UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ARTIFICE: A DESIGN FOR USABLE DENIABLE STORAGE INFORMED BY ADVERSARY THREAT

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Austen Thomas Barker

June 2022

The Dissertation of Austen Thomas Barker
is approved:

Professor Darrell D. E. Long, Chair

Professor Ethan L. Miller

Doctor William Semancik

Doctor Ike Nassi

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures **v**

List of Tables **ix**

Abstract **x**

Dedication **xii**

Acknowledgments **xiii**

Glossary **xv**

1 Introduction **1**

2 Background **8**
- 2.1 A Survey of Steganographic File Systems 9
- 2.2 Snapshot and Traffic Analysis attacks 12
- 2.3 Flash Storage and Mobile Devices 13
- 2.4 Secure Deletion 16
- 2.5 Summary 18

3 Threat Model **20**
- 3.1 Shortcomings of Previous Models 20
- 3.2 Our Threat Model 22
  - 3.2.1 An Example Use Scenario 22
  - 3.2.2 Malware and Information Leakage 23
  - 3.2.3 Suspicion, Escalation, and Coercion 25
  - 3.2.4 The Attack Surface 26
  - 3.2.5 The Challenge of Multi-leveled Deniability 28
- 3.3 Possible attacks under our threat model 29
- 3.4 Related Work 33
  - 3.4.1 Prior Threat Models 33

3.4.2 Existing Attacks 36
3.5 Summary 37

4 The Artifice Deniable Storage System 39
4.1 Design requirements for a Deniable Storage System 39
4.2 Artifice System design 41
4.2.1 Artifice Map 43
4.2.2 Information Leakage, Malware, and Hiding Driver Software 47
4.2.3 Obfuscation and Redundancy through Information Dispersal Algorithms 49
4.2.4 Deniable Writes to Public Free Space 58
4.2.5 Flash Considerations 61
4.3 Secure Deletion and Steganographic Storage 64
4.3.1 Steganographic Storage through a Secure Delete File System 65
4.4 Summary 68

5 Implementation & Operational Security 70
5.1 Implementation 70
5.2 Operational Security Model 72
5.3 Performance 78

6 Survivability Analysis 81
6.1 Theoretical Evaluation 82
6.2 Experimental Evaluation 89
6.3 Mean Time to Data Loss 95
6.4 Summary 103

7 Snapshot & Statistical Analysis Attacks 105
7.1 A Multiple Snapshot Attack Framework 106
7.2 Data Collection & Experiment Methodology 112
7.3 Snapshot Attack Mitigation 116
7.4 Snapshot Attack Results 118
7.5 Entropy Analysis 120
7.6 Dealing with FTLs and TRIM 126
7.7 Summary 129

8 Conclusion and Future Work 131
8.1 Future Work 133

Bibliography 136

List of Figures

2.1 Anderson *et al.*'s second proposed scheme for a steganographic file system. 9

4.1 System overview of Artifice. The Artifice kernel module resides in a separate operating system contained on removable media. The public system includes the public file system that Artifice hides in and the public OS. Free space in the public file system should be filled with pseudo-random blocks. 42

4.2 The design of the Artifice Map. 43

4.3 The process of locating Artifice super blocks through chain hashing a user's passphrase. 45

4.4 Metadata overhead as a percentage of total Artifice available space when implemented as a virtual block device. Metadata overhead is only dependent on the total number of carrier blocks. 46

4.5 Share generation with AONT-RS (All or Nothing Transform and Reed-Solomon), Shamir Secret Sharing, and Reed-Solomon/Entropy. 51

5.1 Architecture of the Artifice block device driver. 71

6.1 Probability of survival for Artifice metadata in a variety of configurations using both Reed-Solomon/Entropy (RS), Shamir Secret Sharing (SSS) and AONT-RS. Probabilities are calculated assuming 512 GB of free space, 5 GB written between repair cycles, 5 GB Artifice volume, and 365 repair cycles. k is the reconstruction threshold in carrier blocks. 84
6.2 Probability of survival for an entire Artifice volume with the same configuration as Figure 6.1 85
6.3 Probability of survival with varying Artifice volume sizes ranging from 256 MB to 4 GB. 5 GB of writes between repair operations and 512 GB of unallocated space. k is the reconstruction threshold in carrier blocks. 86
6.4 Probability of survival with varying sizes of writes between Artifice invocations from 256 MB to 8 GB. 512 GB of unallocated space and a 5 GB Artifice volume. 87
6.5 Probability of survival with varying sizes of unallocated spaces from 64 GB to 512 GB. 5 GB of writes between repair operations and a 5 GB Artifice volume. 88
6.6 Amount of free space on our SSD at each snapshot. 89
6.7 Amount of data written to the SSD between each pair of snapshots. 90
6.8 Amount of data written to newly allocated regions of the disk between each pair of snapshots. 91
6.9 Number of redundant carrier blocks per data block tuple versus the average number of data blocks lost with a 2 GB Artifice instance for Reed-Solomon/Entropy (RS), AONT-RS (AONT), Shamir Secret Sharing (SSS), and basic replication. 92
6.10 Number of redundant carrier blocks per data block tuple versus the average number of carrier blocks overwritten with a 2 GB Artifice instance for a variety of IDAs. 93
6.11 Generalized Markov model with no repair rate. 95
6.12 $(5, 3)$ Markov model with no repair rate. 97

6.13 Generalized Markov model with repair rate v. 98
6.14 $(5, 3)$ Markov model with a repair rate of v. 99
6.15 MTTDL in hours for Artifice data blocks using Shamir Secret Sharing (SSS) with different reconstruction thresholds measured in carrier blocks (k) and basic replication with no repair rate ($v = 0$). . 102
6.16 MTTDL in hours for Artifice data blocks using Shamir Secret Sharing (SSS) with different reconstruction thresholds measured in carrier blocks (k) and basic replication with a repair rate of $v = 1/24$ (once a day). 103

7.1 Theoretical probability of consecutive changes when changes are made uniformly. The probability of c consecutive changes degrades very quickly, especially when the free space is large. 108
7.2 Empirical probability of consecutive changes for 52 change records where the changes are made by an `ext4` file system. Note the occasional spikes in probability found in the long tail due to occasional movements or downloads of unusually large pieces of data. 109
7.3 Hidden volumes over 0.75 GB in size are always identified successfully. 115
7.4 Entropy values per block for a variety of IDAs and AES. Note that the entropy value for each block is around 7.95 with some level of statistical noise and no easily visible outliers that would denote a flaw in one of the systems. 122
7.5 Right: Approximated probability distributions for the entropy of data encoded with a variety of IDAs and AES. Left: Probability distribution for Reed-Solomon/Entropy with compressed data as an entropy source, note the long tail indicative of lower entropy values. 124
7.6 Quantile-Quantile plot showing the relationship between AES entropy values and a Gaussian distribution. 125

7.7 120 GB SSD with 68 GB of free space after a TRIM operation. Black correspond to in use logical blocks, gray pixels are blocks that return only zeroes due to TRIM. 126

7.8 Region of free space on a 120 GB SSD with a 512 MB Artifice volume written. Note the random distribution of written blocks in otherwise unused and zeroed free space. 127

List of Tables

2.1 Summary of Previous Deniable Storage Systems 19

5.1 Possible scenarios and remedies for the compromise of different Artifice components. 75

5.2 Performance comparison of public volume performance with Artifice configured with AONT-RS and with Shamir Secret Sharing (SSS). 80

6.1 Probability of overwrite for Shamir Secret Sharing (SSS), All or Nothing Transform (AONT-RS), Reed-Solomon/Entropy (RS), and Replication with different reconstruction thresholds in carrier blocks (k). 101

7.1 Numeric results from the classifier averaged over 100 runs. Furthermore, 95% confidence intervals for all figures vary only in the thousandths. 116

Abstract

Artifice: A Design for Usable Deniable Storage Informed by Adversary Threat

by

Austen Thomas Barker

With the widespread adoption of disk encryption technologies, it has become common for adversaries to employ coercive tactics to force users to surrender encryption keys and other access credentials. For some users, this creates a need for data storage that provides plausible deniability: the ability to deny the existence of sensitive information to avoid coercive tactics that put their safety at risk. Plausibly deniable storage would benefit groups such as human rights advocates relaying sensitive information, journalists covering human rights stories in a war zone, or NGO workers hiding food shipment schedules from militias.

Most previous systems rely on some form of steganography to conceal sensitive information among innocuous-appearing data on a user's storage device. To accomplish this, they often utilize the unallocated space on a disk to conceal a plausibly deniable hidden volume. Previous approaches all exhibit major design weaknesses stemming from flawed assumptions in their design, like the assumption that the presence of the driver software used to run a deniable volume would not be suspicious to an adversary. The state of the art also does not present solutions to malware installed by the adversary, and does not explore operational characteristics of their systems. Generally, there is a lack of experimental evaluation and available implementations. As a result of these flawed assumptions and other shortcomings, previous deniable storage systems only offer pieces of an implementable and usable solution.

We have developed a new threat model for plausibly deniable storage, designed a system to counter the adversary described in the threat model, and experimentally evaluated both our design and long-held assumptions integral to previous systems. We have designed and implemented Artifice, a deniable storage system that allows us to evaluate our hypotheses. With Artifice, hidden data blocks are split with an information dispersal algorithm to produce a set of obfuscated carrier blocks that are indistinguishable from other pseudo-random blocks on the disk. The blocks are then stored in unallocated space, possess a self-repair capability, and rely on combinatorial security. We have evaluated the reliability and effectiveness of this approach in protecting the integrity of a hidden volume through theoretical models and empirical evaluation.

Unlike existing and proposed systems, Artifice addresses problems regarding flash storage devices and multiple snapshot attacks through comparatively simple block allocation schemes and operational security. To hide the user's ability to run a deniable system and prevent information leakage, Artifice stores its driver software separately from the hidden data. We have also designed and implemented the first multiple snapshot attack against a deniable storage system. This attack has been shown to classify the existence of an Artifice volume on a disk under certain circumstances and we used these results to provide recommendations for how a user can deniably modify their device's characteristics to mitigate the effectiveness of the attack.

This dissertation is dedicated to my parents and family for all their support, and Natalie, without whose patience, love, and support this work would not have been possible.

Acknowledgments

I would like to thank Professor Long, my PhD advisor, for his mentor-ship, guidance, and contributions to this work.

I would also like to thank Professor Miller, Dr. Ike Nassi and Dr. William Semancik for serving on my qualifying exam and dissertation committees and for working with me on editing and fine tuning this dissertation.

I'd like to thank my fiancée Natalie for her support. Without her I would not have finished this work.

My parents, sister, and extended family who have listened to me rant endlessly about myopic details pertaining to this work have my gratitude. I'd like to thank them for nodding along and humoring these ramblings.

My thanks to Thomas Schwarz and Jehan-François Pâris for their assistance in developing the survivability analysis chapter and James Hughes for his assistance with the threat and operational security models.

Yash Gupta, Eugene Chou, and many other graduate students and undergraduates have contributed to this project's repositories over the years. Without these individuals Artifice's prototype implementation would not have taken form as well as it has. I'd also like to thank Sabrina Au for repeatedly reviewing and helping to craft the papers that published much of the content contained within this dissertation.

Kyle Fredrickson was instrumental in developing and implementing a multiple snapshot attack that has formed the core of the Artifice's statistical analysis evaluation.

Lastly I'd like to thank the NSF and the Storage Systems Research Center for providing the funding and a work space needed to complete this work. This work would not have been possible without the collaboration of SSRC members and

long hours spent in front of a lab white board.

Several members of the Storage Systems Research Center have contributed to this work. This dissertation includes work sourced or derived from the following previously published papers.

- Kyle Fredrickson, Austen Barker, Darrell D. E. Long. "A Multiple Snapshot Attack on Deniable Storage Systems." *Proceedings of the 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (MASCOTS 2021), November 2021. With permission from Kyle Fredrickson. Professor Long directed and supervised the work published in this paper. I performed the literature review, data collection, implementation, and drafting of the paper with assistance from Kyle on the theoretical construction of the attack, implementing the program that carries out the attack, and processing experimental results.

Glossary

device mapper The Linux kernel system used to create virtualized block devices on top of other virtual or physical storage devices to modify their behavior in some way. Device mappers are commonly used to implement software RAID or disk encryption. 10

entropy In the field of information theory entropy is the average level of uncertainty in a random variable's possible outcomes. A high entropy stream of data will be more random than one with low entropy. 37, 43

firmware A piece of software, generally small, that is often stored on a small chip to control a piece of hardware. An example of firmware is a computer's BIOS. 14, 47

FTL A flash translation layer is a layer of indirection used with an SSD to make the otherwise unique architecture of flash memory look similar to a mechanical disk. The FTL maps requests made to the drive from a logical block address to physical pages of flash memory. 13, 61

FUSE A kernel module and library used to allow the implementation of file systems as user-level programs. 47

IDA An algorithm used to split a piece of information, sometimes called a secret, into multiple shares (or shards) that can be reconstructed later to recover the original piece of information. IDAs can also provide the ability to generate more shares than needed for reconstruction which provides redundancy, or obfuscate the data to prevent an adversary from learning a piece of the secret. 4, 41, 82, 105, 132

log-structured Referring to an append-only method of writing data to a device, akin to a log. Often used to describe storage devices and file systems that use append-only writes. 14, 32, 63, 135

MLC A type of flash memory that stores multiple bits per cell. Examples include TLC (three bits per cell) and QLC (four bits per cell). 15

MTD A type of raw flash device that leaves the tasks of wear leveling and garbage collection up to the file system instead of an FTL. 14

MTTDL The average length of time a system will operate until component failures result in irrecoverably losing data. 95

multiple snapshot attack Comparing two snapshots of a disk, taken at different points in time, to obtain a list of changes to the disk. These changes are then analyzed to look for abnormalities that would indicate that hidden data was written to the disk. 11, 34, 42, 58, 105, 132

ORAM A technique that obfuscates access patterns to a device, often RAM. With ORAM the access distributions of two programs are indistinguishable from one another. This is applied to deniable storage to obfuscate whether an access is to a hidden or public volume. 12, 30, 58

plausible deniability The user's ability to plausibly deny knowledge of something. In the case of deniable storage systems they would be denying the existence of hidden data. 1, 8

pseudo-random Referring to a number or piece of data that satisfies statistical tests for randomness produced that are produced through mathematical means (a pseudorandom number generator). 6, 14, 41, 124, 125, 130

rootkit A piece of self-hiding malicious software used by an adversary to provide administrator or root level access to a system. 23, 47

rubber hose attack Obtaining a cryptographic key or password from a person through coercion. Often used as a euphemism for torture. 1

SIMD A Single-Instruction-Multiple-Data is a method of computation that allows an operation to be applied to multiple pieces of data with a single processor instruction. 72

SLC A type of flash memory that stores a single bit per flash cell. 15

SSD A storage device that utilizes flash memory. As opposed to a spinning magnetic disk found in a conventional hard disk. 7, 13, 61, 88

steganography The practice of hiding a piece of secret information in something that is not secret. 2, 8, 117

TOR An onion routing network used to provide anonymous communication on the internet. 27

TPM A security focused micro-controller used to secure hardware through integrated cryptographic features and provide certain functionality to applications and operating systems. 18

TRIM A command that allows a file system or operating system to inform an SSD that a given block is no longer in use and can be wiped by the device for future use. 6, 30, 61, 88, 134

Chapter 1

Introduction

As the use of strong encryption for personal data storage becomes more common, it becomes more difficult for those who wish to control the free flow of information to monitor and restrict a user's actions. This prompts regimes and other organizations in opposition to the widespread use of encryption to utilize more unconventional tactics. As a result, in some situations, the possession of an encrypted file or disk can expose a user to coercive cryptanalysis tactics that can vary from threats of legal penalties [123] up to physical violence as in the case of a *rubber hose attack* [124]. With such a threat to the user's safety and well-being while carrying sensitive information, it becomes necessary to not only protect the confidentiality of the data but also provide *plausible deniability*, the ability to deny that the sensitive data even exists.

The lack of plausibly deniable storage can encourage individuals to resort to extreme methods to exfiltrate data from dangerous or restricted environments. For example, in 2011 a Syrian engineer hid a micro SD card in an arm wound to exfiltrate information about atrocities in Hama, Syria [79]. It is also becoming increasingly common for nations and law enforcement agencies to legally require disclosure of encryption keys [123] under a variety of circumstances. Developing tools that enable secure and private information storage are *vital* for journalists, NGOs, peaceful dissent, and the free dissemination of information.

This challenge can be met through the use of *steganography*: hiding sensitive information among innocuous non-secret data. For instance, an example of digital steganography is hiding information among the lower order bits of an image such that the image appears unaltered to the naked eye. A class of software and hardware techniques called *deniable storage systems* use such techniques to create a hidden storage volume, the presence of which can be plausibly denied by a user. In theory, this would allow a user to present a device to an adversary and provide access credentials to a publicly visible volume without revealing the hidden volume contained within it. Some approaches even take this a step further and provide multiple levels of deniability with more sensitive information kept at higher deniability levels. The most common way to achieve this is to disguise and hide data among the unallocated space of an existing storage volume [9, 74].

There have been a wide variety of different proposed systems designed to provide plausible deniability, although there are perennial problems that no approach has addressed. Foremost among these is that no system can provide plausible deniability for the driver software needed to maintain a hidden volume [26, 131]. Some designs assume that deniable storage systems would be widespread to provide deniability for the user's capability of possessing a hidden volume [21, 109]. Widespread use of a privacy tool does not amount to deniability as evidenced by the current suspicion of encryption technologies that serve as the impetus for deniable storage. Flawed assumptions have also led to consistent problems with sensitive information leakage from the hidden to the public volumes [30]. Many systems assume that a device will not contain any malware even after contact with the adversary [26,80,109], which is unlikely given recent trends [128]. Lastly, existing systems that aim to defend against more technically evolved attacks that could be carried out by a more advanced adversary in the process leave themselves vulnerable to much simpler attacks [16, 20, 26]. An example of this last point can be found in the case of a *multiple snapshot attack* in which an adversary compares snapshots of a device from different points in time and looks for suspicious changes. In the process of defending against these sorts of attacks, systems will create other sorts of suspicious markers that betray the existence of a hidden volume.

An examination of previous systems reveals important questions about disk forensics, system design, and the operational characteristics of deniable storage systems. To better explore and test our hypotheses about answers to these questions we have developed a new adversary model that addresses the flawed assumptions of previous approaches. In this new threat model, it is assumed that malware will be deployed by our adversary to surveil the user and gather additional information they can use to find evidence of a hidden volume. We also present the notion that an adversary will *primarily* rely on coercive tactics to uncover and gain access to hidden data and only needs a sufficient level of suspicion to use these tactics against a user. With a sufficiently detailed description of our attack surface, we have been able to identify potential attacks our adversary can leverage to look for indications of the hidden volume. These can include but are not limited to: finding information leakage from applications, analyzing the performance of a system for anomalies, statistical analysis to detect anomalies in measurements taken from a disk, deploying malware, or identifying behaviors specific to flash devices that betray a hidden volume.

From our adversary model, we can derive requirements for a deniable storage system to defend against the described adversary. Since carrying encrypted files or dedicated hardware is inherently suspicious, a deniable storage system must co-exist with an open public file system to maintain *plausible deniability.* Visible drivers or firmware are highly suspicious, as are unconventional partitioning schemes, unusable space in a file system, and unexplained changes to a disk's free space. The hidden file system must therefore operate in such a way that the encapsulating file system and operating system (OS) are unaware of the hidden file system's existence, even when faced with a detailed forensic examination. A deniable system must therefore meet four requirements: effectively hide existence of the data, disguise hidden data accesses, have no impact on the behavior of the public system, and hide the software used to read and write the hidden data.

Following these requirements, we have built a new system, Artifice, a deniable steganographic storage system that seeks to provide functional plausible deniability for both the data and the Artifice system. When the user needs to access the hidden data, they boot into an Artifice-aware operating system, such as a modified version of Linux stored on a USB drive. Booting into a separate operating system provides effective isolation from the host OS. Unlike previous systems, this does not leave behind suspicious drivers on the user's machine. To provide both obfuscation and a level of redundancy to our hidden data, a user's data is encoded with an information dispersal algorithm (IDA) which produces a set of *carrier blocks* that appear to contain random data. These carrier blocks can then be written to the unallocated space of an existing public file system. As the public file system cannot be aware of Artifice's existence, Artifice must protect this data from overwrites made by public operations. To address this risk to the integrity of our hidden data our IDA based approach allows Artifice overwrite tolerance by only requiring a tunable subset of the carrier blocks to reconstruct the original data. We have selected a variety of IDAs that Artifice can use, each of which have pros and cons. Additionally, this approach enables Artifice to repair itself whenever the user boots the Artifice aware OS.

Artifice carrier blocks are generated so that they appear to be random data making them indistinguishable from other random unallocated blocks. We have concluded that filling free space on a drive with random bytes must be done through a deniable process so that the adversary does not see the presence of random unallocated blocks as suspicious. One way to accomplish this is through a system that securely wipes a drive by encrypting data and throwing away the key. To make our hidden data easy to find when needed but difficult for an adversary to guess, Artifice metadata locations are generated algorithmically from a passphrase that must be supplied by the user. Without the correct passphrase, finding an Artifice instance through guessing its location in free space is computationally infeasible.

Artifice addresses the issue of multiple snapshot attacks through a variety of possible countermeasures. These include deniably shuffling blocks under the guise of a deniable operation such as defragmentation, ensuring there are enough accesses to the public volume to act as a smokescreen, or through operational security measures that deny our adversary the ability to carry out such an attack. All these techniques can be used to defend against this problem that previous work has fixated on, while avoiding the vulnerabilities previous approaches expose.

We have explored the often-ignored *operational* characteristics of deniable storage systems as a way to address gaps in a system's deniability. These include procedures for compromised devices or passphrases, best practices for hidden data integrity, navigating different choices of storage medium, and possible strategies for avoiding certain attacks we have previously identified.

In its current form Artifice is implemented as a logical block device through the

Linux device mapper framework. This implementation allowed us to empirically evaluate the validity of both our assumptions and those upon which previous approaches are based.

With both theoretical and applied techniques, we were able to evaluate the resilience of a deniable storage system when faced with operations made by an unaware public file system. To test the effectiveness of our obfuscation schemes, we applied entropy analysis techniques to samples of carrier blocks and were able to show that they are indistinguishable from other pseudo-random blocks, such as those secured by strong disk encryption systems.

We carried out the first demonstrated multiple snapshot attack on deniable storage. This attack entails taking two snapshots of a disk suspected of harboring a hidden volume and analyzing the changes between the two snapshots for suspicious activity or statistical anomalies. Carrying out such an attack that has previously only been described in theory has allowed for a more accurate characterization of an adversary's capabilities. Doing so determines the actual threat these attacks pose to all deniable storage systems, and to better develop countermeasures against such attacks. This implemented attack has shown that even when comparing write behavior based on a single feature, it is possible to distinguish between a disk with and without a hidden volume if the ratio of writes to the public volume versus the number of writes to the hidden volume is low enough.

We have also been able to better analyze the challenges posed by the unique structure of flash devices, specifically TRIM operations and garbage collection. TRIM is a command sent by a file system to a flash device to tell the device when a block is no longer in use so that it may be garbage collected. While it is recommended that Artifice be used on a device with TRIM disabled, this left unanswered questions as to whether or not Artifice could cope with the forensic implications of a solid state drive with TRIM enabled. To address this, we have described and demonstrated not only how Artifice can be detected or accidentally destroyed on a flash storage device but also a variety of different avenues for mitigating these risks and ensuring hidden data can remain undiscovered while providing methods to reliably recover that data.

Chapter 2

Background

Unlike more common cryptographic techniques such as encryption which focuses on preserving the confidentiality of a secret, steganography aims to hide the presence of the secret altogether. In the digital era steganographic techniques are commonly applied to hide information in media files such as images or audio. A common technique is to manipulate the least significant bits in an image to store information [24]. In most cases this leaves the image indistinguishable from the original to the human eye. All these approaches are ill-suited for storing large amounts of information because they are space inefficient and are difficult to scale while maintaining an assurance of plausible deniability.

Over the past two decades there have been numerous attempts at creating a steganographic storage system that will allow a user to store relatively large amounts of data with some level of plausible deniability. While these existing systems all claim to provide deniability, they commonly possess easily detectable traces or behaviors. Such characteristics can betray the existence of the file system itself or the user's capability of running a plausibly deniable system.

2.1 A Survey of Steganographic File Systems

Anderson, *et al.* [9] were the first to propose a steganographic file system and described two possible approaches. The first approach consists of a set of cover files filled with random information, of which a subset is combined with hidden files using an additive secret sharing scheme. The exact subset of the cover files that must be XOR'd together to reveal the hidden data is determined by a passphrase. This approach is more in line with classical steganographic systems that rely on embedding hidden information in the lower-order bits of audio and video files. While this approach is conceptually simple, a relatively large number of cover files is required to provide computational security. Additionally, the presence of unexplained random cover files would be easily detectable and could be considered suspicious by an adversary.

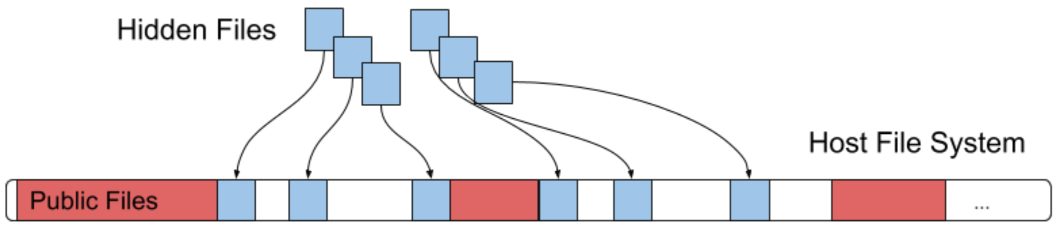


Figure 2.1: Anderson *et al.*'s second proposed scheme for a steganographic file system.

The second construction hides data within the unallocated space of another file system. The hidden files would be located using a one-way hash of some credential, encrypted to prevent detection in a disk filled with random bits, and replicated to protect against overwrite by the encapsulating file system. Due to the challenges posed by the cover file method most deniable storage systems are derived from this construction.

Deniable systems that hide information in free space fall into three general categories as described by Anderson *et al.*

1. *File Systems* that are either implemented as a standalone file system intended to manage both public and hidden volumes [48, 80, 83] or as an extension to an existing file system [74].

2. *Block Devices* that present virtualized hidden and in some cases also public volumes that are then treated by the operating system as disks [16,20,22,26]. These approaches often utilize the Linux device mapper framework [71] to insert themselves into the existing Linux storage system.

3. *Firmware* approaches are often applied to flash devices where one could modify the existing flash controller to support hidden volumes [55, 131].

McDonald and Kuhn implemented Anderson *et al.*'s second scheme as a Linux file system based on `ext2` [4] known as StegFS [74]. StegFS uses a block allocation table to map encrypted data to unallocated blocks. The hidden data is replicated to protect against accidental overwrite by the public file system. Unfortunately, continued writes to the public volume will eventually cause the loss of hidden data. It also possesses the capability of nesting hidden volumes so that the user can reveal less sensitive data in the hope of satisfying an adversary.

Mnemosyne [48] is a distributed approach to steganographic storage that replaces StegFS's simple replication technique with Rabin's Information Dispersal Algorithm [88] to provide greater durability and decrease write amplification across nodes. The primary weakness of such a system is that by distributing hidden information across multiple machines the attack surface is expanded to include any network infrastructure between the nodes.

The on-the-fly-encryption (OTFE) system TrueCrypt [120], its successor VeraCrypt [78], and other similar systems [10, 98] also provide the capability of running a hidden file system within the free space of an ordinary encrypted volume. Its approach is similar to StegFS in that each nested file system has a single key, which grants access to that level's hidden data. Czeskis *et al.* [30] demonstrated that TrueCrypt was vulnerable to information leakage through data writes made by the operating system, file managers, and word processors. In the case of leakage through the operating system, a TrueCrypt hidden volume can be betrayed by the Windows Registry or file shortcuts. Automated recovery files made by Microsoft Word allowed for recovery of hidden files from the public volume if the file was opened using the word processor.

Pang *et al.* [80] implemented their variant of StegFS that improved reliability by removing the risk of data loss in the hidden file system when the public file system writes data. To achieve this both public and private volumes share a block allocation bitmap. The hidden files are stored in free locations according to a secret key and are not tracked in the public file system's data structures. While this does provide perfect resistance to accidental overwrite the bitmap is left in the open which exposes the existence and maximum size of the hidden files. This work was expanded on by Zhou *et al.* [129] which described countermeasures for defending Pang's version of StegFS against two kinds of forensic analysis. In the first case it is assumed that the adversary can view static snapshots of the disk and compare them in what is commonly referred to in later work as a *multiple snapshot attack*. The second type of analysis, called *traffic analysis* involves the adversary viewing the actual IO traffic patterns in real-time. Both countermeasures rely on obfuscating any patterns in disk traffic or differences between snapshots through either relocating blocks and issuing dummy operations in the case of a snapshot attack or through feeding operations through multiple buffers in the case of a traffic analysis attack.

2.2 Snapshot and Traffic Analysis attacks

Defeating snapshot and traffic analysis attacks has become a central problem facing the development of deniable storage systems. As a result, most systems developed after Zhou *et al.*'s work on StegFS attempt to eliminate distinctions between writes to the public and hidden volumes.

Troncoso *et al.* [119] analyzed the countermeasures developed for Pang's StegFS from the perspective of an adversary that could continuously observe disk traffic. They demonstrated that it is possible with high reliability to reveal the existence of a hidden volume defended by those countermeasures.

Datalair [20] and HIVE [16] combine a hidden volume with oblivious RAM (ORAM) [39, 40] techniques. ORAM is intended to prevent an adversary from gaining information about a running program by observing the distribution of memory accesses. This same class of techniques can be applied to deniable storage systems where the distribution of writes to a device could betray the existence of a hidden volume if the distribution does not reflect normal disk behavior. The significant performance penalties of ORAM [23] are somewhat offset by the need to only obfuscate write patterns as read patterns are often ignored because they do not change the state of the disk. Theoretically, this prevents an adversary from successfully carrying out a multiple snapshot attack. When it is applied to a deniable storage system it severely impacts the usability of both the hidden and public volumes. In the case of HIVE, throughput for *both* public and hidden sequential operations is slowed to 1 MB/s [20] compared to raw disk performance of about 220 MB/s. Random disk write patterns and unexplained slow performance compared to the raw disk can be considered suspicious and lead to the compromise of the hidden volume.

Chen *et al.* published PD-DM [26], a virtual block device approach aimed at addressing the poor performance of ORAM derived systems [16, 20]. Although it significantly improves read performance, write performance still suffers and it presents the same distinctive performance characteristics that would betray the existence of a hidden volume.

A variety of systems attempt to defend against multiple snapshot attacks by making dummy writes to the disk with the goal of preventing the adversary from exposing the existence of a hidden volume based on write distributions [21,80,131]. Although this approach could lead to the same abnormal write patterns that weaken ORAM-based approaches.

2.3 Flash Storage and Mobile Devices

SSDs create a set of different issues for deniable storage versus traditional hard drives. The logical block store that the flash translation layer (FTL) presents to the operating system allows the SSD to relocate physical pages so that garbage collection can reclaim pages invalidated by more recent writes independently of the operating system. It is necessary for the SSD to create free flash blocks (encompassing a moderate, but fixed number of pages) that can be erased and made available to future writes. Erased blocks are usually not available via the logical interface as they are not mapped into the logical address space. The FTL will mark any block written by a file system as "in use" whether these writes from the hidden or public system. This creates an opening for detection through forensic analysis where our adversary can compare a list of blocks used by the file system to those marked as in use by the SSD. Alternatively, the FTL may unknowingly erase hidden data as part of opaque and non-standardized garbage collection operations if it is unaware of the deniable volume's presence.

