slackFS – resilient and persistent information hiding framework

Avinash Srinivasan*

Department of Cyber Science,
United States Naval Academy,
597 McNair Road, Hopper Hall – Room 478,
Annapolis, MD 21402, USA
Email: srinivas@usna.edu
*Corresponding author

Christian Rose

Amber Orchard Ct.,
Odenton, MD 21113, USA
Email: christian.j.rose00@gmail.com

Jie Wu

Department of Computer and Information Sciences,
Center of Networked Computing,
Temple University,
SERC 362, 1925 N. 12th Street,
Philadelphia, PA 19122, USA
Email: jiewu@temple.edu

Abstract: The ever-expanding cyberspace, driven by digital convergence, inadvertently broadens the attack surface. Savvy modern cybercriminals have embraced steganography as a key weapon. This paper introduces *slackFS*, a novel steganographic framework utilising file slack space for covert data concealment. Unlike prior methods focusing on individual files, *slackFS* hides entire filesystems, offering a structured means for data exfiltration. It ensures persistence across system reboots, robust detection resistance, portability, and minimal performance impact. Incorporating erasure-code-based fault-tolerance, *slackFS* enables recovery from partial loss due to accidental slack space overwriting. Prototype validation on Ubuntu 20.04 with ext4 filesystems as the cover medium and FAT16 as the hidden malicious filesystem is conducted. The study includes testing of three coding libraries and two Reed-Solomon erasure code implementations – VANDERMONDE and CAUCHY matrices – highlighting *slackFS*'s resilience and effectiveness.

Keywords: attacker; data exfiltration; fault-tolerance; filesystem; information hiding; malicious; operating system; persistence; resilience; security; steganography.

Reference to this paper should be made as follows: Srinivasan, A., Rose, C. and Wu, J. (xxxx) 'slackFS – resilient and persistent information hiding framework', *Int. J. Security and Networks*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Avinash Srinivasan is an Associate Professor in the Cyber Science department at the US Naval Academy. He holds a PhD, an MS in Computer Science, and a BE in Industrial Engineering. His research interests span the broad areas of cybersecurity and forensics focusing on network security and forensics, security and forensics in cyber physical systems, and critical infrastructure, steganography and information hiding. He has published 50 papers in prestigious refereed conferences and journals including IEEE INFOCOM, ICDCS, and ACM SAC. He is a co-inventor on a patent (Patent number: 11210396). He is a Certified Ethical Hacker (CEH) and Computer Hacking Forensics Investigator (CHFI).

Christian Rose is currently an active-duty Lieutenant in the US Navy. He holds a BS in Computer Science and Information Technology from the United States Naval Academy (Class of 2023) and he is working on his MS in Computer Science. His current research interests are in network and hardware security, focusing on adversarial approaches to compromising systems and protocols.

Jie Wu is the Director of the Center for Networked Computing and Laura H. Carnell Professor at Temple University. He also serves as the Director of International Affairs at College of Science and Technology. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu regularly publishes in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Service Computing*, *Journal of Parallel and Distributed Computing* and *Journal of Computer Science and Technology*. He is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.

This paper is a revised and expanded version of a paper entitled 'Steganography with FileSystem-in-SlackSpace' presented at IEEE International Conference on Networking, Architecture and Storage (NAS), Philadelphia, PA, USA, 3–4 October 2022.

1 Introduction

Cyber threats, whether internal or external, pose risks to organisations regardless of size or sector. The ever-expanding cyberspace, driven by digital convergence, widens the potential targets for attacks. Rapidly evolving technology, while beneficial, has an inevitable dark side and that is it affords the malicious users the exact same capabilities. In this dynamic ecosystem, adversaries exhibit intelligence and adaptability, especially those employing *advanced persistent threats*. Their primary goals include maintaining stealth and persistence within target systems to evade detection for as long as possible. Achieving these objectives necessitates operating beneath the detection thresholds of deployed security measures. Prolonged presence within a target system significantly enhances the adversary's likelihood of success, emphasising the importance of effective detection and response mechanisms in mitigating cyber threats.

Steganography originates from the Greek *steganographia*, with *steganós* meaning *covered* or *concealed* and *graphia* meaning *writing*. Concealing files and folders is a commonly used method for hiding information, serving both beneficial and malicious purposes. Beneficial applications involve system administrators concealing certain files to safeguard them from accidental corruption, or operating systems hiding files to prevent inadvertent access by regular users. Conversely, malicious applications entail adversaries modifying file and folder attributes to render them hidden, aiming to evade detection effectively.

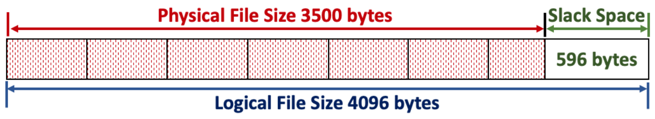
Computer systems utilise filesystems to manage data stored on storage mediums. A file system is a method used by computers and operating systems to organise and store data on storage devices such as hard drives, solid-state drives (SSDs), or external storage media. It provides a structured way to manage files, directories (also known as folders), and metadata associated with them. However, filesystem specifications can sometimes lead to unexpected behaviours, such as concealing data within filesystem data structures or partial and complete unused data blocks (slack). Apart from the filesystem, these storage mediums often contain protected and hidden regions suitable for concealing information. For instance, areas like *host protected area* (HPA) and *device configuration overlay* (DCO) were introduced by manufacturers to offer flexibility to vendors. HPA, situated at the disk's end, allows computer vendors to store data unaffected by user formatting or erasing of disk contents. DCO, on the other hand, enables vendors to limit a hard disk's capabilities by implementing optional features (Carrier, 2005). These regions provide additional avenues for information hiding beyond the conventional filesystem structures.

Other regions on a storage disk that can be exploited for information hiding include *partition slack*, *volume slack*, and *inter-partition gap* among others, all of which occur naturally and cannot be prevented. Furthermore, all of the aforementioned regions are inaccessible by the normal users and more importantly they are transparent to the operating system and commercially-off-the-shelf (COTS) security tools. However, a sophisticated threat actor can readily exploit these regions for hiding information that can be later exfiltrated. Consequently, the last decade has seen the adversaries' cyber arsenal expand significantly to encompass storage disk for steganography and other information hiding techniques that offer robust persistence and stealth.

1.1 The big picture

In this paper, we present *slackFS* – a steganographic information hiding framework that exploits the available slack space on the target system to hide an entire *filesystem volume*. We refer to the filesystem volume to be hidden as the *malicious filesystem volume*, which is denoted as M_{vol}^{fs}. The target system (i.e., file system) that will serve as the cover medium is denoted as T_{vol}^{fs}. After a vanilla installation of an operating system, there are several hundred thousand files created on the T_{vol}^{fs}, prior to any updates or installation of any user applications. Our proposed *slackFS* framework leverages the slack space of these hundreds of thousands of files for hiding the M_{vol}^{fs}. *slackFS* initially stores the files to be hidden in the M_{vol}^{fs}. Then it unmounts the M_{vol}^{fs}, splits it into smaller chunks, i.e., fragments, and distributes the chunks across the T_{vol}^{fs} by hiding each chunk in the slack space of a different file, i.e., each fragment of the entire M_{vol}^{fs} has a separate cover file. To access the information hidden in the M_{vol}^{fs} or to hide new information, *slackFS* reassembles the M_{vol}^{fs} from the hidden chunks and remounts it. The adversary has an optional *advanced model*, which differs from the above described *basic model*, in that it adds fault tolerance to the framework. Therefore, the *advanced model* enables the adversary to recover the entire M_{vol}^{fs} even if the hidden chunks are lost up to a certain threshold, which is a tunable parameter.

Figure 1 Illustration of file slack space (see online version for colours)



1.2 *Unique features and key contributions*

Our *slackFS* framework, unlike existing techniques, uniquely hides entire filesystem volumes in slack space, offering structured storage for numerous files with minimal overhead. In our prototype, we confirm this with a 200MB *FAT16* volume, that the overhead is merely 0.11%. Unlike concealing individual files, this approach enhances security by making recovered filesystem fragments more complex to analyse due to intermingled data from adjacent blocks and filesystem structures.

Our *slackFS* framework is also lightweight, portable and adaptable to the available slack space on the T_{vol}^{fs}. With a modular design, *slackFS* framework can be readily expanded to implement and test various filesystems for their suitability as T_{vol}^{fs} as well as M_{vol}^{fs}. Information hidden using *slackFS* framework has the following key strengths which adds to its stealth and persistence.

- Persistence across system reboots and restarts: the hidden information is part of the non-volatile disk space. Additionally, it is impractical for the operating system to clean the slack space of each and every file on a system since the number of files can be in millions on a production system.
- Robust detection resistance: information is hidden in file slack space that is inaccessible to the operating system and COTS security tools.

