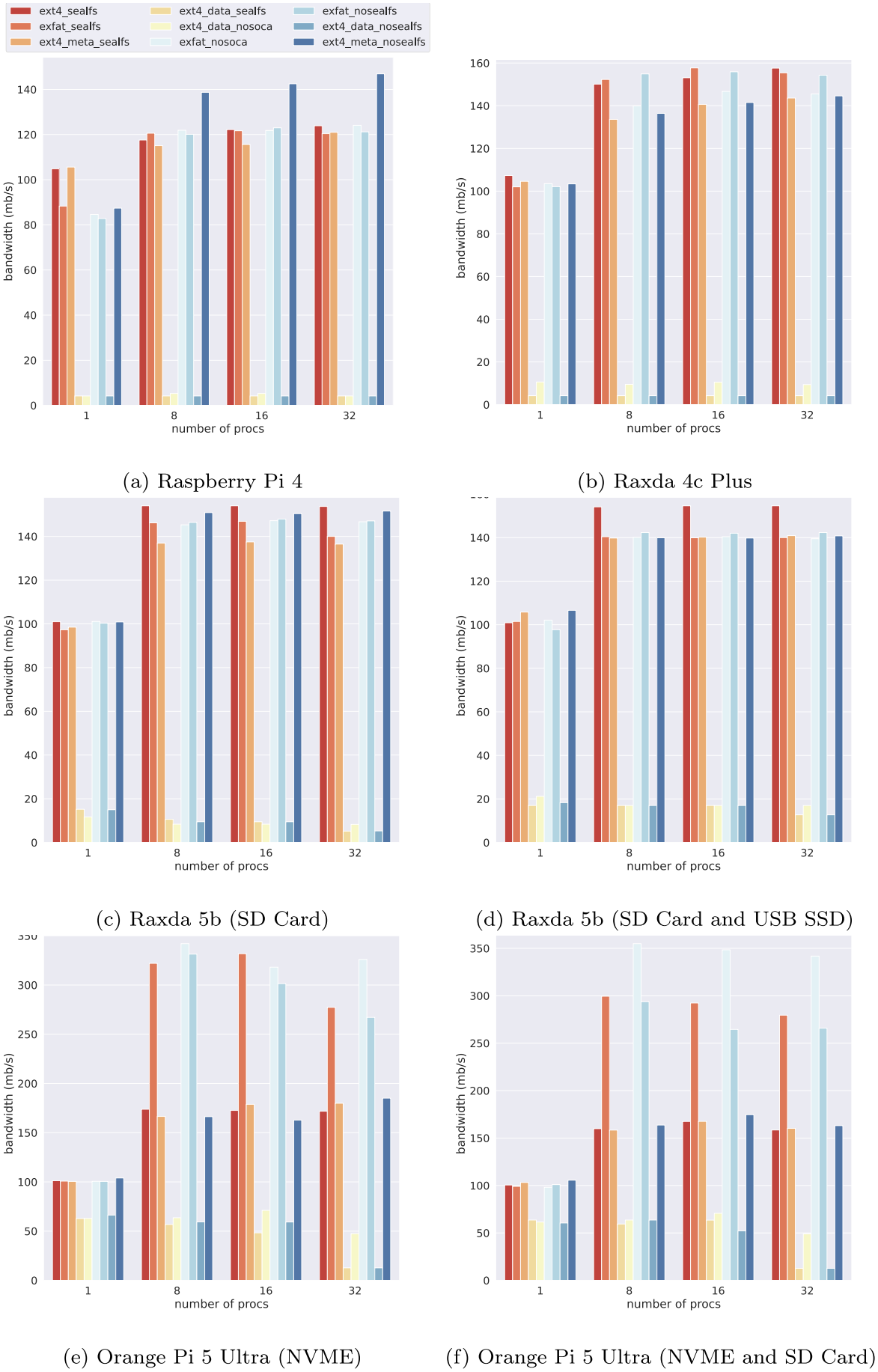

Fig. 4. Bandwidth measured by Filebench.