This layer of abstraction presents a hurdle for deniable storage systems. Most that seek to address these challenges either work on raw flash devices [83] or are intended to operate as drive firmware [131]. Since custom firmware would be suspicious and raw flash devices are still relatively uncommon, Artifice must attempt to address these challenges through other means.

DEFY [83] is a log-structured deniable file system designed for host controlled flash devices and is based on WhisperYAFFS [107], itself an encrypted version of the YAFFS file system. DEFY does not adequately protect against hidden data overwrite unless hidden volumes are constantly mounted and is designed for use on memory technology devices (MTDs). An MTD is a type of raw flash device that relies on the operating system or file system to perform tasks normally performed by the FTL. MTDs are commonly found on mobile devices.

Zuck *et al.* [131] proposed the Ever-Changing Disk (ECD) [131], a firmware design that splits a device into hidden and public volumes where hidden data is written alongside pseudo-random data in a log-structured manner. Although the design makes significant progress towards solving the problem of hidden data overwrite and mitigating multiple snapshot attacks, the lack of deniability for the exposed partitioning scheme and proposed custom firmware are vulnerabilities.

Jia *et al.* [55] present three generalized attacks against conventional deniable volumes operating on flash devices. They describe a capacity analysis attack against DEFY that leverages the fact that it disables garbage collection at its lower deniability levels to prevent overwrite of data at higher levels. They also design and implement an FTL-based deniable storage system called DEFTL. While this system adequately defends against a less rigorous adversary it fails to provide deniability for the flash device's custom firmware or defend against a multiple snapshot attack.

Some systems such as INFUSE [25] utilize the ability of certain flash devices to switch between single bit per cell (single bit per cell) and MLC (multiple bits per cell) to hide information. To the adversary, a flash device looks like it is using SLC while it actually is an MLC device that is using the additional available bits per cell to create a hidden volume. Zuck *et al.* have explored the technical details and limitations and of this technique with their data hiding method VT-HI [130]. While these techniques make it far more difficult to detect the hidden data outright they do not provide protections against malware or information leakage.

Due to the increasing prevalence of mobile devices it follows that some deniable systems would be developed specifically for smartphones and tablets. One of these called Mobiflage [109], fills the disk with random bits where hidden data can possibly be stored. The hidden volume is then placed at a specific hidden offset within a public volume. Like many previous approaches, it relies on the ambiguity of whether or not hidden data is present to provide deniability. Multiple variants of Mobiflage are presented but they all rely on a FAT32 formatted SD card installed in the mobile device.

Many similar systems designed for use in concert with Android's full disk encryption system have incrementally improved upon Mobiflage's basic design. Among these is Yu *et al.*'s Mobihydra [127] which adds support for multiple deniability levels and uses a "shelter volume" to transfer data from the public to hidden volumes without rebooting the device. One of the primary focuses of this work is the boot timing attack [127], a side channel attack in which the adversary directs the user to attempt booting into a system repeatedly with both valid public system passwords and incorrect passwords. The time difference between correct and incorrect authentication operations is then used by the adversary to determine whether a hidden volume is present. Yu *et al.* experimentally verify the efficacy of the boot timing attack against a mobile phone running Mobiflage.

Another system, Mobipluto [22], uses thin provisioning to create virtual hidden and public volumes. Mobimimosa [51] adds an additional security measure that will either delete the hidden data or a threatening application to prevent information leakage to the adversary. Mobiceal [21] aims to defend against multiple snapshot attacks by utilizing dummy writes and modifying certain Android system characteristics to prevent information leakage.

Feng *et al.* while developing their mobile-specific system Mobigyges [37] formally describe what they call the fill to full attack and the capacity analysis attack. In a capacity analysis attack the adversary compares the capacity of the physical device with the capacity of the public storage volumes to determine if a hidden volume is present. Although this attack is commonly avoided using virtual block devices. Complementary to capacity analysis is the fill to full attack in which the adversary fills the public volume(s) and compares the amount of data successfully written to the physical capacity of the device.

Most of these systems assume that the deniable system software is merged into the Android operating system [21,22,109,127]. The resulting widespread adoption is intended to counteract any suspicion resulting from its presence on a device.

2.4 Secure Deletion

Closely related to the topic of deniable steganographic storage is secure deletion. The ability to securely delete information provides plausible deniability because to an attacker the information may as well never existed.

Many approaches to secure deletion have been proposed but most incur significant performance penalties making their use in everyday systems unlikely. Early approaches geared towards traditional magnetic media usually focused on overwriting deleted blocks. Bauer *et al.* [13] implemented a system to work with EXT2 that simply overwrites unused blocks. While simple and reliable it is not a particularly efficient technique as it requires multiple overwrites for each data block and will not work on flash devices due to the difficulty of in-place updates and write endurance limits.

Later the idea of encrypting data and disposing of the key became a popular method of securely deleting data as it drastically cuts down on the amount of data that must be overwritten. This idea was proposed by Boneh *et al.* [31] to allow for the deletion of data from backup tapes without having to mount and write to the tape. Instead the encryption key would be stored separately and controlled by the operating system. When a user wishes to delete data the system would simply “forget” the key.

Peterson *et al.* implemented a secure deletion capable versioning file system utilizing authenticated encryption [84]. Deletion occurs by overwriting a small “stub” appended to the end of each data block. Without this stub it is impossible to recover the information. While this technique is significantly more efficient than Bauer’s system, it assumes the ability to overwrite stubs in-place, which may not always be possible, especially when considering the log-structured nature of flash devices.

Log structured file systems and flash devices pose unique challenges to secure deletion as one cannot simply overwrite the physical blocks: the old copy will still exist in an obviated part of the log. Lee *et al.* designed a secure delete file system designed for NAND flash that modifies YAFFS to encrypt data and delete it by erasing the keys and metadata [67]. Although this technique is reliable and relatively fast, it is limited to use with raw NAND flash devices due to the need for an in-place update. Reardon *et al.* attempted to address the problem with two user-space techniques [90]. The first method is called purging in which all unallocated space is filled with junk data, ensuring all blocks are overwritten. The second is ballooning, where the free space available is artificially constrained. While a combination of these two techniques is demonstrated to be effective, it causes a significant delay between issuing a command to delete a file and the actual deletion.

Zhao and Mannan [69] proposed a secure deletion system called Gracewipe that utilizes a processor's trusted platform module (TPM) to verifiably delete encryption keys when the user is under coercion. It operates by deleting root encryption keys when a special deletion password is entered into the system rendering the device undecipherable.

2.5 Summary

There are many different designs for deniable storage systems that all share a basic common mechanism, hiding data in unused space on the device. We summarize this variety of systems and the general approaches they take to address common challenges in Table 2.1.

A common thread across the development of deniable storage is the presence of a unique "tell" or an exposed component or behavior, for each system. Should the adversary know what to look for with each system it becomes significantly easier to compromise the system's deniability. This and other common flaws with deniable storage systems are more thoroughly discussed in Chapter 3.

Table 2.1: Summary of Previous Deniable Storage Systems

System	**Type**	**Overwrite Resistance**	**Multiple Snapshot Resistance**	**Deniability for Software**	**Available for use**
Veracrypt [78]	OTFE	None	None	None	✓
StegFS (McDonald) [74]	File System	Replication	None	Separate hidden FS driver	✓
StegFS (Pang) [80]	File System	Shared Allocation Map	Dummy Writes	None	×
HIVE [16]	Block Device	Separate Volumes	ORAM	None	✓
Datalair [20]	Block Device	Separate Volumes	ORAM	None	×
DEFY [83]	Flash FS	Separate Volumes	Dummy Writes	None	✓
Mobiflage [109]	Mobile FS	Separate Volumes	None	None	×
Mobipluto [22]	Mobile FS	Separate Volumes	None	None	×
Mobigyges [37]	Mobile FS	Separate Volumes	None	None	×
Ever Changing Disk [131]	Flash Firmware	User intervention Limited Public Writes	Dummy Writes	None	×
PD-DM [26]	Block Device	Separate Volumes	ORAM	None	×
DEFTL [55]	Flash Firmware	FTL Managed Allocation	FTL Managed Allocation	None	×
INFUSE [25]	Flash FS	Replication	Data hidden in flash side channels	Separate hidden FS driver	×
Artifice	Block Device	Erasure Coding and IDAs	Block Re-use Dummy Accesses	Separate hidden volume and OS	✓

Chapter 3

Threat Model

A threat model that more accurately describes the adversaries a user will face when utilizing a deniable storage system is essential to a successful design and deployment. By analyzing previous approaches and identifying attacks against them, we have developed a new more thorough and realistic threat model for deniable storage systems.

3.1 Shortcomings of Previous Models

Many fundamental flaws of existing deniable storage systems can be traced back to flawed assumptions made in their proposed threat model and in some cases a failure to explicitly state a definition of the adversary or their capabilities [74,80].

One of the most flawed assumptions is the fact that the adversary does not have to prove the existence of a deniable volume to compromise the system. Since adversaries are often assumed to apply coercive tactics to access plainly visible encrypted data, they could also apply the same technique to compromise a deniable volume. In the case of encrypted data, the obfuscated bytes are readily apparent but in the case of a deniable volume there must be sufficient indication that the user is running such a system to warrant escalating to coercion and expending additional resources. As a result, we can assume attacks against a deniable storage system do not have to prove the existence of a hidden volume but only convince the adversary that there is a sufficient probability that the user is hiding something. Since the goal of a hidden volume is to avoid coercive tactics, we must avoid arousing the adversary's suspicion or to provide some information to convince the adversary that a hidden volume does not exist.

Some examples of previous work also assume that the user's ability to utilize a deniable storage system would be considered innocuous if that system is bundled with other widely used software such as a specific operating system implementation [22, 109]. Since plausibly deniable storage is a response to scrutiny of disk encryption systems which have seen widespread use, the adversary would also view a deniable storage system with similar, if not more suspicion than an encrypted volume. If the capability of creating a hidden volume is innocuous, even sophisticated multiple-snapshot-resistant approaches would be vulnerable to a significantly simpler adversary. While this more limited adversary would have to know what driver software or system characteristics to look for, we must also assume that the adversary has knowledge of the deniable system's design. Otherwise we risk relying on "security through obscurity" which should be avoided whenever possible as it does not provide a strong security assurance [62]. While keeping the design of the system a secret can add an additional layer of security to hinder an adversary, design secrecy should not be a critical assumption made when evaluating the security of a deniable storage system. Since we must assume they will know the design and capabilities of deniable systems, it would be prudent to also assume they know of weaknesses and attacks against such systems. From this we have concluded that a user's capability to run a plausibly deniable storage system cannot be considered normal. As such, any secure deniable storage system must also take efforts to disguise a user's ability to run said system in addition to any hidden data.

Another flawed assumption is that the adversary will not install malware or take steps to gain more information about that device than they can obtain through static snapshots [22, 26, 80, 83, 109]. It has been observed that officials have installed malware on user's devices when they cross a border [128] and both nation-states [125] and private enterprises [126] have developed robust tool kits for widespread surveillance. We propose that a deniable storage system must assume that malware can be installed on the user's device and that it should be capable of defending against it. Since malware can potentially provide a significant amount of information to the adversary about the user's activities, we must assume that an adversary that can install malware on the user's device must be capable of surveilling the user in real-time, even observing individual disk accesses.

3.2 Our Threat Model

In light of these shortcomings, we propose the following threat model. The adversary is assumed to be an agency or organization that wishes to tightly control the flow of information within and across their regime's borders. We consider our user to be an individual who wishes to carry sensitive information through a region where the user can possibly come under close scrutiny.

3.2.1 An Example Use Scenario

A user enters a region with the intent of obtaining sensitive information that our adversary does not want disseminated. There is also something preventing the user from transmitting the data over a network, either due to strict monitoring of network infrastructure by the adversary or lack of network connectivity. As a result, our user is limited to physically transporting the data out of the region on a storage device. In our case the user decides to do so by using a steganographic or deniable storage system rather than conventional disk encryption to create a hidden volume to hide the fact that information is being leaked.

In our scenario the adversary is some organization that exerts a level of control over a region and tightly controls the flow of information. It is the goal of this adversary to prevent leakage of information. To further this goal our adversary seeks to determine whether the user is attempting to exfiltrate sensitive information on their device. The adversary will inspect devices, flag individuals of interest, and install spyware to gather more information on the individual's activities.

3.2.2 Malware and Information Leakage

Utilizing malware and exploiting existing sources of information leakage are commonly used in digital surveillance. Tools are widely available from both commercial (Pegasus Spyware) [126] and government sources (NSA ANT Catalog) [125]. In the past, it has been shown that state-level adversaries will deploy these sorts of tools at places like border crossings in order to gather information about a user's movements and activities [128]. Since an adversary would have physical access to the user's device and any credentials, they would have the ability to install many different forms of malware ranging from a malicious application to operating system and firmware level rootkits on the user's device. We must assume that as part of a region's practices governing the flow of information, the adversary can and will deploy malware to track and surveil the user.

The primary advantage of malware when used to search for hidden volumes is that it allows our adversary access to significantly more information about a user and their device than when only observing a static device at a border crossing or when the device is unattended in a hotel room. Many pieces of malware could allow the adversary to observe the device's activities in real-time. If this malware is installed in the operating system, it would be possible for the adversary to view storage operations in real-time. This fine granularity of observations has been shown to defeat deniable storage systems in the past by looking at specific write behaviors unique to specific deniable systems [119]. The collected observations can either be covertly transmitted over a network to the adversary or stored on the device for later retrieval when the adversary next has a chance to inspect the device in person.

Malware used by the adversary could come in multiple forms. A simple yet obvious form may be a smartphone application that is given access to the device's underlying storage and I/O hardware such as a camera or microphone. A more sophisticated adversary could employ a rootkit, a self-hiding malicious piece of software that provides access to unauthorized users, to monitor the activities of both the user and storage device. The most sophisticated adversaries could employ lower level attacks where an adversary compromises the hardware or firmware of a device. Of particular risk are pieces of malware such as keyloggers, tools that record buttons pressed on a keyboard, that can reveal access credentials to an adversary.

If an adversary can install malware on a device that would under other circumstances be limited to a single or limited number of interactions with the adversary, it can greatly expand their ability to observe the device and the user. For instance, while an adversary is making an initial inspection of a device as the user is crossing a border, they could install a piece of spyware and proceed to surveil the user at their leisure. That adversary has now gone from being able to observe the device at border crossings to observing the user's activities whenever they wished at low cost. Given the widespread deployment of malware and easily developed tools that can expose deniable storage systems, we consider that any threat model pertaining to deniable storage must assume that the adversary will deploy malware against the user.

3.2.3 Suspicion, Escalation, and Coercion

Unlike the adversary those previous systems have been designed to defend against, our adversary doesn't have to specifically prove that a hidden volume exists. Since it is well known that the user is a vulnerability in most systems, it would follow that instead of relying on relatively involved methods of proving that a user has a hidden volume, the adversary would look for clues to establish whether there is a reasonable suspicion that the user possesses such a volume. The flaw of this sort of logical escalation is that it expects the adversary to follow rules and act logically, which may not always be the case. There is always a chance that the adversary could jump straight to coercive tactics for unknown reasons and there is no reliable defense against this possible outcome. As such, to limit the scope of our problem, we will assume that our adversary will behave in a logical manner.

Ultimately the adversary will follow some sort of process for escalating surveillance of a user based on the suspicion that the user may be attempting to carry out actions in opposition to the adversary's interests. This sort of escalation would begin with relatively quick, low effort attacks against possible hidden volumes. Should they find clues that arouse their suspicion, they can escalate to more involved attacks that could determine with better accuracy whether a hidden volume is present. For instance, the adversary will be suspicious if there is software present on the device that creates or accesses a hidden volume. Even if the tools used to access the hidden volume are bundled by default with a commonly available system, such as in the case of Veracrypt [78], they would still garner significant scrutiny. Once a sufficiently high probability that the user possesses a hidden volume has been established, the adversary will move to coercion to expose the contents of the volume. The nature of these coercive tactics could range from legal penalties to direct bodily harm.

We assume the adversary, although powerful, will wish to apply resources in an efficient manner. Many ways of compromising a hidden volume, especially those reliant on statistical methods, will incur several false positives and false negatives. We assume that in the interest of efficient resource use, our adversary will seek to minimize the number of false positives they must investigate. Some number of false positives will still occur, we assume our adversary will be tolerant of this. Should the false positive rate be too high for a given technique, we assume that the adversary would instead turn to alternatives with a lower rate. Conversely, we assume our adversary will not be tolerant of a significant number of false negatives when designing their investigative techniques, as this would indicate that sensitive information is slipping through their net.

3.2.4 The Attack Surface

We consider anything on the user's machine and any devices the user can interact with using their machine as part of the potential attack surface. Unlike previous adversary models, we also consider the user to be a part of the attack surface, primarily because the user is susceptible to coercive attacks which will compromise the deniable volume.

There are two different ways the adversary can interact with this attack surface. The first is direct, where the adversary has physical access to the device, such as in the case of a border crossing. This allows for direct observations of the hardware and software on the device at a single point in time. The second is when the adversary can indirectly surveil or observe the device or user, by observing the user's network traffic or malware installed on the user's device. The secondary method would allow the adversary a fine-grained view of the user's activities over time, whereas direct physical access would allow the user to view the device in snapshots.

The adversary might be able to take control of the device at certain points in time and perform any static forensic analysis they want. This could be with the user present, such as at a border crossing or checkpoint or with the user absent, when the device is left unattended in a place like a hotel room. When in possession of the device, the adversary can install applications, modify files, manipulate disk sectors, inspect hardware, and make copies of the disk. In these situations the adversary would gain information about the device at a specific point in time or install malware to enable remote observation to gather further information about the user.

The adversary may monitor the user's interactions with a public network infrastructure for suspicious behavior. An example of suspicious behavior would be uploading images and video to a remote file server or accessing the TOR network. It is also possible that certain outgoing or incoming connections would be blocked within the adversary's sphere of influence.

Although the adversary we have presented is relatively powerful, we must lastly assume that they can't always monitor a user. There must be a time where the user is safe enough to access and write to a hidden volume. If this assumption does not hold for a specific adversary, a deniable storage system is not the correct tool for that use case.

The adversary will very likely have access to credentials for the user's normal operating system or publicly visible encrypted volumes. These credentials would be provided freely by the user when requested to avoid the use of coercive tactics. The adversary can easily verify any access credentials obtained from the user. This is because the adversary will likely possess the user's device and can verify whether any supplied credentials can access what they are supposed to.

3.2.5 The Challenge of Multi-leveled Deniability

Many deniable storage systems support multiple levels of hidden information where the lower levels contain increasingly sensitive information [74, 80, 83]. The intended purpose is so that when subjected to coercion, the user could reveal the less sensitive levels of the deniable system while keeping lower levels secret. This is similar to running a whole disk encryption system alongside a deniable system where the user would surrender the encryption key to an adversary while not revealing the presence of the deniable system.

The multi-level approach assumes the adversary will deduce the existence of a hidden volume without significant hassle. While such an approach may satiate an ignorant or less determined adversary, if they possess knowledge of the storage system's capabilities, we have to assume they will press the user for access to further hidden levels until they are satisfied. In fact, once the user exposes a hidden level, it would make sense that the adversary would know exactly which deniable storage system is in use. At this point a highly motivated adversary could pressure the user to disclose all levels of the deniable storage system.

As we assume that our adversary knows of a deniable system's design and capabilities, it becomes clear that multiple levels of deniability are of dubious value as a primary defensive measure against a determined adversary. For example, we cannot predict how the adversary will react once they are given credentials to a less sensitive hidden level. If they know that a system can possess multiple levels there is nothing stopping them from pressuring the user to provide credentials for every level on the device. Considering this, a safer course of action would be to avoid disclosing any aspect of the deniable system to the adversary and rely only on secrecy as a first line of defense. That is not to say that the capability isn't worthwhile but that it should be relied on only as a fallback if other methods of protecting the user have failed.

3.3 Possible attacks under our threat model

We have identified a series of potential attacks against existing and possible future deniable storage systems. The goal of each attack is to gather enough information to establish whether there is a sufficient probability that the device contains a hidden volume. If the probability is high enough to arouse the adversary's suspicions, we assume they will move to coercive tactics.

1. *Capacity Analysis*: This is a relatively simplistic class of attacks in which the adversary can look at the characteristics of visible volumes on a device and compare that to the advertised capacity of the device. A sufficiently large discrepancy between the two would indicate an abnormality that could indicate a side effect of a hidden volume. For instance, if an adversary notes that there is a relatively large amount of used space on the device that is not accessible through visible volumes, it could indicate a hidden volume.

2. *Detecting Exposed Software, Data structures, and Firmware*: If the adver-

sary can identify a data structure (for example StegFS's exposed allocation bitmap) or some software that corresponds to the user's ability to run a deniable storage system, they could conclude the user possesses such a volume. If the device is known to have a suspicious firmware version available, it is possible for the adversary to regard all devices of that model to be suspicious.

3. *Information leakage:* The adversary could check common locations for temporary or hidden files created by applications that may have accessed a hidden volume. If there is no evidence that those files are linked to innocuous public files, the adversary can possibly conclude that those files are traces of a deniable volume [30].

4. *Performance Analysis*: The adversary could run a benchmark on the public device and compare the results to the advertised or normal performance results for the storage device. Since some deniable storage devices, most notably ORAM-based approaches, significantly impact the performance of both the public and hidden volumes [16, 20, 26]. A sufficiently large discrepancy between advertised and actual performance could indicate the existence of a hidden volume.

5. *Statistical Analysis*: Assuming that the hidden data has been obfuscated in some manner (such as encryption) the adversary could perform statistical analysis on the unallocated space of a hidden volume. If there is a significant difference in the statistical distribution of bits from one block to another, it could indicate the existence of a hidden volume.

6. *TRIM analysis*: Should TRIM be enabled on the user's device, the adversary could look at which logical block addresses have not been TRIMmed but also

don't have valid blocks from the file system's perspective. This will provide the adversary with a far smaller set of blocks to analyze further. Since most devices utilize periodic TRIM, some blocks will have been flagged as false positives.

7. *Malware*: We consider the primary threat posed by adversary controlled malware to be information gathering about the user's activities on their device, specifically those pertaining to their storage devices. Intelligence regarding the user's movements and network traffic could also be useful to the adversary. The adversary realistically could install malware anywhere on the device where persistent writable storage is found.

8. *Boot-timing*: This attack specifically targets Mobiflage [109] as described by Yu *et al.* [127]. In this attack it is assumed that the adversary will possess the password to the public volume. The adversary will boot into the public volume, noting the time needed for authentication. The adversary repeats this with an incorrect password while also recording the authentication time. In the case of Mobiflage there is a noted difference in the overall authentication time and the difference between successful and failed attempts. It is unknown whether other deniable storage systems are as susceptible to this attack.

9. *Denial of Service*: The adversary fills all free space on the public volume. Since data is stored within the free space eventually will be overwritten unless the space occupied by hidden data is protected by the software writing to the public volume. In this case the system would be vulnerable to capacity analysis, expose suspicious partitioning schemes, or expose driver software. This attack also demonstrates an interesting behavior in some systems where

they will fail securely when attacked in this manner. Data will be lost but the confidentiality of that data and the safety of the user are not compromised.

10. *Multiple Snapshot Analysis*: The adversary compares a snapshot of the disk obtained at a previous point in time to one or more from a later point in time. In theory differences between the snapshots should follow a certain pattern or distribution based on a user's workload or the public file system. For instance in a log-structured file system [96] it can be assumed that new blocks will always be written to new segments at the head of the log and blocks will be freed throughout the log as segments are marked for deletion and later garbage collected. Unusual deviations from the "normal" pattern could imply the existence of a hidden volume.

    As with previous work we consider there might be different observation frequencies for this attack [16]. A minimum of two snapshots would be required but this number could increase to one snapshot for each individual disk operation. We will call this latter case continuous traffic analysis. If the adversary can obtain this granularity of snapshots, we would expect it to become significantly easier to determine the existence of any suspicious writes to the hidden volume.

11. *Continuous Traffic Analysis*: As a worst case scenario the adversary could have access to a full record of accesses to the disk and other system events. With such a breadth of information the adversary would be able to detect the presence of hidden volume [119]. One way the adversary could obtain such information is through spying on the user via malware.

3.4 Related Work

While the specific details of previous threat models vary, we can generally categorize them by how often they assume the adversary can possibly access the device to inspect it for traces of hidden information or other suspicious activity [30].

- **Single snapshot**: A single snapshot adversary can only view a device once. Most existing deniable systems have some level of resilience against this sort of attack. This is the simplest and generally considered the least capable adversary considered by previous work. Examples of systems designed to defend against this adversary are McDonald's StegFS [74] and Mobiflage [109].

- **Multiple snapshot**: A multiple snapshot adversary can view the device two or more times. Multiple snapshots of the disk can be compared and the changes analyzed for anomalies that could reveal a hidden volume on the user's device. It is important to note that with this class of attacks, the adversary is only able to view static snapshots of the disk a limited number of times.

- **Continuous observation** is when an adversary can continuously observe writes to the user's device or make a snapshot of the device for each write. This adversary capability is sometimes also called *continuous traffic analysis*. This sort of attack would likely require a form of malware to be installed on the user's device for information gathering. Technologies such as ORAM are employed to defend against this class of adversary [16, 20].

3.4.1 Prior Threat Models

Anderson *et al.*'s work makes the initial assumption that the adversaries can and will use coercive tactics against the user to expose sensitive information on a device and that the adversary has a complete understanding of both the steganographic and the public components of the system [9]. Subsequently, McDonald *et al.* does not provide an explicit threat model but does note a weakeness, an adversary discovering an exposed StegFS driver could lead to them pressing the user for access to a hidden volume [74]. The original description of Pang *et al.*'s StegFS also doesn't explicitly make assumptions concerning the adversary's capabilities [80]. In most cases it implicitly assumes the same use case and adversary as McDonald *et al.*'s design. The key weakness of their model is their solution to the accidental overwrite problem, which is an exposed allocation map. As such it implicitly assumes that an exposed data structure without a deniable explanation is not inherently suspicious.

Further work on Pang *et al.*'s system includes the author's description of a possible multiple snapshot attack mitigation and introduces elements of a formal adversary model [129]. They introduce the idea of categorizing attackers by the level of granularity at which they can make observations of a system running a hidden volume. Specifically, they make a distinction between what later work would call a snapshot and traffic analysis adversaries.

Hive, while lacking an explicit threat model, assumes that the security of a multiple snapshot-resistant system relies on rendering accesses to a hidden volume and a public volume indistinguishable [16]. While they, along with Datalair [20] and PD-DM [26], can provably achieve this, they do not assume that the adversary will not find two things suspicious. First, they create random appearing write patterns for the public volume which along with an exposed driver would be considered suspicious. Second, they do not assume that the adversary can deploy malware. If faced with an adversary that deploys malware on a device they consider suspicious due to previously stated reasons, the adversary would likely be able to view the state of the machine in operation, which the authors assume is impossible. Lastly, since the authors assume that the adversary can utilize coercion to force a user to reveal a password, the presence of suspiciously random write patterns and a piece of software would give the adversary reason to employ coercion. As a result of these shortcomings, the notion of indistinguishability as presented does not entirely provide plausible deniability.

Another very common flawed assumption found in previous work is that the adversary will not find a piece of software capable of creating a hidden volume suspicious if it is widely deployed on devices [21,22,51,109,127]. This assumption is flawed as existing disk encryption schemes are commonly deployed but still face significant scrutiny. There is no reason that widespread adoption of systems capable of creating deniable volumes would lead to a different outcome. Therefore we assume that our system is not widely adopted, and we must disguise the presence of the software used to create and access a hidden volume.

The threat model utilized by INFUSE makes some departures from preceding systems. Most importantly it assumes that the adversary will find suspicious the presence of software used to access a hidden volume. Although, unlike our approach they make no assumptions regarding malware. It can be implicitly stated that they assume that the adversary will not deploy malware by stating that the adversary cannot view the running state of the system, which is a clear capability with certain types of malware.

Kedziora *et al.* look at threat models of existing deniable storage systems and their threat models with a specific focus on Veracrypt [61]. They propose that the categories of continuous observation and multiple snapshots are similar enough to be considered a single type of adversary and introduce a third stronger adversary type called Live Response Access. In a Live Response Access model, the adversary is assumed to be capable of direct or remote live access to the user's device while they are running a deniable volume. This includes access to any hidden operating system, and access to any network environment or cloud application that the hidden system is interacting with or contained in. This last model, while not explicitly stated, is closely related to our assumption that an adversary can install malware on a user's device, which would lead to similar adversary capabilities. Instead, Kedziora *et al.* propose that an adversary would obtain this level of access through physical access to the device or remote access tools like Team Viewer or Windows Remote Desktop.

3.4.2 Existing Attacks

While many threat models and constructions for deniable storage systems have been proposed, there is a notable deficit in the number of proposed attacks against such systems. The first thoroughly described and published attack against a deniable storage is Troncoso *et al.*'s traffic analysis attack [119] against Zhou *et al.*'s proposed enhancements for Pang *et al.*'s StegFS [80, 129]. Troncoso *et al.* demonstrates that if an adversary can perform continuous traffic analysis on a system containing a hidden volume, presumably through some form of spyware, it is possible to detect distinctive block access patterns that correlate to the dummy writes and block moves that Zhou *et al.*'s mechanisms use to hide writes to the hidden volume. It is from this paper that the power of being able to directly observe a record of disk writes is made apparent.

Following another avenue, Czeskis *et al.* attack the deniable volume feature found in the TrueCrypt by looking at its interactions with the Windows Operating System and user applications [30]. They show that Windows can leak information about the existence of a hidden volume through automatically generated shortcuts (`lnk` files), the Windows Registry, Microsoft Word auto-save files, and through the application Google Desktop. All of these locations would be accessible to most adversaries inspecting a device. As a result we assume that the adversary is capable of exploiting this weakness in existing systems.

Kedziora *et al.*, in addition to describing a new take on threat models for deniable storage, also demonstrates attacks against the Veracrypt hidden OS feature which, is an operating system installed in a hidden volume. Through entropy analysis of their test disk, they observed that the beginning and end of the hidden volume corresponded to sectors with lower entropy than the surrounding parts of the disk. Their second attack compares two disks, one where a hidden OS has run and one where a decoy OS has run and demonstrates that because of how Veracrypt writes data, it is possible to look at the ranges of sectors changed and determine the approximate size and possible existence of the hidden volume.

3.5 Summary

In this chapter we have examined the shortcomings of assumptions, both explicit and implicit, made by previous deniable storage systems. Chief among the shortcomings of previous models are the assumption that the adversary does not know about deniable systems and their weaknesses and the assumption that the adversary will not install malware on the user's device. In the case of the former it has been widely accepted that if a system relies on the secrecy of its design to provide security, that secret will eventually leak leading to the compromise of the system in question. It follows that we must design our system under the assumption that our adversary knows what a deniable storage system and what points to attack. The second assumption is malware, which can provide an adversary an easy means to escalate their capabilities and in some cases possibly even compromise access credentials used to secure a deniable system. In short, we have to assume our adversary is competent and will use existing observation techniques effectively.

From this examination we have defined a sample use case for our deniable storage system; where a user crosses a border into a country, obtains some sensitive information, and then crosses the border again to leave. We discuss how an adversary may utilize malware, a framework for how an adversary may escalate their tactics based on observations, and define our attack surface. Additionally, we examine the consequences that may arise from the use of multiple levels of deniability in a single volume.

We define a list of potential attacks our adversary can employ against a deniable storage system. These attacks can include exploiting information leakage, denial of service, multiple snapshot analysis, TRIM analysis, malware, and more. Lastly, we cover the adversary models and assumptions presented by previous systems and more thoroughly examine their individual shortcomings. Following this we can then look at the relatively small number of implemented or well defined attacks against existing systems.