Furthermore, *slackFS* framework has minimum impact on the target system's performance, which we confirm in Section 5. As part of the *slackFS* framework, we propose and implement a *basic model* and an *advanced model*. A key distinction between the basic and advanced models is that the advanced model is designed to be fault-tolerance given the volatile nature of a filesystem volume. The advanced model, due to in-built fault-tolerance, is robust to accidental modification or deletion of files on the T_{vol}^{fs} and therefore enables the adversary to reconstruct the entire M_{vol}^{fs}. Traditional COTS security tools that scan for deviations in disk space utilisation will fail to detect the information hidden using the *slackFS* framework for the following key reasons:

1 writing to the slack space of a file does not alter the file's content, i.e., the file size remains unaltered since the slack space of a file is considered to be part of the allocated disk space as it is factored in the file's logical size (Figure 1)

2 writing to the slack space of a file does not alter the file's accessed or modified date and time stamps.

We confirm the capacity, stealth, persistence, and robustness of the proposed basic and advanced models through prototype implementation on a *Ubuntu 20.04* on *ext4* filesystem as the T_{vol}^{fs}. Additionally, for the advanced model, we implement fault-tolerance with erasure codes and confirm its performance. Finally, our proposed *slackFS* framework satisfies the three key requirements of steganography:

1. uses the T_{vol}^{fs} as the cover file
2. hides the very existence of M_{vol}^{fs}
3. writes to the slack space of files on T_{vol}^{fs} without altering their contents.

1.3 *Road-map*

In Section 2, we discuss background information and review research literature relevant to the presented research work. In Section 3, we present the overview and the working of the proposed *slackFS* framework. In Section 4, we provide the implementation details of basic model of the *slackFS* framework followed by discussion of results and analysis. In Section 5, we present the implementation details, results and analysis of the advanced fault-tolerant model of the *slackFS* framework, followed by conclusions in Section 6.

2 Background and related work

Digital information hiding can be broadly categorised into three types – cryptography, steganography, and watermarking (Rasmi et al., 2019). While they all strive to achieve secure or proprietary communications, there are some key differences (Berghel et al., 2006) and we will specifically focus on the strength of and distinction between cryptography and steganography. Cryptography is a very powerful security tool that can protect data at rest and in motion from unauthorised disclosures or modifications. However, cryptography cannot hide the very existence of such protected data and or communications. Furthermore, in the ever-evolving cyber domain with constantly expanding attack surface, the adversary seeks stealth and strives to hide the very existence of any hidden data or communications for which cryptography is not the preferred tool. Consequently, over the last couple of decades, steganography has evolved into a weapon of choice for cyber-crime and cyber-espionage as the adversaries increasingly seek to hide their malevolence in plain sight – embedded in banner ads, text messages, or images (Cameron, 2016).

Steganography has been and continues to be used extensively for information hiding and exfiltration. It is important to note that a well designed steganographic information hiding technique can hold a significant payload with no discernible impact on the cover file or the system's performance.

The quality of a steganographic technique is determined by the following key characteristics (Metcheka and Ndoundam, 2020; Evsutin et al., 2020) –

1 *Stealth* – how hard is it to detect?

2 *Capacity* – how much data can it hide?

3 *Robustness* – how much modification to the cover medium can it survive?

Some of the existing works that consider hiding data or individual files in slack space include Srinivasan et al. (2013) and Thampy et al. (2018). In Srinivasan et al. (2017), the authors use Shamir's secret sharing (Shamir, 1979) for fault-tolerance in hiding of individual files. Other similar fault-tolerant methods include Rabin (1989), where in the authors present an information dispersal based on non-systematic erasure codes and Reed-Solomon codes that are block-based error correcting codes (Reed and Solomon, 1960).

2.1 *Files – physical size, logical size and slack space*

In this section, we provide background information relevant to this work, more details on the geometric structure and nested data structures of a filesystem and the basic hard disk geometry can be found in Berghel et al. (2006).

A filesystem is the data structures used by the operating system for storing files and folders in an organised way and for retrieving them (Wirzenius et al., 1991). Furthermore, it is the filesystem that manages access to both the contents and metadata of all files and folders. Some popular traditional filesystems used by mainstream operating systems include:

- Windows operating systems: FAT12/16/32 and NTFS
- Unix/Linux operating systems: ext2/3/4, XFS, and JFS
- Macintosh operating systems: APFS and HFS+.

A file has two associated sizes – physical size and logical size. Physical size of a file is the actual number of bytes that constitute the file. Logical size is the total number of bytes allocated to the file and is always a multiple of 512. Furthermore, on a given filesystem, there are two types of allocation units – *physical allocation unit* and *logical allocation unit*. Typically, a group of two or more contiguous physical allocation units constitute a logical allocation unit. On Windows systems, a physical allocation unit is called a *sector* and a logical allocation unit is called a *cluster*. On a Unix/Linux system, a physical allocation unit is called a *block* and a logical allocation unit is called an *IO block*. Physical-to-logical allocation mapping is presented in Figures 2 and 3. Specifically, Figure 2 depicts the physical-to-logical allocation mapping of a typical Windows *FAT16* filesystem volume and Figure 3 depicts the physical-to-logical allocation mapping of a typical *ext4* filesystem volume.

Figure 2 Physical-to-logical allocation mapping: M_{vol}^{fs} (see online version for colours)

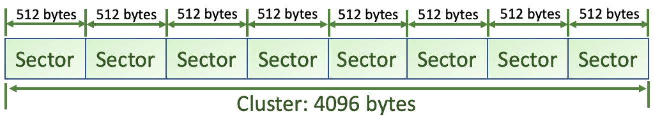
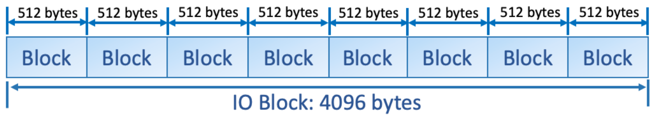


Figure 3 Physical-to-logical allocation mapping: T_{vol}^{fs} (see online version for colours)



When files are allocated space on a storage drive, the underlying filesystem can only allocate whole blocks to individual files. The block (or sector) size on a given filesystem is typically a multiple of 512 bytes. Since file size, i.e., physical size, is rarely exactly a multiple of IOBlock size, such storage allocation results in wasted space on the last IOBlock allocated to a file. This wasted space within the last allocated IOBlock of a file is called *slack space*. An example illustrating a file's physical and logical sizes, and the resulting slack space in illustrated in Figure 1. The quantity of storage lost due to slack space may seem trivial for an individual file. However, the total amount of storage space lost due to file slack space can be quite substantial across a given filesystem (Mulazzani et al., 2013a). This is especially the case if the system uses large block sizes to improve performance while the user predominantly stores small files.

While slack space cannot be entirely eliminated, it can be mitigated by optimising the logical allocation unit size. Moreover, certain filesystems incorporate a *block sub-allocation* feature, inherently addressing the slack space issue. This feature, exemplified by a clever BSD implemented algorithm, enables the storage of partially-filled blocks from multiple files within a single block, efficiently utilising slack space at the end of large files (Wikipedia, 2022). However, our proposed method is tailored for traditional filesystems lacking this *block sub-allocation* capability.

2.2 *Related work*

Numerous ideas have been proposed that exploit various aspects of a filesystem for steganography and information hiding. Below is a summary of the broad ideas existing in literature:

- *Bad cluster marking* – adversary hides data by writing it to clusters (a.k.a. IOBlocks) in the unallocated space of the filesystem. Subsequently adversary marks those specific clusters as 'BAD' clusters in a data structure the operating system uses to track free clusters. This forces the operating system to not use those clusters, thereby leaving the attacker's data unaltered.
  Note: unallocated space is the region of the filesystem volume (i.e., storage) that is unused and available for allocation for incoming storage requests.
- *Hidden partitions* – attacker creates a partition that is hidden from the operating system which stores exfiltration data, malware, etc. (Hassan and Hijazi, 2017). Since the partition is hidden from the operating system, it successfully evades detection by security tools. Protection against hidden partition-based attacks should scan and account for every byte of used and unused disk space.
- *Hidden files and folders* – the adversary can change the attribute of files and folders to hidden in order to hide information and evade detection. On Linux/Mac systems this is as simple as prefixing the file name with a period ('.'). *Okrum*, a Windows backdoor seen in use since December 2016 is known to have used hidden files to store logs and outputs from backdoor commands before exfiltration (Microsoft, 2020; MITRE T1564.001, 2020). Similarly, *MacSpy*, a *malware-as-a-service* offered on the dark-web, hides itself in the '$\sim$/Library/.DS_Stores/' folder (MITRE T1564.001, 2020; Ewane, 2017).
- *Hiding filesystem* – adversary may use a hidden filesystem to evade detection. *BOOTRASH* (MITRE T1564.005, 2020), a bootkit that targets Windows operating system, is a *VBR bootkit* that uses the VBR to maintain persistence. It is know to have used a hidden filesystem stores components of the *Nemesis bootkit* in the unallocated disk space between partitions. Information hiding leveraging hidden filesystems is the primary focus of the research presented in this paper.
- *Hiding in slack space* – the adversary hides information in the file slack space, i.e., the unused allocated bytes beyond the physical end of the file (see Figure 1). Over time a majority of files see very little change during system updates and therefore are better suited for information hiding. This is one of the key aspects that our proposed *slackFS* framework leverages. Some of the more relevant works that focus on information hiding in slack space include Srinivasan and Dong (2018), Srinivasan et al. (2013), Mulazzani et al. (2013b) and Huebner et al. (2006).