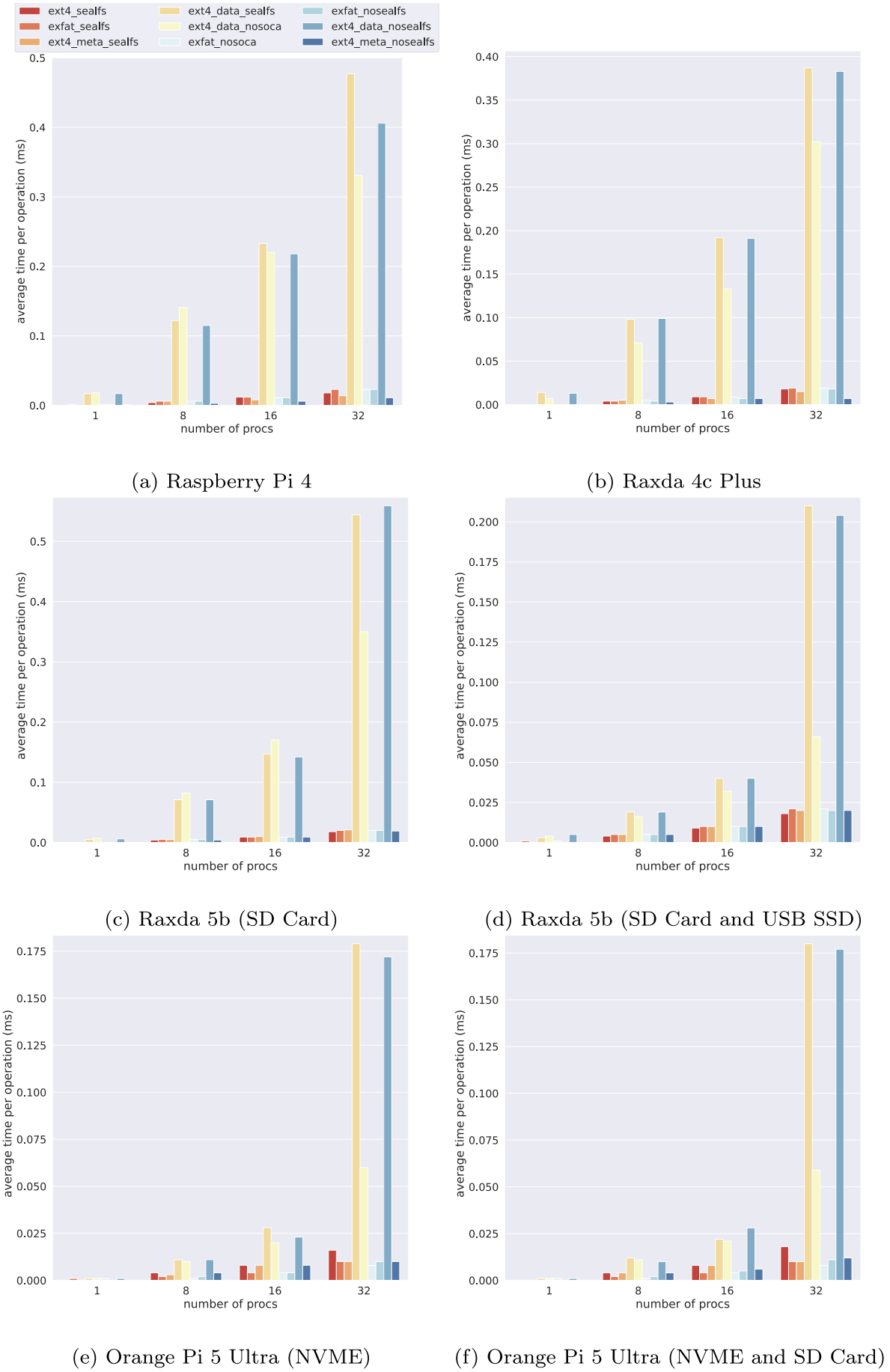

Fig. 5. Average latency measured by Filebench.