Chapter 4

The Artifice Deniable Storage System

With a well-defined threat model we can now create a set of design requirements with which to direct the construction of Artifice, our steganographic deniable storage system. In this chapter we discuss these design requirements, how Artifice approaches meeting these design requirements, our Artifice implementation, and an operational security model to help dictate the proper use of our system, and evaluate its performance.

4.1 Design requirements for a Deniable Storage System

By estimating our adversary's characteristics and considering our identified attacks, we have identified the following problems central to the successful design and implementation of a deniable storage system:

1. *Hide the system's driver software.* Due to the specific use cases for deniable

volumes, we must consider that the existence of driver software on a device might imply the existence of a hidden volume on a user's device. Our system must be able to provide a deniable reason for the presence of the suspicious software or hide the driver software through some secondary mechanism to avoid arousing the adversary's suspicion. Ideally, our system would be capable of running on off-the-shelf hardware.

2. *Obfuscate the hidden data.* To deniably store data, we must obfuscate the hidden data and render it undiscoverable. In most existing systems this involves encrypting the data and hiding it among random bytes in a volume's unallocated space. We consider this to be the primary mechanism for hiding information, and as such it is imperative to thoroughly examine available obfuscation techniques and their limits.

3. *Mitigate the effects of information leakage.* Hidden data must coexist with publicly visible data on a drive. This presents the risk that sensitive information leaks from the secure environment provided by the hidden volume to an insecure public volume, whether this is through ordinary programs operating on the hidden data, the operating system, or malware installed by an adversary. Since closely vetting all the software present on the average computer is an almost infeasible task, we must study design decisions and practical measures that can be taken to mitigate the effects of information leakage.

4. *Preserve the integrity of hidden data.* Since the hidden data may need to exist on an unaware publicly visible volume, there is a risk that the volume will unwittingly destroy some hidden information. As such a deniable system, to the extent possible, should take measures to ensure that its data

is not overwritten without compromising the secrecy of the volume or its driver.

5. *Providing deniable reasons for changes made to a disk by writing hidden data.* Writing hidden data to the disk will inevitably result in changes visible to an adversary. Previous work has shown that if the adversary possesses sufficient background information about the user's device, a hidden volume can be detected. As a result, a secure deniable storage system must be able to provide innocuous reasons for the inevitable changes to the disk that are not made by the public file system.

6. *Address flash and emerging storage tech.* Compared to mechanical disks, flash storage presents different storage characteristics that impact the design of a deniable storage system. Flash translation layers and the inherent characteristics of flash devices present additional opportunities for compromising a deniable volume designed for a mechanical disk. These characteristics can lead to information leakage, a higher risk of data overwrite, or expose the user's capability of possessing a hidden volume. Since most user devices now utilize flash memory for storage, we must seek to mitigate these additional risks when designing a deniable storage system.

4.2 Artifice System design

To address the previously discussed design requirements, we have designed *Artifice*, a plausibly deniable virtual block device in software using the Linux device mapper kernel interface. Artifice obfuscates data and protects against accidental overwrite using an IDA such as Shamir Secret Sharing [105] to generate a set of pseudo-random shares or *carrier blocks* from a user's data blocks that are uniformly distributed throughout the free space of a device. These carrier blocks provide *combinatorial security* where an adversary must select the correct blocks out of the free space to reconstruct a valid data block. Adding redundant carrier blocks enables Artifice to repair itself when it is inevitably damaged by the public file system. This IDA-based approach and flexible block allocation allows the user to configure Artifice for use with a variety of public file systems and mitigate the effectiveness of a multiple snapshot attack.

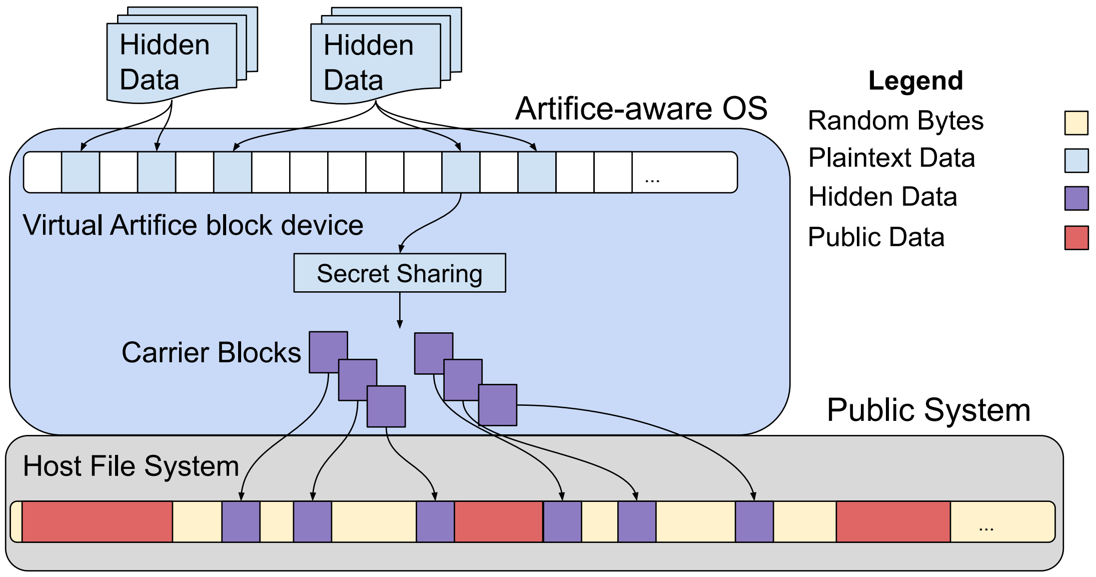
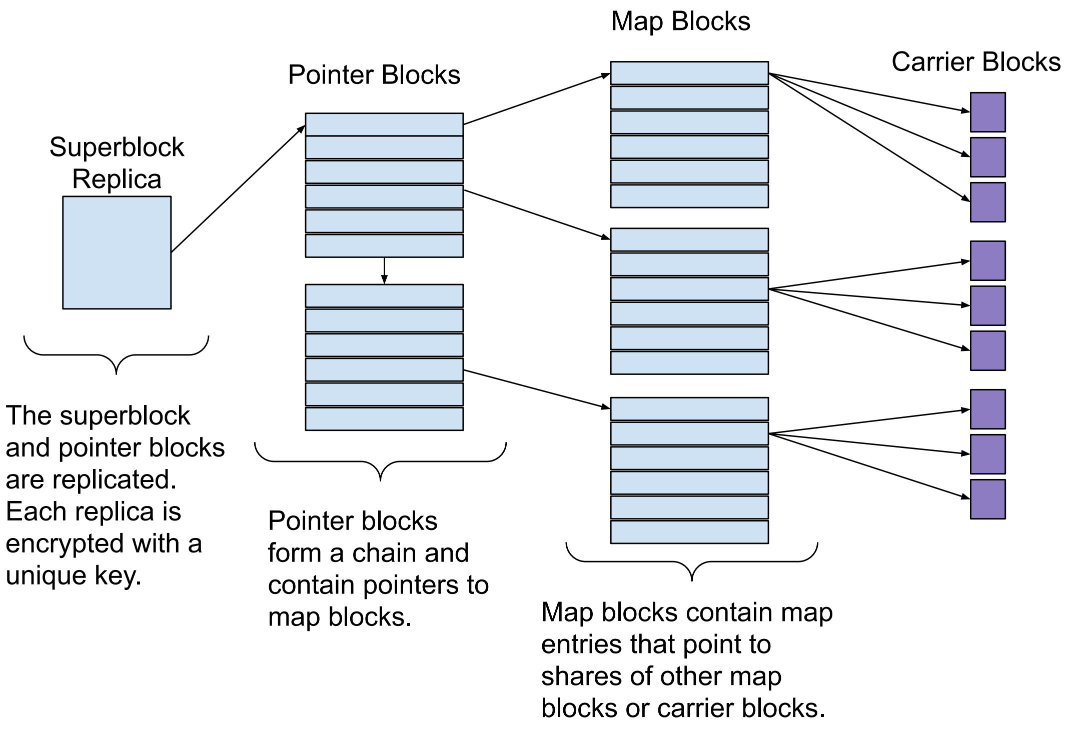


Figure 4.1: System overview of Artifice. The Artifice kernel module resides in a separate operating system contained on removable media. The public system includes the public file system that Artifice hides in and the public OS. Free space in the public file system should be filled with pseudo-random blocks.

Unlike previous approaches that require driver software to be installed on the user's device, a user accesses Artifice by booting a separate live Linux installation on a USB drive containing the Artifice driver. Isolating the driver from the public operating system prevents information leakage and protects the Artifice volume from most malware. Separating the hidden data from the driver software helps prevent the adversary from noticing the existence of the Artifice software and thereby inferring the existence of hidden data. It is also important to note that the user does not need to possess a copy of the bootable USB drive at all times; it would be advantageous for them to not be carrying it on their person when under the scrutiny of an adversary.

4.2.1 Artifice Map



The superblock and pointer blocks are replicated. Each replica is encrypted with a unique key.

Pointer blocks form a chain and contain pointers to map blocks.

Map blocks contain map entries that point to shares of other map blocks or carrier blocks.

Example Map Entry:

Artifice block number.	Carrier block hashes and pointers	Data block hash
0	<10, 0xBE>, <90, 0xA1>, <17, 0xD0>	0xCAFE

Figure 4.2: The design of the Artifice Map.

Artifice reads data by identifying the carrier blocks and entropy blocks associated with the logical block address through a metadata structure called the Artifice Map (shown in Figure 4.2). The Artifice Map is a multi-level tree that stores mappings from logical data blocks to physical carrier block locations and vice versa. The map is made repairable in the face of overwrites by our IDA scheme and is stored alongside the carrier blocks in unallocated space.

A hash of a user-specified passphrase is used to determine the location of a super block, which provides general information about the metadata structures and the locations of the Artifice Map carrier blocks (Figure 4.3). The super block is replicated to protect against overwrite and each replica is encrypted with a different unique key derived from the passphrase. The possible block offset of each replica is derived from a hash of the previous replica's possible location. Should a specified location be in use by a valid public block, the next possible location is used for that replica. A possible location is confirmed to contain a super block replica if the decrypted replica starts with a known value. If a possible location for a super block is found to not begin with the expected value when mounting an Artifice volume, the driver moves to the next location and repeats the process until a replica is found. The number of these replicas written to the disk is defined by the user. Artifice should be configured to generate more replicas of the super block than shares of the data blocks to provide a better probability of survival when faced with overwrites made by the public system. If Artifice sustains significant damage beyond its ability to repair, we can still recover some data if a superblock replica can be found. If all copies of the superblock are lost, then no data can be recovered from the hidden volume. Should a user need to revoke the passphrase, the super blocks are re-encrypted with a new passphrase and moved to new locations determined by the new passphrase.

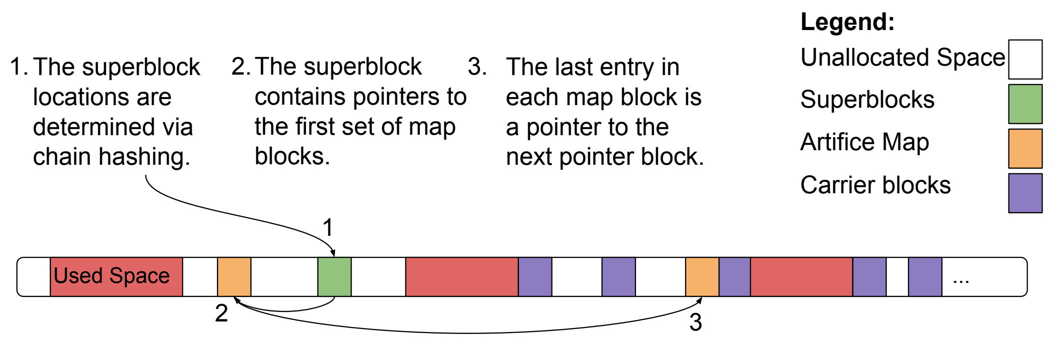

1. The superblock locations are determined via chain hashing.
2. The superblock contains pointers to the first set of map blocks.
3. The last entry in each map block is a pointer to the next pointer block.
Legend:
Unallocated Space
Superblocks
Artifice Map
Carrier blocks
Used Space
1
2
3
...

Figure 4.3: The process of locating Artifice super blocks through chain hashing a user's passphrase.

Each entry in the map contains a set of carrier block pointers, checksums of each carrier block, and a hash of the original data block that is used to verify the reconstruction succeeded. In the case of an encoding scheme that requires an external entropy source, identifying information about that entropy source is also included. These entries are arranged into *map blocks*.

To support information dispersal for the map blocks, a multi-level approach is used. Additional levels of map blocks are used to track the shares of the next lowest level of map blocks. This technique reduces the size of the top-level of map blocks. The location of the top level of map blocks are stored by a set of *pointer blocks*. Both these top level map blocks and the pointer blocks are replicated and encrypted much like the super block. Information about the pointer blocks is stored in the super block. When Artifice is running, map and pointer blocks are reconstructed, and a working set is cached in memory. As the map is modified by new writes, it is periodically flushed to the disk.

The size of this metadata structure will grow linearly as the size of the Artifice volume increases. Figure 4.4 shows the metadata overhead assuming 128-bit data block hashes, 16-bit checksums, and 32-bit block pointers. Metadata overhead for our proposed scheme in a virtual block device is relatively high, especially for information dispersal algorithms where additional metadata must be stored to enable decoding.

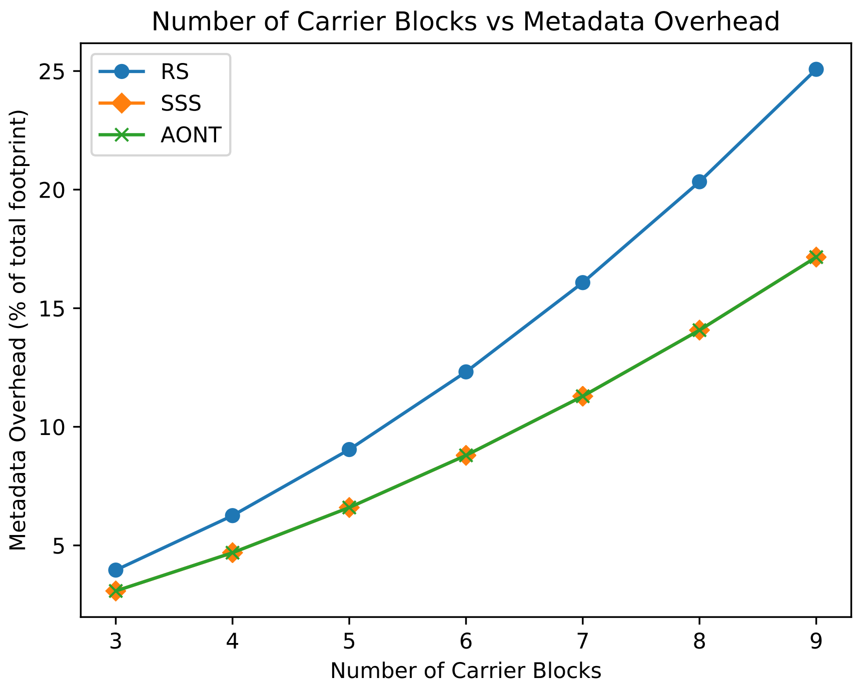


Figure 4.4: Metadata overhead as a percentage of total Artifice available space when implemented as a virtual block device. Metadata overhead is only dependent on the total number of carrier blocks.

The Artifice metadata structures can also be modified to allow it to run as a standalone file system instead of a block device driver. In this case our pointer block and map block structures would be modified to function as inodes, the data structure that stores metadata for a file or directory in Unix file systems and their relatives. These inodes would have to be replicated or encoded into shares with an IDA the same as normal Artifice metadata blocks. Instead of single data block pointers like in a normal file system (`ext4`), each inode will contain tuples pointing to the encoded carrier blocks that correspond to one data block out of the file that the inode addresses. The worst case scenario in terms of space used for this metadata approach is one file per data block in the volume. Whenever a file is updated changes would need to cascade up the chain of inodes for each file eventually to the super block which is still located using chain hashing. Allowing an unlimited length inode file similar to WAFL [50] would allow for efficient updates to the metadata and let it expand to just the size needed to support the files contained within a volume.

Such an approach would improve ease of deployment through tools such as File System in Userspace (FUSE) [70]. However, it would also introduce additional complexity through the need to implement and ensure the security of all the file system operations and semantics as opposed to simple read and write operations required for a virtual block device.

4.2.2 Information Leakage, Malware, and Hiding Driver Software

Since existing deniable storage systems inherently possess significant drawbacks we assume that it is unlikely for the average user to keep a copy of the driver on their devices thus making possession of such software suspicious. As a result, the presence of a deniable system's driver implies that the user's device contains a hidden volume. Detecting the driver software is perhaps the least computationally intensive way to detect a deniable system as it only involves inspection of the storage software stack, device firmware, behavioral characteristics, or partitioning scheme. In the case of some systems there is a significant performance impact for both the hidden and public volumes such that it would be simple to infer the existence of a deniable system [16, 20, 26] through a storage benchmark. While it is possible to hide such software through the use of a rootkit (an often self hiding piece of software used to gain root access to a machine) or other malware, this technique is "security through obscurity" and as such is unreliable.

With Artifice the driver software exists independently of the hidden volume on a separate device with no trace of the driver software left on the device containing the public and hidden volumes.

Since deniable storage systems coexist with the public operating system, challenges arise concerning information leakage through programs and hardware that interact with the hidden volume. In the case of TrueCrypt, Czeskis *et al.* [30] demonstrated that the system was plagued by information leakage through both the features of the Windows operating system and through temporary files generated by applications such as a word processor. This persistent system and application data, lacking a publicly visible reason for its presence, could be considered an indication of a hidden volume. It has also been made apparent by Troncoso *et al.* [119] that should an adversary install malware on a device to continuously leak disk traffic information, it is possible to reveal the existence and location of hidden files. Since our adversary can freely manipulate the user's device it is safe to assume that they can and likely will install some form of malware on the device to monitor the user's activities, including writes to any mounted storage devices. Currently, no known deniable storage system provides explicit protections against compromise through accidental or intentional information leakage through user applications or malware respectively.

To address the problem posed by malware and information leakage we restrict access to a deniable volume to a separate and secure operating system contained on separate bootable media. To access the hidden volume the user reboots the computer using the separate operating system, performs necessary operations on the hidden volume, and switches back to the public OS for normal use. By not accessing the hidden volume through the public operating system we do not expose the hidden volume to any programs capable of storing application data in the public volume and do not expose it to any malware that may have been installed on the public volume, but as we will discuss later, it does allow for the public file system to overwrite data in the hidden volume.

This should prevent malware installed by the adversary on the public operating system from leaking information about the hidden volume so long as the user does not use the public operating system to access the hidden volume. However, this approach does not protect against malware installed on a device at the firmware level [35].

4.2.3 Obfuscation and Redundancy through Information Dispersal Algorithms

As most deniable storage systems rely on hiding information among random data in the free space of a device there is a need for both obfuscation and some form of overwrite resistance. While most systems avoid the need for additional data integrity protections by preventing the hidden volume from sharing the same set of disk blocks as the public volume [80], these schemes often leak information about the hidden volume through either exposed metadata structures or significant differences between the advertised and usable capacity of a device.

Our strategy to solve both obfuscation and redundancy is to encode hidden data using an IDA and hide the resulting encoded blocks in unallocated space filled with random bytes generated by a secure-deletion-capable drive wiping utility or an encrypted file system. In this approach the hidden *data blocks* are processed through the IDA to produce a set of random appearing *carrier blocks*. These blocks will be written to the unallocated space that has been deniably filled with random bytes. Without knowledge of which carrier blocks correspond to what data blocks and with carrier blocks indistinguishable from other free space on the disk. To brute force this system an adversary is forced to reconstruct every possible combination of blocks on the disk. Through this we obtain combinatorial security as an additional provision alongside standard encryption techniques. Most IDAs can generate redundant shards of the original secret, much like standard erasure codes. As a result, the IDA allows a system to repair itself and protect against accidental overwrite by the public volume. This strategy has limits to how many blocks the hidden volume can lose before irreparable damage is done, if that happens the system fails securely by not revealing information to the adversary.

Information Dispersal Algorithms and Erasure Codes

With an IDA pieces of data are split into a series of shares, a subset of which are required to reconstruct the original data. This is similar to and in some cases the same as an (n, k) erasure code, such as Reed-Solomon [91], with which there are n symbols in the code-word and $k \leq n$ symbols are required to reconstruct the data. When applied to steganographic storage the most important aspects of an IDA are the ratio of write amplification to number of recoverable erasures and its ability to prevent the adversary from gleaning information about the plaintext from some number of shares less than the threshold k.

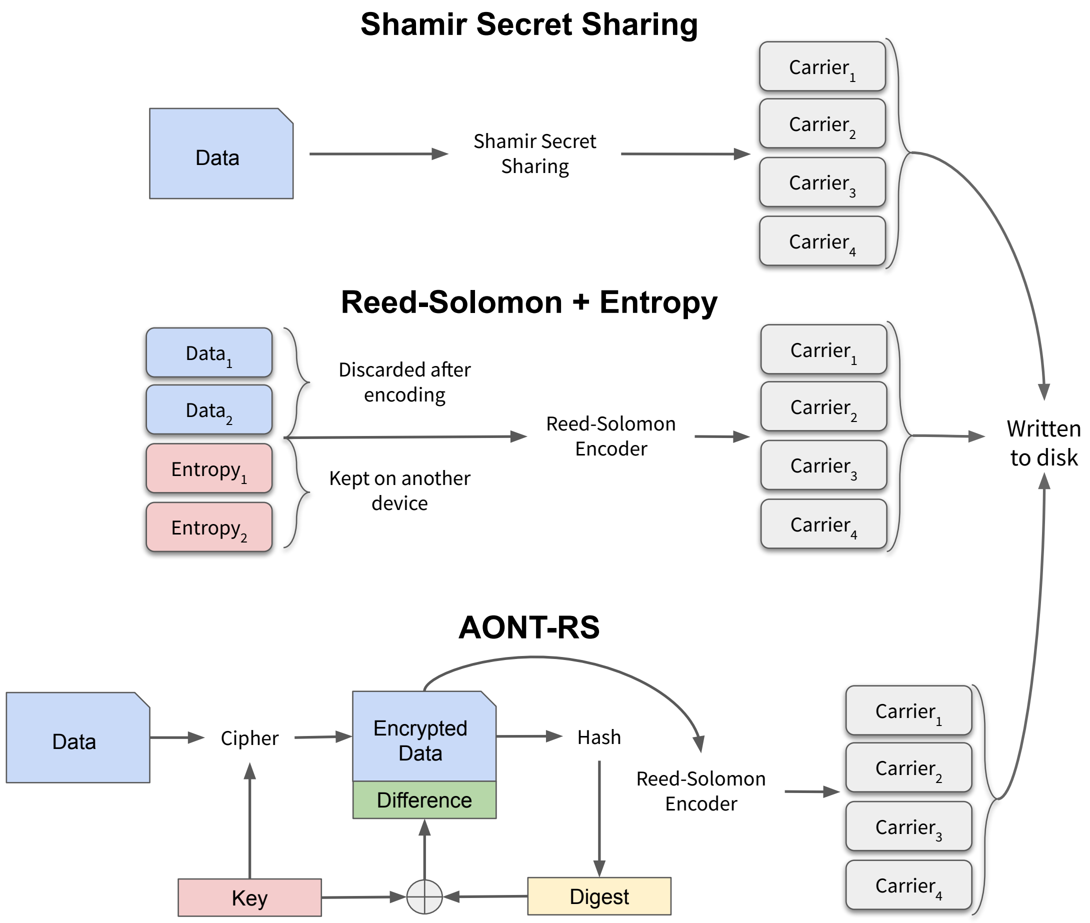


Figure 4.5: Share generation with AONT-RS (All or Nothing Transform and Reed-Solomon), Shamir Secret Sharing, and Reed-Solomon/Entropy.

Shamir Secret Sharing

One of the best known IDAs is Shamir's Secret Sharing algorithm [105]. In this scheme n shares are generated from the secret and any $k \leq n$ shares are needed to reconstruct the original secret. This relies on the fact that any set of k points will define a unique polynomial of degree $k-1$. The secret is used to create this polynomial of degree $k-1$ which is evaluated at n random points. When at least k points are known it is possible to reconstruct the original secret through polynomial interpolation. The benefit of this approach is that with any less than

k points, it is impossible to infer any information about the original secret. One share does not reveal any information about the potential contents of other shares. This set of characteristics provides us with information theoretic security.

AONT-RS

A similar but computationally secure threshold guarantee can be provided using the All or Nothing Transform (AONT) [93]. In standard symmetric encryption modes such as Cipher Block Chaining (CBC) it is possible to decrypt information one block at a time, so the adversary need not know the entirety of the data to reveal some plaintext. The AONT is an encryption mode that prevents decrypting a piece of information without possessing all of it. With the AONT a block of data D is encrypted with key K to produce ciphertext C. A cryptographic hash H is used to produce digest $d = H(C)$ that is equal in length to K. K and d are combined with a bit-wise XOR operation and the resulting difference is appended to the encrypted data to produce the AONT payload. To recover D the entirety of the payload is required. The digest d is recomputed from C and the digest is used to recover K from the difference. Combining this technique with an erasure code such as Reed-Solomon results in an algorithm that provides error correction abilities and a computationally secure threshold scheme [92]. Although theoretically weaker than Shamir's information theoretic approach, AONT-RS provides the advantage that the individual shares can be smaller than the input data. AONT-RS splits information of length L into n pieces with a threshold of k where each share is only of size $L/k \leq L$. This improved space efficiency for the same number of shares is a significant advantage when applied to deniable storage where the total footprint on disk relates to both overall deniability and reliability.

Reed-Solomon and Entropy

We have also explored a new IDA that combines non-systematic Reed-Solomon erasure codes with existing blocks of pseudo-random entropy data as a sort of one time pad. This algorithm will produce random appearing carrier blocks with a similar space efficiency to AONT-RS. As shown in Figure 4.5 our Reed-Solomon/Entropy approach uses a set of one or more data blocks and a set of one or more pseudo-random entropy blocks in the initial Reed-Solomon code-word. The generated random appearing carrier blocks are stored on the disk and the original data blocks are discarded. When it comes time to reconstruct the data any surviving carrier blocks and the entropy blocks are used to recover the data blocks. It is important to use a non-systematic Reed-Solomon code so that the original data blocks are not retained.

For example, if we have $d \leq k$ data blocks and $e = k - d$ entropy blocks, after encoding we are left with $m = n - k$ carrier blocks and e entropy blocks. The data blocks are discarded, and the carrier blocks are stored in the unallocated space of the file system. The entropy blocks are stored in a known, external location. If $m < e + d$, we require entropy blocks to reconstruct the original data. Whereas if $m \geq e + d$ we do not require entropy blocks to reconstruct. For example, if we have $d = 2$ data blocks and $e = 3$ entropy blocks resulting in $k = 5$, and if $n = 9$, we arrive at a set of $m = 4$ carrier blocks after encoding. Since the two plaintext data blocks are discarded and not written to disk, we are left with seven blocks that can be used to reconstruct the original data. Out of this set of $n - d$ blocks, only k are needed to reconstruct the original data. The numbers m, d, and e can each be adjusted by the user to provide more resiliency, performance, or security as desired.

Artifice has multiple ways to acquire high entropy data from deniable sources such as a user's DRM (Digital Rights Management) protected media files. The presence of which on a publicly visible file system is not suspicious. So long as $m < d + e$, without the entropy blocks, the original data is unrecoverable.

With an erasure coding scheme, it is possible to map multiple data blocks to a single pool of carrier blocks and provide improved space efficiency over secret sharing. However, we must weigh the advantage of improved space efficiency against the additional complexity and inconvenience of requiring additional entropy data.

Other IDAs

Rabin presented another IDA [88] that similarly to Shamir's scheme splits a piece of information into shares smaller than the original data like AONT-RS but through a different means than Reed-Solomon encoding. Although more efficient, Rabin's scheme provides weaker security assurances as without the random appearing encoded shares provided by Shamir's scheme an adversary can infer the contents shares that they do not possess [92]. The basic principles of Rabin's scheme are very similar to those employed by Reed-Solomon erasure codes [91].

There have been a few attempts at providing the space and computational efficiency of Rabin's approach with the security assurances of Shamir's. An example of this is Secret Sharing Made Short (SSMS) [65]. In SSMS the secret S is encrypted with a key K which is split using Shamir's scheme into a set of n shares $K_0, K_1, ..., K_n$ with a threshold m. The encrypted secret is split into n shares with a threshold of m using Rabin's scheme to produce $S_0, S_1, ..., S_n$. The result is a set of shares that are significantly more space efficient than Shamir's scheme alone. The downside is that this approach can only be considered computationally secure much like AONT-RS and does not provide any additional benefits as far as space efficiency.

Combinatorial Security

One interesting aspect of IDAs is the possibility of relying on combinatorial security either in addition to or in place of a conventional encryption type scheme that protects data with some secret known by the user. While this does save us the hassle of worrying about key management a combinatoric guarantee can only be provided given certain preconditions regarding the size of the Artifice volume, the size of the free space we are hiding a volume in, and the number of shares written to the disk.

For instance, if we assume that the adversary cannot determine which unallocated blocks contain hidden data the time needed for a brute force attempt to reconstruct the volume is $O(n!/(k!(n-k)!)$ where N is the total number of unallocated blocks on the disk. In this case the threshold k can be considered constant, or limited to a small range of realistic values, so the computational complexity can be simplified to $O(N^k)$. While polynomial time does not necessarily provide a strong security guarantee on its own, the number of blocks N can be quite large. In the case of a 1 TB disk with 512 GB of free space there are 2^{27} 4 KB unallocated blocks. If we assume each data block is divided into a set of 12 carrier blocks with a reconstruction threshold of 8 then there are $(2^{27})^8$ or 2^{216} possible combinations for reconstructing each data block.

In this scenario, even if an adversary can determine which blocks contain hidden data, a brute force attack is still infeasible. Using the same secret sharing parameters as the previous example and assuming our Artifice volume has a footprint of 8 GB or 2^{21} 4 KB carrier blocks, there are still $(2^{21})^8$ or 2^{168} possible block combinations for the adversary to attempt. Each data block can be reconstructed through $\binom{12}{8} = 495$ different combinations. With this and the approximate size of the volume we can approximate that out of the 2^{168} block combinations, on the order of 2^{27} combinations will yield a valid data block.

There is of course a trade-off between resilience, security, and performance but using an IDA allows for a measure of tunability. The larger the threshold k required to rebuild the data, the more secure the system as the number of possible block combinations has increased. However, there is a minimum threshold for a reasonable combinatorial guarantee based on the size of the Artifice volume, size of the free space, and the number of shares. Should we have too little free space, too few shares per data block, or too small of an Artifice volume the number of combinations the adversary must try becomes a feasible, if still expensive task. For example, an Artifice volume of 1 GB with four shares per block presents $(2^{18})^4 = 2^{72}$ possible combinations. Shrink this to a 256 MB volume and you have 2^{64} possible combinations. This is still a difficult problem but possibly feasible given past successful efforts at brute forcing encryption keys [1]. These small volume sizes become a significant concern when one keeps in mind that the smaller the footprint the easier it is to hide sensitive data.

Luckily both our proposed Reed-Solomon + Entropy scheme and AONT-RS can rely on some additional provisions to ensure the security of the encoded data either encryption or entropy blocks. As such, Shamir Secret Sharing is the algorithm that stands to benefit the most from combinatorial guarantees as any correct selection of blocks can reveal data without the need for additional information. This should be considered when selecting an IDA for use with a deniable storage system. If the volume is not large enough or there is not enough free space to provide a combinatorial security guarantee using Shamir Secret Sharing would be ill-advised. In the case of our other two IDAs combinatorial security provides an additional layer of security alongside other mechanisms.