In Alji and Chougdali (2021), the authors present a tool that can extract information available in file slack associated with each regular file within the new technology file system (NTFS) formatted partition. The presented tool can dump the file slacks in a RAW format and comes with hashing capabilities using MD5 and SHA1 algorithms. The limitation of this tool is that it is developed with forensics analysis as the intended application looking for fragments of deleted evidentiary files. This tool will fall short of recovering and reassembling fragments of a file or malware that is intentionally split into smaller chunks and spread across the slack space of numerous files. In Göbel and Baier (2019), the authors present *fishy*, a framework designed to implement and analyse different filesystem-based data hiding techniques on ext4, FAT and NTFS filesystems. Subsequently, in Göbel et al. (2019), the authors extend the *fishy* framework with a separate module with specific data hiding techniques for APFS. In Koolhaas and van Steenbergen (2020), the authors present their findings regarding the automated detection of hidden and/or modified data within APFS data structures and slack. In Göbel and Baier (2018), the authors analyse the feasibility of using timestamps of the ext4 file system to hide data and the results reveal that the nanoseconds part of the ext4 timestamps can be used to build a system with steganographic strength.

3 *slackFS* – information hiding framework

3.1 Adversary capability and attacker model

In our proposed *slackFS* framework, the attacker(s) can operate independently or in collusion. The attacker(s) can be insider(s), outsider(s), or a combination thereof. We assume that the attacker(s) has access to the T_{vol}^{fs} either through a colluding insider or via an existing vulnerability. Our proposed *slackFS* framework assumes that the adversary has penetrated and established a foothold on the target system. The attacker's objective is to utilise a data exfiltration technique that is stealthy and more importantly persistent across system reboots and restarts.

3.2 The framework

With the *slackFS* framework, upon initial entry, the adversary runs a reconnaissance scan on the target system's secondary storage, i.e., T_{vol}^{fs}. The objective of the reconnaissance scan is to enumerate all files and compute the total available slack space, i.e., cumulative slack space of all files on the T_{vol}^{fs}. The reconnaissance scan can target specific file types (e.g., system files) or files within certain folders (e.g., '/bin', '/sys'). In particular, system files are better suited for *slackFS* framework since system files are less likely to be modified with the exception of log files. However, in the prototype that we develop, we utilise the files under '/usr' folder which is less likely to raise alert compared to '/bin' or '/sys' folders.

During the reconnaissance scan, the adversary generates a map_file. The map_file maps the amount of slack space available for each file scanned and stores it as a tuple $\langle file\_name, slack\_space\rangle$. Once the map_file is generated, the adversary knows the total available slack space on the entire or with a specific folder(s) on the T_{vol}^{fs}. Subsequently, the attacker creates a M_{vol}^{fs}, whose size is less than the total available slack space on the T_{vol}^{fs} or a specific folder (e.g., '/usr'). Details of *slackFS* framework's reconnaissance scan are presented in Algorithm 1. Note that the input $required\_slack$ is a non-zero value. Also, $map\_file.T_{vol}^{fs}$ is the map_file for the T_{vol}^{fs} generated by Algorithm 1.

Algorithm 1 *slackFS* – target disk reconnaissance

```
Input: $required\_slack$; $T_{vol}^{fs}$;
Output: $map\_file.T_{vol}^{fs}$
Initialisation: $map\_file \leftarrow NULL$; $cover\_file \leftarrow NULL$;
        $file\_slack \leftarrow 0$; $total\_slack \leftarrow 0$;
        $slack\_offset \leftarrow 0$;
1: while (! $EOF[\ T_{vol}^{fs}]$ ) do
2:     $file \leftarrow getNextFile(T_{vol}^{fs})$
3:     $file\_name \leftarrow file.getName()$;
4:     $logSize \leftarrow file.getSize(logical)$;
5:     $phySize \leftarrow file.getSize(physical)$;
6:     $file\_slack \leftarrow [logSize - phySize]$;
7:     $file\_slack\_offset \leftarrow [\ file.getLastByteOffset() + 1\ ]$;
8:     $map\_file \leftarrow [\ file\_name,\ file\_slack,\ file\_slack\_offset)\ ]$
9:     $total\_slack \leftarrow total\_slack + file\_slack$;
10:    if ($total\_slack > required\_slack$) then
11:        break;
12:    end if
13: end while
14: return $map\_file.T_{vol}^{fs}$;
```

Algorithm 2 *slackFS* – STRIP_NULL(M_{vol}^{fs})

```
Input: $M_{vol}^{fs}$;
Output: $null\_map$; ${}^{noNull}M_{vol}^{fs}$
Initialisation: $null\_map \leftarrow NULL$;
        ${}^{noNull}M_{vol}^{fs} \leftarrow NULL$;
1: while ( ! $EOF[M_{vol}^{fs}]$ ) do
2:     $currByte \leftarrow M_{vol}^{fs}.getNextByte()$
3:     if currByte $\neq NULL$ then
4:         ${}^{noNull}M_{vol}^{fs} \leftarrow {}^{noNull}M_{vol}^{fs} + currByte$
5:     for each null_byte_sequence do
6:         $null\_map \leftarrow [null\_start\_offset, null\_seq\_len]$
7:     end for
8: end while
9: return $null\_map,\ {}^{noNull}M_{vol}^{fs}$
```

The data to be hidden is now saved to the M_{vol}^{fs} like any normal file and when complete the M_{vol}^{fs} is unmounted. The adversary then processes splits the M_{vol}^{fs} into smaller chunks based on the available slack space of individual files referring to the map_file and writes them to the slack space of the corresponding files on the T_{vol}^{fs}. Note that the attacker has the option of sorting the map_file from largest to smallest based on file slack space and using only files that have a threshold minimum slack space as cover files. The advantage of sorting the map_file from largest to smallest slack space is that it will minimise the number of chunks the M_{vol}^{fs} has to be split into. Details of the slackFS hiding process is presented in Algorithm 4.

Algorithm 3 *slackFS* – basic model hiding process

```
Input: $map\_file.T_{vol}^{fs}$; $M_{vol}^{fs}$;
Initialisation: $file\_name \leftarrow NULL$; $read\_size \leftarrow 0$;
        $write\_offset \leftarrow NULL$;
    $write\_buf \leftarrow NULL$;
        $cover\_file\_ctr \leftarrow map\_file.T_{vol}^{fs}.getLen()$
    $STATUS \leftarrow success$;
1: STRIP_NULL($M_{vol}^{fs}$)
2:         return ( $null\_map,\ {}^{noNull}M_{vol}^{fs}$ )
3: for ($i = 0$; $i < cover\_file\_ctr$; $i++$) do
4:     $curr\_cover\_file \leftarrow map\_file.getEntry(i)$
5:     $file\_name \leftarrow curr\_cover\_file.file\_name$;
6:     $read\_size \leftarrow curr\_cover\_file.file\_slack$;
7:     $write\_offset \leftarrow curr\_cover\_file.slack\_offset$;
8:     $write\_buf \leftarrow$ READ ( ${}^{noNull}M_{vol}^{fs}$, $read\_size$ );
9:     WRITE ($write\_buf, write\_at\_offset, read\_size$);
       $write\_buf \leftarrow NULL$;
10: end for
11: if ( ! $EOF[\ {}^{noNull}M_{vol}^{fs}\ ]$ ) then
12:     $STATUS \leftarrow failure$;
13: end if
14: return $STATUS$
```

Algorithm 4 *slackFS* – basic model retrieving process

```
Input: $map\_file.T_{vol}^{fs}$; $null\_map$;
Initialisation: ${}^{retrvd}M_{vol}^{fs} \leftarrow NULL$
        $cover\_file\_ctr \leftarrow map\_file.T_{vol}^{fs}.getLen()$
        $STATUS \leftarrow success$;
1: for ($i = 0$; $i < cover\_file\_ctr$; $i++$) do
2:     $curr\_cover\_file \leftarrow map\_file.getEntry(i)$
3:     $file\_name \leftarrow curr\_cover\_file.file\_name$;
4:     $read\_size \leftarrow curr\_cover\_file.file\_slack$;
5:     $read\_offset \leftarrow curr\_cover\_file.slack\_offset$;
6:     $write\_buf \leftarrow$
           READ ($file\_name$, $read\_size$,
       $read\_offset$);
7:     APPEND (${}^{retrvd}M_{vol}^{fs}, write\_buf$);
       $write\_buf \leftarrow NULL$;
8: end for
9: RESTORE_NULL(${}^{retrvd}M_{vol}^{fs}, null\_map$)
10:        return ${}^{rstord}M_{vol}^{fs}$
11: if hash(${}^{rstord}M_{vol}^{fs}$) != hash($M_{vol}^{fs}$) then
12:     $STATUS \leftarrow failure$;
13: end if
14: return $STATUS$
```

In our prototype implementation we only use files from the '/usr' folder on the T_{vol}^{fs}. Subsequently, to access the data stored in the hidden M_{vol}^{fs}, the adversary uses the above process in reverse order. In addition, the adversary uses the same map_file that he used for hiding to reassembles the hidden chunks and recreate the M^{fs}_{vol}, and remounts it on the T^{fs}_{vol}.