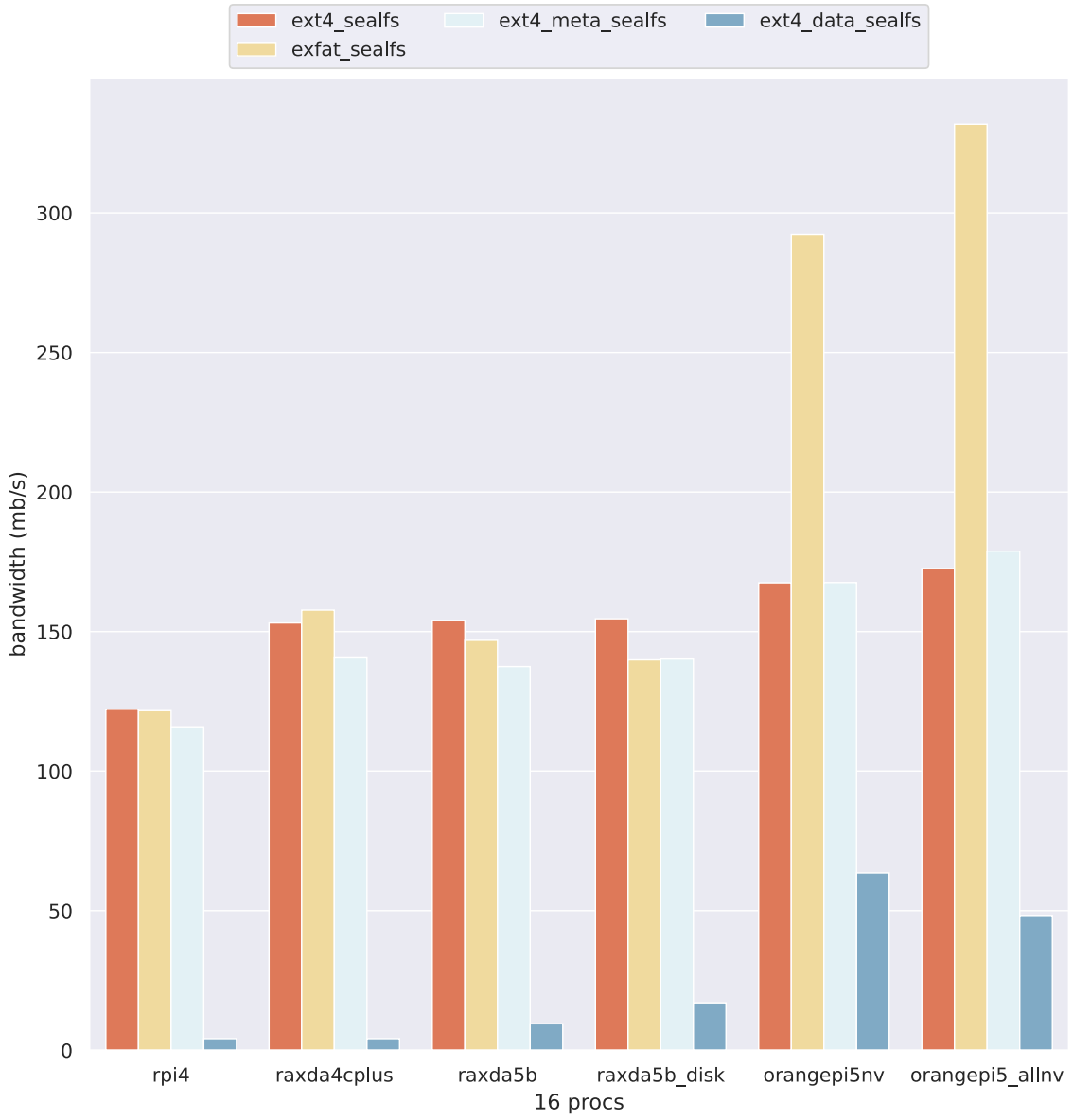

Fig. 6. Comparison of the results of Filebench (bandwidth) for 16 concurrent processes in different platform configurations.

and Soriano-Salvador, 2022). The modification has a negligible performance impact. Note that Filebench is run for $60\,s$ on each test.

In all scenarios, the host is a 12th Gen Intel(R) Core(TM) i7-1280P with 20 cores and 48 GB of RAM, running Ubuntu Linux 24.04, with a Linux kernel 6.8.0.