Overwrite Resistance

To avoid the security pitfalls of previous approaches, a deniable volume must be designed to exist in an environment where accidental overwrite is a constant threat to data integrity. Some approaches utilize basic replication [74] but this leads to significant write amplification. This gives the hidden volume a significant footprint on the disk which requires significantly more free space in which to store data. This increases the chances of accidental overwrite and changes more blocks for each write which can make detection more likely. We believe that a deniable storage system must be able to efficiently repair itself and recover from a limited amount of damage dealt by the public volume. We aim to achieve this using the same IDA scheme to obfuscate the data. Most IDAs can produce more redundant shares than the number needed to reconstruct the secret. It follows that they can reconstruct the original secret and regenerate the overwritten shares. The system can remap the reconstructed shares to new locations on the disk, restoring the full erasure correcting capability of the volume. By adjusting the total number of shares and the reconstruction threshold along with careful placement of the data the deniable system can adapt to different rates and distributions of overwrites.

The primary drawback of this approach is that hidden data survival is probabilistic. We can minimize the probability of catastrophic data loss through either maintaining a small hidden volume relative to the size of the public system's unallocated space or through the user consciously limiting the amount of data they write to the public volume between repair operations. We have theoretically and empirically evaluated the effectiveness of multiple IDAs with a variety of hidden volume sizes, amounts of unallocated space, and the amount of data written to the public volume per repair operation. Additionally, we have developed methods by which Artifice can determine the appropriate settings for the IDA.

However, Artifice would still be vulnerable to the “fill to full” attack [37]. We consider the result of silently losing the hidden volume to be a secure failure as it does not compromise the confidentiality of the hidden data or alert the adversary to the presence of the hidden volume.

4.2.4 Deniable Writes to Public Free Space

Of significant concern for previous systems among the is the multiple snapshot attack in which the adversary can capture images of the disk and infer the existence of a hidden volume through analysis of the changes. To tackle this problem there must be a level of plausible deniability for the inevitable changes made to the unallocated space of a public volume when data is written to the hidden volume.

Current strategies to provide a provable defense against a multiple snapshot attack inevitably weaken the system against attacks far simpler than snapshot analysis. As we have discussed in Sections 2.2 and 3.1, ORAM-based approaches [16,20] primarily compromise information leakage resistance and leave the deniable system’s driver open to the adversary. As a result, we have assumed both of these weaknesses are less resource intensive for an adversary to exploit than analyzing disk access patterns from a limited number of snapshots. Alternate approaches rely on the system constantly writing dummy blocks to locations on the disk independent of writes to the public or hidden volumes [80, 131]. Additionally, outside of Troncoso *et al.*’s work on traffic analysis attacks [119], there is a lack of empirical evaluation for the effectiveness of snapshot attack countermeasures.

We instead prioritize securing the system against information leakage and hiding the driver software over provable snapshot analysis countermeasures while still providing some defenses which rely on providing deniable reasons for the changes in a disk’s free space.

The first solution is rooted in operational security. Avoiding the scenario of a multiple snapshot attack is the most foolproof way to defeat it. When an adversary gains access to the device, the user must assume that either a snapshot has been taken or malware installed. The easiest and most reliable response is to replace either the whole device or the disk, or deniably scramble the contents of the disk rendering the previous snapshot meaningless. With any data already contained in the public and hidden volumes copied to the new device or scrubbed disk, there is nothing for the adversary to meaningfully compare to previous snapshots. In the case of a mechanical disk, a defragmentation operation between two snapshots would render the first meaningless and provide a deniable reason for the changes. Only then would data be written to the hidden volume. Although relying on operational security is ideal, it will not always be practical for a user to take such relatively drastic measures.

Another approach is to write data to a portion of the disk where the contents change frequently or recently. This reduces the problem to selecting suitable blocks for storing hidden data. Artifice would be limited to writing new hidden data only to blocks that have been freed by the public file system after the most recent opportunity for the adversary to take a snapshot. To accomplish this Artifice would store an allocation bit vector describing which blocks are in use as of the last hidden data access. When it is next initialized the current state of the disk would be compared to the previous state. Since these "hot" regions on the disk change frequently there would be a deniable reason for changes in the free space. A drawback is that hiding data in frequently changed sections of the disk increases the probability of overwrite. We would need to store larger sets of carrier blocks to provide a reasonable probability of survival. Lastly for this approach to be feasible the user must be sure to delete enough data to provide free blocks before writing to an Artifice volume.

Even our best efforts at rendering write patterns indistinguishable still run the risk of missing something that the adversary notices as has happened with multiple attempts to disguise TOR traffic by mimicking the behavior of another type of network [52]. As such while we implement and propose defenses for Artifice (see §7.1 it would still be advisable to take an operational security approach like the one we have proposed to maintain deniability more reliably.

Deniably Random Free Space

Tangentially related to the problem of deniable writes is how we prepare the free space of the device to receive those writes. The primary challenge is that filling the unallocated portions of the disk with unexplained pseudo-random information could be considered suspicious. Consequently, there must be a deniable reason for the free space of an existing file system to be filled with pseudo-random bytes.

One possible solution to this problem is maintaining a large amount of deleted random appearing files such as compressed archives to fill unallocated space with random bytes. Although there is the possibility of statistical tests showing hidden data as being "too random" or otherwise standing out from the surrounding blocks. Such a mismatch could be detected using widely available randomness testing programs such as the NIST Statistical Test Suite (STS) [97]. The most reliable way to produce large amounts of sufficiently random information is to rely on cryptographic ciphers. Although we must assume that the user would surrender keys for a file system or whole disk encryption system and that the free space could be decrypted revealing our obfuscated hidden data. To avoid this, we must render the hidden space undecipherable. The use of a secure drive wiping utility that encrypts data and discards the key would accomplish this but the number of random blocks on the disk would decrease as data is written through normal use. The use of an encrypted file system that deletes information securely by overwriting the key material used to encrypt a block or file would result in deniably random unallocated space that replenishes itself through normal operations. We discuss how to address this challenge in §4.3.

4.2.5 Flash Considerations

SSDs create a set of different issues for Artifice as flash technologies possess different technical characteristics and features when compared to traditional mechanical disks. The primary culprit in this problem is the FTL which can mark hidden blocks as written which leaks information about the location and presence of hidden data. Alternatively, the FTL may unknowingly erase hidden data as part of opaque and non-standardized garbage collection operations if is unaware of the hidden volume's writes which presents another possible source of hidden data overwrites.

This layer of abstraction presents a hurdle for deniable storage systems. Many designs seek to address these challenges by operating on raw flash devices [83] or are intended to be implemented as drive firmware [131]. Since custom firmware could be suspicious in a similar way to publicly visible drivers and raw flash devices are still relatively uncommon, Artifice primarily addresses these challenges through other means.

TRIM

Most modern file systems support the TRIM function, which notifies an SSD that certain blocks are no longer in use by the host, and thus need not be copied to new locations during garbage collection. Ideally for the public file system, the hidden data would be TRIMmed, therefore marked as unallocated by the SSD, and would treat garbage collection operations as another form of accidental overwrite. However only one kind of TRIM (Non-deterministic TRIM) possibly allows access to the original data after a block has been subject to TRIM. When reading from TRIMmed blocks the SSD could either return the original data or some other information if the block has been subjected to garbage collection.

The other two types of TRIM are far more damaging for a deniable storage system, Deterministic Read After TRIM (DRAT) and deterministic Read Zero After TRIM (RZAT). Both will return a consistent pre-defined value for any logical block address that has been TRIMmed. In this case it would be necessary for a deniable storage system to leave all of its blocks listed as allocated on the SSD and therefore vulnerable to forensic analysis. Additionally, deterministic trim will cause most, if not all, free space on the device to appear uniform, eliminating the ability for a deniable storage system to hide within pseudo-random free space.

The challenge posed by TRIM is somewhat mitigated by the fact that most operating systems utilize *periodic TRIM*, where the operating system will periodically send a TRIM command for all blocks deleted after the previous TRIM operation [6, 116]. This is viewed as preferable to *continuous TRIM* where a TRIM command is sent to the disk each time a file is deleted. The common use of periodic TRIM allows for a small region of accessible untrimmed free space on an SSD between TRIM invocations. It is unknown whether the size and lifespan of this region are sufficient to address the problem that TRIM presents.

To address this issue, we have concluded that TRIM should be disabled when a device contains a hidden volume. Fortunately, it is common to disable TRIM if using a drive encryption system as it could leak the locations of the unallocated blocks and reveal the possible size of stored data [18, 109, 120]. In the case of a deniable storage system disabling TRIM is an ideal choice as we need not worry about hiding data in blocks that would be altered by TRIM. Effectively causing the SSD to behave like a mechanical disk from the perspective of Artifice and the public file system.

NVMe and Zoned Namespaces

Multiple reasonable alternatives to FTL based SSDs have emerged, such as open channel SSDs. A related recent development has been the move towards the adoption of Zoned Namespaces (ZNS) [15]. This new approach breaks an SSD up into a series of sequentially written and host controlled "zones" that are written sequentially like a log-structured file system. Zoned block device support is already included in the Linux kernel through software such as `dm-zoned` and `F2FS` [66,72]. Zones behave similarly to segments in a log-structured file system or erase blocks on a flash device. Artifice could extend its block allocation functionality to support hiding data in deleted blocks of zones that have not yet been garbage collected by the file system or `dm-zoned` if those zones already contained pseudo-random or encrypted information.

There are two possible routes Artifice can take when interacting with zoned namespaces. The first is where Artifice is oblivious of the fact that it is operating on a zoned device by running the Artifice block device on top of an existing `dm-zoned` device. This is the simplest approach that eliminates the need for any additional functionality in the Artifice driver. The second technique involves Artifice working in concert with a secure deletion utility as previously described in §4.3. Artifice data would be written normally to a series of zones and would be marked as deleted in the same manner as with a secure deletion utility on a conventional drive. It would be best to spread carrier blocks from the same data block tuple across multiple zones to increase the probability of survival should an entire zone be deleted.

While zoned namespaces and other host controlled flash devices present compelling options for bypassing the challenges posed by FTL controlled flash, they have not yet achieved wide adoption and some standards like Zone Namespaces have not yet been integrated into commonly available consumer hardware.

4.3 Secure Deletion and Steganographic Storage

As previously stated in §4.2.3 Artifice, and most other deniable storage systems require some deniable justification for unallocated space to be filled with undecipherable pseudo-random blocks. If discovered, unexplained random blocks would be considered suspicious by the adversary and provide justification for further investigation. Without pseudo-random free space Artifice blocks will be relatively easy to spot. Our proposed solution was the use of a secure deletion utility that encrypts data and throws away the key to render the data undecipherable. This technique is commonly used to wipe drives before disposal and by modern self-encrypting drives (SEDs).

With a secure deletion utility, the user would first wipe a drive before use and then create a hidden volume within the free space of the volume. If the device allows in-place updates and has not been given any TRIM commands the device will contain pseudo-random blocks throughout its unallocated space. While this technique is relatively simple to implement it is not without its drawbacks. The biggest of which being that limited number of random free blocks. As the device is used and blocks are freed by normal use the number of random blocks in the unallocated space will shrink, reducing the space available to Artifice to hide in. Additionally, it is unclear how this technique would fare when used on an SSD due to overprovisioning and the behaviors of the FTL.

As an alternative to using a secure deletion utility the user could decide to wipe a drive by overwriting it entirely with an Artifice volume with a significant number of redundant blocks generated. Then the user could write normal looking cover files onto the disk. Since blocks encoded with Artifice IDAs would produce psuedo-random appearing blocks it would appear as if the user had simply used a secure deletion utility at some point in the past. The high redundancy factor would then provide Artifice ample ability to absorb overwrites until the data could be extracted. While this approach places data integrity and deniability of random information at the forefront it limits the user's ability to write new data to the Artifice volume over time and limits the user's ability to perform repair operations on the data once it is written to the device. As such this method would only be advisable when a piece of information only must be hidden for a short time.

A more ideal solution to this problem would be to utilize a technique that would automatically generate new pseudo-random free blocks as the file system is used. Such a system would be an encrypted file system that deletes data by discarding an encryption key.

4.3.1 Steganographic Storage through a Secure Delete File System

To demonstrate how Artifice would interact with a secure delete capable file system we will assume a theoretical system that fulfills the following requirements.

- The file system securely deletes data by provably forgetting an encryption key corresponding to this data. Ideally, these keys would be at a finer granularity than the entire file system such as a key per block.

- The primarily used version of this file system must have no capability of interacting with a hidden volume to avoid suspicion of the software itself.

To explore how Artifice would be modified for use with such a system we will be using a system called *Lethe* which provides file system encryption and efficient secure deletion. With Lethe keys are generated for individual blocks through a Keyed Hash Tree (KHT) [68]. With a KHT, individual block keys can be derived from a single root key. Each file possesses its own list of KHTs where generated leaf keys are used to encrypt individual blocks from a set of root file keys. These root file keys are stored in a metadata file similar to systems like WAFL and ZFS [50]. The root file keys in this key file are managed and encrypted with their own list of KHTs like any other file. By using KHTs Lethe reduces the number of keys that must be forgotten to only the root key for the file system and avoids re-encrypting data with new keys. Instead of changing the root key with each deletion operation it is done periodically with each file system *epoch*, some length of time or number of writes after which we roll over to a new root key. When the root key is changed any keys or sub-trees for still valid files or blocks are rolled forward to the next epoch. Anything not carried forward is securely forgotten, as is the data encrypted with the forgotten key assuming the system can provably forget a single root key. Eventually data will need to be re-encrypted with new keys once the KHT lists reach some level of fragmentation caused by carrying forward old keys to a new epoch. This is addressed through garbage collection operations that lazily re-encrypt and remap data with new keys to defragment KHT lists.

There are two ways that Artifice could interact with Lethe. In the first approach Artifice treats Lethe like any other file system and hides within unallocated blocks. The second method of utilizing Artifice alongside Lethe involves writing data through the encrypted file system, deleting said data, and preserving the relevant keys elsewhere. Data would be written to a volume encrypted with Lethe using the IDA techniques that Artifice normally uses. These newly written blocks and their corresponding inodes will then be “deleted” with the user being sure to maintain a copy of the root encryption key for those blocks elsewhere. An IDA would still be necessary to protect against accidental overwrite as the “deleted” blocks are reused by the file system. We must be sure to only write to this hidden volume using another operating system like the live disk used in a conventional Artifice deployment. This is to maintain the same isolation that ensures resistance to information leakage and malware. The primary challenge of this technique is that the saved key(s) used to decrypt the Artifice blocks must be stored somewhere. These could be stored using more conventional steganographic techniques such as embedded in an image or hidden in a smaller Artifice volume on another device. Ideally only the root key for the entire file system corresponding to the epoch in which the Artifice data was written is needed to read the Artifice volume.

To modify Artifice for use alongside Lethe as described in our second method we can drastically simplify the Artifice driver. Instead of existing as a virtual block device Artifice could instead be a simpler program that takes a set of files the user wishes to write to a hidden volume and writes them to a Lethe file system. This streamlined Artifice version would still need IDAs and metadata structures to provide redundancy, but the obfuscation provided by Lethe’s encryption system would be sufficient. The most important function of this Artifice variant is that it must record which blocks it has written to and return a copy of the corresponding keys to the user. To recover data the user would provide Artifice with the root keys corresponding to the hidden data, decrypt the relevant blocks, perform error correction to account for overwritten blocks, and provide the reconstructed and decrypted data to the user.

Either approach automatically provides a replenishing set of apparent pseudo-random blocks with a deniable reason for their existence on the device. The second method has the the benefit that the write patterns will match what is “normal” for the public file system, especially if it is a file system that utilizes append-only or read-modify-write operations.

Since Lethe would perform garbage collection operations to mitigate KHT list fragmentation Artifice must contend with an additional source of overwrites by an unknowing file system. As a result, it may be necessary to increase the redundancy of Artifice IDA configuration or decrease the frequency at which garbage collection is run to minimize the number of overwrites for a given duration.

A drawback of this approach is that we would need a modified version of the Artifice driver that interfaces directly with Lethe. This could make use of an ordinary Lethe instance suspicious to an adversary if they knew of a version of the Artifice driver tailored for use with Lethe. Also when hiding data by writing into the Lethe volume, instead of a single passphrase we have a set of one or more root keys we must keep track of. Changing access credentials for the hidden volume is also more complicated and expensive than with a conventional Artifice deployment as we would have to re-write the hidden data to new locations in a different epoch and save new keys.

4.4 Summary

To defend against our adversary described in Chapter 3 we have developed Artifice, a plausibly deniable virtual block device. Artifice defends against malware and mitigates information leakage from the hidden volume through isolating the hidden volume from the public system. A user accesses Artifice through a separate bootable drive that contains the Artifice driver. The public OS is not running, and the public volume is not mounted. Reducing the chances of information leaking from one volume to the other and preventing malware present on the public volume from running. This isolation also allows us to keep the driver software and hidden volume separate, which helps to hide the existence of the driver software.

Artifice utilizes IDAs to both provide redundancy in the face of the public volume overwriting Artifice blocks and obfuscates the data to render it indistinguishable from other random blocks in the free space of a device.

To mitigate the compromising characteristics of flash devices we propose disabling TRIM when using an Artifice volume, as is commonly done with disk encryption systems.

Secure deletion through encryption provides us with a deniable reason for random blocks in free space that Artifice blocks hide in. We extend this by considering how we can modify Artifice to make the most of a file system called Lethe (§4.3) that performs secure deletion as a default behavior.

Chapter 5

Implementation & Operational Security

Core to our objective of designing *usable* deniable storage is to implement our proposed design and evaluate it against our design criteria. To demonstrate and test its viability we have implemented Artifice as a loadable kernel module intended to be run from a Linux live flash drive. With an implementation of Artifice we empirically evaluate how well it meets our design requirements, benchmark its performance, and draw up an operational security model for its use.

5.1 Implementation

Artifice uses the Linux device-mapper framework [71] to present the user with a virtual block device. This block device maps block IO operations from logical data blocks to carrier blocks that are written into the free space of an existing file system. As with most device-mapper targets, Artifice can be layered with other device mappers such as `dm-crypt` and `dm-zoned` to modify its behavior. Also like other block device drivers, our implementation of Artifice must be formatted with a file system before it is used like a regular physical disk. While Artifice can be implemented as a file system instead of a virtual block device, choosing the latter reduces the number of operations Artifice must handle to only block reads and writes and simplifies the required metadata structures. That said, a file system built on a user-space pass-through system like FUSE (File System in Userspace) would eliminate the added complexity of a loadable kernel module and allow easier deployment on other Unix-like operating systems such as FreeBSD or MacOS. It is important to note that by keeping Artifice at the block device layer or higher in the storage stack, we are able to eliminate the need for the specialized hardware or firmware that some previous approaches require and allow Artifice to run on common commodity devices.

The architecture of the current implementation of Artifice, as shown in Figure 5.1, is easily extensible to support multiple public file system types and currently supports `ext4` [5] and `FAT32` [3] and we plan to support `NTFS` [2] and `APFS`.

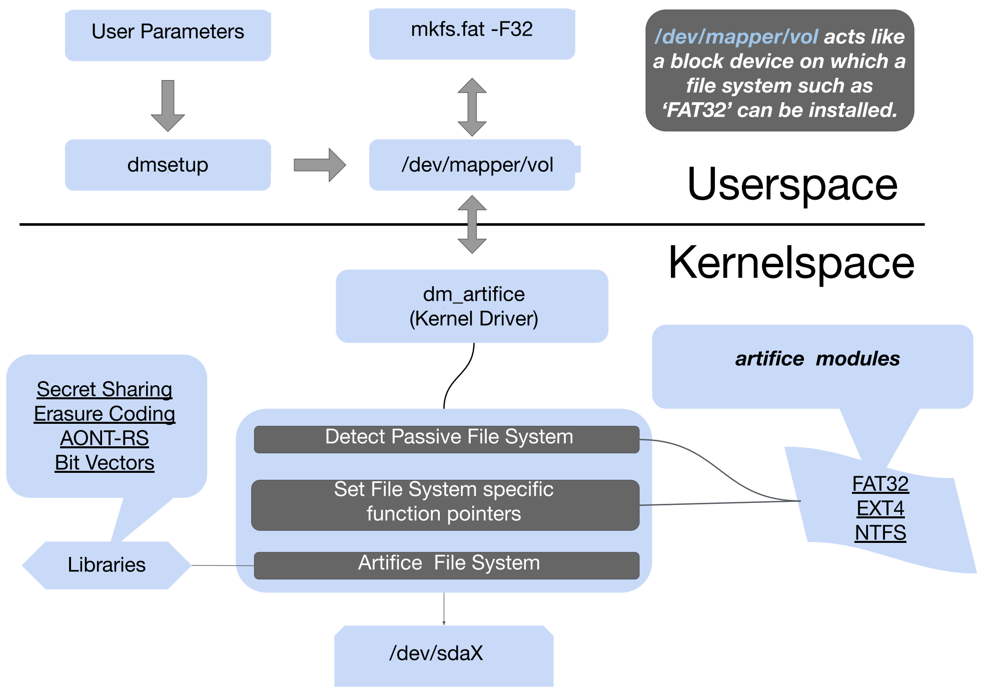

User Parameters
mkfs.fat -F32
/dev/mapper/vol acts like a block device on which a file system such as 'FAT32' can be installed.
dmsetup
/dev/mapper/vol
Userspace
Kernelspace
dm_artifice (Kernel Driver)
artifice modules
Secret Sharing
Erasure Coding
AONT-RS
Bit Vectors
Detect Passive File System
Set File System specific function pointers
Artifice File System
FAT32
EXT4
NTFS
Libraries
/dev/sdaX

Figure 5.1: Architecture of the Artifice block device driver.

This implementation is designed to utilize Shamir Secret Sharing [105], non-systematic Reed-Solomon erasure codes [91], or systematic erasure codes combined with an all or nothing transform [92] as IDAs to generate carrier blocks. All of which provide an (n, k) scheme where at least k carrier blocks out of a set of n are needed to reconstruct the original data and no data is recovered if fewer than k blocks are available.

For its IDAs, Artifice includes a variant of the libgfshare [108] Shamir Secret Sharing library ported for use in the Linux kernel, an AONT-RS library built on a SIMD-optimized Reed-Solomon library and the SPECK symmetric cipher [14], and a Reed-Solomon/Entropy library using the same SIMD library. The current implementation operates on 4 KB logical blocks as it is a common default block size for file systems such as `ext4` and because it is the default Linux page size. Block checksums use a modified version of the cityhash library [41] and the passphrase is hashed with `SHA256`.

5.2 Operational Security Model

Operational security is an often-overlooked aspect of deniable systems and something that we must consider in the design phase. Exploring the procedures and operational circumstances surrounding the deployment of such storage systems are critical and likely to be the deciding factor in the actual security of the system. In the case of Artifice we must consider the deniability of the driver software, the security of the passphrase, the physical security of the device, and the measures that should be taken if any of the components necessary to use Artifice are compromised.

As mentioned previously, Artifice aims to provide deniability for the storage system's driver as the presence of a relatively uncommon and suspicious program on the user's device would imply the existence of a deniable volume. Keeping the volume and software on separate devices assists in this but is still heavily reliant on the user's practices. We assume that the deniability of the software is tied to its probability of being present on an average user's device. The more common and innocuous the software is, the less suspicious it is to our adversary. Due to the many drawbacks inherent in deniable storage systems, we can assume that a user will not have it installed unless they intend to maintain a hidden volume. Many previous solutions assume that if a deniable storage system was to be included in some other common software package, such as the Linux or Android kernel [22, 109], the user would have a deniable reason for the driver's presence on their device. The flaw in this approach is that there would have to be enough machines with the software sitting unused to make its presence appear innocuous. We consider it unlikely for a maintainer of a mainstream piece of software to bundle such a niche application into a release due to the relatively narrow use case and significant compromises that deniable storage systems must make.

When designing operational security procedures for a system we must first consider what components must be secured. With Artifice we must at minimum secure the passphrase to the hidden volume, the device the volume is stored on, and the driver software to ensure that the hidden volume remains secure and deniable. An important question of how to keep Artifice deniable is what the user should do when one or more of these components is compromised. These procedures are summarized in Table 5.1.

One of the more critical components to secure is the passphrase to an Artifice volume. Artifice does not persistently store any material related to the passphrase on a device. To further improve security multiple factors can be utilized alongside the passphrase to locate and secure an Artifice volume. If a passphrase is compromised by an adversary Artifice provides an easy means for revoking and replacing a set of credentials. When a new passphrase is chosen Artifice will determine new locations for the super block and pointer blocks. These will be remapped to those new locations and encrypted using the new passphrase. The old super block replicas and pointer blocks will be overwritten with random information. If the user can change the passphrase before the adversary gains access to the storage volume, they will be unable to find the hidden volume using the old passphrase.

In an ideal scenario, the user would ensure that they do not possess a copy of Artifice on a live disk when the adversary is most likely to inspect their device. This assumes the user has discarded their original live disk and made arrangements to obtain a copy once the present danger has passed to access or repair the volume. To carry out this more secure procedure the user in possession of the hidden volume may need to coordinate with multiple other individuals. When this is not practical the user could fall back on classic steganographic techniques such as hiding data in the lower order bits of images to hide the software on the live disk. This approach would require less overhead as the driver is significantly smaller than the hidden volume. Additional options to avoid exposing the driver include downloading it through secure means like TOR [33] or an HTTPS secured website and carrying it on an easily concealed MicroSD memory card. Should the user carry the live disk on their person without additional measures to hide the software and it falls into the hands of the adversary we must be concerned about the adversary escalating efforts to monitor the user's actions and we must assume the adversary knows that the user is possibly in possession of a hidden volume.

Perhaps the most complicated factor to consider is the device the user is keeping the hidden volume on. We have a few different kinds of devices to contend

Table 5.1: Possible scenarios and remedies for the compromise of different Artifice components.

✓: The item is still in the user's possession or not compromised by the adversary.
×: The item has been lost or compromised.
-: Irrelevant to the scenario.

Passphrase	Device	Live Disk	Loss of Security	**Description**	**Remedy**
×	✓	-	no	Passphrase is compromised.	The passphrase should be changed.
✓	×	✓	no	Device containing the volume is compromised.	Change the passphrase, deniably scramble information on the disk, or destroy the device.
✓	✓	×	no	Disk containing the driver is compromised.	The user should be concerned about the adversary escalating monitoring efforts.
✓	✓	✓	no	Surveilled by the adversary while accessing Artifice.	The user should be concerned about the adversary escalating monitoring efforts.
×	×	-	yes	Passphrase and the device are compromised.	If the adversary knows that the password is to an Artifice volume, it is advisable to destroy it as it is compromised.
✓	×	×	yes	The device and live disk are compromised.	If the user is found in possession of the device and live disk simultaneously then the adversary can obtain the passphrase through coercion.
×	✓	×	no	Passphrase and live disk are compromised.	If the passphrase and live disk are compromised then the Artifice volume should be destroyed.
×	×	×	yes	All the components are compromised.	There is no action that will protect the user or hidden data.

with that influence what the user must keep in mind when using a deniable storage system. The simplest of these is a mechanical disk. This disk could either be a boot disk or an external disk kept separately from the user's machine. We must also consider whether or not this device is one that uses flash. Flash introduces another dimension of possible device consideration. Some flash devices like compact memory cards or USB flash drives often lack an FTL and can be treated as an ordinary mechanical disk. Should the drive have an FTL the user should consider disabling TRIM or utilizing other possible countermeasures discussed in §4.2.5.

A significant operational concern for the device used to store and/or access the hidden volume is malware. Should the adversary gain physical access to the user's device we must assume that they have installed malware to monitor the user's actions and taken a snapshot of the disk. Isolating Artifice to a separate operating system helps protect against leakage through OS-level malware, although Artifice cannot help protect against hardware or firmware level malware. If it is suspected that the adversary has tampered with or compromised the device the safest option is to destroy and replace the device. If this is not possible, the user could scramble the information on the disk through an operation such as disk defragmentation or system cleanup utilities and change the Artifice passphrase. While the best solution to malware is to scrub the device or replace it there can be an advantage to leaving it be if the malware cannot impact or observe Artifice operations. This is because the adversary may consider it suspicious if the device no longer contains malware when they next get the opportunity to interact with it.

If all the components of the system are compromised there is little that can be done to maintain security. Although if the user possessed multiple hidden volumes and only the passphrase for a subset of these are compromised then they may be able to satiate the adversary by revealing one of the volumes. It is unknown if this would be a viable strategy because we have assumed the adversary would be aware of this capability and may continue pressing the user for more information.

In addition to developing procedures for the compromise of each component, the user must also remain aware of specific Artifice behaviors such as the integrity of the hidden data. The user must remain aware of their activities on the public system and how much data is being written to the disk to ensure that the hidden data survives. For instance, if the user cannot take an operational security approach to mitigate the threat of multiple snapshot attacks (§4.2.4) then we must manage the deniability of the volume, e.g. the number of disk updates, versus how much we want to store and how much the public file system is writing. Ideally, the user would be able to minimize how much the public file system is writing once hidden data has been stored on the device. This is more controllable when the user is writing data to an external drive that does not see writes from the operating system or user applications.

Perhaps the most difficult aspect of using a deniable storage system is that the user must be aware of the level of scrutiny the adversary has placed them under. If the adversary has already targeted the user for closer observation the number of options available to ensure the secrecy of a hidden volume becomes significantly limited. If the user is being closely tracked by our adversary, the movements may provide sufficient reason for an adversary to escalate to coercion at the next available opportunity. If the user suspects that the adversary has paid particularly close attention to them the safest course of action would be to not attempt to use Artifice and destroy any device that may have stored a hidden volume or suspicious software.

5.3 Performance

In general, we consider the performance of a deniable storage system a low priority. Artifice is *not* a high-performance system; its ultimate goal is protecting the user and only requires sufficient throughput to process small amounts of information. That said, performance must be sufficiently fast so that Artifice does not become a dangerous hindrance to the user as is the case with some previous systems [16,20,26]. We consider this threshold to be the performance of a small removable storage device such as a USB flash drive. The largest sources of overhead are the additional processing that secret sharing or Reed-Solomon will require and write amplification from writing multiple carrier blocks for each data block. Performance oriented Shamir Secret Sharing and erasure code implementations can be achieved using vector instructions [86] and fast Fourier transforms [64] to accelerate Galois field operations. Despite these methods, reading blocks from scattered locations will hinder performance.

Fortunately, the use of magnetic hard drives is rapidly decreasing, and with them painfully long seek times. SSDs impose no significant seek penalty and have high read performance. Scattered blocks on an SSD pose less of a performance hindrance.

Contrarily, writing redundant carrier blocks will inevitably impose excess writes and CPU overhead. Traditional buffering techniques can be used to mitigate these delays. Simple methods applied in traditional storage devices, such as contiguous allocation, are not applicable as they introduce correlations that would render Artifice vulnerable to multiple snapshot attacks and an increased risk of accidental overwrite.

It is also important that a deniable storage system does not impact the performance of the public system as is the case with some approaches aimed at tackling the multiple snapshot attack [16, 20, 26].

Our test machine was equipped with an Intel i7-4790 CPU, 32 GB of RAM, and a 480 GB Intel 660p SSD. To better model an average laptop computer we ran all benchmarks on a VirtualBox virtual machine provisioned with 4 processor cores, 4 GB of RAM, and a virtual disk formatted with `FAT32` containing 100 GB of free space. This virtual machine was running Ubuntu 18.04 with kernel v4.15.0. Our Artifice volume was 16 GB and formatted with `ext4`. For our benchmark we used the disk benchmark bonnie++ (version 1.97) with no special options enabled.