Algorithm 5 *slackFS* – advanced model hiding process

```
Input: $map\_file$; $cover\_file\_ctr$; $M^{fs}_{vol}$;
       back-end; DATA_FRAGS;
       PARITY_VALUE (in %);
Initialisation: $file\_name \leftarrow NULL$; $read\_size \leftarrow 0$;
       $write\_offset \leftarrow NULL$;
   $write\_buf \leftarrow NULL$;
       $\mathcal{B} \leftarrow$ back-end; $\mathcal{M} \leftarrow M^{fs}_{vol}$;
       $\mathcal{D} \leftarrow$ DATA_FRAGS;
       $\mathcal{P} \leftarrow$ PARITY_VALUE;
       $\mathcal{T} \leftarrow \mathcal{D} \times (1 + \mathcal{P})$;
       $STATUS \leftarrow success$;
1:  $fragments$ = ERASURE_CODE$(\mathcal{B}, \mathcal{M}, \mathcal{P}, \mathcal{T})$
2:  $frag\_size \leftarrow fragments.getSize()$
3:  for $(i = 0;\ i < cover\_file\_ctr;\ i++)$ do
4:      $curr\_cover\_file \leftarrow map\_file.getEntry(i)$
5:      if ( $curr\_cover\_file.file\_slack \leq frag\_size$ ) then
6:          continue;
7:      end if
8:      $curr\_frag \leftarrow fragments.getFrag()$
9:      if ( $curr\_frag == NULL$ ) then
10:         $STATUS \leftarrow success$;
11:         break;
12:     end if
13:     $file\_name \leftarrow curr\_cover\_file.file\_name$;
14:     $write\_offset \leftarrow curr\_cover\_file.slack\_offset$;
15:     $write\_buf \leftarrow$ READ ($curr\_frag$, $frag\_size$);
16:     WRITE ($write\_buf, write\_at\_offset, frag\_size$);
        $write\_buf \leftarrow NULL$;
17: end for
18: if ( $fragments.getFrag()\ != \ NULL$ ) then
19:     $STATUS \leftarrow failure$;
20: end if
21: return $STATUS$
```

4 *slackFS* – basic model

4.1 Implementation

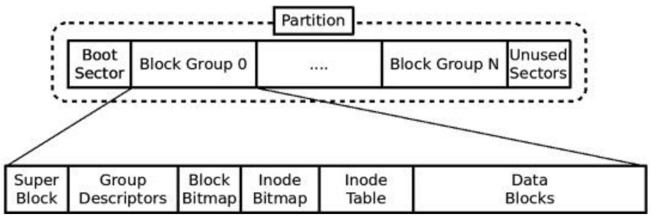
We have implemented and tested a working prototype of the proposed *slackFS* basic information hiding method on a *Ubuntu Linux 20.4 system* with *ext4* filesystem as the T^{fs}_{vol}. No updates, patches or any user applications are installed on the T^{fs}_{vol} with the exception of those tools that were necessary for this research. However, we have excluded all of the files that were created as a result of installing tools necessary for this research, which provides a good baseline estimate for the total available slack on a vanilla *Ubuntu Linux 20.04* installation with *ext4* filesystem. A generic *ext4* filesystem layout is shown in Figure 4.

We have used a 200 MB non-bootable (i.e., storage only volume) *FAT16* volume as our M^{fs}_{vol}, key specifications of which are as follows:

- *bytes-per-sector:* 512
- *sectors-per-cluster:* 8
- *bytes-per-cluster:* 4,096

Detailed layout of the entire M^{fs}_{vol} is presented in Table 1. The generic layout of a *FAT16* filesystem is shown in Figure 5. The *ext4* filesystem, i.e., the T^{fs}_{vol} has a block size of 512 bytes and an IO block size of 4,096 bytes, which is same as the 'sector' and 'cluster' sizes respectively on *FAT16*, i.e., the M^{fs}_{vol}.

Figure 4 An *ext4* filesystem layout

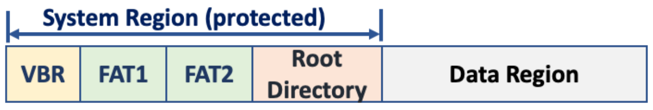


Source: Bovet and Cesati (2005)

Table 1 M^{fs}_{vol} layout used in prototype

M^{fs}_{vol} *region (FAT16)*	*Size (in sectors)*	*Size (in bytes)*	*Sector range*	
			Start	*End*
VBR	1	512	0	0
FAT1	191	97,792	1	191
FAT2	191	97,792	192	382
Root fir.	32	16,384	383	414
Data	390,211	199,788,032	415	390,625

Figure 5 Typical *FAT16* filesystem layout (see online version for colours)



4.2 Results and analysis

During the reconnaissance scan, we analyse the T^{fs}_{vol} and compute the total available slack space. Additionally, during the reconnaissance scan, we identify folders under the root directory, i.e., '/', of the filesystem that has the maximum amount of slack space. Figures 6(a), 6(b) and 6(c), respectively present the available slack space for three important folders under the root directory on the T^{fs}_{vol} – '/bin', '/usr', and '/etc'. For each of the three folders, we map the slack space for the following three different scenarios:

Figure 6 Comparison of slack space (in bytes) for first 100 files (sorted and unsorted) and top 100 files, (a) '/bin' slack space for 100 files (b) '/usr' slack space for 100 files (c) '/etc' slack space for 100 files (see online version for colours)

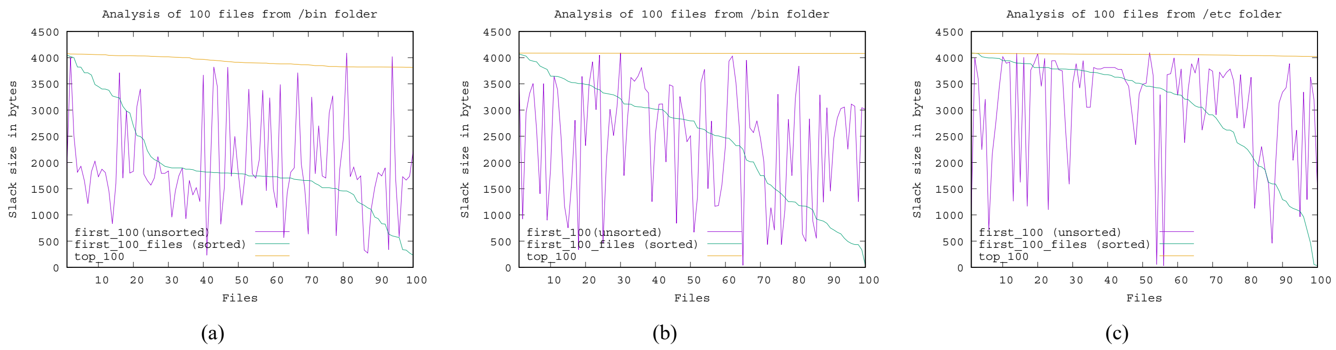
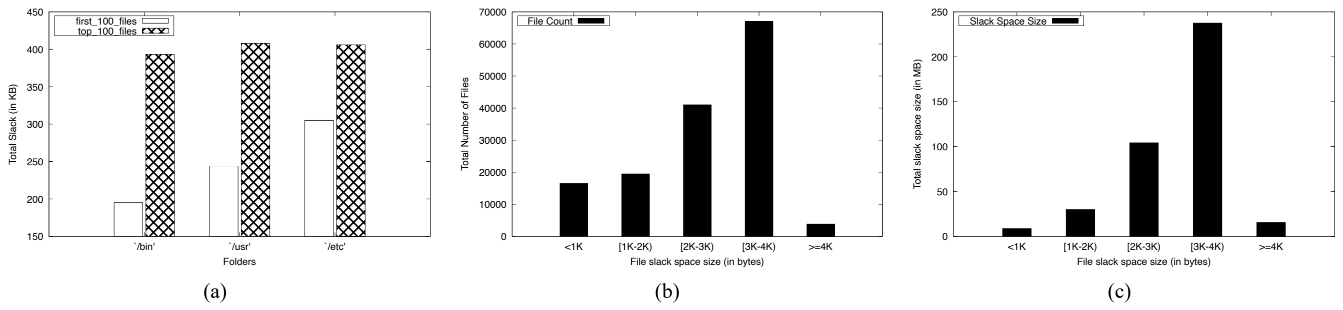


Figure 7 Target filesystem volume slack space information, (a) total slack space (100 files) (b) '/usr' file count.(c) '/usr' total slack space



1 slack space distribution for the first 100 files (unsorted)

2 slack space for the first 100 files (first 100 files on T_{vol}^{fs} or within a folder T_{vol}^{fs} sorted largest to smallest slack space)

3 slack space for top 100 files (top 100 files with all files on the T_{vol}^{fs} or within a folder on the T_{vol}^{fs} sorted largest to smallest slack space).