We tested Socarrat on four different architectures:

- Raspberry Pi 4 model B running in 64-bit mode. It is based on a BCM2711, with a quad core Cortex-A72 (ARM v8) 64-bit SoC with a clock running at 1.8GHz and 4GB of RAM. It has a USB 2.0 OTG port (high-speed). Its operating system is Debian 12 (Bookworm) with a Linux kernel 6.12.34. The storage is an SD card SanDisk Ultra 10 HCI, with read speeds of 98 MB/s and write speed of 10 MB/s (Class 10 rating with Ultra High Speed Class 1 rating). This is the cheapest device evaluated in the experiments (≈ 50 €).
- Raxda ROCK 4c + in 64-bit mode. It is based on RK3399-T, which has 6 cores: a dual Cortex-72 at 1.5GHz and a quad Cortex-A53 at 1.0GHz. It has 4GB of RAM and a USB 3.0 OTG port (super-speed). The storage is an SD card SanDisk Extreme, microSDXC Class 10 U3 A2 V30 with 170 MB/s of read speed and 90 MB/s of write speed. Its operating system is an Armbian 25.5.0-trunk.550 (based on debian Bookworm) with a Linux kernel 6.12.28.
- Raxda ROCK 5b in 64-bit mode. It is based on RK3588, which has a quad Cortex-A76 at 2.2 2.4GHz and a quad Cortex-A55 at 1.8GHz. It has 12 GB of RAM a USB 3.0 OTG port (super-speed). The storage is an SD card SanDisk Extreme, microSDXC Class 10 U3 A2 V30 with 170 MB/s of read speed and 90 MB/s of write speed. Its operating system is an Armbian 25.5.2 (based on debian Bookworm) with a Linux kernel 6.1.115.
- Orange Pi 5 Ultra in 64-bit mode. It is also based on RK3588, with the same processors as the Raxda ROCK 5b, and 16 GB of RAM. It has a USB 3.0 OTG port (super-speed). The boot storage is a SD card SanDisk Extreme PRO, Class V30 U3 A2 XCI, with 200 MB/s of read speed and 120 MB/s of write speed. The additional storage is a 1 TB WD BLACK SN850X NVME drive. Its operating system is an Armbian 25.5.2 with a Linux kernel 6.1.115. This is the most expensive device evaluated in the experiments (≈ 230 €).

We had to backport SealFS to the kernel 6.1 to make it work in the ROCK 5b and the Orange Pi 5 Ultra (it worked out of the box for the

