FROST – Anti-Forensics Digital-Dead-DROp Information Hiding RobuST to Detection & Data Loss with Fault tolerance

Avinash Srinivasan
Temple University
Philadelphia, Pennsylvania
avinash@temple.edu

Hunter Dong
Temple University
Philadelphia, Pennsylvania
hunter@temple.edu

Angelos Stavrou
George Mason University
Fairfax, Virginia
astavrou@gmu.edu

ABSTRACT

Covert operations involving clandestine dealings and communication through cryptic and hidden messages have existed since time immemorial. While these do have a negative connotation, they have had their fair share of use in situations and applications beneficial to society in general. A "Dead Drop" is one such method of espionage trade craft used to physically exchange items or information between two individuals using a secret rendezvous point. With a "Dead Drop", to maintain operational security, the exchange itself is asynchronous. Information hiding in the slack space is one modern technique that has been used extensively. Slack space is the unused space within the last block allocated to a stored file. However, hiding in slack space operates under significant constraints with little resilience and fault tolerance.

In this paper, we propose FROST – a novel asynchronous "Digital Dead Drop" robust to detection and data loss with tunable fault tolerance. Fault tolerance is a critical attribute of a secure and robust system design. Through extensive validation of FROST prototype implementation on Ubuntu Linux, we confirm the performance and robustness of the proposed digital dead drop to detection and data loss. We verify the recoverability of the secret message under various operating conditions ranging from block corruption and drive de-fragmentation to growing existing files on the target drive.

CCS CONCEPTS

•Applied computing → Computer forensics; Data recovery; •Computer systems organization → Redundancy; •Information systems → *Storage recovery strategies;*

KEYWORDS

Anti-forensics, Detection, Fault Tolerance, File Systems, Hashing, Information Hiding, Robust, Security, Slack Space, Steganography, Threshold Secret Sharing

ARES '17, Reggio Calabria, Italy

978-1-4503-5257-4/17/08…$15.00
DOI: 10.1145/3098954.3106069

ACM Reference format:
Avinash Srinivasan, Hunter Dong, and Angelos Stavrou. 2017. FROST – Anti-Forensics Digital-Dead-DROp Information Hiding RobuST to Detection & Data Loss with Fault tolerance. In *Proceedings of ARES '17, Reggio Calabria, Italy, August 29-September 01, 2017,* 8 pages.
DOI: 10.1145/3098954.3106069

1 INTRODUCTION

Hiding information in *slack space* is one of the numerous information hiding (IH) techniques proposed over the last two decades. Slack space based IH techniques, while simple to implement, are vulnerable and operate with serious limitations. Slack space is the unused space within the last allocated block of a file on a secondary storage. Hiding in slack space not only suffers due to the dynamic nature of the storage volume, especially when the target is a bootable partition, but also from file system actions such as *file modification* and *file deletion* and system actions such as *defragmentation*.

The slack space of a file is the first place the system will write to when the file grows in size. Consequently, a critical requirement of various slack space IH tools and technique is *reliability* and *fault tolerance*. Finally, the IH capacity of existing IH tools and techniques is dictated by available slack space on the target volume. Note that a sector is the smallest amount of physical allocation unit that a drive controller can write to. A block (a.k.a. cluster) on the other hand is a logical storage unit, which is a collection of two or more sectors. A block is the smallest amount of space that can be assigned to a file stored on a secondary disk.

Now, a *"Dead Drop"* is a container, one that is not easily found, such as a magnetized box attached to a metal rack in an out-of-sight alley[1]. It should be possible for a user to approach the dead drop to either drop off or pick up information/items. Neither the drop box itself nor the user approaching it should be easily observed. The dead drop enables information exchange between two individuals using a predetermined secret location, thus not requiring them to meet directly and thereby maintaining operational security.

The dead drop method stands in contrast to the live drop method, which necessitates that two people meet in order to exchange information. From a real-world perspective, the last known *Dead Drop* case was in 2006 when the Russian FSB accused Britain of using wireless dead drops concealed inside hollowed-out rocks to collect information from agents in Russia.

In this paper, we propose "FROST" – a digital dead drop for covert asynchronous information exchange. This digital dead drop is a tunable fault-tolerant IH technique that is robust to detection

[1] https://en.wikipedia.org/wiki/Dead_drop

and data loss for various reasons including block corruption and file modification. FROST employs a storage volume's (bootable or non-bootable) allocated files' slack space as the information exchange drop point. Finally, FROST achieves all three core information security requirements – *confidentiality*, *integrity*, and *availability*. Our approach provides fault tolerance as a tunable parameter making it very robust to data loss. We provide the design details and validate the performance of FROST through a prototype implementation on a Ubuntu Linux machine.