As can be seen from the results in Figure 6, all three folders have different distribution in the slack space of first 100 files (unsorted). When the first 100 files in each of the three folders are sorted from largest to smallest slack space, we have the following results – the '/bin' folder has the smallest fraction of the first 100 files, with approximately 30%, each having a slack space of 2,000 bytes or more; in the '/usr' folder approximately 70% of the first 100 files each has a slack space of 2,000 bytes or more; and the '/etc' folder has the highest fraction of the first 100 files, with approximately 85%, each having a slack space of 2,000 bytes or more. Finally, due to the existence of very small files, when top 100 files are selected, based on all files on the T_{vol}^{fs} sorted largest to smallest slack space, almost all of the top 100 files in the '/usr' and '/etc' folders have a slack space of $\geq$4,000 bytes but $\leq$4,095 bytes. This is because, the size of an IOBlock (i.e., cluster) on our T_{vol}^{fs} *ext4* is 4096 and at least one byte in an IOBlock (i.e., cluster) has to contain a file's data for the IOBlock (i.e., cluster) to be considered allocated. It is important to note that the maximum possible slack space on a given filesystem volume is the size of the IOBlock (a.k.a. cluster) in bytes minus 1 byte.

In Figure 7, we present the slack space information for our ext4 T_{vol}^{fs}. Figure 7(a) presents a summary of the cumulative slack space for the first 100 and top 100 (slack space sorted largest to smallest) files in the following three folders – '/bin', '/usr', and '/etc'. As can be seen from the results, '/etc' folder with approximately 305 KB has the most available slack space for the first 100 files while the '/usr' folder with approximately 410 KB has the most available slack space for the top 100 files.

Figure 7(b) shows the number of files in '/usr' folder for different sizes of slack space and Figure 7(c) presents the total slack space available from files with slack space in a specific range. As can be seen, a total of about $67,000$ files have slack space in the 3,000–3,999 bytes range [Figure 7(b)], which adds up to a total slack space of about 240 MB [Figure 7(c)]. Similarly, a total of about $41,000$ files have slack space in the 2,000–2,999 bytes range [Figure 7(b)], which adds up to a total slack space of about 110 MB [Figure 7(c)].

4.3 Caveats

In this section we discuss the caveats of the *slackFS* basic model, motivating the need for the *slackFS* advanced model with fault-tolerance. With the basic model, one particular scenario wherein the hidden M_{vol}^{fs} could potentially be lost partially or completely is when a file on the T_{vol}^{fs}, whose slack space contains a portion of the M_{vol}^{fs} changes in size. Here again, there are two possible scenarios. If the file grows in size, then it will overwrite the hidden contents in the slack space. If the file shrinks in size, then the operating system can potentially de-allocate one or more blocks and mark it as free for other incoming disk space requests. Since both the scenarios are realistic on a production system, we have used a vanilla instance of the Ubuntu operating system to identify files that are less likely to change in size over long periods of time.

Another realistic operations scenario the hidden M_{vol}^{fs} could be lost is if the operating system on the target system uses a tool to overwrite the slack space of each and every file stored on the T_{vol}^{fs}. Although realistic, such a tool comes with a significant disk input/output overhead that would affect the performance of the system. However, if faced with a tool that achieves robust slack space sanitisation, no existing method of information hiding in slack space will be successful.

5 *slackFS* – advanced model

5.1 Adding fault tolerance for resilience

To address the caveats of the basic model presented in Section 4.3, we have incorporated fault-tolerance and designed the *slackFS* advanced model. The primary objective of the advanced model is to render *slackFS* robust to data loss resulting from modification or deletion of files on the T_{vol}^{fs}. Over time, files tend to change, some more than others, and hence data hidden the the slack space of those file can be lost partially or completely. As a first step to address this challenge, in the proposed *slackFS* framework, we focus on *system files* in the '/usr' folder that are the least likely to change.

Table 2 Number of unmodified files (in the '/usr' folder) over different time periods

Last modified (in days)	*Number of files (in '/usr')*
>1,000	99,874
>750	114,965
>500	118,682
>365	126,881
>250	131,139
>100	164,304
>50	210,714
>30	221,704

Table 2 presents data on the number of files that are unmodified within the '/usr' folder of an *ext4* filesystem on a production Ubuntu Server 20.04.6. As can be seen from the results, there are $114,965$ files that have not been modified in over 750 days (2+ years), $164,304$ files that have not been modified in over 100 days (3+ months), and $221,704$ files that have not been modified in over over 100 days. Therefore, there are hundreds of thousands of files on a target system that remain unmodified over long periods that the adversary can exploit using a tool like the proposed *slackFS*.

5.2 Erasure codes

To further augment the survivability of the hidden M_{vol}^{fs}, we have leveraged *erasure codes*, a forward error correction code, to build fault-tolerance into the *slackFS* framework. Erasure coding uses a set of algorithms to allow the recovery of missing data using a subset of original data (*How Erasure Coding is Configured in Object Storage*, 2014). *Erasure coding* works as follows: input data is split into multiple fragments, known as data fragments. Subsequently, it creates additional fragments, known as parity fragments, over the data fragments that can be used for data recovery in case of loss of or errors in the original data fragments. For illustration of how each parity fragment is created based on the original data fragments consider a $\mathcal{D} + p$ *erasure coding* scheme. Here, $\mathcal{D}$ is the number of data fragments an input message $\mathcal{M}$ is split into and p is the number of parity fragments generated. Such a coding scheme will have an overhead of $\frac{p}{\mathcal{D}}\%$. In the remainder of this paper, we shall consider parity the number of parity fragments p as a percentage of data fragments $\mathcal{D}$ denoted as $\mathcal{P} = \frac{p}{\mathcal{D}}\%$.

Reed-Solomon codes are used to detect and correct errors in the data introduced during transmission or on the storage devices (Reed and Solomon, 1960). Our prototype leverages *Reed-Solomon* implementation of *erasure code* for achieving fault-tolerance with *slackFS* framework with the following three popular coding libraries available from the open-source *liberasurecode* project (*Erasure Code API Library Written in C with Pluggable Erasure Code Backends*, 2014) – INTEL STORAGE ACCELERATION (ISA), JERASURE LIBERASURECODE.

Specifically, we have two different implementations of *Reed-Solomon* codes – VANDERMONDE matrix and CAUCHY matrix. In Table 3, ISA_RS_VAND, JERASURE_RS_VAND and LIBERASURECODE_RS_VAND are back-end implementations of the *Reed-Solomon* Encoding with VANDERMONDE matrix. ISA_RS_CAUCHY is a *Reed-Solomon* code implementation based on CAUCHY matrices over finite fields (MacWilliams and Sloane, 1977; Luby and Zuckermank, 1995; Wicker and Bhargava, 1999). All possible combinations of the back-ends and parity percentages ($\mathcal{P}$) used in our implementation are presented in Table 3.

Table 3 Table summarising the various back-ends and parity combinations implemented and tested

		Parity		
		12.50%	*25%*	*50%*
Back-end	ISA_L_RS_CAUCHY (i-rs-c)	CASE 1	CASE 2	CASE 3
	ISA_L_RS_VAND (i-rs-v)	CASE 4	CASE 5	CASE 6
	JERASURE_RS_VAND (j-rs-v)	CASE 7	CASE 8	CASE 9
	LIBERASURECODE_RS_VAND (l-rs-v)	CASE 10	CASE 11	CASE 12

Table 4 Average hiding and retrieval run time performance for 10 MB M_{vol}^{fs} for various combinations of $\mathcal{P} \in \{12.5\%, 25\%, 50\%\}$ and $\mathcal{T} \in \{16, 62, 64\}$

Back-end	*$\mathcal{P}$ = 12.5%*		*$\mathcal{P}$ = 25%*		*$\mathcal{P}$ = 50%*	
	Hide	*Retrieve*	*Hide*	*Retrieve*	*Hide*	*Retrieve*
			$\mathcal{T}$ = 16			
i-rs-c	2.6483	1.3797	2.6473	1.2444	2.7069	1.3348
i-rs-v	2.6096	1.2623	2.5588	1.2326	2.615	1.2778
j-rs-v	2.6218	1.2704	2.5408	1.1723	2.6028	1.3347
l-rs-v	2.5187	1.2181	2.5576	1.207	2.7028	1.3769
			$\mathcal{T}$ = 32			
i-rs-c	2.3469	0.5072	2.2962	0.511	2.3211	0.5031
i-rs-v	2.3447	0.5043	2.2961	0.5313	2.335	0.5389
j-rs-v	2.3281	0.5424	2.3166	0.5037	2.4064	0.5186
l-rs-v	2.3609	0.5072	2.334	0.5158	2.3527	0.5155
			$\mathcal{T}$ = 64			
i-rs-c	2.2678	0.4981	2.372	0.5264	2.3851	0.5874
i-rs-v	2.3696	0.5272	2.3621	0.5124	2.3889	0.5241
j-rs-v	2.3815	0.5107	2.2777	0.4891	2.4217	0.5656
l-rs-v	2.3414	0.5034	2.3508	0.5332	2.3367	0.5056