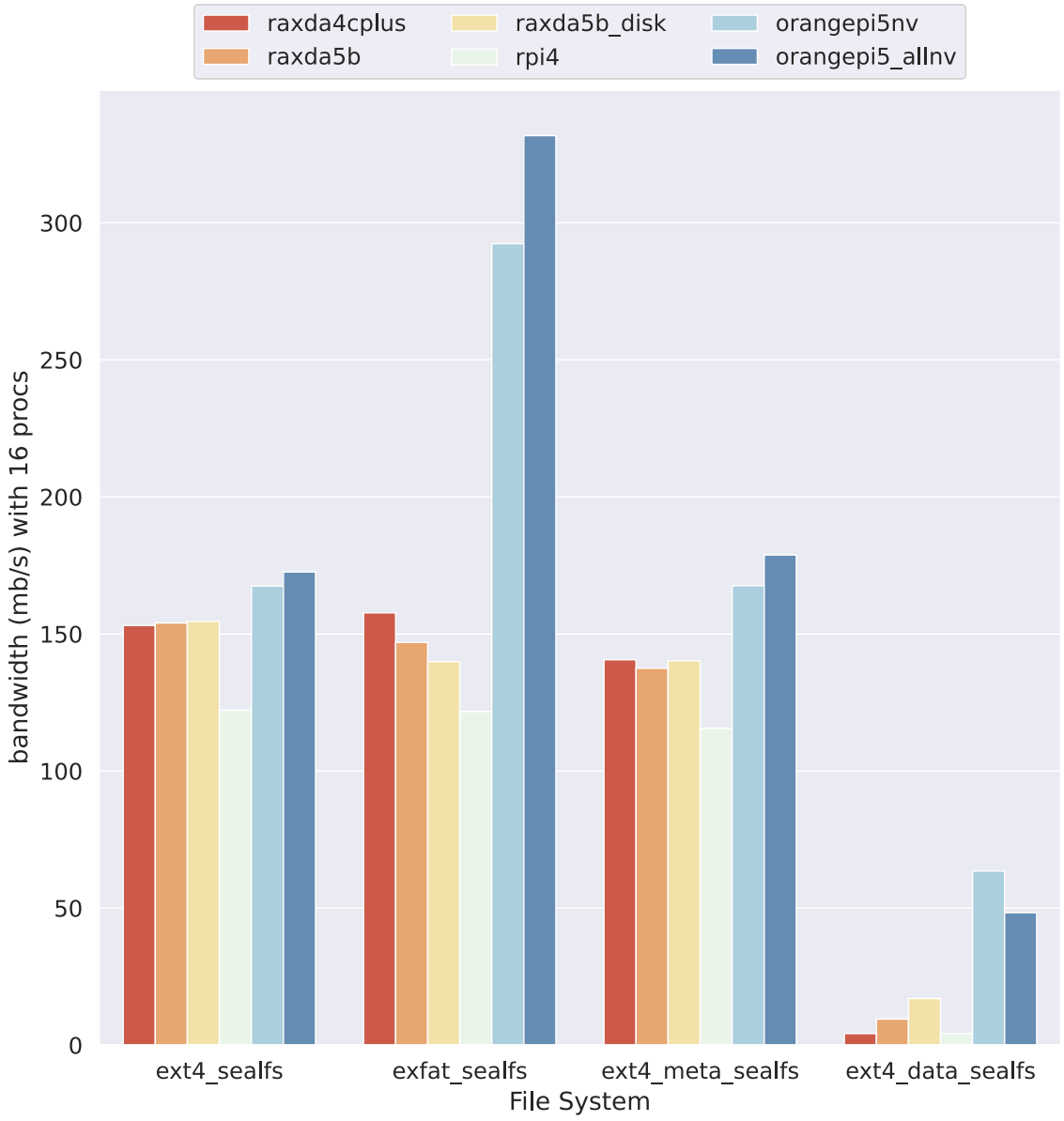

Fig. 7. Comparison of the results of Filebench (bandwidth) for 16 concurrent processes, for different file systems.

other devices). Even though there are other versions of Armbian with more modern kernels, USB OTG is not supported yet.

The Raxda ROCK 5b architecture has been measured with two configurations:

- Both the Socarrat image and the real logs stored on the boot SD Card (labeled `raxda5b` in the plots).
- The Socarrat image stored on an external USB SSD disk and the real logs stored on the boot SD Card (labeled `raxda5b_disk` in the plots).

The Orange Pi 5 Ultra architecture has been measured with two configurations:

- The Socarrat image stored on the NVME disk and the real logs stored on the boot SD Card (labeled `orangepi5nv` in the plots).
- Both the Socarrat image and the real logs stored on the NVME disk (labeled `orangepi5_allnv` in the plots).

In the rest of the architectures, both the Socarrat image and the real logs are stored on the boot SD Card.

Figs. 4 and 5 show the results of the measurements performed by Filebench for different number of concurrent processes. Fig. 4 shows the bandwidth and Fig. 5 shows the latency (the average time for an operation). In those plots:

- Bars labeled `_nosoca` depict measurements where the device exports directly an image filesystem file using USB OTG. That is, this is the baseline for the measurements, because `soca` is not involved (i.e. there is no Reverse File System). Note that, in this case, the block device is exported directly (there is no NBD).

  The other cases export an NBD device served by soca using USB OTG.
- Bars labeled `_nosealfs` use soca but do not keep the real logs under a SealFS mounted directory. They provide a baseline to measure the impact of SealFS. Bars labeled with `_sealfs` use soca with SealFS.
- Bars labeled `ext4_meta` use ext4 with a journal configured as journal_data_ordered.
- Bars labeled `ext4_data` use ext4 with a journal configured as journal_data.
- The rest of bars labeled `ext4_` use ext4 without journaling.
- Bars labeled `exfat_` use exFAT.

Figs. 4 and 5 reflect the impact of concurrency in the host. As the host has 20 CPU cores, we have selected a constant number of processes (16 processes, less than 20 cores) for the next figures, which are focused on bandwidth. Fig. 6 shows the bandwidth results for 16 concurrent processes, grouped by hardware configuration. Fig. 7 shows the bandwidth results, grouped by file system configuration. Those figures only depict the file system configurations with all the security properties explained in our model (i.e. Socarrat and SealFS).

7.3. Analysis and discussion

Note that this is a complex system of two interconnected machines with many different hardware and software components involved, so analyzing the results is not trivial.