IH in slack space – Overview. While slack space is one of the most popular areas on the secondary storage system for hiding information, there are numerous other areas on a secondary storage volume that provide similar capabilities when it comes to IH. Some of the more popular IH techniques include – *obfuscated file names* [15], *file encryption*, *disk and volume encryption (e.g., TrueCrypt, BitLocker)*, *Encrypting File Systems (EFS)*, *partition slack*, *volume slack*, *Host Protected Area (HPA)*, and *Device Configuration Overlay (DCO)*.

Most users are unaware of the existence of these protected areas and even the OS is not allowed to access the HPAs and DCOs. Information stored in HPAs and DCOs can be accessed only with specific privileged commands by users. Particularly, crypto-based approaches are more for confidentiality preservation.

Hiding in slack space is markedly different from hiding in unallocated space. Slack space is part of the allocated space where as unallocated space is the set of all blocks available to the OS. Unallocated space consists of unused blocks and/or freed blocks that contain remnants of deleted files. There are numerous tools that can wipe the unallocated space, and the modern day PC comes preloaded with system utilities to this aim.

In addition, since deletion of a file only marks the corresponding block(s) as free, which can be used by the OS to store data in the future, there is no certainty that all of the blocks released will be used by the OS. Data in slack space has significantly higher odds of surviving situations that other popular IH tools and techniques fail to. Therefore, the only way to permanently get rid of content in the slack space of files is to erase/wipe the corresponding blocks.

1.1 Summary of Contributions

Our contributions in this paper can be summarized as follows. To the best of our knowledge, FROST presents the first work to propose an information hiding technique in slack space with the following features –

(1) First IH technique with built-in tunable fault tolerance capabilities. It is also the first IH technique that employs strong and provably secure cryptographic primitives.
(2) First of its kind digital dead drop designed, implemented, and analyzed extending the idea of the physical world counterpart. A Ubuntu Linux VM prototype implementation is used to validate FROST.
(3) Robust to information loss resulting from dynamic nature of system events such as generation temporary files, storage volume corruption, disk de-fragmentation, etc.
(4) Is unique compared to contemporary steganography techniques since it does not alter the payload of the cover file. Hence, FROST design is inherently resilient to detection tools and techniques utilizing payload analysis.
(5) Recovery of secret shares is based on physical address as a byte offset from the absolute first byte of the disk. This property enables FROST to recover secret shares even if the cover file is deleted or the storage volume is de-fragmented.

1.2 Road Map

The remainder of this paper is organized as follows. In section 2, we present the necessary background information and preliminaries of our work. We then provide a detailed discussion of FROST in section 3. We present the implementation details of a working prototype in section 4, which also contains the results from the evaluation of our prototype on real-world systems. In section 5, we provide detailed analysis of FROST's security robustness and performance followed by a review of related literature in section 6. Finally, we conclude our work by highlighting the significance of FROST along with directions for future research in section 7.

2 PRELIMINARIES

In this section, we discuss the preliminaries of our work including necessary cryptographic building blocks. The target storage disk is referred to as a storage volume denoted as $\mathcal{V}$, which can be either bootable or storage-only. Additionally, $\mathcal{V}$ can be either internal or an external disk.

Although files systems can be categorized based on numerous features, one key classification feature that is of significance to our work and in alignment with its key objectives is based on *block sub-allocation*. Block sub-allocation is a feature of some file systems that allows large blocks to be used while making efficient use of *slack space* at the end of large files.

This classification results in two broad categories based on whether or not file systems support the implementation of block suballocation. File systems that support block suballocation include the two very popular and widely used file systems – *FreeBSD UFS2* and *Brtfs* [18]. However, contemporary file systems used predominantly today include *NTSF*, *ext3*, *ext4*, and *HFS+* that do not support file system *block sub-allocation*.

Let the set of all files on $\mathcal{V}$ be denoted as $F = \left\{ f_1, f_2, \cdots f_N \right\}$. Now, F can be categorized into the following two distinct subsets: i) F_{nor} – the set of files that don't hold secret shares in their slack space; and ii) F_{cov} – the set of files that are cover files, each holding a unique secret share. Therefore, we have the following: $F = F_{nor} \bigcup F_{cov}$.