5.3 Results and analysis

In this section, we shall denote and refer to our prototype slackFS framework as a 4-tuple: $EC(\mathcal{B}, \mathcal{M}, \mathcal{P}, \mathcal{T})$ where:

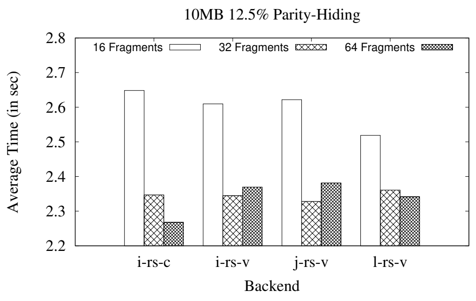
- $\mathcal{B} \in$ {i-rs-c, i-rs-v, j-rs-v, l-rs-v}
- $\mathcal{M} \in \{10, 25, 50\}$ MB
- $\mathcal{P} \in \{12.5, 25, 50\}\%$
- $\mathcal{T} \in \{16, 32, 64\}$ fragments

Due to paper length restrictions and in appreciation of the reader's time, we specifically discuss results for $\mathcal{P} = 25\%$ and $\mathcal{T} = 32$, i.e., $EC(\mathcal{B}, \mathcal{M}, 25, 32)$. We chose these two values as they provide a good mid-point for evaluating the performance of our prototype implementation. We have evaluated the performance for all 12 CASES (Table 3) over 50 trial runs. Average performance of $EC(\mathcal{B}, \mathcal{M}, \mathcal{P}, \mathcal{T})$ in terms of hiding and retrieval times of all 12 CASES is summarised in Tables 4, 5 and 6, which are plotted in Figures 8, 9 and 10 respectively. Furthermore, we are evaluating the prototype *slackFS* framework with empty malicious filesystem volumes. This gives the baseline performance for hiding and retrieval times. As the adversary adds or removes files from the the M_{vol}^{fs} volume, the hiding and retrieval times will change accordingly.

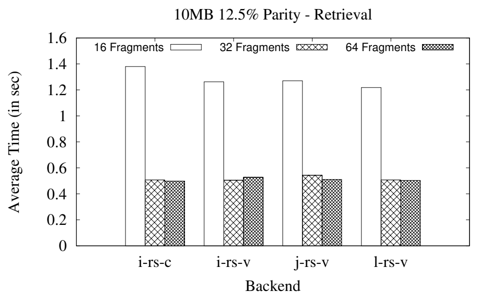
From these results, it can be seen that while the hiding and retrieval times have similar trends, each CASE (see Table 3) has a unique performance. From Figure 9, it can be seen that EC(i-rs-c, $25, 50, 32$) [Figure 9(e)] achieves the fastest 50-run average hiding time. Similarly, EC(i-rs-v, $25, 25, 64$) achieves the fastest 50-run average retrieval time. In Figure 9(c) it can be seen that with $\mathcal{P} = 25\%$, the hiding time increases by over 3% from $\mathcal{D} = 32$ to $\mathcal{D} = 64$, which is the most significant for all cases with 25MB volume.

Other important observations are as follows. The retrieval times for $EC(\mathcal{B}, 10, \mathcal{P}, 16)$ [Figure 8(b)] is approximately 2.5 times longer than the retrieval times for $EC(\mathcal{B}, 10, \mathcal{P}, 32)$ [Figure 8(d)] and $EC(\mathcal{B}, 10, \mathcal{P}, 64)$ [Figure 8(f)]. Similarly, the retrieval times for $EC(\mathcal{B}, 25, \mathcal{P}, 16)$ is over 4 times longer than the retrieval times for $EC(\mathcal{B}, 25, \mathcal{P}, 32)$ and $EC(\mathcal{B}, 25, \mathcal{P}, 64)$. Finally, the retrieval times for $EC(\mathcal{B}, 50, \mathcal{P}, 16)$ is approximately 10 times longer than the retrieval times for $EC(\mathcal{B}, 50, \mathcal{P}, 32)$ and $EC(\mathcal{B}, 50, \mathcal{P}, 64)$. Based on these observations, it can be concluded that for all four back-ends and across all three parity values, as the total number of fragments increases from 16 to 32 or 64, the time to both hide and retrieve dramatically decreases. However, the time difference is not significant going from 32 to 64 fragments, especially for retrieval.

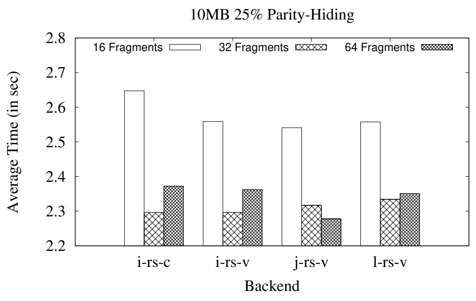
Figure 8 Average hiding and retrieval run times for 10 MB M_{vol}^{fs} for various combinations of $\mathcal{P}$ and $\mathcal{T}$, (a) hiding ($\mathcal{P} = 12.5\%$) (b) retrieval ($\mathcal{P} = 12.5\%$) (c) hiding ($\mathcal{P} = 25\%$) (d) retrieval ($\mathcal{P} = 25\%$) (e) hiding ($\mathcal{P} = 50\%$) (f) retrieval ($\mathcal{P} = 50\%$)



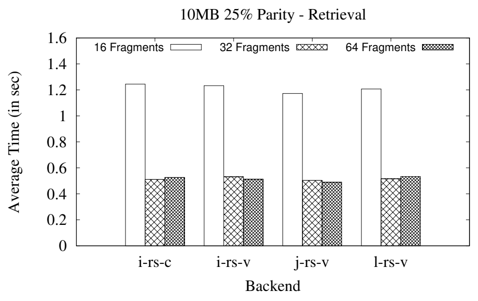
(a)



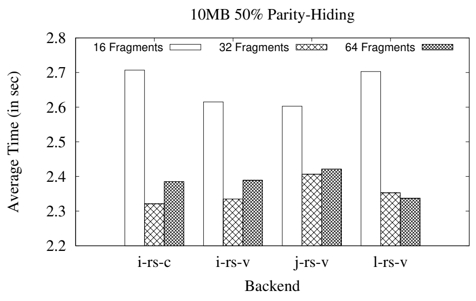
(b)



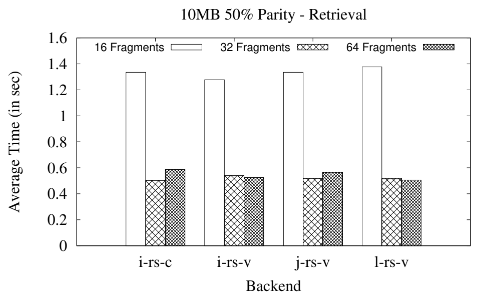
(c)



(d)



(e)



(f)

Table 5 Average hiding and retrieval run time performance for 25 MB M_{vol}^{fs} for various combinations of $\mathcal{P} \in \{12.5\%, 25\%, 50\%\}$ and $\mathcal{T} \in \{16, 62, 64\}$

Back-end	$\mathcal{P}$ = *12.5%*		$\mathcal{P}$ = *25%*		$\mathcal{P}$ = *50%*	
	Hide	*Retrieve*	*Hide*	*Retrieve*	*Hide*	*Retrieve*
			$\mathcal{T}$ = *16*			
i-rs-c	5.518	2.2906	5.5905	2.4383	5.6778	2.3692
i-rs-v	5.6318	2.3236	5.5843	2.2814	5.6387	2.4193
j-rs-v	5.7548	2.4256	5.6828	2.569	5.7632	2.5927
l-rs-v	5.5384	2.3043	5.5371	2.3192	5.605	2.44
			$\mathcal{T}$ = *32*			
i-rs-c	5.3366	0.4094	5.3261	0.4068	5.2209	0.4243
i-rs-v	5.3688	0.4093	5.2646	0.4136	5.326	0.4099
j-rs-v	5.398	0.4338	5.2461	0.4077	5.3229	0.4181
l-rs-v	5.3477	0.4098	5.276	0.4131	5.3277	0.414
			$\mathcal{T}$ = *64*			
i-rs-c	5.3272	0.4083	5.317	0.4146	5.317	0.4102
i-rs-v	5.2871	0.4063	5.3097	0.406	5.4058	0.4099
j-rs-v	5.2899	0.4197	5.4175	0.416	5.3653	0.4179
l-rs-v	5.3504	0.4139	5.3119	0.4105	5.3553	0.4125

Figure 9 Average hiding and retrieval run times for 25 MB M_{vol}^{fs} for various combinations of $\mathcal{P}$ and $\mathcal{T}$, (a) hiding ($\mathcal{P} = 12.5\%$) (b) retrieval ($\mathcal{P} = 12.5\%$) (c) hiding ($\mathcal{P} = 25\%$) (d) retrieval ($\mathcal{P} = 25\%$) (e) hiding ($\mathcal{P} = 50\%$) (f) retrieval ($\mathcal{P} = 50\%$)