As seen in Table 5.2 our Artifice implementation using a relatively slow secret sharing library running on a commodity SSD provides performance on par with USB 2.0 flash drives [118] and thoroughly surpasses the write throughput of recent competing systems [16, 20, 26] without compromising the performance of the public volume. As the current bottleneck is a naïve secret sharing implementation, further improvements can be made by leveraging processor vector instructions or fast Fourier transforms as previously discussed. Even without those improvements, Artifice's performance is sufficient for most basic tasks including compressed 1080p video playback.

Table 5.2: Performance comparison of public volume performance with Artifice configured with AONT-RS and with Shamir Secret Sharing (SSS).

	Public Volume		**Artifice AONT**		**Artifice SSS**	
	Throughput	***CPU***	***Throughput***	***CPU***	***Throughput***	***CPU***
Read	795.399 MB/s	59.970%	124.556 MB/s	8.313%	46.946 MB/s	3.000%
	±31.580	±2.419	±3.157	±0.264	±0.986	±0.210
Write	523.758 MB/s	30.731%	82.512 MB/s	17.072%	66.965 MB/s	13.410%
	±12.591	±1.186	±0.812	±0.224	±0.792	±0.356

Chapter 6

Survivability Analysis

While the idea of adding redundancy to ensure the survivability of a deniable volume is not new [74], there has never been an evaluation of what level of redundancy is needed to provide a reasonable probability of survival for a deniable volume. This sort of evaluation is essential to demonstrating the efficacy of the proposed techniques and informs users of the limitations they must be aware of when using a deniable storage system.

Conventional storage systems are predominantly designed for use with highly reliable devices. Traditional magnetic drives have an uncorrectable error rate on the order of 10^{-13} to 10^{-15} [42]. If a block can be read at all it is extremely unlikely to be incorrect after normal error correction techniques are employed by the disk. Marginal blocks can be remapped by the drive or the file system. In contrast, a deniable storage system following Artifice's design requirements would have constant destruction of data blocks as normal behavior because public file system operations will overwrite some Artifice carrier blocks. Without a constantly running mechanism to prevent the public file system from overwriting carrier blocks, the survival of the hidden information is at best probabilistic. Although this may appear as a problematic situation, we have found through both theoretical models and an experimental evaluation that it is relatively simple to reliably ensure the survival of a small hidden volume under the right conditions. We have also developed a means to provide users with approximations of how their hidden volumes will behave through a combination of experimental observations and reliability analysis techniques commonly applied to arrays of conventional storage devices.

6.1 Theoretical Evaluation

Recall from §4.2.3 that in our Information Dispersal Algorithm IDA-based approach we require k carrier blocks out of a set of n to reconstruct our original data when using secret sharing. By calculating the probability that we will lose no more than $n-k$ blocks from each set of carrier blocks, we can determine the probability of survival for an Artifice volume. We can model this using Equation 6.1 that utilizes a binomial distribution to approximate a series of Bernoulli trials where the number of blocks successfully overwritten X is less or equal to the number of redundant blocks, $n-k$. In this case probability p is the probability of overwrite for any one given block on our disk as given in Equation 6.2.

$$\Pr[X \leq n-k] = \sum_{i=0}^{n-k} p^i \binom{n}{i} (1-p)^{n-i} \tag{6.1}$$

$$p = \frac{\text{number of writes to unallocated blocks}}{\text{number of total unallocated blocks}} \tag{6.2}$$

With the probability of survival for a single reconstructed data block we can determine the probability of survival for the entirety of the Artifice volume. This is shown in Equation 6.3 where s is the logical size of the Artifice volume and the function $\text{SizeSSS}(s, n, k)$ is the effective size of our Artifice volume when accounting for write amplification incurred by Shamir Secret Sharing. The variable t is the number of times the user writes a given amount of data and then performs a repair process. For instance, each iteration approximates the user writing 5 GB of data and then booting into the Artifice aware OS to repair the hidden volume.

$$\Pr_{\text{Survival}}[k, n] = \left(\sum_{i=0}^{n-k} p^i \binom{n}{i} (1-p)^{n-i} \right)^{\text{SizeSSS}(s,n,k) \times t} \tag{6.3}$$

In the case of our Reed-Solomon/Entropy scheme, we must also account for the entropy blocks, e, and the possibility of multiple data blocks, d, mapping to a single set of carrier blocks m. In this case, we can lose up to $m - d$ blocks out of $e + m$ stored. It is important to note that, unlike the secret sharing approach, the reconstruction threshold is dependent on the number of carrier blocks. The number of vulnerable blocks is given as SizeRS(s, m, e, d).

$$\Pr_{\text{Survival}}[e, d, m] = \left(\sum_{i=0}^{m-d} p^i \binom{e+m}{i} (1-p)^{e+m-i} \right)^{\text{SizeRS}(s,m,e,d) \times t} \tag{6.4}$$

With these two functions, we can evaluate the probability of survival for a given number of carrier blocks. We assume that the drive has 512 GB of unallocated space and an Artifice instance has a usable space of about 5 GB. In the case of our Reed-Solomon scheme, we assume that our code word contains one entropy block and either one or two data blocks. It is assumed that the user overwrites about 1% of the public volume between each time Artifice is initialized to start a repair cycle that rebuilds any overwritten blocks.

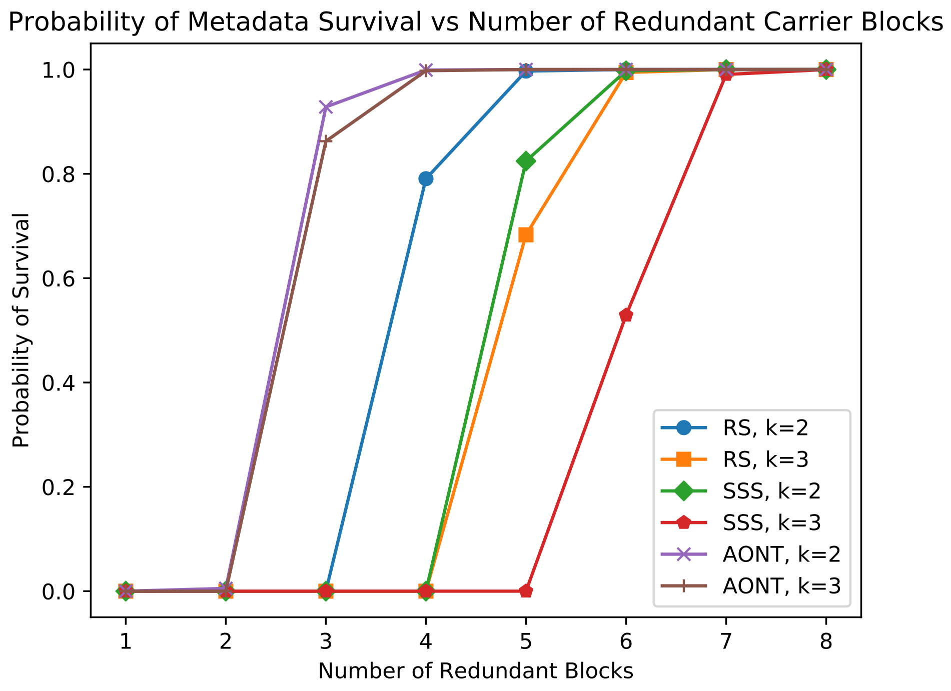


Figure 6.1: Probability of survival for Artifice metadata in a variety of configurations using both Reed-Solomon/Entropy (RS), Shamir Secret Sharing (SSS) and AONT-RS. Probabilities are calculated assuming 512 GB of free space, 5 GB written between repair cycles, 5 GB Artifice volume, and 365 repair cycles. k is the reconstruction threshold in carrier blocks.

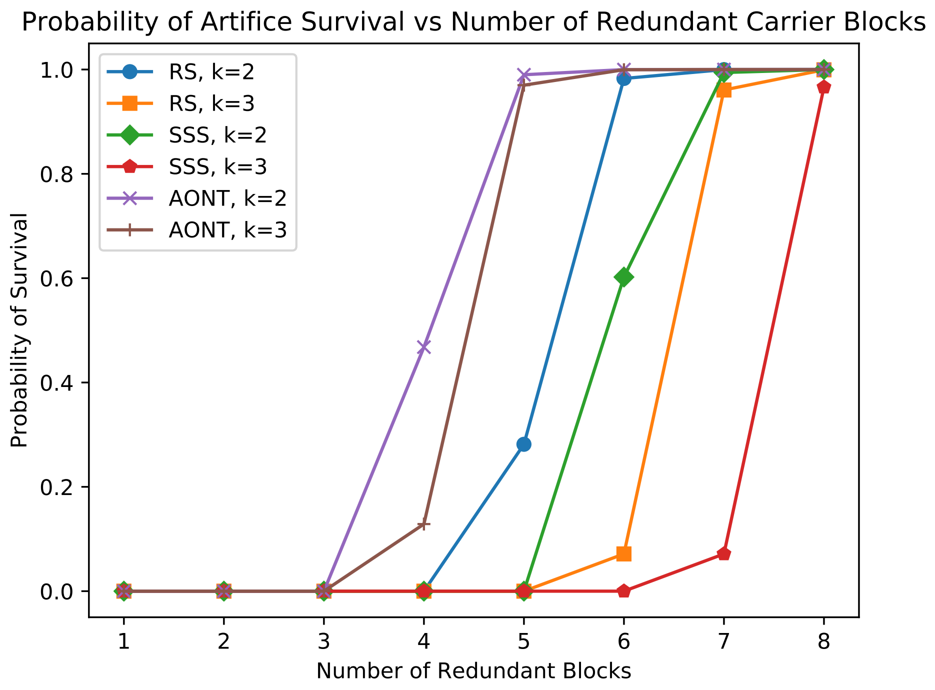


Figure 6.2: Probability of survival for an entire Artifice volume with the same configuration as Figure 6.1

Figures 6.1 and 6.2 show the survival probability of our example instance over one year assuming a repair cycle run each day for both the metadata and the entire Artifice instance with a variety of different encoding techniques and numbers of carrier blocks. From these figures, we can see that there is a specific number of carrier blocks for each configuration where the probability of survival asymptotically approaches one, which depends on the reconstruction threshold k. We can also observe that while a Reed-Solomon erasure code can provide better reliability due to improved error correction capabilities and a smaller footprint on the disk, it is at the cost of additional operational overhead due to the required entropy blocks. On the other hand, Shamir Secret Sharing would usually require one additional carrier block to provide a similar level of reliability.

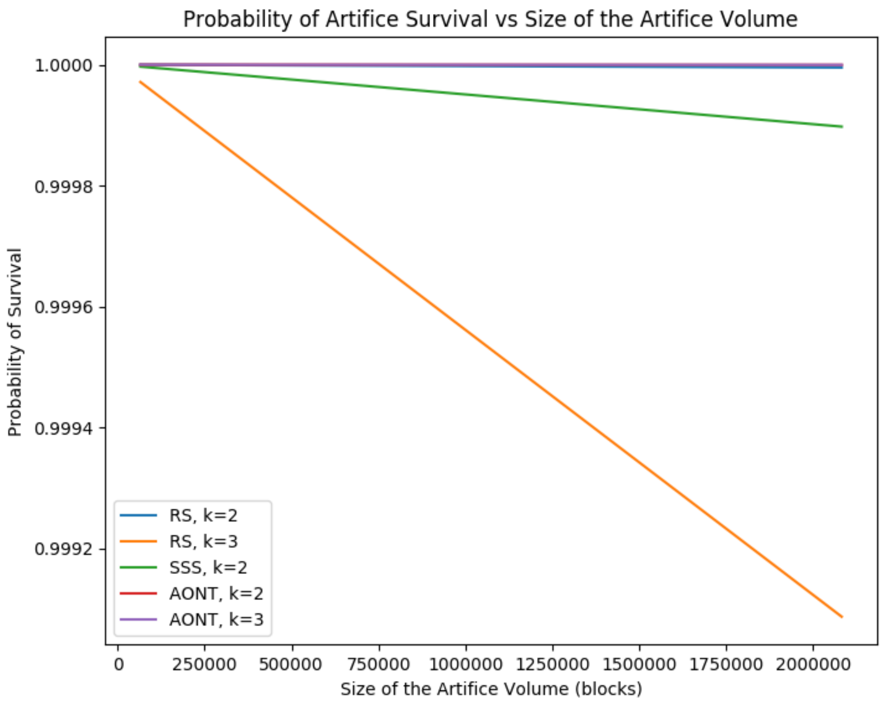


Figure 6.3: Probability of survival with varying Artifice volume sizes ranging from 256 MB to 4 GB. 5 GB of writes between repair operations and 512 GB of unallocated space. k is the reconstruction threshold in carrier blocks.

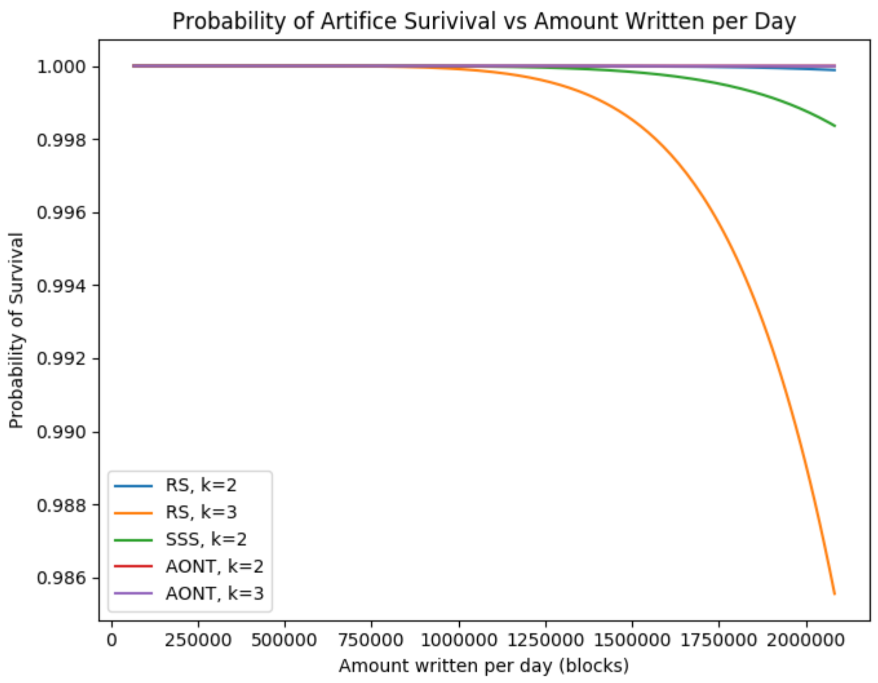


Figure 6.4: Probability of survival with varying sizes of writes between Artifice invocations from 256 MB to 8 GB. 512 GB of unallocated space and a 5 GB Artifice volume.

We can also model survivability based on the size of the Artifice volume, the size of the free space, and the amount written to the public file system between repair operations. For these figures, we assume that each data block corresponds to a set of eight carrier blocks. As shown in Figure 6.3, the smaller the Artifice volume, the higher the probability of survival. We see overall marginal decreases in reliability with different IDAs and but the probability of survival remains rather high even in the case of Shamir Secret Sharing when $k = 3$, which lags behind the other configurations. Overall, we observe a linear relationship between the size of the Artifice volume and reliability. In the case of the amount written to the public volume between repairs (Figure 6.4) we can observe an exponential decrease in reliability after approximately 4 GB. Finally, when the amount of free space available to Artifice (Figure 6.5) we can see that 256 GB of unallocated space provides a promising probability of survival for our Artifice instance.

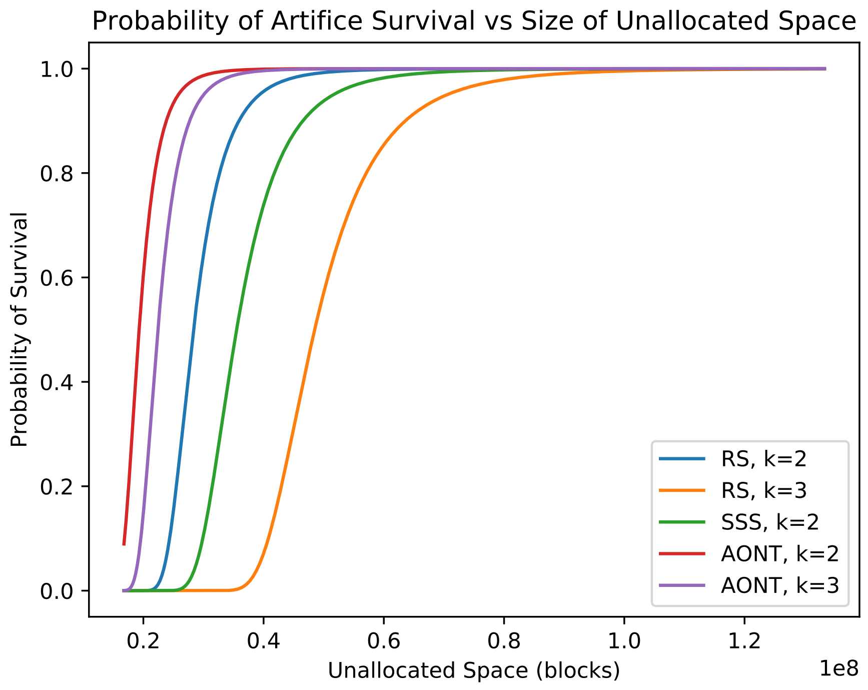


Figure 6.5: Probability of survival with varying sizes of unallocated spaces from 64 GB to 512 GB. 5 GB of writes between repair operations and a 5 GB Artifice volume.

Figure 6.5 shows that Artifice can sustain severe damage, as long as the user (i) maintains a certain percentage of the encapsulating file system remains available for Artifice to occupy, and (ii) regularly remounting the hidden volume so Artifice can repair any lost shares. It should be noted that these figures do not specifically take into account the probability of overwrite from additional sources such as garbage collection on an SSD utilizing non-deterministic TRIM operations. Although Artifice cannot escape the probabilistic block overwrite behaviors that arise as a result of our stronger adversary model, it is possible with the right IDA configuration to effectively nullify the issue.

6.2 Experimental Evaluation

To simplify experimental evaluation of Artifice's survivability characteristics with a variety of IDA configurations we have collected records of block changes and allocation information from a real-world device that we can replay onto a simulated disk containing an Artifice volume. This allows us to compare the survivability of different IDA configurations and allocation schemes with the same set of disk changes.

We have collected four months (March through June of 2021) of snapshots from a 1 TB NVMe SSD formatted with `ext4` and in use as the boot disk of a desktop computer running Ubuntu 18.04. We collected 53 snapshots in total, giving us 52 sets of changes to replay against a variety of different Artifice hidden volumes.

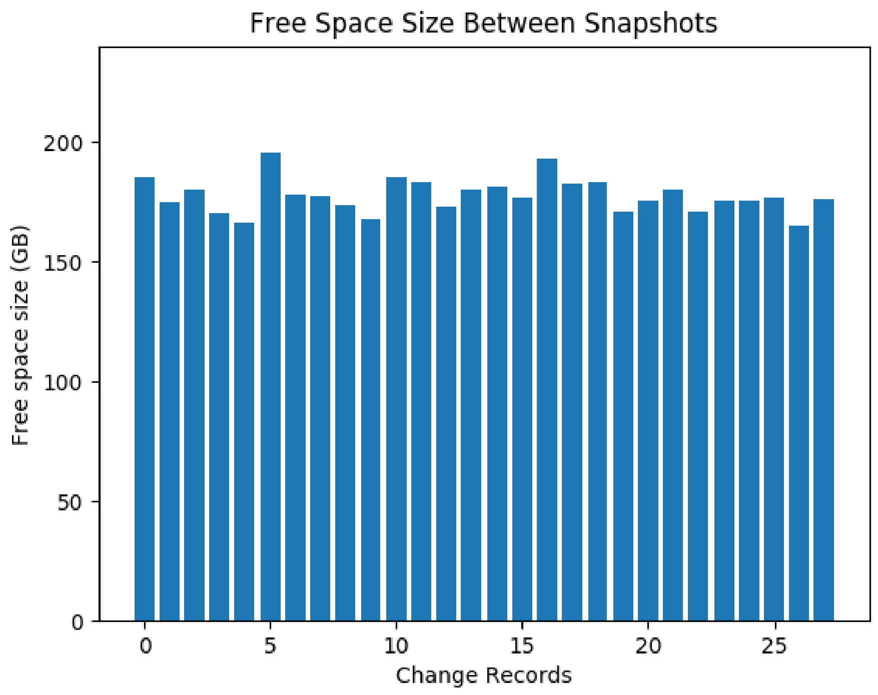


Figure 6.6: Amount of free space on our SSD at each snapshot.

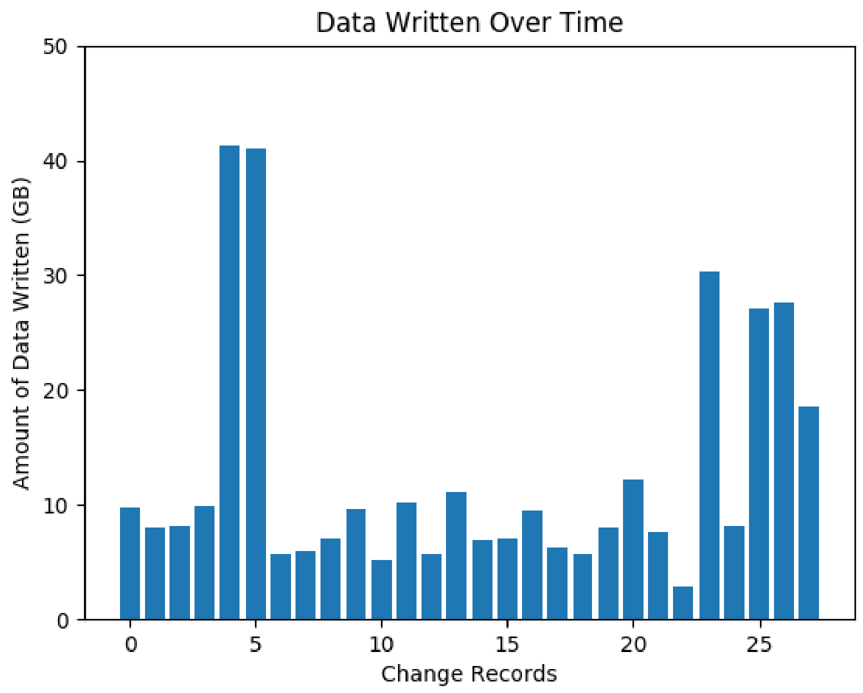


Figure 6.7: Amount of data written to the SSD between each pair of snapshots.

Figures 6.6, 6.7, and 6.8 characterize the changing state of the disk's free space and amount written with each data point generated from a pair of snapshots. While this data was collected from an SSD it is from the perspective of the logical block addressing scheme that the Artifice block device interacts with. In Figure 6.6 we can see that the free space on the disk remains relatively consistent in size with an average of 177.7 GB of free space. Figure 6.7 shows the total amount of data that changed from one snapshot to the next. Usually, this number is below 10 GB but we can see occasional spikes up to 40 GB which correspond to the user moving or downloading large media files. The average amount of data written between snapshots is 12.7 GB. In Figure 6.8 we can see that the actual amount of data written between snapshots is mostly in-place updates to already allocated blocks. This presents us with a much smaller number of blocks written between snapshots that Artifice must endure. Out of the average 12.7 GB written per data point, we see that only an average of 1.45 GB of this is written to newly allocated locations on the disk.

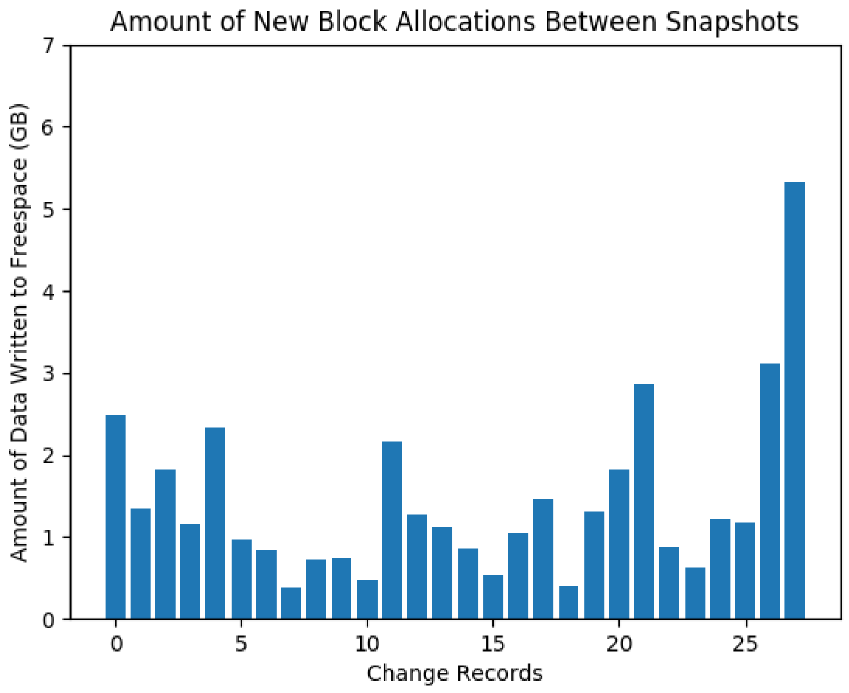


Figure 6.8: Amount of data written to newly allocated regions of the disk between each pair of snapshots.

With each pair of snapshots we can generate a list of blocks changed between the two. By combining this list of changed blocks with lists of unallocated blocks on the disk from the same time as those snapshots were taken we get a set of multiple block address arrays that we call a *change record*. With these change records we can characterize the writes to a disk and the changes in free space between two snapshots. These change records were fed into our Artifice simulator with Artifice volume sizes of 256 MB, 512 MB, 1 GB, 2 GB, and 4 GB using our Reed-Solomon/Entropy, All or Nothing Transform plus Reed-Solomon (AONT-RS), and Shamir Secret Sharing IDAs. We also added simple replication, or replicating and encrypting a data block, to our collection of IDAs as a point of comparison to previous systems [74, 80].

Our simulator reconstructs the state of the disk using the first of a pair of snapshots that make up a given change record. In the unallocated space of this snapshot we construct an Artifice volume with a given size and IDA configuration. The simulator will then use the list of changed blocks from our change record to mark which blocks were overwritten between the two snapshots. The simulator pays specific attention to which Artifice blocks were overwritten and outputs a record of what types of Artifice blocks were overwritten, how many of each type, and ultimately how many metadata or data blocks were rendered unrecoverable and lost.

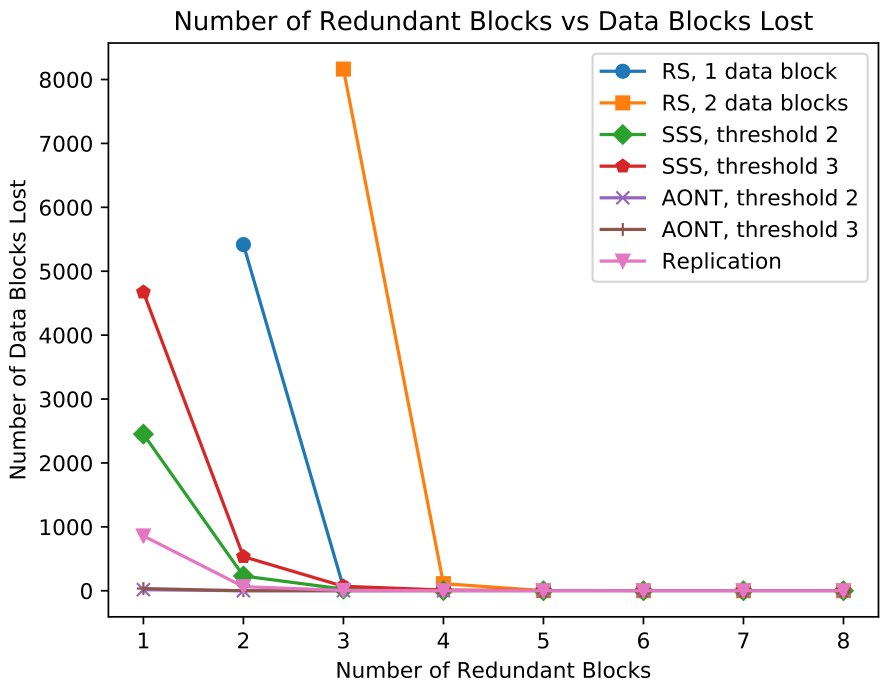


Figure 6.9: Number of redundant carrier blocks per data block tuple versus the average number of data blocks lost with a 2 GB Artifice instance for Reed-Solomon/Entropy (RS), AONT-RS (AONT), Shamir Secret Sharing (SSS), and basic replication.

We can see in Figure 6.9 the number of data blocks lost due to public file system writes. The lower the average number of blocks lost for a given configuration, the more effective the redundancy scheme is at ensuring the survival of the Artifice volume. It is apparent that when compared with previous approaches utilizing a basic replication scheme, SSS and Reed-Solomon/Entropy lose more blocks on average for a given number of redundant blocks per tuple than AONT-RS or replication. This is due to a combination of the overall higher write amplification factor and higher reconstruction threshold for a given number of redundant blocks. Unsurprisingly, AONT-RS loses fewer blocks than replication. This is primarily due to the increased space efficiency of the algorithm despite requiring more shares of data to reconstruct than replication.

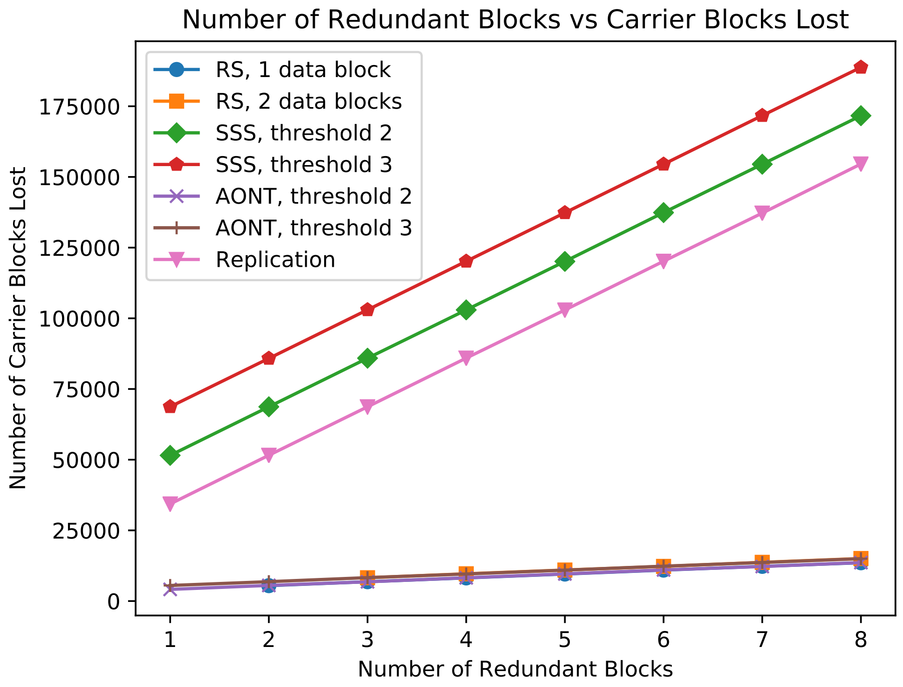


Figure 6.10: Number of redundant carrier blocks per data block tuple versus the average number of carrier blocks overwritten with a 2 GB Artifice instance for a variety of IDAs.

Figure 6.10 shows the average total number of redundant carrier blocks overwritten by the public file system. This graph shows us part of the relationship between a given scheme's write amplification and the probability of a collision between blocks. We can see two general groupings. Replication and Shamir Secret sharing incur higher write amplification because each share is equal in size to the original data block. Reed-Solomon/Entropy, and AONT-RS allow for carrier blocks smaller than the original data block and therefore have smaller write amplification factors. These graphs ultimately let us know approximately how many redundant carrier blocks must be added to our original reconstruction threshold to ensure data is not lost given real-world write patterns. For instance, we can observe from Figure 6.10 that with a 2 GB Artifice volume that AONT-RS and replication provide far greater reliability with each additional carrier block added than Reed-Solomon/Entropy with only three redundant blocks needed to provide near zero data loss versus the latter's four or five, depending on the reconstruction threshold.