Regarding the latency, it grows with the number of processes in all cases. This is due to contention in the host device (e.g. spin locks in the kernel). When the number of processes is greater than the number of CPU cores, the performance is notably worse. In the case of ext4 with data journaling, the latency is extremely worse. This is due to the same effect that we will describe later, in the bandwidth analysis.

The bandwidth is worse for one process (versus multiple processes), in all cases. One plausible explanation is that the host is underusing the Socarrat device. While the host is waiting for the response of a USB request, the link is idle, because there are not other concurrent requests to process. With more processes injecting requests, there is bigger utilization of the capabilities of the device (so the bandwidth grows).

In general, the bandwidth is not worse with 32 processes, even when the host has 20 CPU cores. This is expected, because they are I/O bound processes. Nevertheless, in some cases there is exhaustion with 32 processes. This is the case of ext4 with data journaling in the most powerful architectures. In this cases, the journal is the bottleneck.

There is a clear performance gap between `ext4_data` and the rest of file systems. This is the cost of strong guarantees. If we want the strong guarantees offered by the journal, `soca` has to wait synchronously for the points in time where the storage device commits the data and the file system is coherent. Moreover, the journal is a circular data structure that gets full and we have to wait for parts of it to be evicted before overwriting them, which compounds this effect (it is much worse when there is data is also journaled).

When only the metadata is journaled, there is not so much traffic. Note that in the worst cases (Raspberry Pi 4 and Raxda 4c +), the performance is greater or equal than 4.2 MB/s, which should be more than enough for a regular logging scenario. In the best scenario, we provide more than 63.5 MB/s. Note that recommended bandwidth to stream Ultra High Definition (UHD) video is 15 Mbps or higher.[11]

As expected, the performance of ext4 with journaling improves in the cases with faster storage devices (e.g. NVME disk) and dual storage configurations (i.e. mixing the SD Card and the NVME disk). In the second case, performance benefits may be derived from the load balancing effect of having two different drives.

A surprising result is that, in some cases, using Socarrat is slightly faster than exporting the block device directly. Note that we measured the latter as a baseline. This could be a consequence of two compounded effects:

- `soca` caches heavily (in different ways, including keeping a queue of answered but not attended operations).
- A regular block device in Linux has a small block size (normally between 4 KB and 16 KB). NBD devices, on the other hand, can handle bigger writes of contiguous data, making operations more efficient.

The effect of these factors along the chain ends up delivering bigger chunks to write to disk at the other end, which is also more efficient.

[11] See for example Netflix recommendations: https://help.netflix.com/en/node/306

Comparing architectures (Fig. 6), the best hardware in all cases is the Orange Pi 5 Ultra, as expected. Surprisingly, the Raspberry Pi 4 (the cheapest and more widely available configuration) is not much worse than the Raxda 4c + , even though it only supports USB OTG 2.0.

Regarding file systems, the fastest is exFAT, because there is no journaling and the blocks (i.e. clusters) are bigger, and allocated contiguously when possible. Also, the file system data structures are simpler (i.e. it requires updating less blocks per write).

SealFS does not affect latency or bandwidth significantly, as can be observed in Figs. 4 and 5.

8. Conclusions

In this work, we have presented Socarrat, a novel solution for implementing Write Once Read Many (WORM) storage devices using a low-cost, widely available single-board computers. Socarrat achieves strong data immutability guarantees without relying on proprietary hardware or complex distributed infrastructures.

Our approach, based on the Reverse File System, enables transparent integration with standard file systems such as ext4 and exFAT, requiring no modifications to the host's operating system or additional drivers. Socarrat not only ensures WORM append-only files, but also incorporates tamper-evident features that enhance its resilience against physical compromise.

The system is fully implemented in Go and released as free/libre software, promoting transparency and reproducibility. Our evaluation demonstrates that Socarrat exhibits reasonable performance when running on cost-effective hardware: It can be practically employed for secure logging purposes, even in demanding scenarios such as video logging.

As future work, it would be interesting to implement the local Linux interface using ublk (Ublk, 2025) instead of NBD. It would probably have performance benefits (less copies of memory and system call batching, due to the io_uring interface). Another possible change would be to switch the local Linux interface to netlink communication with the kernel (Netlink, 2025). Another idea left as future work is the possibility of sealing files. Once a file is done with, it could be sealed and become immutable. For example, in ext4, this can be done by changing its attributes. Once a file is sealed, a trail is written to it (so that SealFS logs it) and `soca` closes it. This file is considered read only and the real log file for it cannot be changed.