2.1 File Systems Internal Fragmentation

On a typical storage volume $\mathcal{V}$, the file system uses a fixed size block (b_{fs}^{size}) allocation algorithm. This fixed size of blocks is specific to $\mathcal{V}$ and is specified within the volume boot record. When a new file is created on/copied to $\mathcal{V}$, each file starts at the beginning of a new block. This simplifies their organization and makes it easier to track as each file grows.

An important thing to note is that files come in all types and sizes. Note that, even if the actual data being stored requires less storage than the block size, an entire block is reserved for the file. The pair $\langle b_i, f_x \rangle$ always exhibits a *one-to-one* relationship while

$\langle f_x, b_y \rangle_{(\forall y \in \{1,2,\cdots,m\})}$ exhibits a *one-to-many* relationship. Every single file on $\mathcal{V}$, irrespective of the underlying file system, has two associated sizes: i) *logical size* (f_x^{ls}) – cumulative size of all the blocks allocated to the file; and ii) *physical size* (f_x^{ps}) – actual size of the file representing the actual contents of the file. While the *logical size* of a file is almost always greater than its *physical size*, the two sizes can be equal – $f_x^{ls} \geq f_x^{ps}$. Let file f_x be assigned blocks $\langle b_1, b_2, \cdots, b_m \rangle$ where b_m is the last block assigned to f_x. The amount of space unused and wasted inside b_m is the slack space denoted as f_x^{ss} and $f_x^{ss} = f_x^{ls} - f_x^{ps}$.

The slack space itself can be categorized into the following two types: *i) RAM slack – unused space from end of physical file to end of sector;* and *ii) Drive slack – unused sectors within the last block assigned to the file.* Slack space is unusable since it is internal to the last allocated block of a file. Allocating such space to more than one file is prohibited as noted above.

Two key actions on files that will alter the file payload are – *adding content to the file* and *deleting content from the file.* Among these two actions, deleting content from a cover file is not as serious a threat as adding content. This is because when content is deleted from a file, depending on the size of the deleted content, one or more blocks are freed and released to the OS for reuse.

At this point, logical access to the block with the secret share is lost, while the secret share itself remains intact. Therefore, as long as the freed blocks of a f_x^{cov} are not overwritten, the corresponding secret share can be retrieved. To this aim, FROST is designed to access secret shares using their physical address. Similarly, FROST is robust to disk de-fragmentation as long as the the blocks containing the secret shares are not overwritten.

2.2 Threshold Secret Sharing Schemes

The concept of secret sharing among n parties has been employed in a wide array of applications including numerous cryptographic protocols. Some popular applications include *secure multiparty computation* [1, 2], *proactive secret sharing* [5], *secure key management* [7, 11], and *Byzantine agreement among participants* [12]. However, with the original secret sharing scheme, there is one major problem – it has *zero fault tolerance* since all n shares are necessary to recover the original secret.

To overcome this problem, Shamir presented (t, n)-threshold secret sharing [14]. A (t, n)-threshold secret sharing is a method of sharing a secret among a given set of n users such that any subset of t participants constitutes an authorized set and can recover the secret by pooling their shares together while no subset of less than t participants can do so [14].

3 INFORMATION HIDING WITH TUNABLE FAULT TOLERANCE – FROST

FROST generates the secret shares from the original secret file using a polynomial method of computation, computes a SHA-1 for each share, concatenates each share with its corresponding SHA-1, and then it writes each concatenated $[share \,||\, SHA\text{-}1]$ in the slack of a unique cover file. FROST also generates a map file for each hidden secret file. The map file consists of one entry for each hidden $[share \,||\, SHA\text{-}1]$, which is a tuple of the form $<s_{id}, pa^{bo}>$, where pa^{bo} is the physical address of the start byte offset of the share in the slack space.

Each $[share \,||\, SHA\text{-}1]$ is subjected to integrity verification prior to processing it. Once t valid shares are recovered, where t is called the threshold, the original secret file is reconstructed with the inverse of the polynomial function that was used to generate the shares. FROST is an enhanced information hiding mechanism leveraging fundamental cryptographic primitives that are provably secure. This augmented technique provides reliability when saving secret data in the slack space of existing allocated files. This section outlines the design and functional details of the FROST IH mechanism.