From the experimental results, it is apparent that Artifice, if configured with the proper level of redundancy, stands a good chance of surviving on a system's primary drive in the configuration we observed, although it should be noted that the pattern and number of blocks written by the public file system depends heavily on the user's behavior and the behavior of their computer's file system. As such users must be acutely aware of their activities when using Artifice to assist Artifice's built-in redundancy mechanisms in ensuring the survival of their hidden data.

6.3 Mean Time to Data Loss

While our experimental evaluation provides us empirical evidence that Artifice can survive the inadvertently destructive actions of the public volume, it is difficult to translate those into recommendations for the user. Although they provide us with some starting data with which to estimate the amount of redundancy with which an Artifice volume should be configured. With an estimate of the rate at which individual blocks are overwritten we can model the mean time to data loss (MTTDL) for an Artifice volume, a more directly applicable and easily calculated metric than running a set of simulations and collecting the required data sets. To do this we construct a simple Markov model from which we can derive the MTTDL for each set of carrier blocks in a method described by Pâris *et al.* [81]. While MTTDL is far from an ideal measurement of storage system reliability [43], it is relatively easy to calculate from our gathered measurements and provides a reasonable approximation for the purposes of comparing the effectiveness of different IDA configurations when given a block overwrite rate.

As an example, we have constructed a model and determined the MTTDL for a set of carrier blocks where $n = 5$, $k = 3$, and λ is the failure rate for individual carrier blocks in Figure 6.11. From this model we can derive a system of differential equations where $p_n(t)$ is the probability of being in state n at time t.

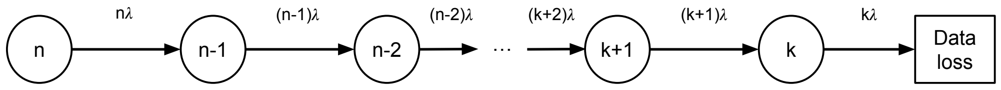


Figure 6.11: Generalized Markov model with no repair rate.

$$
\begin{aligned}
p'_n(t) &= -n\lambda p_n(t) \\
p'_{n-1}(t) &= n\lambda p_n(t) - (n-1)\lambda p_{n-1}(t) \\
&\dots \\
p'_k(t) &= (k+1)\lambda p_{k+1}(t) - k\lambda p_k(t)
\end{aligned}
$$

To simplify the problem of solving this system of differential equations we can apply the Laplace transform to either side of each equation. When we take the Laplace transforms of this series of differential equations where the initial conditions are $p_n(0) = 1, p_{n-1}(0) = 0, ..., p_k(0) = 0$ it gives us a system of linear equations in the domain of s.

$$
\begin{aligned}
sp^*_n(s) - 1 &= -n\lambda p^*_n(s) \\
sp^*_{n-1}(s) &= n\lambda p^*_n(s) - (n-1)\lambda p^*_{n-1}(s) \\
&\dots \\
sp^*_k(s) &= (k+1)\lambda p^*_{k+1}(s) - k\lambda p^*_k(s)
\end{aligned}
$$

The solutions of this system of Laplace transformed differential equations can be used to compute the MTTDL with the following sum.

$$
\text{MTTDL} = \sum_i p^*_i(0)
$$

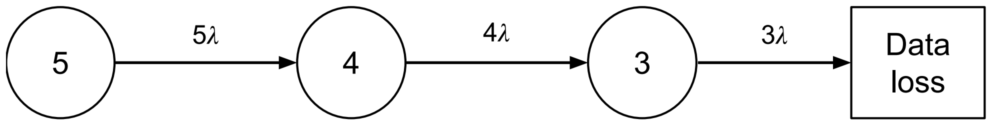


Figure 6.12: $(5, 3)$ Markov model with no repair rate.

By taking the Laplace transform of $p_i(t)$ at time zero we get the average amount of time spent in state i. To calculate the MTTDL we add these average times for each non-failure state to determine the average time before entering our failure state. For example, if we have an IDA where $n = 5$ and $k = 3$ (shown in Figure 6.12) we get the following system of differential equations

$$\begin{aligned} p_5'(t) &= -5\lambda p_5(t) \\ p_4'(t) &= 5\lambda p_5(t) - 4\lambda p_4(t) \\ p_3'(t) &= 4\lambda p_4(t) - 3\lambda p_3(t) \end{aligned}$$

This combined with the initial conditions of our Markov model, $p_5(0) = 1$, $p_4(0) = 0$, and $p_3(0) = 0$ we can take the Laplace transforms of this system of differential equations.

$$\begin{aligned} sp_5^*(s) - 1 &= -5\lambda p_5^*(s) \\ sp_4^*(s) &= 5\lambda p_5^*(s) - 4\lambda p_4^*(s) \\ sp_3^*(s) &= 4\lambda p_4^*(s) - 3\lambda p_3^*(s) \end{aligned}$$

The MTTDL of the set of carrier blocks is given by the sum of the solutions to the system of Laplace transformed differential equations evaluated at $s = 0$.

$$\text{MTTDL} = \sum_i p_i^*(0) = \frac{1}{5\lambda} + \frac{1}{4\lambda} + \frac{1}{3\lambda} = \frac{47}{60\lambda}$$

This can be generalized into the following for a single set of carrier blocks with n total shares, a reconstruction threshold of k, $n \geq k$, and without a repair rate.

$$\text{MTTDL}(n, k, \lambda) = \sum_{i=0}^{n-k} \frac{1}{(n-i)\lambda} = \frac{1}{n\lambda} + \frac{1}{(n-1)\lambda} + \frac{1}{(n-2)\lambda} + ... + \frac{1}{k\lambda}$$

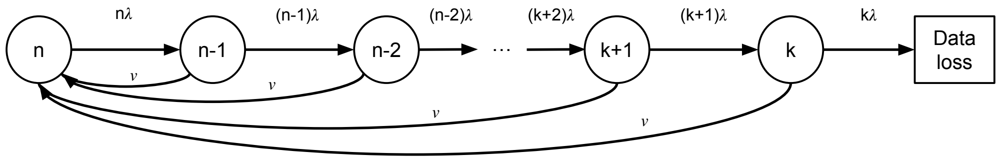


Figure 6.13: Generalized Markov model with repair rate v.

Since Artifice can repair the volume when mounted at some rate v we can modify our Markov model with a new set of possible state transitions. If we mount Artifice for repair once a day, then we have a probability of $v = 1/24$ that the user will mount Artifice in any given hour. This yields a new set of differential equations derived from a revised Markov model (Figure 6.13).

$$p'_n(t) = -n\lambda p_n(t) + \sum_{i=k}^{n-1} v p_i(t)$$

$$p'_{n-1}(t) = n\lambda p_n(t) - (n-1)\lambda p_{n-1}(t) - v p_{n-1}(t)$$

$$...$$

$$p'_k(t) = (k+1)\lambda p_{k+1}(t) - k\lambda p_k(t) - v p_k(t)$$

From this new set of differential equations, we can compute their Laplace transforms.

$$sp_n^*(s) - 1 = -n\lambda p_n^*(s) + \sum_{i=k}^{n-1} vp_i^*(s)$$

$$sp_{n-1}^*(s) = n\lambda p_n^*(s) - (n-1)\lambda p_{n-1}^*(s) - vp_{n-1}^*(s)$$

$$\dots$$

$$sp_k^*(s) = (k+1)\lambda p_{k+1}^*(s) - k\lambda p_k^*(s) - vp_k^*(s)$$

As an example, we can compute the MTTDL for a $(5,3)$ code shown in Figure 6.14.

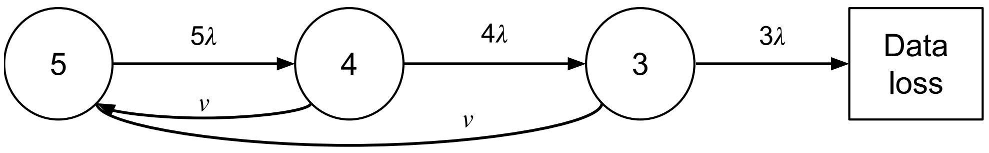


Figure 6.14: $(5,3)$ Markov model with a repair rate of v.

$$p_5'(t) = -5\lambda p_5(t) + vp_4(t) + vp_3(t)$$

$$p_4'(t) = 5\lambda p_5(t) - 4\lambda p_4(t) - vp_4(t)$$

$$p_3'(t) = 4\lambda p_4(t) - 3\lambda p_3(t) - vp_3(t)$$

Then using the same initial conditions as our model without a repair rate we get the following set of Laplace transformed equations.

$$sp_5^*(s) - 1 = -5\lambda p_5^*(s) + vp_4^*(s) + vp_3^*(s)$$
$$sp_4^*(s) = 5\lambda p_5^*(s) - 4\lambda p_4^*(s) - vp_4^*(s)$$
$$sp_3^*(s) = 4\lambda p_4^*(s) - 3\lambda p_3^*(s) - vp_3^*(s)$$

Then by solving this system of transformed equations when $s = 0$ we get the following solution for the MTTDL.

$$\text{MTTDL}(5, 3, \lambda, v) = \frac{1}{3\lambda} + \frac{3\lambda + v}{12\lambda^2} + \frac{12\lambda^2 + 7\lambda v + v^2}{60\lambda^3} \quad = \frac{47\lambda^2 + 12\lambda v + v^2}{60\lambda^3}$$

The failure rate (λ) is the total number of failures over time.

$$\lambda = \frac{\text{Total Number of Failures}}{\text{Total Operating Time}}$$

Our simulator used in §6.2 tracks both the number of overwritten carrier blocks and the number of overwritten metadata blocks for a given Artifice volume. Also, because we know the amount of time over which the original disk write measurements were taken we can compute the average failure rate for that period of time.

Recall that our data was collected from a 1 TB SSD that had on average 171 GB of free space, and 1.45 GB of new unallocated blocks written to between snapshots. On average each measurement period between snapshots was 47.14 hours or about two days. With these measurements we can compute the failure rate for a variety of IDAs and (n, k) combinations for a 2 GB Artifice volume.

Table 6.1: Probability of overwrite for Shamir Secret Sharing (SSS), All or Nothing Transform (AONT-RS), Reed-Solomon/Entropy (RS), and Replication with different reconstruction thresholds in carrier blocks (k).

SSS	Replication	AONT-RS ($k = 2$)	AONT-RS ($k = 3$)	RS ($k = 2$)	RS ($k = 3$)
0.0007	0.0007	0.00011	0.00017	0.00011	0.00017

Since our MTTDL equations require λ to be expressed as a probability of failure in a given length of time, we can divide the failure rate by the total footprint of the Artifice volume on the disk in blocks to express it as a probability that any one block in an Artifice volume will be overwritten in an hour (Table 6.1). We can plug that probability into our formulas for the MTTDL derived using the methods we have previously described. We can see in Figure 6.15 the MTTDL of a 2 GB Artifice volume with 1.45 GB of new blocks written every 47.14 hours assuming that there is no repair rate ($v = 0$). In this figure we can see that the MTTDL increases relatively gradually as the number of redundant carrier blocks is increased but in general, we see reliability on the order of a few thousand hours and with a lower reconstruction threshold resulting in an overall higher MTTDL. This can tell our user a rough idea of how long they have until the Artifice volume is damaged beyond repair, or, the effective operational lifespan of a hidden volume.

Although, Figure 6.15 ignores Artifice's self-repair capability. If we apply a repair rate where a user boots into the Artifice aware operating system to perform self-repair once per day we get a probability of repair in any given hour of 1/24. We can see in Figure 6.16 that even a relatively infrequent repair rate for the expected lifetime of an Artifice volume provides a greatly improved MTTDL that increases exponentially as the number of redundant blocks per carrier block tuple increases. In addition to the MTTDL being orders of magnitude higher than if a user was to not repair their hidden volume.

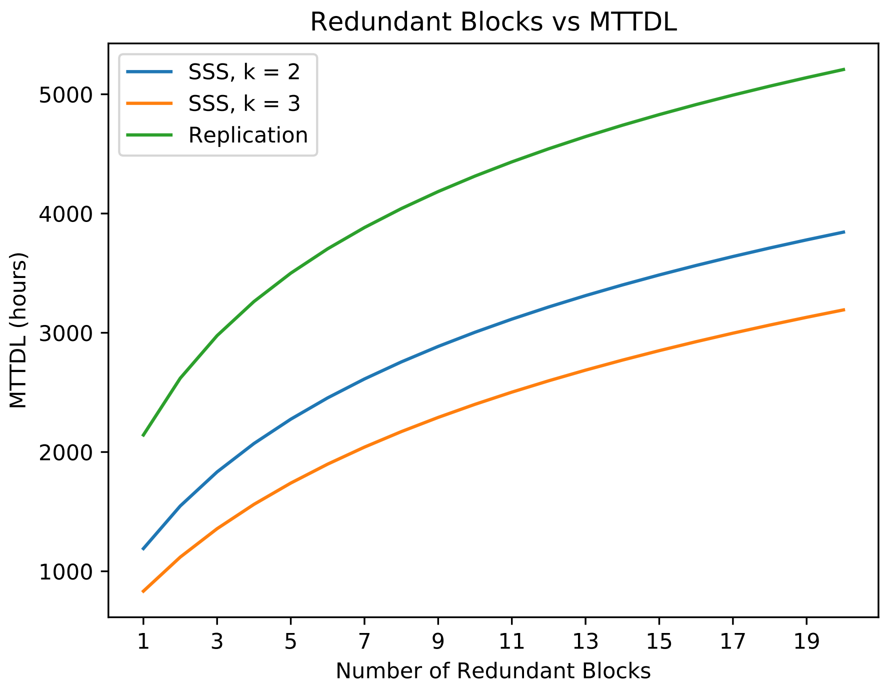


Figure 6.15: MTTDL in hours for Artifice data blocks using Shamir Secret Sharing (SSS) with different reconstruction thresholds measured in carrier blocks (k) and basic replication with no repair rate ($v = 0$).

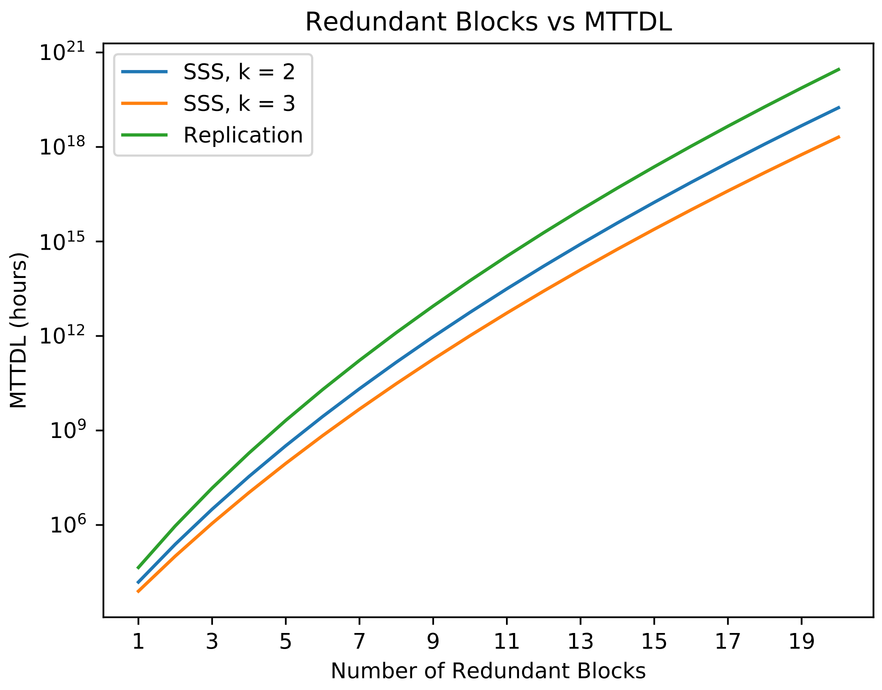


Figure 6.16: MTTDL in hours for Artifice data blocks using Shamir Secret Sharing (SSS) with different reconstruction thresholds measured in carrier blocks (k) and basic replication with a repair rate of $v = 1/24$ (once a day).

With a way to calculate the MTTDL we can equip the user with a means to estimate on average how long the user can hide an Artifice volume before hidden data is lost and approximate how often the user should boot into the Artifice-aware OS to repair the hidden volume.

6.4 Summary

In this chapter we have shown that Artifice if configured right can reliably withstand accidental overwrites that are naturally a result of hiding Artifice data within the unallocated space of an existing public file system. Our original theoretical evaluation gave us a model with which we could estimate the probability that an Artifice volume would be able to survive for a given number of overwrite/repair cycles. While it may be an imperfect model this analysis showed that if configured correctly Artifice can survive for a long period of time consistent with the indicated conditions.

The drawback of our theoretical evaluation is that the uniform write patterns the model assumes are unlikely to to be found on a user's device. This shortcoming prompted us to gather data on real-world disk activity so that we could determine the reliability of an Artifice volume in a setting in which the write patterns would match those of a real device. With our simulator we were able to confirm our findings from our theoretical evaluation that Artifice should be able to survive if used as recommended. The primary limitation of this approach is that gathering data and running a set of simulations is a labor and computationally intensive process. This limitation makes it unrealistic for a user to replicate these methods to estimate how they should configure Artifice and and provide and estimation of how long their hidden volume can survive.

To address the shortcomings of our simulator-based approach, we observed the failure rate of individual Artifice blocks and used that in conjunction with techniques commonly applied to redundant disk arrays to derive expressions with which we can calculate the MTTDL of an Artifice volume. While allowing us an easier way to estimate of a volume's lifespan with a given configuration, this approach also allows us to easily include repair rates so that the user can determine how often to repair their hidden volume.

Chapter 7

Snapshot & Statistical Analysis Attacks

To address the lack of extensive experimental evaluation and attempted attacks against deniable storage systems we have explored a variety of possible avenues to compromise a deniable storage system.

First, we have implemented a multiple snapshot attack that analyzes the average length of a disk update. The overall goal of this proposed attack is to reliably differentiate between devices that contain a hidden volume and devices that do not. In particular, we pay attention to the rates of false positives and false negatives that should both be sufficiently low for an attack to be considered successful. Using our implemented attack as a guideline we propose countermeasures that Artifice can use to mitigate the effectiveness of our attack.

Next we explore the matter of entropy analysis to differentiate Artifice carrier blocks produced by an IDA from those filled with random bytes. Through our experimentation we were able to verify the effectiveness of our IDAs in rendering data indistinguishable from surrounding random blocks. As well as demonstrating a situation where blocks generated by a misconfigured IDA can be detected through the use of entropy analysis.

Lastly we demonstrate how TRIM on a flash device can invalidate our assumption that hidden data blocks are indistinguishable from other blocks on the disk. We also discuss possible ways that Artifice can still be used to effectively hide data on a device where TRIM is enabled.

7.1 A Multiple Snapshot Attack Framework

Recall from §4.2 that Artifice by default distributes its blocks uniformly across a disk's free space. Depending on the size of the free space, writes made uniformly are very likely to result in isolated changes on disk, which we call *singletons.* Additionally, other deniable storage systems such as Pang's StegFS [80], HIVE [16], and Datalair [20] also exhibit a pseudo-random distribution of writes to the disk. This is in contrast to normal file systems, which do not uniformly distribute writes, and are much more likely to place writes that are part of long strings of consecutive changes that we call *chains*, which are a series of changed consecutive blocks. We call a chain of c consecutive changes a c-chain.

Example 7.1.1. *Assume that in a list of changes to a block device, a 1 denotes a changed block and a 0 denotes no change between two snapshots of the block device taken at different points in time. In Figure 7.1 there are two singletons (or 1-chains) and one 3-chain.*

$$\begin{array}{|c|c|c|c|c|c|c|}\hline 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline \end{array} \tag{7.1}$$

An adversary analyzing the change records produced from a pair of disk snapshots would be able to see the lengths of the chains those changes produce. Crucially, if the disk contains a hidden volume, the adversary would also see singleton changes made by writes to that volume.

We will represent the theoretical distribution of these chains by representing our disk with an array of blocks A of length n with k changes made uniformly where 1 denotes a change and 0 denotes no change to a specified block. A can be represented as $p = (p_1, p_2, ..., p_k)$, an ordered partition of k, where each $p_i \in \mathbb{Z}$ represents the i-th string of p_i consecutive 1s separated by one or more 0s and $|p|$ is the length of the partition of k. By our construction of p, A is uniquely represented by p and

$$p_1 + p_2 + ... + p_{|p|} = k. \tag{7.2}$$

Since any array A can be represented by $p \in P$, where P is the partition of k, we can compute the probability of c length chains $\Pr(C = c)$, as

$$\Pr(C = c) = \sum_{p \in P} \Pr(C = c \mid p) \Pr(p), \tag{7.3}$$

by marginalizing over P, the size of which is $\binom{2k-1}{k-1}$. Since there are $\binom{n}{k}$ possible arrays, counting the number of arrays represented by p is sufficient to compute $\Pr(p)$.

In our construction arrays that are represented by p must have the form

$$\star \underbrace{11\cdots 10}_{p_1} \star \underbrace{11\cdots 10}_{p_2} \star \cdots \star \underbrace{11\cdots 1}_{p_{|p|}} \star$$

Notice that there are k 1s and $|p| - 1$ 0s in the string above, so there are $n - k - (|p| - 1)$ 0s, whose locations are variable. There are $\binom{n-k-1}{|p|}$ different ways to place these 0s, thus $\Pr(p)$ is defined as

$$\Pr(p) = \frac{\binom{n-k-1}{|p|}}{\binom{n}{k}} \tag{7.4}$$

The probability of a chain of c consecutive changes in A is given by Equation 7.5, where P is the set of partitions of k, $|p|$ is the number of elements in a partition p, and $\Pr(C = c \mid p)$ is the probability of c in a partition p.

$$\Pr(C = c) = \sum_{p \in P} \frac{\binom{n-k+1}{|p|}}{\binom{n}{k}} \Pr(C = c \mid p) \tag{7.5}$$

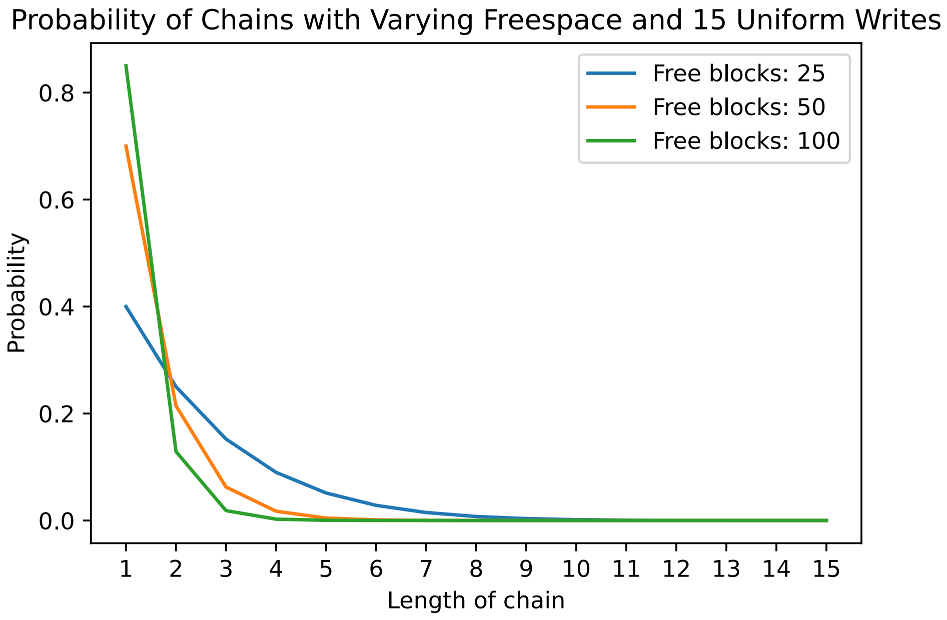


Figure 7.1: Theoretical probability of consecutive changes when changes are made uniformly. The probability of c consecutive changes degrades very quickly, especially when the free space is large.

A plot of Equation 7.5 for a variety of free space sizes and chain lengths (Figure 7.1) shows that as the free space grows relative to the number of writes, the probability of a singleton increases. A plot of chains found in our collected dataset (Figure 7.2) shows that for real disks, the probability of a singleton is much smaller, and the tail of the distribution is typically much heavier. Together they show the disparity between the distributions of chains due to a hidden volume and the chains due to a public file system. This becomes more pronounced as the hidden volume writes more data. Consequently, we can construct features based on these probabilities to distinguish between the distribution of consecutive changes made by a public file system and those made by a public file system with a hidden volume.

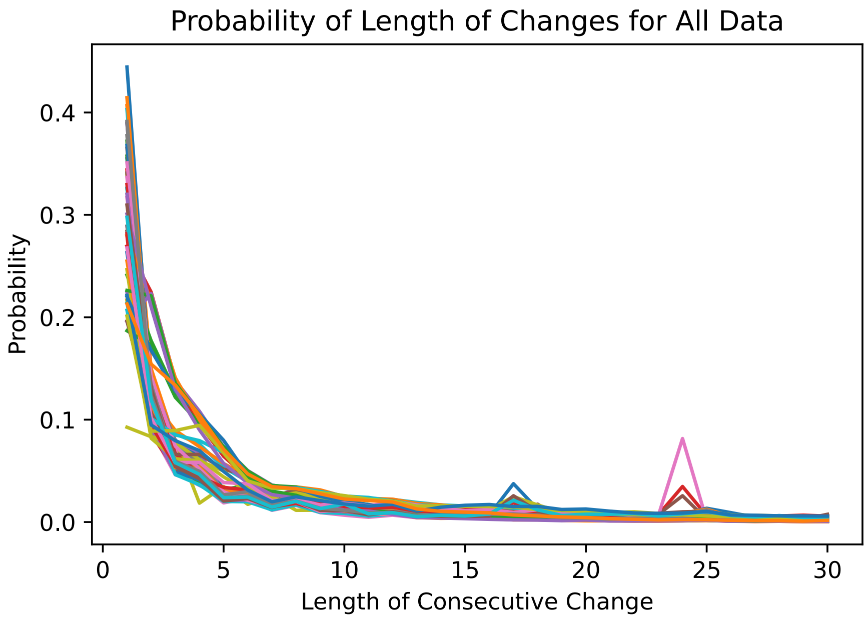


Figure 7.2: Empirical probability of consecutive changes for 52 change records where the changes are made by an `ext4` file system. Note the occasional spikes in probability found in the long tail due to occasional movements or downloads of unusually large pieces of data.

To carry out this attack, we assume that the attacker has access to a large set of disk images, both from disks that contain hidden volumes and from those that do not. Images in this set will be organized into pairs of images from the same disk at different points in time. Comparing these pairs of images produces a list of changes made to that disk. Moving forward we will call pairs that do not have an instance of a hidden volume *clean* and those that do *dirty*. Assuming our adversary is well-funded and motivated we consider this sort of data obtainable.

Since for our proposed attack we only need to determine whether individual blocks have changed, we can take snapshots of the clean and dirty disks in a space efficient manner by hashing each block on the disk and constructing a Merkle tree [75] over the hashed blocks. This gives us a very efficient method of finding changes and producing change records. These change records are further processed into lists of integers, D_i, recording the lengths of chains found in each change record. We will denote these lists of chains $\{D_i\} = \mathcal{D}$.

From $\mathcal{D}$ the adversary has several options for constructing an arbitrary n number of features for use in a classification algorithm. The first is to remove a set of clean disks that we will call $\mathcal{C}$. Using $\mathcal{C}$ the adversary estimates the probability of c-changes from 1 to n, the number of features, for each D_i by counting the occurrences of each c-chain and dividing by the total number of chains in D_i. Using these probabilities on $\mathcal{D} - \mathcal{C}$, the adversary can then estimate the probability of a disk containing more than k c-chains with the cumulative distribution function (CDF) of the binomial distribution, $F(k; n, p_c)$ where n is the length of the change record and p_c is the estimated probability of a chain of c consecutive changes. We will denote the event of an adversary observing k consecutive changes of length c as X_c. The probability of an adversary observing more than k consecutive c-chains is given by Equation 7.6.

$$P(X_c > k) = 1 - F(k; n, p_c). \tag{7.6}$$

Using these values we construct our final set of features $\mathcal{F}$, a matrix of size $\mathcal{D} \times c$, for our list of chains $\mathcal{D}$. This method has greater sensitivity to small variations in probabilities for small disks; however, on large disks, it tends to underflow when computing the equation shown above. The pseudocode for this feature construction algorithm is provided in Algorithm 1. For large datasets, we take a simpler approach to computing $\mathcal{F}$ by computing the probabilities of chains of length 1 to n for each D_i and feeding these to a classifier.

Algorithm 1: Feature Construction. This method is suitable for smaller disks, where greater sensitivity is required, but suffers from underflow on large disks.

```
Input: $\mathcal{D}$, a set of processed disks; $\{p_1, ..., p_c\}$, the estimated
       probabilities of consecutive changes of length $1...c$.
Output: $\mathcal{F}$, a $|\mathcal{D}| \times c$ matrix.
$\mathcal{F} := \{\}$
foreach $D_i \in \mathcal{D}$ do
    $f := \{\}$
    foreach $c_i \in \{1...c\}$ do
        $k :=$ number of $c$-consecutive changes in $D_i$
        $a := 1 - F(k; |D_i|, p_{c_i})$
        append$(a, f)$
    end
    append$(f, \mathcal{F})$
end
return $\mathcal{F}$
```

Our classification problem has two possible outcomes, either the disk contains a hidden Artifice volume or it does not. Since our output has two possible outcomes we can use a simple binary logistic regression fit to our training dataset as our classifier. This logistic regression allows us to estimate the probability that a given disk contains a hidden volume from our set of features, in this case, the probability of a chain of length n. Based on this probability and a threshold value (we use the default value of 0.5) the model will classify a disk as containing a hidden volume or not.

Recall that the dataset is entirely constructed by the adversary, so it has ground truth labels describing whether each row in $\mathcal{F}$ corresponds to a disk containing a hidden volume or not. The adversary now trains a supervised classification algorithm on $\mathcal{F}$ split into standard train and test sets. On new pairs of disks, the adversary runs through the feature construction process, then runs the classification algorithm on those features and responds accordingly. Furthermore, if the adversary can confirm that some disks did contain a hidden volume, it can update its model using various online learning techniques [104], further refining their model.

As a note on the selection of the number of features, we observe from Equation 7.5 that when the free space on a disk is large relative to the number of writes, the vast majority of writes will result in singleton changes. As a consequence, the adversary could learn based on a single feature derived from singletons. This may be desirable in some situations for the sake of efficiency; however, Artifice could simply be modified so that when writing blocks they would be grouped in chains of two or more, thereby defeating the attack as described. The adversary can in turn thwart this countermeasure by increasing the number of features, n. We go into more detail regarding this problem in §7.3.

7.2 Data Collection & Experiment Methodology

One of the more daunting challenges of carrying out a multiple snapshot attack is the availability of pairs of disk images. These are necessary to learn what normal chains look like. While collecting hundreds, or thousands, of disk images may be feasible for a nation-state level adversary (or a large IT department), we were unable to collect such a large amount of data.

Instead, we have used the dataset collected for the survivability experiments described in §6.2. This data set gives us 53 snapshots in total and comparing these snapshots gives us 52 change records. By observing the distribution of lengths of changes over our collected data (Figure 7.2), and the theoretical distribution of consecutive changes (Figure 7.1) when changes are made uniformly, the potential strangeness of a disk running a deniable volume becomes clear.