Socarrat is free/libre software (GPL). The source code is available in the our public repositories: https://gitlab.eif.urjc.es/publications/soca

CRediT authorship contribution statement

Gorka Guardiola Múzquiz: Conceptualization, Data curation, Investigation, Methodology, Resources, Software, Supervision, Validation, Writing – original draft, Writing – review & editing; **Juan González-Gómez:** Software, Writing – review & editing; **Enrique Soriano-Salvador:** Conceptualization, Data curation, Funding acquisition, Investigation, Methodology, Software, Validation, Writing – review & editing.

Data availability

No data was used for the research described in the article.

Funding

This work has been funded by MICIU/AEI/10.13039/501100011033 and "ERDF/EU" (European Union), grant PID2021-126592OB-C22 CASCAR/DMARCE.

Declaration of competing interest

There are no competing interests in this work.

Acknowledgements

The authors would like to thank Fermín Galán Márquez for constructive criticism of the manuscript. Generative AI software tools (Microsoft Copilot[12]) have been used exclusively to edit and improve the quality of human-generated existing text.

References

Amazon, 2025. Locking objects with Object Lock. https://docs.aws.amazon.com/AmazonS3/latest/userguide/object-lock.html.

Bellare, M., Yee, B.S., 1997. Forward integrity for secure audit logs. Technical Report. University of California at San Diego.

Configfs. Linux usb gadget configured through configfs, 2025. https://www.kernel.org/doc/html/latest/usb/gadget_configfs.html.

Davies, A., Orsaria, A., 2013. Scale out with glusterfs. Linux Journal 2013 (235), 1.

Debiez, J., Hughes, J., Aprille, A., 2003. Virtual worm method and system. US Patent 6615330.

exfat filesystem, [accessed sep-2025]. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/fileio/exfat-specification.

Ext4 filesystem, accessed sep-2025]. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/ext4/.

EU-GDPR. General data protection regulation (GDPR). European Union, 2016. https://gdpr-info.eu.

Finlayson, R., Cheriton, D., 1987. Log files: an extended file service exploiting write-once storage. In: Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Ser. SOSP '87. Association for Computing Machinery, pp. 139–148. https://doi.org/10.1145/41457.37516

Ghemawat, S., Gobioff, H., Leung, S.-T., 2003. The google file system. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Ser. SOSP '03. Association for Computing Machinery, pp. 29–43. https://doi.org/10.1145/945445.945450

Gluster, 2025. Gluster Specs Repository: WORM (Write Once Read Many). https://github.com/gluster/glusterfs-specs/blob/master/done/Features/worm.md.

Guardiola-Múzquiz, G., Soriano-Salvador, E., 2022. Sealfsv2: combining storage-based and ratcheting for tamper-evident logging. Int. J. Inf. Secur 22 (2), 447–466. https://doi.org/10.1007/s10207-022-00643-1

Helland, P., 2015. Immutability changes everything: we need it, we can afford it, and the time is now. Queue 13 (9), 101–125. https://doi.org/10.1145/2857274.2884038

Hoare, C. A.R., 1978. Communicating sequential processes. Commun. ACM 21 (8), 666–677. https://doi.org/10.1145/359576.359585

Hsu, C.-C., Wu, W.-C., Jhang, W.-C., Xiao, Z.-K., Chang, H.-C., Hsu, M.-Y., Nanda, U., 2024. Barium titanate write-once read-many times resistive memory with an ultra-high on/off current ratio of 108. J. Alloys Compd. 988, 174252. https://www.sciencedirect.com/science/article/pii/S0925838824008399.

Huawei, 2025. Oceanstor dorado v6 series 6.1.x and v700r001 hyperlock feature guide. https://support.huawei.com/enterprise/en/doc/EDOC1100264821/426cffd9/about-this-document.

Ibm, 2025. IBM storage scale: Immutability and appendonly features. https://www.ibm.com/docs/en/storage-scale/5.2.2?topic=scale-immutability-appendonly-features.

Jbd2. 2025. Journal (jbd2). https://www.kernel.org/doc/html/latest/filesystems/ext4/journal.html.