With FROST, a secret message f_{sec} is processed into n file shares each of which is hidden in the slack space of a unique files on a target storage volume $\mathcal{V}$. To recover the entire secret file f_{sec}, any subset of t valid shares are required, where $t \leq n$. FROST provides in-place integrity verification of extracted shares ensuring secret file recovery is attempted only when t valid shares are available.

Secret message shares are hidden within the slack space of other allocated files on the target volume $\mathcal{V}$. To this end, FROST provides a choice between sequential and random selection of cover files for hiding the secret shares along with its hash.

Once hidden, the secret file f_{sec} can be recovered in its original form only when a minimum of t valid shares are input to the recovery function. Since t is a tunable parameter, its value can be set to any value such that $t \in \{2, 3, \cdots, (n-1)\}$. The values of $t = 1$ and $t = n$ present trivial boundary conditions, each of which has a security weakness. A value of $t = 1$ implies that each secret share is the complete secret message. This scenario is vulnerable since the adversary need only to obtain a single share to extract the entire secret file in its original form.

Despite being encrypted, it is fairly simple for the adversary to decrypt a single share using the popular cryptanalytic *ciphertext only* attack. Additionally, if the adversary scans the system slack space and locates large number of slack spaces with identical content – which can be easily determined through hashing, it can be a easy give-away.

On the other hand, a value of $t = n$ is very secure and robust to cryptanalytic attacks since each share in the slack of a unique file will have a different payload, consequently making it hard for the adversary to obtain any type of information regarding slack space data. However, the scheme is extremely sensitive and has zero fault tolerance. Loss of even a single share due to cover file modifications or block corruption will make it impossible for the original secret message to be recovered. In the following paragraphs, we provide the step-by-step working of the proposed FROST:

Step-1: Process secret file into n secret shares. The secret message is first processed into n shares of equal size. Let each share of f_{sec} be denoted as s_j and the set of all secret shares of f_{sec} be denoted as S. f_{sec} is interpreted as a binary string as shown in equation 1, where n_p is a prime number such that $n_p \geq n$, and $d > 0$ denotes the bit-length

of the secret share.

$$s \in \{0,1\}^{d(n_p-1)} \tag{1}$$

$$S = \bigcup_{j=1}^{n_p-1} s_j \in \{0,1\}^d \tag{2}$$

The set of all shares of f_{sec} is presented in equation 2. The contents f_{sec} are pre-processed by mapping it to $\mathbb{Z}_p$ such that $f_{sec} \in \mathbb{Z}_p$. Subsequently, $t-1$ elements are randomly chosen from $\mathbb{Z}_p$, and these $(t-1)$ elements are denoted as $\{a_1, a_2, \cdots, a_{(t-1)}\}$. Additionally, the user needs to set $a_0 = f_{sec}$. We have s_0, which is a zero string such that –

$$s_0 = 0^d \tag{3}$$

$$s_0 \oplus a = a \tag{4}$$

At this point, the shares can be computed as presented in equation 5.

$$y_i = \left\{ a(x_i) \mid i = \{0, 1, \cdots, n-1\} \right\} \tag{5}$$

Additionally, an MD5 hash value is generated for each share s_i and appended to the share. The process of generating secret shares, a hash value for each share, and appending each share and its corresponding hash value is performed off-line by the user intending to share the message. Note that FROST indeed appears to have key characteristics of a steganographic technique. However, a closer look clearly draws its distinction from contemporary steganography techniques.

Unlike steganography-based IH, FROST never alters the actual payload of the cover file. Instead it relies on effectively utilizing the unused space within the last allocated cluster of the cover file, which is otherwise inaccessible to the user and the OS.

Step-2: **Hide each secret share in distinct slack space.** The secret shares generated from the given secret file are now each written to the slack space of unique files on the target storage volume $\mathcal{V}$. When hiding the secret shares, the user creates a map file f_{map} with the physical address as a byte offset for each share.

The f_{map} also includes a hash of the original secret message $SHA\text{-}1[f_{sec}]$. The f_{map} can be optionally encrypted using an asymmetric algorithm making it easy to exchange encrypted data with colluding partners. Finally, the user can exchange the f_{map} off-line with colluding partners to ensure complete secrecy.

Step-3: **Reconstruct the original secret message.** The secret message $extractor()$ binary reads the map file f_{map} and extracts one share of the secret message at a time, verifying the integrity of each extracted share. This process continues until t valid shares have been extracted. At this point, the $rebuild()$ binary is invoked, which will take as input the t valid shares and output the reconstructed original message.