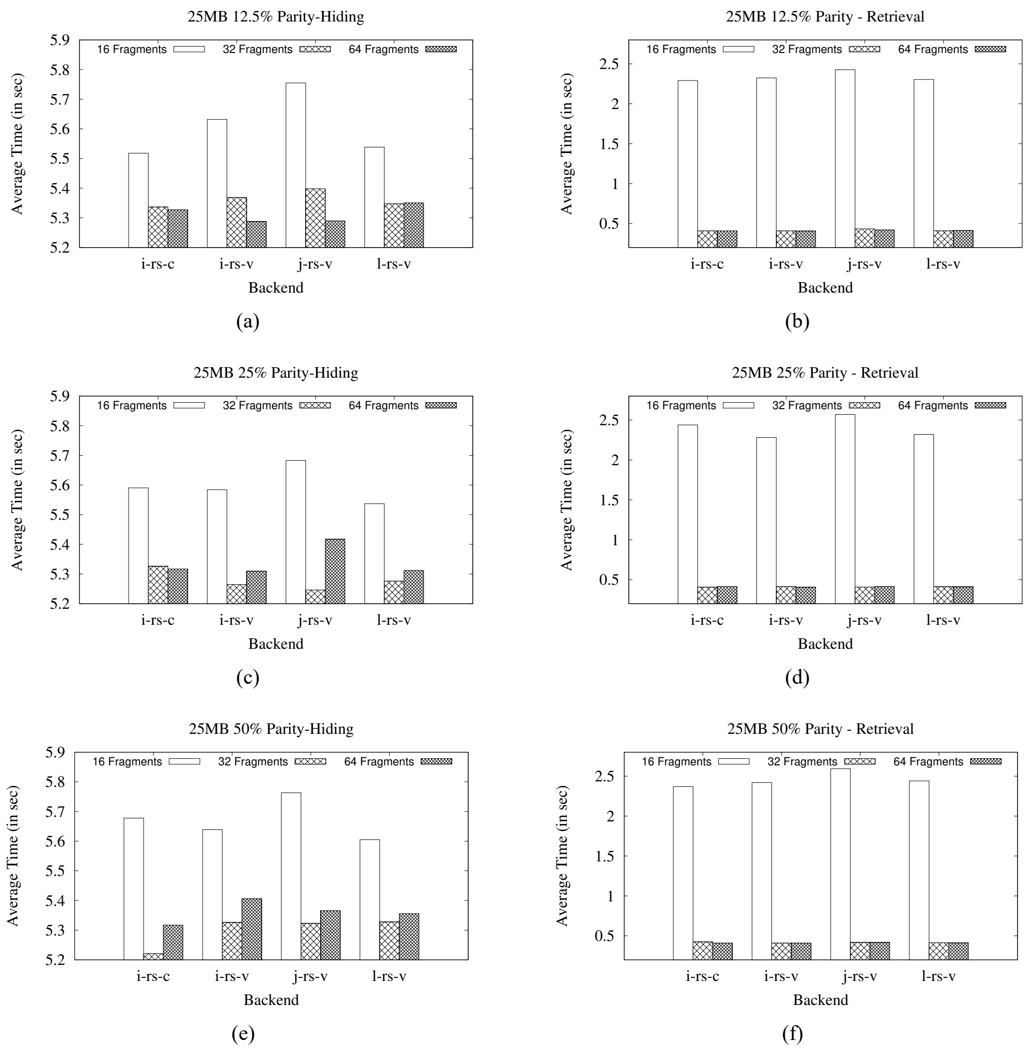
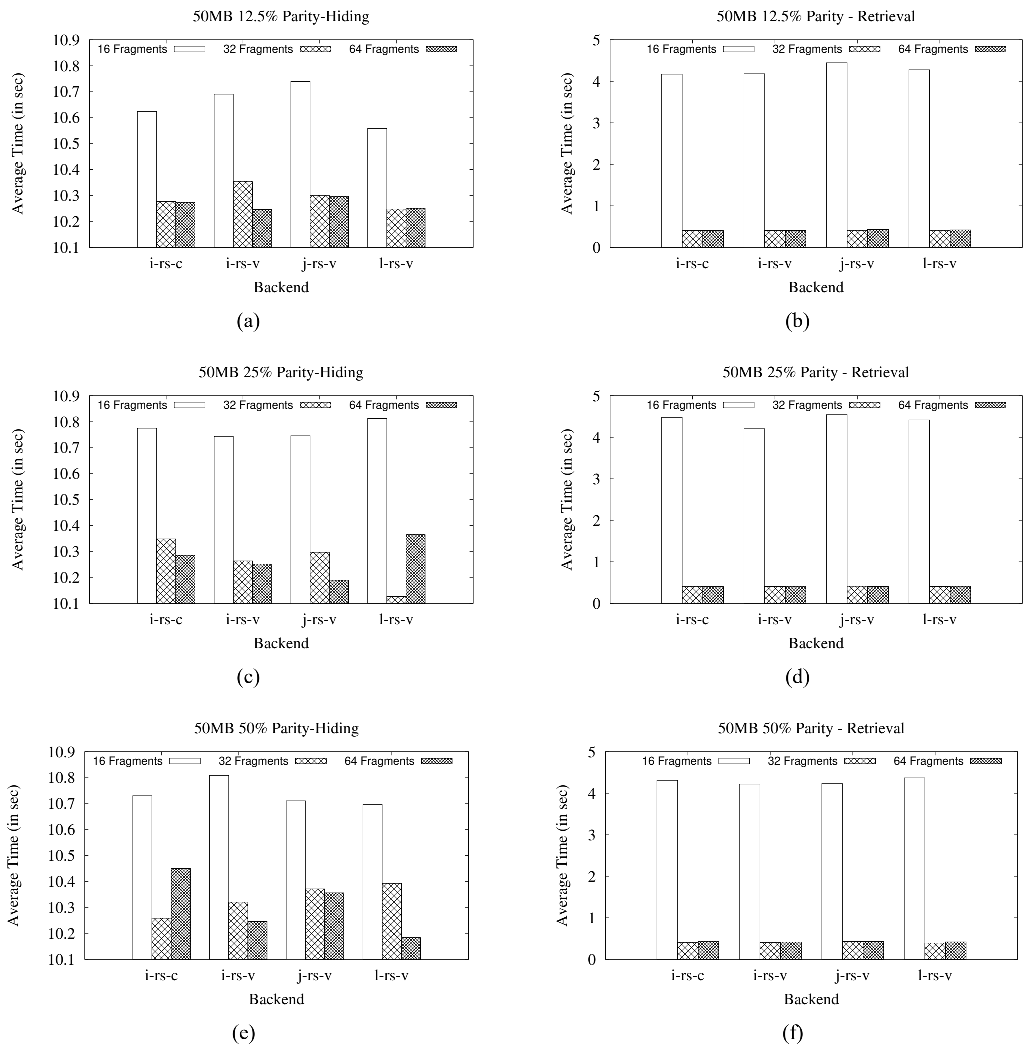


Table 6 Average hiding and retrieval run time performance for 50 MB M_{vol}^{fs} for various combinations of $\mathcal{P} \in \{12.5\%, 25\%, 50\%\}$ and $\mathcal{T} \in \{16, 62, 64\}$

Back-end	*$\mathcal{P}$ = 12.5%*		*$\mathcal{P}$ = 25%*		*$\mathcal{P}$ = 50%*	
	Hide	*Retrieve*	*Hide*	*Retrieve*	*Hide*	*Retrieve*
			$\mathcal{T}$ = 16			
i-rs-c	10.6234	4.173	10.775	4.4786	10.7306	4.3104
i-rs-v	10.6906	4.1817	10.7435	4.2069	10.8087	4.2223
j-rs-v	10.7393	4.447	10.7458	4.5448	10.7108	4.2323
l-rs-v	10.5581	4.2785	10.8124	4.4168	10.6965	4.3703
			$\mathcal{T}$ = 32			
i-rs-c	10.2768	0.4068	10.3482	0.4085	10.2586	0.4085
i-rs-v	10.3527	0.4072	10.2634	0.4058	10.3205	0.3997
j-rs-v	10.3006	0.4005	10.2971	0.4119	10.3713	0.4259
l-rs-v	10.2477	0.41	10.1259	0.4051	10.3929	0.3917
			$\mathcal{T}$ = 64			
i-rs-c	10.2716	0.4028	10.2854	0.4019	10.4498	0.4212
i-rs-v	10.2462	0.4057	10.2513	0.4095	10.2454	0.4154
j-rs-v	10.295	0.4276	10.1894	0.4052	10.3562	0.4298
l-rs-v	10.2508	0.4192	10.3642	0.4105	10.183	0.4175

Figure 10 Average hiding and retrieval run time performance for 50 MB M_{vol}^{fs} for various combinations of $\mathcal{P}$ and $\mathcal{T}$, (a) hiding ($\mathcal{P} = 12.5\%$) (b) retrieval ($\mathcal{P} = 12.5\%$) (c) hiding ($\mathcal{P} = 25\%$) (d) retrieval ($\mathcal{P} = 25\%$) (e) hiding ($\mathcal{P} = 50\%$) (f) retrieval ($\mathcal{P} = 50\%$)



Our proposed method has very robust detection resistance as it introduces a strong asymmetry between the attacker and the target. We would like to restate that the M_{vol}^{fs} is completely hidden by splitting it up into chunks and spreading it across the T_{vol}^{fs} in the slack space of multiple files. Since the slack space is not accessible to the operating system or to the administrator without special tools, it is extremely hard, if not impossible, to even detect the presence of the M_{vol}^{fs}. In the rare circumstance where it is detected, even then its recovery by a third party tool is impractical due to how it is hidden in a distributed manner. To further strengthen the asymmetry in favor of the adversary, simply regrouping the distributed chunks of M_{vol}^{fs} using the map_file and mounting it on the T_{vol}^{fs} provides ready access to all hidden data. Furthermore, since the slack space is considered as part of the allocated disk space, security tools that scan for deviation in disk space utilisation or modifications to file date and time stamps will not be able to detect information hidden with *slackFS* framework.