Leppäniemi, J., Mattila, T., Kololuoma, T., Suhonen, M., Alastalo, A., 2012. Roll-to-roll printed resistive worm memory on a flexible substrate. Nanotechnology 23 (30), 305204.

Loggly, 2019. Remote Logging Service. https://www.loggly.com/solution/remote-logging-service/.

Logsentinel, 2019. Official Web Page. https://logsentinel.com.

Múzquiz, G.G., Soriano-Salvador, E., 2025. Socarrat for plan 9: building 9p worm devices. 11th International Workshop on Plan 9.

Nbdkit. Nbd virtual floppy disk from directory, 2025. https://libguestfs.org/nbdkit-floppy-plugin.1.html

NBD. Network block device github. https://github.com/NetworkBlockDevice, 2024.

Netlink, 2025. Introduction to netlink. https://www.kernel.org/doc/html/next/userspace-api/netlink/intro.html.

Oliveira, F., Guardiola, G., Patel, J.A., Hensbergen, E.V., 2007. Blutopia: stackable storage for cluster management. In: 2007 IEEE International Conference on Cluster Computing, pp. 293–302. https://doi.org/10.1109/CLUSTR.2007.4629243

Peterson, Z., Burns, R., 2005. Ext3cow: a time-shifting file system for regulatory compliance. ACM Trans. Storage 1 (2), 190–212. https://doi.org/10.1145/1063786.1063789

Pike, R., Presotto, D., Thompson, K., Trickey, H., 1990. Plan 9 from bell labs. In: Proceedings of the Summer 1990 UKUUG Conference, pp. 1–9.

Quinlan, S., 1991. A cached worm file system. Software: Practice and Experience 21 (12), 1289–1299.

Quinlan, S., Dorward, S., 2002. Venti: a new approach to archival data storage. In: Conference on File and Storage Technologies.

Quinlan, S., Mckie, J., Cox, R., 2003. Fossil, an archival file server. Technical Report. World-Wide Web document.

Rosa, M., Barraca, J.P., Rocha, N.P., 2019. Logging integrity with blockchain structures. In: Rocha, A., Adeli, H., Reis, L.P., Costanzo, S. (Eds.), New Knowledge in Information Systems and Technologies. Springer International Publishing, pp. 83–93.

Samba, 2017. Using the worm vfs module. https://wiki.samba.org/index.php/Using_the_worm_VFS_Module.

Selvaganesan, M., Liazudeen, M.A., 2016. An insight about glusterfs and its enforcement techniques. In: 2016 International Conference on Cloud Computing Research and Innovations, pp. 120–127.

Shvachko, K., Kuang, H., Radia, S., Chansler, R., 2010. The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10.

Soriano-Salvador, E., Guardiola-Múzquiz, G., 2021. Sealfs: storage-based tamper-evident logging. Computers and Security 108, 0102325.

Stackdriver, 2019. Stackdriver Logging. https://cloud.google.com/logging/.

Suk, J., No, J., 2010. Worm-based data protection approach. In: 2010 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, pp. 410–416.

Tarasov, V., Zadok, E., Shepler, S., 2016. Filebench: a flexible framework for file system benchmarking. login Usenix Mag. 41, 56553130.

Ublk. Userspace block device driver (ublk driver), 2025. https://docs.kernel.org/block/ublk.html.

USB Implementers Forum, 2012. On-the-go and embedded host supplement to the usb.

USSEC. Electronic storage of broker-dealer records. U. S. Securities and Exchange Commission, 2003. https://www.sec.gov/rule-release/34-47806.

Vvfat, Qemu block driver for virtual vfat, 2025. https://github.com/qemu/qemu/blob/master/block/vvfat.c.

Walter, J., Tulledge, D., 2024. Compliant worm storage using netapp snaplock. Technical Report. NetApp Technical Report.

Wang, H., Yang, D., Duan, N., Guo, Y., Zhang, L., 2018. Medusa: blockchain powered log storage system. 2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS) 11, 518–521.

Wang, Y., Zheng, Y., 2003. Fast and secure magnetic worm storage systems. In: Second IEEE International Security in Storage Workshop, pp. 11.

White, R., Caiazza, G., Cortesi, A., Cho, Y., Christensen, H., Black block recorder: immutable black box logging for robots via blockchain. IEEE J. Robot. Autom. 4, 2019.

[12] https://www.bing.com/chat