Because just 52 data points are insufficient to train a classifier and would be inconclusive regarding the performance of that classifier, we instead used this data to generate a synthetic dataset on which to train and test our classifier. While in the real world different file systems may produce different patterns of changes, this does not change the reality that a deniable volume writing many single blocks, and thereby causing many singleton chains in the change record, would be considered abnormal regardless of the public file system in use. Though the fact that our attack only utilizes data from `ext4` file systems is a limitation, no file system in widespread use writes blocks randomly, so we expect our attack will generalize to other data sources.

Our experiments are conservative in terms of the operational security measures the user of a hidden volume might take and we discuss them here to better describe our experimental methodology. There are several things that the user of a deniable volume could do to decrease the odds of detection. To start, assume that a single snapshot has been taken, and no deniable volume yet exists on the drive. A prudent user would make many changes to the disk through the public file system so that if there are overwhelmingly many chains distributed according to normal disk behavior, the singletons made by writing to the hidden volume could be made to look like noise. To illustrate this, consider a user that does not produce a single change through the public file system after the deniable volume is created and written to. In this case, the adversary would see, after taking a second snapshot and computing the differences, only chains produced by the deniable volume. These chains would principally be singletons, which would surely be conspicuous. As an extreme measure, the user could simply wipe the disk and then create the deniable volume, but this may be considered suspicious or may be undesirable for other reasons. As a less drastic alternative, the user could produce an overwhelming number of changes to the disk through the public file system. In our experiments we chose this middle ground by using our real data to simulate 25 GB worth of public changes on a 1 TB disk with 100 GB of free space. This produces a sufficient number of public changes to hide private changes while still behaving as a normal user might and is in line with the occasional spikes in write activity that we observed in §6.2.

Each pair of disk snapshots can be regarded as producing a distribution over consecutive change lengths, so to construct our synthetic dataset we simply draw chains from these distributions. Realistically, the fraction of disks containing hidden volumes would be relatively small, and the size of hidden volumes would also be variable. For our hidden volumes, we assume that we have instances that are from 250 MB to 1.25 GB in increments of 250 MB. In addition to its realism, this allows us to determine a point at which the number of uniform writes becomes conspicuous. For our Artifice parameters we chose those that minimized the number of writes while achieving survival probabilities over 80% with 25 GB of cover changes. This led us to copy data blocks 6 times, where the survival of a single block is sufficient for reconstructing the data. Since our adversary can generate an arbitrary number of disk snapshots with and without hidden volumes, to allow our classifier to learn to distinguish disks more quickly our training set contains an even split of disks with and without hidden volumes. However, to reflect the rarity of hidden volumes in the real world only 5% of our test set contains disks with a hidden volume. We generate a training set of size 10,000 and a test set of size 2,500. We repeat this generation, training, and testing cycle 100 times to ensure the reliability of our results.

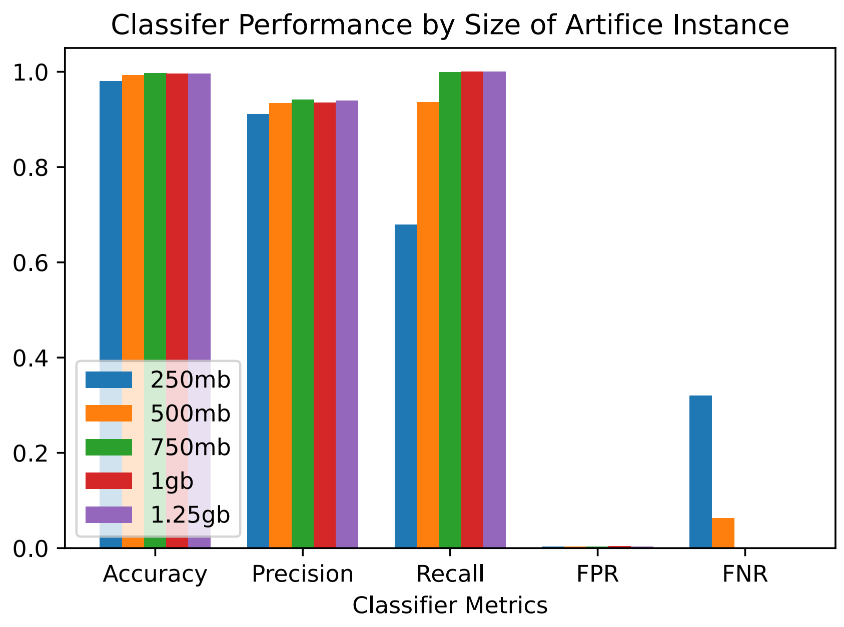


Figure 7.3: Hidden volumes over 0.75 GB in size are always identified successfully.

The presence of a point where the false negative rate (FNR) becomes negligible also offers an explanation for the relatively constant false positive rate (FPR). This being that a certain percentage of simulated clean disks will naturally have disproportionately many singletons and thus get misclassified as containing an Artifice volume. Interestingly, there were very few disks that naturally had enough singletons to exceed the learned threshold.

Future work may combine the features we use in this attack with other features. Such as what proportion of changes is made to blocks in free space versus allocated blocks. The introduction of more features would serve to better characterize disk behavior and further improve the efficacy of the attack.

Table 7.1: Numeric results from the classifier averaged over 100 runs. Furthermore, 95% confidence intervals for all figures vary only in the thousandths.

Size (GB)	Accuracy	Precision	Recall	FPR	FNR
0.25	0.981	0.911	0.680	0.004	0.320
0.5	0.993	0.934	0.937	0.004	0.063
0.75	0.997	0.942	0.999	0.003	0.001
1.0	0.996	0.935	1.0	0.004	0.0
1.25	0.997	0.939	1.0	0.004	0.0

7.3 Snapshot Attack Mitigation

With a practical multiple snapshot attack described in detail, we can develop countermeasures that Artifice and other deniable storage systems can use to defend themselves against this class of attacks. In §4.2.4 we proposed mitigating multiple snapshot attacks through operational security measures. The rationale is that if the user can produce a deniable reason for changes to the entire disk, such as re-installing the operating system or defragmentation, the adversary's previously gathered data would be rendered useless. This defense relies heavily on the ability of the user to out-maneuver the adversary and will not always be practical. As a result, it would be prudent to develop some other countermeasures against our proposed attack and future types of snapshot analysis.

We can observe in Figure 7.3 that the FNR of our classifier decreases as the effective size of the Artifice instance increases while the amount of data written by the public volume remains the same. With this in mind, we can conclude that the operational strategy proposed in §7.2, to keep the proportion of hidden to public writes sufficiently skewed in favor of the latter, is a viable strategy. Unfortunately, this severely limits the effective size of an Artifice instance.

In our proposed attack, we use only one feature associated with single block changes to the disk. A naïve approach would be to simply write all blocks in pairs, producing consecutive changes of two blocks. This would defeat our attack, but the adversary could expand its feature set to include changes two blocks long and so on, forcing the user of the hidden volume to mimic the public file system or risk detection. Furthermore, given the myriad of measurable features in the average file system and that the adversary is unlikely to publish details of its attack, we can conclude it would be difficult to accurately determine what features are and are not being tracked by the adversary, making mimicry or operational measures safer options.

While our proposed mitigation technique is a promising means of defense against snapshot analysis it is still worthwhile to explore the possibility of a mechanism that does not require such close user involvement. For a deniable storage system to defend against a snapshot attack without operational measures, the designer of such a system could attempt to mimic the distribution of writes that the public file system is making. Similar to our operational approach, the survivability of a hidden volume may be impacted by changing the distribution of writes from uniform to something more closely resembling a normal file system.

To accurately mimic expected access patterns, more work needs to be done to quantify exactly what a change pattern for a disk without a deniable volume looks like. A significant body of work has been published in the context of network steganography and pattern mimicking cryptography and could be used to inform designers of ways to design future multiple snapshot resistance systems. For instance, initial work into format-transforming encryption by Dyer *et al.* [34] showed that it was possible to efficiently encrypt data so that it would conform to a target regular expression. An application of this is found in censorship resistant networking. In this case, a user might be running a blacklisted protocol, such as Tor, but transforms the protocol messages in such a way that they look like HTTPS. Similar techniques could be applied to deniable storage to disguise suspicious write patterns. Unfortunately, there is also evidence that more capable adversaries utilizing more sophisticated attacks can easily identify these false protocols. Houmansadr *et al.* [52] argue that it is unlikely that one could mimic a protocol perfectly without running the actual protocol because there are often sub-protocols one would also need to mimic or differences in implementations that allow for version fingerprinting. Assuming snapshot analysis becomes more sophisticated it is likely that mimicry techniques applied to deniable storage would also need to evolve.

Since Artifice relies on pseudo-random data in free space and the use of secure deletion utilities to produce cover changes, one way to potentially sidestep the issues of artificial mimicry could be found in actually using these deleted files. Artifice could keep track of changes in free space on the public file system, and when it sees a block added to free space it could overwrite this with an Artifice block. As noted above this is not uniformly written and so may risk corruption of files, but it also may provide stronger mimicry guarantees than other methods. We leave it to future work to investigate this and the other techniques we have presented for mitigation.

7.4 Snapshot Attack Results

By implementing the experimental methodology described in §7.2 and running it against our dataset we collected a set of results that show the efficacy of our proposed multiple snapshot attack. It should be noted from the start that at the core of our implementation is a simple logistic regression based classifier that only takes into account the probability of single block changes on the disk. For this experiment, we utilized the Scikit-learn library's logistic regression classifier [101].

In choosing to implement such a simple learning algorithm we highlight the distinguishing power of analyzing consecutive block changes in detecting anomalous disk behavior.

We collected five different metrics on our classifier: accuracy, precision, recall, false positive rate, and false negative rate. Because only 5% of our test set contains Artifice instances, accuracy is not a very informative metric, and we include it for completeness only. Precision is the ratio of true positives to predicted positives. Recall is the ratio of true positives to the sum of true positives and false negatives, giving the ratio of Artifice instances that were identified from the test set. False-positive and false-negative rates are useful for understanding how frequently our classifier makes errors in both directions. We consider false classification rates to be the most important metrics for our adversary.

In our experiments the 250 MB Artifice instances were often able to pass undetected, implying that 25 GB of cover changes were sufficient to hide these volumes. However, the largest four sizes were reliably detected, with the three largest sizes, 0.75 GB, 1.0 GB, and 1.25 GB, being detected nearly 100% of the time. This highlights a feature of our attack, namely that for 25 GB of cover changes every Artifice instance above a certain size will be detected with high probability. This is because of our use of logistic regression, and because the probability of singletons is so overwhelming. Eventually, as free space fills up, Artifice will begin to make changes that are parts of longer chains, but if the reconstruction threshold is low, this will severely impact the survivability of the volume.

While these results show that it is possible to reliably detect an Artifice instance on a device by analyzing snapshots of the device, it is far more difficult than the theoretical attacks described in previous work. To carry out this attack an adversary must collect a large dataset of previous snapshots to build a statistical model with which they can identify suspicious patterns, obtain at least two snapshots of a suspect device, and then feed a comparison of those two snapshots through a classifier compared to other attacks that only require one observation of the device and no large dataset to compare against.

7.5 Entropy Analysis

An often untested assumption underlying the security of deniable storage systems is the theoretical impossibility of distinguishing obfuscated hidden data from random free space, as described in §4.2.3. Although indistinguishability provided by random unallocated and hidden blocks is a strong security property, it can be undermined by any blocks with lower entropy than their surroundings. Since Artifice uses IDAs instead of the more common symmetric ciphers to obfuscate data we must verify whether these IDAs reliably produce blocks with a high enough level of entropy to evade detection through entropy analysis. We assume that the presence of anomalous blocks with lower entropy than surrounding bytes would be suspicious and may be just as insecure as random obfuscated blocks residing in free space filled with only zeroes. This would mean that we must verify that all of our IDAs produce pseudo-random output that is indistinguishable from other pseudo-random blocks on the disk and whether there are any circumstances under which they may not.

To evaluate the effectiveness of our proposed IDA-based scheme we first tested each IDA with the `dieharder` [94] statistical test suite. The `dieharder` suite is a battery of statistical tests used to measure the quality of a random number generator that includes tests from both the older `diehard` [38] test suite and the NIST Statistical Test Suite [97]. With `dieharder`, we tested the quality of the pseudo-random output produced by AONT-RS, Shamir Secret Sharing, our Reed-Solomon/Entropy Scheme, and the Linux Kernel's Cryptographic API implementation of AES-CBC with a 256-bit key. Any pseudo-random numbers needed for AONT-RS, Secret Sharing, or AES were generated using the Linux kernel's random number generator. Entropy blocks for the Reed-Solomon scheme were generated by a cryptographically secure pseudo-random number generator based on the Speck cipher seeded with values from encrypted files. To utilize the test suite we had each algorithm encode a stream of data taken from a disk full of an assortment of text documents, music files, and videos. The output was then fed into the test suite. We found that each algorithm passed the battery of tests without issue which would imply that each IDA produces similar quality random numbers as our AES control. Although these tests are a good benchmark for the overall quality of cipher or IDA implementation and would indicate that encoded blocks should be indistinguishable from one another, it is not suited to identifying blocks with lower entropy than their surroundings.

With the quality of the output randomness of each algorithm verified using the `dieharder` tests, we move on to exploring tests that would allow us to identify lower entropy blocks in the free space of a file system. Specifically we aim to search for outliers or easily discernable differences in the level of entropy for each block. This is similar to what was done by Kedziora *et al.* [61] to find the start and end sectors of a hidden Veracrypt volume. To do this we compute the Shannon Entropy [106] of our encoded data. The entropy of an encoded block gives us a measurement of how uniformly the individual values for a set of bytes are distributed among the possible 256 values. The closer to a uniform distribution we get, the higher the entropy. Differences in this entropy between sets of bytes allow us to compare them and see if one is more "random" than the other. In the case of our experiment, the calculated entropy ranges from 0, or no entropy, to

8, a perfect uniform distribution of possible values for each byte. The equation for Shannon entropy given possible byte values x_i and the probability of a given byte value $P_{\text{byte}}(x_i)$ is shown in Equation 7.7. Each probability $P_{\text{byte}}(x_i)$ is derived from how frequently each possible byte value appears in a set of bytes.

$$H(X) = -\sum_{i=0}^{n} P_{\text{byte}}(x_i) \log_2(P_{\text{byte}}(x_i)) \tag{7.7}$$

As Artifice and most other block devices write data in discrete blocks we computed the Shannon entropy of each 4 KB block in our test dataset. The plotted entropy values for a 400 MB dataset of assorted images and PDF documents encoded with a variety of IDAs and AES operating in CBC mode can be seen in Figure 7.4.

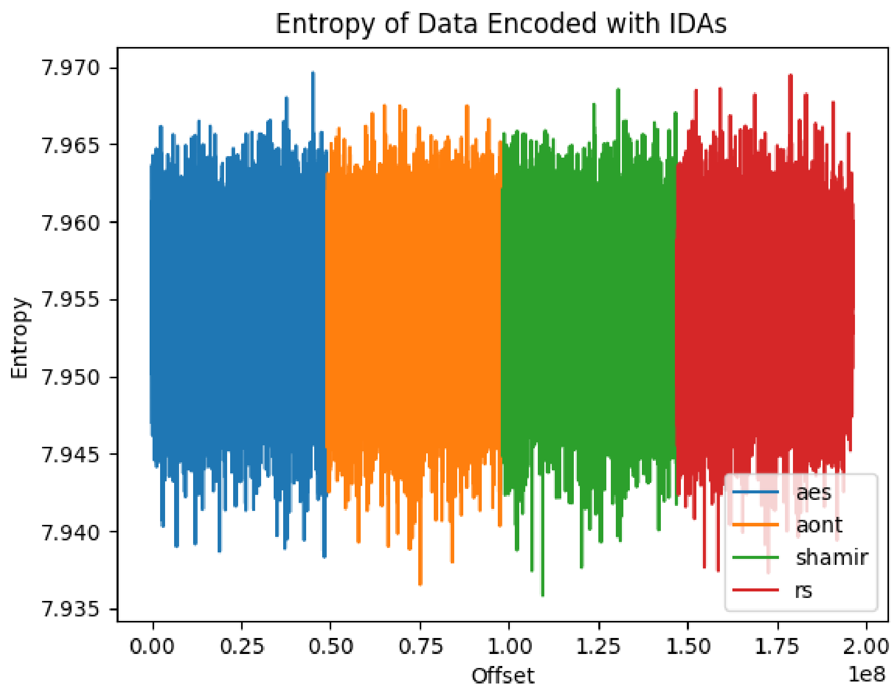


Figure 7.4: Entropy values per block for a variety of IDAs and AES. Note that the entropy value for each block is around 7.95 with some level of statistical noise and no easily visible outliers that would denote a flaw in one of the systems.

The same behavior was observed for each of our IDAs. No obvious outliers are apparent which is what would be expected based on the results of the previous battery of tests run.

To create a more precise test for distinguishing IDAs and encryption through entropy analysis we can look at the probability distribution of entropy values for a large array of blocks. To build a large experimental data set to look at these distributions, we processed 141 GB of data through our IDAs and AES. In the case of Reed-Solomon/Entropy, we used three different entropy sources, a cryptographically secure pseudo-random number generator seeded with values from a known source, encrypted data, and compressed data. As shown in Figure 7.5 these distributions when represented by a kernel density estimation taken over the dataset mostly follow the pattern of a Gaussian distribution with expected values close enough that the plots cannot be easily distinguished.

We can then verify that they follow a Gaussian distribution with the quantile-quantile plot shown in Figure 7.6. A cursory examination of this would indicate that our IDAs produce blocks that all follow the same distribution as those encrypted with AES. The notable exception to this pattern is Reed-Solomon/Entropy with a compressed entropy source which exhibits a relatively suspicious long tail. This long tail is indicative of a small number of encoded blocks with low levels of entropy outside of the expected range. While this number of low entropy blocks appears small, if found within unallocated space filled with high entropy blocksthey would be easy to identify and would appear suspicious.

Lastly to verify through another test whether the distributions for our IDAs are indistinguishable we have used a χ-squared test. A χ-squared test allows us to compare two sets of measurements and determine whether there is a statistically significant difference between the two. Our null hypothesis for this test is that each sample of entropy values came from the same distribution. We implemented this test in Python using the SciPy χ-square library and compared the distribution for AES to the distributions for each IDA. In almost every case our null hypothesis that both samples come from the same distribution held with a significance level of 0.05. From this and our previous tests we can conclude that data encoded with most of our IDAs is indistinguishable from blocks encrypted with AES. The exception to this pattern was Reed-Solomon/Entropy with compressed data as an entropy source, which returned a value lower than our significance level and thus rejected our null hypothesis. This indicates that there is a very high probability that the two compared samples came from different distributions. In short, we can conclude that in this last case the data encoded by this specific IDA configuration is not indistinguishable from pseudo-random blocks that we assume fill a public volume's unallocated space. Overall, from these results we can conclude that the chances of distinguishing carrier blocks from pseudo-random appearing data blocks in the free space of a file system are negligible if proper care has been taken when implementing the IDAs and choosing entropy sources.

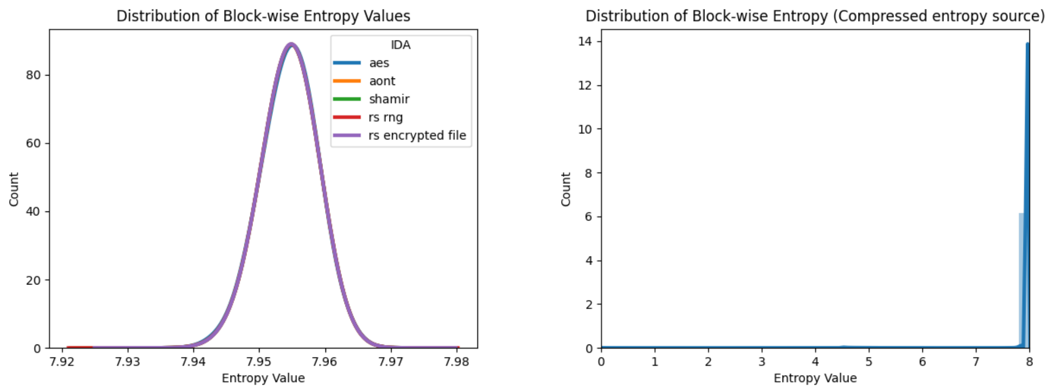


Figure 7.5: Right: Approximated probability distributions for the entropy of data encoded with a variety of IDAs and AES. Left: Probability distribution for Reed-Solomon/Entropy with compressed data as an entropy source, note the long tail indicative of lower entropy values.

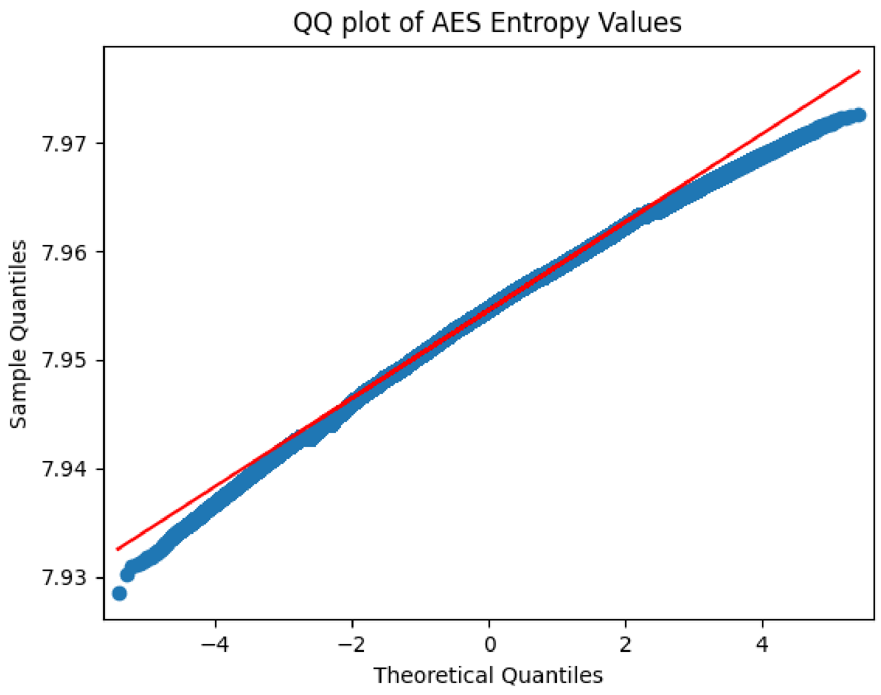


Figure 7.6: Quantile-Quantile plot showing the relationship between AES entropy values and a Gaussian distribution.

While an IDA producing lower entropy blocks is not acceptable when free space is full of random bytes, it may be acceptable in some situations depending on the contents of the device's free space. If the user does not fill their free space with random bytes but also ensures it has not been overwritten with zeroes, it could be difficult for our adversary to detect anomalies based on the level of entropy. The presence of lower entropy blocks in the free space of a disk is to be expected in a situation where the free space of the drive is filled by a single secure deletion operation as normal use of the drive will replace pseudo-random blocks with freshly deleted data over time. The resulting problem for our adversary is similar to the one posed by the multiple snapshot attack. It is resource-intensive to define what the "normal" level of entropy is. Although, relying on non-random free space to provide some obfuscation cannot provide an indistinguishability guarantee but it may provide a sufficient level of security in the case where random free space may be considered suspicious by an adversary.

7.6 Dealing with FTLs and TRIM

As previously stated in §4.2.5 we assume that the ideal situation when using a flash storage device with Artifice would be to disable TRIM. Disabling TRIM on a device with a deniable reason to do so, such as running disk encryption, is not an inherently suspicious behavior. While this solution is both deniable and effective it still leaves open questions regarding how an attacker can find an Artifice volume on a device with TRIM enabled and what can we do with Artifice to mitigate this threat.

First, we must look at how Artifice on a device with TRIM enabled would appear under basic forensic examination. For this example we utilized a 512 MB Artifice volume on a 120 GB Samsung 850 Evo SSD using periodic RZAT TRIM where reading trimmed logical blocks returns only zeroes. We wrote 68GB of regular data to the drive and then ran a TRIM operation to clear any unallocated blocks. To analyze the disk we computed the Shannon Entropy for each block on the device. The results of this test are shown in Figure 7.7 where each pixle is an individual block on the disk. Black pixels correspond to used logical blocks whereas gray pixels correspond to blocks containing only zeroes.

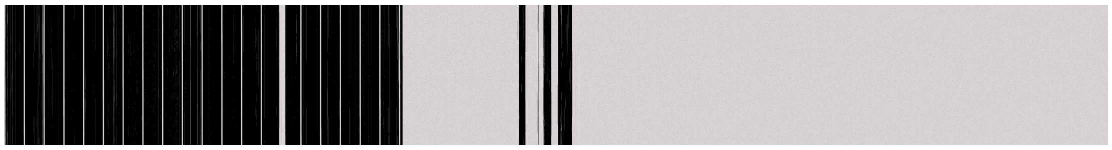


Figure 7.7: 120 GB SSD with 68 GB of free space after a TRIM operation. Black correspond to in use logical blocks, gray pixels are blocks that return only zeroes due to TRIM.

By creating and filling a hidden volume on the device with the default Artifice write behavior of placing carrier blocks uniformly across the disk, we can see a sprinkling of lone single blocks in otherwise unallocated space zeroed by TRIM. The distinctive pattern is shown more closely in Figure 7.8. This pattern of disk writes, even with a relatively small Artifice volume would be distinctive enough to raise suspicion. As demonstrated in §7.4 such a pattern of isolated single block writes can be easily and reliably detected through the user of a classifier trained on the probability of singleton blocks.

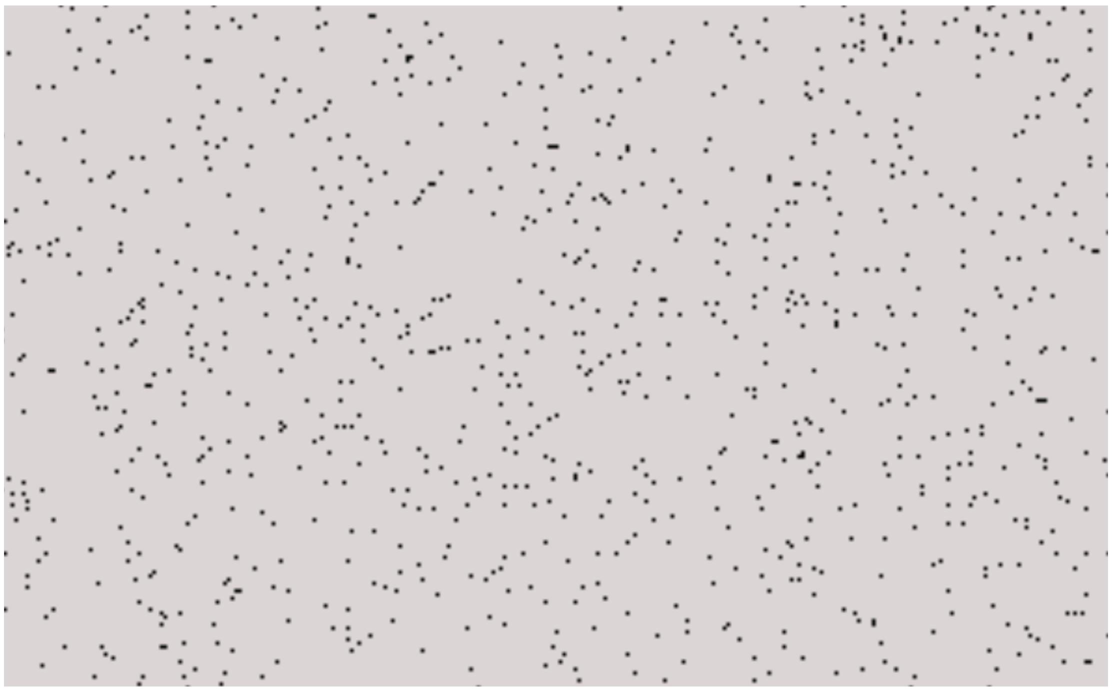


Figure 7.8: Region of free space on a 120 GB SSD with a 512 MB Artifice volume written. Note the random distribution of written blocks in otherwise unused and zeroed free space.

Although this write pattern is distinctive we can look to our multiple snapshot attack evaluation for a possible countermeasure. If a sufficient number of write operations are made to the public volume as cover traffic after the last TRIM operation, we can make the blocks distributed throughout free space appear less suspicious by manipulating the probability of single block changes. A drawback of this approach is that it may not entirely obscure the random pattern of writes if the adversary chooses to manually examine the disk in the manner that we have done in Figures 7.7 and 7.8. To avoid this, Artifice blocks could be contiguously allocated or limited to specific regions of free space to avoid the distinctive write pattern produced by random allocation. Such a limitation on block allocation would negatively impact the survivability of a hidden volume and result in a functional limit on the size of the volume and a subsequent increase in the level of redundancy needed to prevent data loss. Additionally, leaving even periodic TRIM enabled would significantly shorten the lifespan of an Artifice volume. At maximum this lifespan would be however long it is until the next TRIM operation.

The second possible countermeasure is to allow Artifice blocks to be trimmed and retrieve them at a later point in time through forensic techniques. There are two techniques for recovering data from flash after a TRIM operation is carried out. In the first approach the physical flash chips are removed from the drive's circuit board and the data is recovered using specialized tools in what is called off-chip analysis. This technique is reliable but very costly in terms of the labor and time needed to recover data. Alternatively, using aftermarket forensic tools we can put an SSD into something called Factory Access Mode [7]. Factory access mode allows us to read trimmed logical blocks or physical pages without having to completely disassemble the flash device. Once the data is read from either directly from the flash chips or through factory access mode, the process for recovering an Artifice volume is similar to what would be done normally. As long as enough carrier blocks were written to survive garbage collection operations performed by the device controller it would be possible to identify an Artifice superblock and recover the data. While factory access mode and off-chip analysis can sidestep the challenges posed by flash devices this technique for information hiding is not without its drawbacks.

Both techniques require specialized software and in some cases hardware to access the trimmed data. The procedures for enabling Factory Access Mode differ depending on the device manufacturer. The two techniques, while varying in their invasiveness both require disassembling a machine to remove the drive. These limitations mean that both methods of recovering trimmed data should only be carried out in a safe location that the adversary cannot observe or control. This approach also eliminates Artifice's ability to carry out self-repair operations by rendering the hidden blocks inaccessible by normal software.

While there are ways to have a hidden volume implemented at the block device or file system level that coexists with TRIM operations, we can see that the possible techniques for preserving deniability in the face of TRIM impose significant drawbacks. As a result, we can only recommend that these techniques be used in a situation where it would suspicious to disable TRIM.

7.7 Summary

While there is a large variety of existing deniable storage system designs, as stated in §3.4.2 there is very little in the way of demonstrated or thoroughly described attacks against these systems. We have demonstrated and explored a variety of avenues through which our adversary can establish whether they suspect a user is in possession of a hidden volume.

To start we have described the first multiple snapshot attack that analyzes numbers of consecutive block updates, or chains, to determine whether there is a statistically anomalous number of single block changes. Through gathering a dataset of block change records we established a distribution that described the probability of single block updates on the disk. With this dataset and simulated Artifice volumes we trained our model to reliably classify whether or not there was a hidden volume present on the test device. Additionally, were were able to demonstrate the circumstances underwhich our attack was unable to reliably distinguish whether a device did or did not contain a hidden volume. With this information we developed possible countermeasures the user could use to inflate the test's false-negative rate which include artificially skewing the ratio of public to hidden writes in favor of the public writes.

We verified the effectiveness of our IDAs in obfuscating data in a hidden volume through entropy analysis. We showed that if configured correctly our IDA-based approach produces carrier blocks that are indistinguishable from surrounding pseudo-random blocks in the unallocated space on the disk. We also showed circumstances where an insufficient source of entropy can produce lower entropy carrier blocks that can be easily identified under forensic examination.