6 Conclusions and future work

In this paper, we introduce *slackFS*, a pioneering steganographic method that conceals entire filesystems within the slack space of another filesystem. Unlike previous approaches focusing on individual file concealment, *slackFS* leverages the entirety of slack space, offering robust detection resistance against standard security tools. Since slack space persists across system reboots and restarts, *slackFS* ensures persistent data hiding, enhancing its effectiveness for covert operations. Moreover, the ability to hide and un-hide the filesystem at runtime provides a structured means for adversaries to extract data clandestinely. The *slackFS* framework incorporates two models: a basic version without fault tolerance and an advanced model with fault tolerance. The advanced model facilitates recovery from accidental cover file modifications or deletions, ensuring the reliability of the hidden malicious filesystem. Extensive testing of the prototype validates its robustness and computational overhead, with and without fault tolerance. Additionally, our reconnaissance scan of target filesystem volumes reveals significant slack space availability, highlighting *slackFS*'s potential for efficient information hiding. Furthermore, the framework exhibits characteristics of a robust steganographic technique, laying the foundation for future research into expanding *slackFS* and exploring further avenues for covert data concealment within target systems.

References

Alji, M. and Chougdali, K. (2021) 'File slack handling tool', *2021 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, pp.1–3.

Berghel, H., Hoelzer, D. and Sthultz, M. (2006) 'Data hiding tactics for windows and unix file systems', *Advances in Computers*, Vol. 74, pp.1–17, http://www.berghel.net/publications/data_hiding/data_hiding.php.

Bovet, D.P. and Cesati, M. (2005) *Understanding the Linux Kernel*, 3rd ed., 208 Incremental Checkpointing for Grids.

Cameron, L.M. (2016) *With Cryptography Easier to Detect, Cybercriminals Now Hide Malware in Plain Sight. Call it Steganography. Here's How It Works*.

Carrier, B. (2005) *File System Forensic Analysis*, Addison-Wesley Professional.

Erasure Code API Library Written in C with Pluggable Erasure Code Backends (2014) [online] https://github.com/openstack/liberasurecode (accessed 25 June 2023).

Evsutin, O., Melman, A. and Meshcheryakov, R. (2020) 'Digital steganography and watermarking for digital images: a review of current research directions', *IEEE Access*, Vol. 8, pp.166589–166611.

Ewane, P. (2017) *MacSpy: OS X Mac RAT as a Service*, June [online] https://cybersecurity.att.com/blogs/labs-research/macspy-os-x-rat-as-a-service (accessed 7 June 2022).

Göbel, T. and Baier, H. (2018) 'Anti-forensics in ext4: on secrecy and usability of timestamp-based data hiding', *Digital Investigation*, Vol. 24, pp.S111–S120.

Göbel, T. and Baier, H. (2019) 'Fishy – a framework for implementing filesystem-based data hiding techniques', *Digital Forensics and Cyber Crime: 10th International EAI Conference, ICDF2C 2018, Proceedings*, 10–12 September, Springer, New Orleans, LA, USA, pp.23–42.

Göbel, T., Türr, J. and Baier, H. (2019) 'Revisiting data hiding techniques for apple file system', *Proceedings of the 14th International Conference on Availability, Reliability and Security*, pp.1–10.

Hassan, N.A. and Hijazi, R. (2017) 'Chapter 6 – Data hiding forensics', in Hassan, N.A. and Hijazi, R. (Eds.): *Data Hiding Techniques in Windows OS*, pp.207–265, Syngress, Boston, https://doi.org/10.1016/978-0-12-804449-0.00006-3.

How Erasure Coding is Configured in Object Storage (2014) [online] https://docs.openio.io/latest/source/admin-guide/configuration_ec.html (accessed 25 June 2023).

Huebner, E., Bem, D. and Wee, C.K. (2006) 'Data hiding in the NTFS file system', *Digital Investigation*, Vol. 3, No. 4, pp.211–226, https://doi.org/10.1016/j.diin.2006.10.005.

Koolhaas, A. and van Steenbergen, W. (2020) 'APFS slack analysis and detection of hidden data', *Security and Network Engineering*, pp.2019–2020.

Luby, M. and Zuckermank, D. (1995) *An XOR-Based Erasure-Resilient Coding Scheme*, Tech report.

MacWilliams, F.J. and Sloane, N.J.A. (1977) *The Theory of Error-Correcting Codes*, 1st ed., 1 January, Vol. 16, Elsevier.

Metcheka, L.M. and Ndoundam, R. (2020) 'Distributed data hiding in multi-cloud storage environment', *Journal of Cloud Computing*, Vol. 9, No. 1, p.68.

Microsoft (2020) *Trojan:win32/okrum!msr Threat Description* [online] https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:Win32/Okrum!MSR&ThreatID=2147755899 (accessed 7 June 2022).

MITRE T1564.001 (2020) *Hide Artifacts: Hidden Files and Directories, Sub-Technique T1564.001 – Enterprise*, March [online] https://attack.mitre.org/techniques/T1564/001/ (accessed 7 June 2022).

MITRE T1564.005 (2020) *Hide Artifacts: Hidden File System, Sub-Technique T1564.005 – Enterprise*, June [online] https://attack.mitre.org/techniques/T1564/005/ (accessed 7 June 2022).

Mulazzani, M., Neuner, S., Kieseberg, P., Huber, M., Schrittwieser, S. and Weippl, E. (2013a) 'Quantifying windows file slack size and stability', *Advances in Digital Forensics IX: 9th IFIP WG 11.9 International Conference on Digital Forensics, Revised Selected Papers*, 28–30 January, Springer, Orlando, FL, USA, pp.183–193.

Mulazzani, M., Neuner, S., Kieseberg, P., Huber, M., Schrittwieser, S. and Weippl, E. (2013b) 'Quantifying windows file slack size and stability', in Peterson, G. and Shenoi, S. (Eds.): *Advances in Digital Forensics IX*, pp.183–193, Springer Berlin Heidelberg, Berlin, Heidelberg.

Rabin, M.O. (1989) 'Efficient dispersal of information for security, load balancing, and fault tolerance', *J. ACM*, Vol. 36, No. 2, p.335–348.

Rasmi, A., Arunkumar, B. and Anees, V.M. (2019) 'A comprehensive review of digital data hiding techniques', *Pattern Recognition and Image Analysis*, Vol. 29, pp.639–646.

Reed, I.S. and Solomon, G. (1960) 'Polynomial codes over certain finite fields', *Journal of the Society for Industrial and Applied Mathematics*, Vol. 8, No. 2, pp.300–304.

Shamir, A. (1979) 'How to share a secret', *Commun. ACM*, Vol. 22, No. 11, p.612–613.

Srinivasan, A. and Dong, H. (2018) 'SURE-FIT – secure and adaptive framework for information hiding with fault-tolerance', *Journal of Cyber Security and Mobility*, Vol. 6, pp.427–456, https://doi.org/10.13052/jcsm2245-1439.643.

Srinivasan, A., Dong, H. and Stavrou, A. (2017) 'Frost: anti-forensics digital-dead-drop information hiding robust to detection & data loss with fault tolerance', *Proceedings of the 12th International Conference on Availability, Reliability and Security, ARES '17*, Association for Computing Machinery, New York, NY, USA.

Srinivasan, A., Nazaraj, S.T. and Stavrou, A. (2013) 'HIDEINSIDE – a novel randomized & encrypted antiforensic information hiding', *2013 International Conference on Computing, Networking and Communications (ICNC)*, pp.626–631, https://doi.org/10.1109/ICCNC.2013.6504159.

Thampy, R.V., Praveen, K. and Mohan, A.K. (2018) 'Data hiding in slack space revisited', *International Journal of Pure and Applied Mathematics*, Vol. 118, No. 18, pp.3017–3025.

Wicker, S.B. and Bhargava, V.K. (1999) *Reed-Solomon Codes and Their Applications*, John Wiley & Sons.

Wikipedia (2022) *Block Sub-Allocation*, December [online] https://en.wikipedia.org/w/index.php?title= Blocksuballocation&oldid=1127107338 (accessed 7 June 2022).

Wirzenius, L., Oja, J., Stafford, S. and Weeks, A. (1991) *The Linux System Administrator's Guide – Filesystems* [online] https://tldp.org/LDP/sag/html/filesystems.html (accessed 25 June 2023).