Lastly we demonstrated how an adversary could use the unique behavior of flash devices that utilize TRIM to allow for easy identification of blocks written to the free space of an existing file system. Our originally proposed approach of simply disabling TRIM may not always be the best choice for a user so we discuss two methods of combating TRIM on a flash device. The first is to produce sufficient cover traffic and change Artifice's block allocation scheme to make our carrier blocks less obvious. Secondly, we can utilize niche features of flash devices such as factory access mode to TRIM hidden blocks and recover them later using off-the-shelf software tools.

Chapter 8

Conclusion and Future Work

Deniable storage systems such as Artifice stand to fill an ever more important niche in situations where normal disk encryption is not enough to ensure both secrecy and a user's safety. Previous attempts to fill this niche have been plagued with flawed assumptions, insecure design characteristics, and usability problems. It is Artifice's goal to address the shortcomings of previous deniable storage systems in light of a more realistic threat model. This encompasses not only implementing and evaluating a system but also exploring how such a system would be used.

To start, we had to first revisit the assumptions and priorities behind the design of existing deniable storage systems. These previous approaches frequently exhibit similar problems in how they assume an adversary will behave. Usually this entails counting on the adversary's incompetence and seem to imply that their security relies on their systems "flying under the radar". To address this fundamental problem and others, we have introduced a new threat model for deniable storage that assumes the adversary will be knowledgeable about this class of systems and be familiar with their weaknesses. Unlike previous models we assume the adversary will deploy malware to surveil the user and collect information to establish suspicion of whether the user possesses a hidden volume or not. In addition, this threat model makes assumptions about how our adversary would escalate their attacks against a user until they establish reasonable suspicion the user may possess a hidden volume before moving on to coercive tactics. From our new assumptions and a description of an example use case, we have defined the attack surface and described a series of attacks against deniable storage systems that can be used by our adversary and that Artifice must contend with.

To defend against this adversary, we have designed and implemented Artifice, an operationally secure, tunable, and self repairing deniable storage system that addresses the flaws of previous approaches. Unlike previous systems, Artifice provides a means to both deny the existence of data and the Artifice software itself. It obfuscates and protects the integrity of hidden data through the use of information dispersal algorithms (IDAs), mitigates the threats posed by information leakage and malware, and provides functional means for defending against the attacks described in our adversary model including the much-maligned multiple snapshot attack.

With a prototype of Artifice we were able to verify the effectiveness of our proposed IDA scheme in both obfuscating and protecting hidden data from accidental overwrite. We have implemented and demonstrated the first multiple snapshot attack against a deniable storage system, showing not only its effectiveness but also better characterizing the attack's limitations and identifying possible mitigations Artifice can deploy to defend itself. With a design and implementation of Artifice we have been able to develop an operational security model that would inform a user of what to do in the event that a component of the system is compromised and suggestions on system configuration to mitigate Artifice's weaknesses. Lastly, through our prototype, we have explored the threats posed by the unique design characteristics of flash devices and multiple avenues for mitigating them.

8.1 Future Work

While the evaluation of Artifice in Chapter 7 has shed light on many unknown and untested factors that can compromise a deniable storage system there is still room for further evaluation in the areas of snapshot and statistical analysis. There is also a need to explore the implications of new developments in the realm of flash storage technologies as they evolve.

Secure Deletion and Deniable Storage

While we have proposed an approach to modifying Artifice to co-exist with a secure-delete file system called Lethe (§4.3), we have yet to empirically evaluate Artifice's behavior when used alongside it. This is because Lethe is currently still a work in progress and consequently not yet ready for experimentation alongside Artifice. If we take our first proposed approach of having Artifice treat Lethe like any other file system, the primary question is, "what impact would the additional overwrites caused by garbage collection operations have on Artifice's reliability?" To measure this we would have to revisit the experiments described in §6.2 and §6.3 with new measurements that show the patterns and numbers of Lethe-incurred overwrites.

In the case of the second approach where Artifice blocks are written and then immediately deleted, we would not only have to revisit our survivability experiments, but also modify the current Artifice implementation and perform new statistical analysis experiments to determine whether Artifice blocks handled in this manner produce a pattern easily recognizable by the adversary.

Snapshot and Statistical Analysis

In §7, our multiple snapshot attack when used against Artifice showed the effectiveness of training a classifier based on only a single feature, the frequency of single block updates. This attack was relatively simple and provided avenues for additional research regarding the use of additional features and show how a more sophisticated model would impact our proposed mitigations. Additionally, due to the lack of suitable implementations for other deniable storage systems we were not able to apply our attack against systems that exhibit different write behavior. We hypothesize that other deniable storage systems would be vulnerable to our attack based on the write patterns their designs would produce, such as the uniformly distributed blocks of StegFS implementations [74, 80] or ORAM-based solutions [16, 20]. We consider it worthwhile to verify whether these hypotheses hold under experimental evaluation.

Zone Namespaces and Raw Flash

As we have previously stated in §4.2.5, newer storage technologies and methods for managing flash are currently under active development. A promising technology that could eliminate the need for an flash translation layer's current features, such as TRIM operations, is Zoned Namespaces (ZNS). The strict sequential writes of ZNS devices change how we must think about how Artifice writes to the disk, what constitutes a deniable operation, and what we consider unallocated space on the disk.

ZNS breaks up the logical block address space into fixed size zones that enforces strict sequential writes to a zone. On the surface this means that we cannot perform easy in-place updates to the device and alters how free space is structured. With these new design characteristics in mind, it is unclear which blocks Artifice should write to in a deniable manner.

There is functionality in zoned devices that allows for random writes in specific namespaces or areas of a device's address space. While this would allow Artifice to operate similarly to how it does now, it would also limit the amount of space on the disk in which we have to hide data and would negatively impact the total size and survivability of an Artifice volume.

A possible solution to these challenges could be found in the log-structured approaches to Artifice described in §4.3 but more investigation into the specifics of how these devices fit into existing storage stacks will be required as they begin to appear in consumer devices. As these devices become more readily available, Artifice's design decisions should be re-examined to ensure that deniable storage can adapt to this new paradigm.

Bibliography

[1] "Project RC5," https://www.distributed.net/RC5, [Online; accessed 27-September-2021].

[2] *mkfs.ntfs(8) Linux System Manager's Manual*, January 2006. [Online]. Available: https://linux.die.net/man/8/mkntfs

[3] *mkdosfs(8) Linux System Manager's Manual*, October 2017. [Online]. Available: https://man7.org/linux/man-pages/man8/mkdosfs.8.html

[4] *ext2(5) Linux File Formats Manual*, March 2020. [Online]. Available: https://man7.org/linux/man-pages/man5/ext2.5.html

[5] *ext4(5) Linux File Formats Manual*, March 2020. [Online]. Available: https://man7.org/linux/man-pages/man5/ext4.5.html

[6] Aaron Griffin and others, "The Arch Linux Wiki: Solid state drive," https://wiki.archlinux.org/index.php/Solid_state_drive, [Online; accessed 11-February-2020].

[7] O. Afonin, "Life after Trim: Using Factory Access Mode for Imaging SSD Drives," 2019. [Online]. Available: https://blog.elcomsoft.com/2019/01/life-after-trim-using-factory-access-mode-for-imaging-ssd-drives/

[8] R. Anderson, *Information Hiding*, ser. Proceedings of the International Workshop on Information Hiding. Springer, 1996.

[9] R. Anderson, R. Needham, and A. Shamir, "The Steganographic File System," in *International Workshop on Information Hiding*, D. Aucsmith, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 73–82.

[10] J. Assange, R. P. Weinmann, and S. Dreyfus, "Rubberhose," https://web.archive.org/web/20100915130330/http://iq.org/~proff/rubberhose.org/, [Online; accessed 20-January-2022].

[11] A. Barker, Y. Gupta, S. Au, E. Chou, E. L. Miller, and D. D. E. Long, "Artifice: Data in Disguise," in *Proceedings of the Conference on Mass Storage Systems and Technologies (MSST '20)*, Oct. 2020.

[12] A. Barker, S. Sample, Y. Gupta, A. McTaggart, E. L. Miller, and D. D. E. Long, "Artifice: A Deniable Steganographic File System," in *9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/foci19/presentation/barker

[13] S. Bauer and N. B. Priyantha, "Secure Data Deletion for Linux File Systems," in *Proceedings of the 10th Conference on USENIX Security Symposium (SSYM '01)*, vol. 10. USENIX Association, 2001.

[14] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK Lightweight Block Ciphers," in *Proceedings of the 52nd Annual Design Automation Conference (DAC '15)*. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2744769.2747946

[15] M. Bjørling, "From Open-Channel SSDs to Zoned Namespaces." Boston, MA: USENIX Association, Feb. 2019, VAULT-2019: 1st USENIX Conference on Linux Storage and Filesystems. [Online]. Available: https://www.usenix.org/conference/vault19

[16] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu, "Toward Robust Hidden Volumes Using Write-Only Oblivious RAM," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. New York, NY, USA: ACM, 2014, pp. 203–214. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660313

[17] H. Bojinov, D. Sanchez, P. Reber, D. Boneh, and P. Lincoln, "Neuroscience meets cryptography: Crypto primitives secure against rubber hose attacks," *Communications of the ACM*, vol. 57, pp. 33–33, 08 2012.

[18] M. Broz and V. Matyás, "The TrueCrypt On-Disk Format–An Independent View," *IEEE Security Privacy*, vol. 12, no. 3, pp. 74–77, May 2014.

[19] M. Broz and A. Kergon, "dm crypt: optionally support discard requests," [Online; accessed 20-January-2022]. [Online]. Available: https://github.com/torvalds/linux/commit/772ae5f54d69c38a5e3c4352c5fdbdaff141af21

[20] A. Chakraborti, C. Chen, and R. Sion, "DataLair: Efficient Block Storage with Plausible Deniability against Multi-Snapshot Adversaries," *Computing Research Repository (CoRR)*, vol. abs/1706.10276, 2017. [Online]. Available: http://arxiv.org/abs/1706.10276

[21] B. Chang, F. Zhang, B. Chen, Y. Li, W. Zhu, Y. Tian, Z. Wang, and A. Ching, "MobiCeal: Towards Secure and Practical Plausibly Deniable Encryption on Mobile Devices," in *Proceedings of the 48th Annual IEEE/IFIP*

International Conference on Dependable Systems and Networks (DSN), 2018, pp. 454–465.

[22] B. Chang, Z. Wang, B. Chen, and F. Zhang, "MobiPluto: File System Friendly Deniable Storage for Mobile Devices," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: ACM, 2015, pp. 381–390. [Online]. Available: http://doi.acm.org/10.1145/2818000.2818046

[23] Z. Chang, D. Xie, and F. Li, "Oblivious RAM: A Dissection and Experimental Evaluation," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 1113–1124, Aug. 2016. [Online]. Available: https://doi.org/10.14778/2994509.2994528

[24] A. Cheddad, J. Condell, K. Curran, and P. M. Kevitt, "Digital image steganography: Survey and analysis of current methods," *Signal Processing*, vol. 90, pp. 727–752, March 2010.

[25] C. Chen, A. Chakraborti, and R. Sion, "Infuse: Invisible plausibly-deniable file system for nand flash," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 4, pp. 239 – 254, 01 Oct. 2020. [Online]. Available: https://content.sciendo.com/view/journals/popets/2020/4/article-p239.xml

[26] ——, "PD-DM: An Efficient Locality-preserving Block Device Mapper with Plausible Deniability," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, pp. 153–171, 01 2019.

[27] S.-H. Chen, M.-C. Yang, Y.-H. Chang, and C.-F. Wu, "Enabling File-Oriented Fast Secure Deletion on Shingled Magnetic Recording Drives," in *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317817

[28] J. Collins and S. S. Agaian, "Trends Toward Real-Time Network Data Steganography," *CoRR*, vol. abs/1604.02778, 2016. [Online]. Available: http://arxiv.org/abs/1604.02778

[29] L. Constantin, "Newly Found TrueCrypt Flaw Allows Full System Compromise," *PCWorld*, Sep. 2015.

[30] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier, "Defeating Encrypted and Deniable File Systems: TrueCrypt V5.1a and the Case of the Tattling OS and Applications," in *Proceedings of the 3rd Conference on Hot Topics in Security (HOTSEC '08)*. Berkeley, CA, USA: USENIX Association, 2008, pp. 7:1–7:7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1496671.1496678

[31] Dan Boneh and Richard J. Lipton, "A revocable backup system," in *Proceedings of the 6th USENIX Security Symposium.* USENIX Association, 1996. [Online]. Available: https://www.usenix.org/conference/6th-usenix-security-symposium/revocable-backup-system

[32] S. M. Diesburg and A.-I. A. Wang, "A Survey of Confidential Data Storage and Deletion Methods," *ACM Computing Surveys CSUR*, vol. 43, no. 1, pp. 2:1–2:37, Dec. 2010. [Online]. Available: http://doi.acm.org/10.1145/1824795.1824797

[33] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation Onion Router," in *Proceedings of the 13th Conference on USENIX Security Symposium (SSYM '04)*, vol. 13. USA: USENIX Association, 2004, p. 21.

[34] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, "Protocol Misidentification Made Easy with Format-Transforming Encryption," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013, pp. 61–72.

[35] J. Edge, "Thwarting the Evil Maid," *lwn.net*, July 2015.

[36] S. C. Elan Ashendorf, "Design of a Steganographic Virtual Operating System," vol. 9409, 2015, pp. 9409 – 9409 – 10. [Online]. Available: https://doi.org/10.1117/12.2079875

[37] W. Feng, C. Liu, Z. Guo, T. Baker, G. Wang, M. Wang, B. Cheng, and J. Chen, "MobiGyges: A mobile hidden volume for preventing data loss, improving storage utilization, and avoiding device reboot," *Future Generation Computer Systems*, vol. 109, pp. 158–171, 2020.

[38] George Marsaglia, "The Marsaglia Random Number CDROM including the Diehard Batter of Tests of Randomness," 1995, [Online; accessed 13-February-2022]. [Online]. Available: https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/

[39] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87).* ACM, 1987, pp. 182–194. [Online]. Available: https://doi.org/10.1145/28395.28416

[40] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, May 1996. [Online]. Available: http://doi.acm.org/10.1145/233551.233553

[41] Google, “The CityHash family of Hash Functions,” 2013, [Online; accessed 10-February-2020]. [Online]. Available: https://code.google.com/p/cityhash

[42] J. Gray, “Empirical Measurements of Disk Failure Rates and Error Rates,” Tech. Rep., December 2005. [Online]. Available: https://www.microsoft.com/en-us/research/publication/empirical-measurements-of-disk-failure-rates-and-error-rates/

[43] K. M. Greenan, J. S. Plank, and J. J. Wylie, “Mean time to meaningless: MTTDL, markov models, and storage system reliability,” in *2nd Workshop on Hot Topics in Storage and File Systems (HotStorage 10)*. Boston, MA: USENIX Association, Jun. 2010. [Online]. Available: https://www.usenix.org/conference/hotstorage-10/mean-time-meaningless-mttdl-markov-models-and-storage-system-reliability

[44] Y. Gubanov and O. Afonin, “White Paper: SSD Forensics 2014: Recovering Evidence from SSD drivers: Understanding TRIM, Garbage Collection, and Exclusions,” Belkasoft, Tech. Rep., 2014.

[45] ——, “White Paper: SSD and eMMC Forensics 2016,” Belkasoft, Tech. Rep., 2016.

[46] P. Hamamoto, “Letter to David Kaye, Special Rapporteur on the Promotion of the Right to Freedom of Opinion and Expression,” February 26 2015. [Online]. Available: www.ohchr.org/Documents/Issues/Opinion/Communications/States/USA.pdf

[47] J. Han, M. Pan, D. Gao, and H. Pang, “A Multi-user Steganographic File System on Untrusted Shared Storage,” in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC ’10)*. New York, NY, USA: ACM, 2010, pp. 317–326. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920309

[48] S. Hand and T. Roscoe, “Mnemosyne: Peer-to-Peer Steganographic Storage,” in *Proceedings of the International Workshop on Peer-to-Peer Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 130–140.

[49] ——, “Mnemosyne: Peer-to-peer steganographic storage,” in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 130–140.

[50] D. Hitz, J. Lau, and M. Malcolm, “File System Design for an NFS File Server Appliance,” in *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC ’94)*. USA: USENIX Association, 1994, p. 19.

[51] S. Hong, C. Liu, B. Ren, Y. Huang, and J. Chen, "Personal Privacy Protection Framework Based on Hidden Technology for Smartphones," *IEEE Access*, vol. 5, pp. 6515–6526, 2017.

[52] A. Houmansadr, C. Brubaker, and V. Shmatikov, "The Parrot Is Dead: Observing Unobservable Network Communications," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013, pp. 65–79.

[53] M. IDRASSI, "VeraCrypt, Known Issues & Limitations." [Online]. Available: https://veracrypt.codeplex.com/wikipage?title=Issues%20and%20Limitations

[54] S. Jia, L. Xia, B. Chen, and P. Liu, "NFPS: Adding Undetectable Secure Deletion to Flash Translation Layer," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16).* New York, NY, USA: Association for Computing Machinery, 2016, pp. 305–315. [Online]. Available: https://doi.org/10.1145/2897845.2897882

[55] ——, "Deftl: Implementing plausibly deniable encryption in flash translation layer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17).* New York, NY, USA: Association for Computing Machinery, 2017, pp. 2217–2229. [Online]. Available: https://doi.org/10.1145/3133956.3134011

[56] N. F. Johnson and S. Katzenbeisser, "A Survey of Steganographic Techniques," in *Proceedings of the International Workshop on Information Hiding.* Norwood, MA: Artech House, 2000, pp. 43–78.

[57] N. Joukov, H. Papaxenopoulos, and E. Zadok, "Secure Deletion Myths, Issues, and Solutions," in *Proceedings of the Second ACM Workshop on Storage Security and Survivability (StorageSS '06).* New York, NY, USA: Association for Computing Machinery, 2006, pp. 61–66. [Online]. Available: http://doi.acm.org/10.1145/1179559.1179571

[58] I. Jozwiak, M. Kedziora, and A. Melinska, "Theoretical and Practical Aspects of Encrypted Containers Detection – Digital Forensics Approach," in *Dependable Computer Systems.* Springer, 2011, pp. 75–85.

[59] S. Kaçamak and A. Uka, "Sound steganography using Shamir secret sharing scheme," in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, 2017, pp. 1–4.

[60] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable Secure File Sharing on Untrusted Storage," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03).* USA: USENIX Association, 2003, pp. 29–42.

[61] M. Kedziora, Y.-W. Chow, and W. Susilo, "Threat Models for Analyzing PlausibleDeniability of Deniable File Systems," *Journal of Software Networking*, vol. 2017, pp. 241–264, 2017.

[62] A. Kerckhoff, "La Cryptographie Militaire," *Journal des Sciences Militaires*, vol. IX, 1883.

[63] A. Kiayias and M. Yung, "Cryptographic Hardness Based on the Decoding of Reed-Solomon Codes," *IEEE Transactions on Information Theory*, vol. 54, no. 6, pp. 2752–2769, June 2008.

[64] D. E. Knuth, *The Art of Computer Programming, volume 2 (3rd ed.): Seminumerical Algorithms.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, p. 505. [Online]. Available: http://portal.acm.org/citation.cfm?id=270146

[65] H. Krawczyk, "Secret Sharing Made Short," in *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '93).* Berlin, Heidelberg: Springer-Verlag, 1993, pp. 136–146.

[66] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *13th USENIX Conference on File and Storage Technologies (FAST 15).* Santa Clara, CA: USENIX Association, Feb. 2015, pp. 273–286. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee

[67] J. Lee, J. Heo, Y. Cho, J. Hong, and S. Y. Shin, "Secure Deletion for NAND Flash File System," in *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC '08).* New York, NY, USA: Association for Computing Machinery, 2008, pp. 1710–1714. [Online]. Available: https://doi.org/10.1145/1363686.1364093

[68] Y. Li, N. S. Dhotre, Y. Ohara, T. M. Kroeger, E. Miller, and D. D. E. Long, "Horus: Fine-grained encryption-based security for large-scale storage," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13).* San Jose, CA: USENIX, 2013, pp. 147–160. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/li_yan

[69] Lianying Zhao and Mohammad Mannan, "Gracewipe: Secure and Verifiable Deletion under Coercion," in *Proceedings of the 22nd Annual Network and Distributed System Security Sympossium (NDSS)*, 2015.

[70] Linux Documentation Maintainers, "The Linux Kernel Documentation: FUSE," https://www.kernel.org/doc/html/latest/filesystems/fuse.html, [Online; access 24-February-2022].

[71] ——, "The Linux Kernel User's and Administrator's Guide: Device Mapper," https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/index.html, [Online; accessed 30-September-2020].

[72] ——, "The Linux Kernel User's and Administrator's Guide: Device Mapper: dm-zoned," https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-zoned.html, [Online; accessed 11-February-2020].

[73] E. S. Mario Bartiromo, "CNBC Interview," www.youtube.com/watch?v=A6e7wfDHzew, December 2009.

[74] A. D. McDonald and M. G. Kuhn, "StegFS: A steganographic file system for Linux," in *Proceedings of the International Workshop on Information Hiding.* Springer, 1999, pp. 463–477.

[75] R. C. Merkle, "A Digital Signature based on a Conventional Encryption Function," in *Proceedings of the Conference on the Theory and Application of Cryptographic Techniques.* Springer, 1987, pp. 369–378.

[76] Milan Broz, "dm-crypt," https://gitlab.com/cryptsetup/cryptsetup, [Online; accessed 20-January-2022].

[77] ——, "TRIM & dm-crypt ... problems?" http://asalor.blogspot.com/2011/08/trim-dm-crypt-problems.html, [Online; accessed 20-January-2022].

[78] Mounir Iddrassi, "Veracrypt," https://www.veracrypt.fr/en/Home.html, [Online; accessed 20-January-2022].

[79] J. Mull, "How a Syrian Refugee Risked His Life to Bear Witness to Atrocities," *Toronto Star Online*, March 2012. [Online]. Available: https://www.thestar.com/news/world/2012/03/14/how_a_syrian_refugee_risked_his_life_to_bear_witness_to_atrocities.html

[80] H. Pang, K. Tan, and X. Zhou, "StegFS: A Steganographic File System," in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, March 2003, pp. 657–667.

[81] J.-F. Pâris, T. Schwarz, D. D. E. Long, and A. Amer, "When MTTDLs are not good enough: Providing better estimates of disk array reliability," in *Proceedings of the 7th International Information and Telecommunication Technologies Symposium (I2TS '08)*, Dec. 2008.

[82] R. Perlman, "File system design with assured delete," in *Proceedings of the Third IEEE International Security in Storage Workshop (SISW'05)*, Dec 2005, pp. 6 pp.–88.

[83] T. Peters, M. A. Gondree, and Z. N. J. Peterson, "DEFY: A Deniable, Encrypted File System for Log-Structured Storage," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2015.

[84] Z. N. J. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. D. Rubin, "Secure Deletion for a Versioning File System," in *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies (FAST '05)*, vol. 4. USA: USENIX Association, 2005, p. 11.

[85] S. Piper, M. Davis, G. Manes, and S. Shenoi, "Detecting hidden data in ext2/ext3 file systems," in *IFIP International Conference on Digital Forensics*. Springer, 2005, pp. 245–256.

[86] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast Galois Field arithmetic using Intel SIMD instructions," in *Proceedings of the 11th Usenix Conference on File and Storage Technologies (FAST '13)*. San Jose, CA: USENIX Association, February 2013.

[87] C. Plumb, "shred (1)-delete a file securely, first overwriting it to hide its contents," Linux Manual Pages, http://linux.die.net/man/1/shred, 2004.

[88] M. O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance," *Journal of the ACM*, vol. 36, no. 2, pp. 335–348, Apr. 1989. [Online]. Available: http://doi.acm.org/10.1145/62044.62050

[89] J. Reardon, D. Basin, and S. Capkun, "SoK: Secure Data Deletion," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. USA: IEEE Computer Society, 2013, pp. 301–315. [Online]. Available: https://doi.org/10.1109/SP.2013.28

[90] J. Reardon, C. Marforio, S. Capkun, and D. Basin, "User-Level Secure Deletion on Log-Structured File Systems," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)*. New York, NY, USA: Association for Computing Machinery, 2012, pp. 63–64. [Online]. Available: https://doi.org/10.1145/2414456.2414493

[91] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960. [Online]. Available: https://doi.org/10.1137/0108018

[92] J. K. Resch and J. S. Plank, "AONT-RS: Blending Security and Performance in Dispersed Storage Systems," in *Proceedings of the 9th USENIX*

Conference on File and Stroage Technologies (FAST '11). USA: USENIX Association, 2011, p. 14.

[93] R. L. Rivest, "All-or-nothing encryption and the package transform," in *Procceedings of the International Workshop on Fast Software Encryption*, E. Biham, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 210–218.

[94] Robert G. Brown, Dirk Eddelbuettel, David Bauer, *Dieharder: A Random Number Test Suite*, https://webhome.phy.duke.edu/~rgb/General/dieharder.php, [Online; accessed 13-February-2022]. [Online]. Available: https://webhome.phy.duke.edu/~rgb/General/dieharder.php

[95] M. K. Rogers and K. Seigfried, "The Future of Computer Forensics: A Needs Analysis Survey," *Computers and Security*, 12 2003.

[96] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, Feb. 1992. [Online]. Available: https://doi.org/10.1145/146941.146943

[97] A. Ruhkin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, National Institute of Standards and Technology (NIST), April 2010.

[98] Sarah Dean, "FreeOTFE," https://web.archive.org/web/20130305192701/http://freeotfe.org/docs/Main/version_history.htm, [Online; accessed 20-January-2022].

[99] K. A. Scarfone, W. Jansen, and M. Tracy, "SP 800-123. Guide to General Server Security," Gaithersburg, MD, United States, Tech. Rep., 2008.

[100] T. Schwarz, A. Amer, T. Kroeger, E. Miller, D. Long, and J.-F. Pâris, "RESAR: Reliable Storage at Exabyte Scale," in *Proceedings of the 24th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '16)*, September 2016.

[101] Scikit-learn Contributors, "Scikit-learn," https://github.com/scikit-learn/scikit-learn, [Online; accessed 3-February-2022]. [Online]. Available: https://github.com/scikit-learn/scikit-learn

[102] SD4L Project, "ScramDisk for Linux," sd4l.sourceforge.net.

[103] A. Shah, V. Banakar, S. Shastri, M. Wasserman, and V. Chidambaram, "Analyzing the impact of GDPR on storage systems," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '19).* Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotstorage19/presentation/banakar

[104] S. Shalev-Shwartz *et al.*, "Online learning and online convex optimization," *Foundations and trends in Machine Learning*, vol. 4, no. 2, pp. 107–194, 2011.

[105] A. Shamir, "How to Share a Secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[106] C. E. Shannon, "A Mathematical Theory of Communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 1948. [Online]. Available: http://plan9.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf

[107] SignalApp, "Github: WhisperYAFFS," https://github.com/signalapp/WhisperYAFFS/wiki, [Online; accessed 20-January-2022].

[108] D. Silverstone, "Library for Shamir Secret Sharing in Galois Field 2**8," https://github.com/jcushman/libgfshare, 2006, [Online; accessed 20-January-2022].

[109] A. Skillen and M. Mannan, "On Implementing Deniable Storage Encryption for Mobile Devices," in *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, February 2013. [Online]. Available: https://spectrum.library.concordia.ca/975074/

[110] C. H. Sobey, "Recovering Unrecoverable Data - The Need for Drive-Independent Data Recovery," ChannelScience white paper, 2004.

[111] C. H. Sobey, L. Orto, and G. Sakaguchi, "Drive-Independent Data Recovery: The Current State-of-the-Art," in *IEEE Transactions on Magnetics*, ser. TMRC 2005, 2005, pp. 188–193.

[112] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," *Journal of the ACM*, vol. 65, no. 4, Apr. 2018. [Online]. Available: https://doi.org/10.1145/3177872

[113] Stegfs Development Team , "Stegfs," albinoloverats.net/projects/stegfs.

[114] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti, “POTSHARDS: Secure Long-term Storage Without Encryption,” in *2007 USENIX Annual Technical Conference (ATC ’07)*. USENIX Association, 2008.

[115] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger, “Self-securing storage: protecting data in compromised systems,” in *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, Dec 2003, pp. 195–209.

[116] The Debian Project, “Debian Wiki: SSD Optimization,” https://wiki.debian.org/SSDOptimization, [Online; accessed 11-February-2020].

[117] The Tails Project, “Tails: The Amnesic Incognito Live System,” https://tails.boum.org/, [Online; accessed 11-February-2020].

[118] Z. Throckmorton, “USB 3.0 Flash Drive Roundup,” *Anandtech*, July 2011. [Online]. Available: https://www.anandtech.com/show/4523/usb-30-flash-drive-roundup/3

[119] C. Troncoso, C. Diaz, O. Dunkelman, and B. Preneel, “Traffic analysis attacks on a continuously-observable steganographic file system,” in *Proceedings of the 9th International Workshop on Information Hiding*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 220–236.

[120] Truecrypt Foundation, “Truecrypt,” http://truecrypt.sourceforge.net, [Online; accessed 20-January-2022].

[121] N. Tyagi, M. H. Mughees, T. Ristenpart, and I. Miers, “BurnBox: Self-Revocable Encryption in a World Of Compelled Access,” in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 445–461. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/tyagi

[122] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, “Reliably Erasing Data from Flash-Based Solid State Drives,” in *Proceedings of the 14th Usenix conference on file and storage technology (FAST ’11)*, vol. 11, 2011, pp. 8–8.

[123] Wikipedia contributors, “Key Disclosure Law — Wikipedia, The Free Encyclopedia,” 2020, [Online; accessed 27-January-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Key_disclosure_law

[124] ——, “Rubber-hose Cryptanalysis — Wikipedia, The Free Encyclopedia,” 2020, [Online; accessed 27-January-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Rubber-hose_cryptanalysis

[125] ——, “NSA ANT catalog — Wikipedia, The Free Encyclopedia,” 2021, [Online; accessed 26-July-2021]. [Online]. Available: https://en.wikipedia.org/wiki/NSA_ANT_catalog

[126] ——, “Pegasus (spyware) — Wikipedia, The Free Encyclopedia,” 2021, [Online; accessed 26-July-2021]. [Online]. Available: https://en.wikipedia.org/wiki/Pegasus_(spyware)

[127] X. Yu, B. Chen, Z. Wang, B. Chang, W. T. Zhu, and J. Jing, “MobiHydra: Pragmatic and Multi-level Plausibly Deniable Encryption Storage for Mobile Devices,” in *Information Security*, S. S. M. Chow, J. Camenisch, L. C. K. Hui, and S. M. Yiu, Eds. Cham: Springer International Publishing, 2014, pp. 555–567.

[128] R. Zhong, “China Snares Tourists’ Phones in Surveillance Dragnet by Adding Secret App,” *The New York Times*, July 2019. [Online]. Available: https://www.nytimes.com/2019/07/02/technology/china-xinjiang-app.html

[129] X. Zhou, H. Pang, and K. Tan, “Hiding Data Accesses in Steganographic File System,” in *Proceedings 20th International Conference on Data Engineering*, April 2004, pp. 572–583.

[130] A. Zuck, Y. Li, J. Bruck, D. E. Porter, and D. Tsafrir, “Stash in a flash,” in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST ’18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 169–188. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/zuck

[131] A. Zuck, U. Shriki, D. E. Porter, and D. Tsafrir, “Preserving Hidden Data with an Ever-Changing Disk,” in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS ’17)*. New York, NY, USA: ACM, 2017, pp. 50–55. [Online]. Available: http://doi.acm.org/10.1145/3102980.3102989