At this point, $msg\_verify()$ binary is invoked to authenticate the recovered secret message. This is done by computing the hash of the recovered secret message and comparing it to the hash of the original secret message included in the map file. If the $extractor()$ binary completes extracting all hidden secret shares without successfully recovering t valid shares, then it terminates with a message to the user. Note that a user intending to recover the original secret message has to successfully extract a minimum of t valid shares. This set of t valid shares constitute a qualified subset of secret shares $S' \subseteq S$.

4 FROST VALIDATION RESULTS

4.1 Environment and Evaluation Parameters

In our prototype implementation, we are utilizing the data on occurrence and size of slack space on the real-world drives presented in [9, 10]. With threshold secret sharing mechanisms, three important parameters are – i) total number of shares n, ii) threshold value t, and iii) the difference $(n-t)$. All three parameters can be tuned to optimize the performance of FROST.

We implement a working prototype of FROST on a Ubuntu Linux machine and validate the performance of FROST under different scenarios by systematically fine tuning one parameter at a time. Parameters critical to a robust and fault-tolerant solution, such as n, t, $\mathcal{V}_{size}$, f_{sec}, and F among others, were tuned measuring the fault tolerance of FROST in terms of secret shares' survivability and secret message recoverability. Below, we present two scenarios discussing the impact of the three parameters on the robustness and fault tolerance capabilities of FROST.

Case-1: Lower threshold value with higher $(n-t)$ augments resilience of FROST with robust fault tolerance. User can recover the secret message in its entirety even if a significant number of shares are lost. Essentially, if the number of shares that survive $n_{survive} \geq t$, the original secret message can be successfully reconstructed from the surviving shares. However, keeping t to be low, irrespective of n, significantly increasing the chance of either a third-party successfully retrieving the secret message through guessing or brute-force methods.

Case-2: Keeping both n and t large increases the storage space and computational time of recovering the secret file f_{sec} in its entirety, even if a significant portion of the secret is lost or gets corrupt over time due to normal system operations and not enough survive in order to recover the secret file.

4.2 Evaluation Parameters

In our emperical evaluations, we construct the secret message shares using the threshold secret sharing algorithm [14]. Below are some of the key parameters we consider in our prototype implementation and validation of FROST.

- Number of shares $n = \{25, 30, 35, 40, 45, 50\}$
- Threshold value $t \leq \lceil \frac{n}{2} \rceil$
- Target $\mathcal{V}_{size}$: 100K, 250K, and 500K blocks
- Target volume block size: 2048 bytes

Furthermore, during extraction of secret shares for message reconstruction, we can divide the set of n shares into two subsets – the set of shares '$n_{survive}$' that survived and are intact, and the set

of shares '$n_{corrupt}$' that were corrupted and are now unusable – during extraction of secret shares for message reconstruction.

4.3 Cover File Modifications & Recoverability

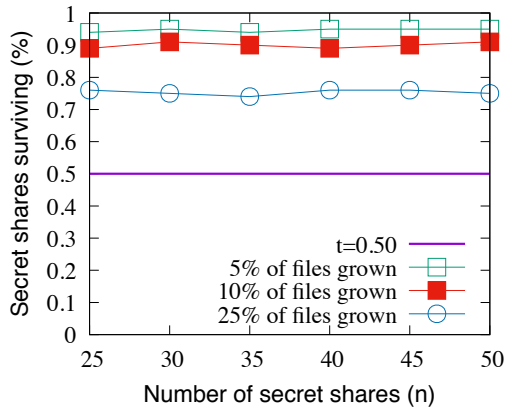


Figure 1: FROST survivability when files on $\sqsubseteq$ grow in size.

To evaluate the performance of our proposed IH techniques when cover files are modified, we measure the probability of survival of cover files against the threshold parameter t. Now, let the set of files used as cover files for hiding the secret shares be denoted as follows:

$$F_{cover} = \bigcup_{y=1}^{p} f_y \tag{6}$$

Once the secret shares are hidden in the slack space of cover files, let the set of files whose contents grow in size be denoted as follows:

$$F_{mod} = \bigcup_{z=1}^{q} f_z \tag{7}$$

We randomly select files on target disk $\mathcal{V}$ and modify them by growing them by a few bytes. Then, we measure the percentage of the n shares that were lost due to growth in cover file size and if our mechanisms survive with at least t valid recoverable shares.

For empirical analysis, we vary the percentage of files on $\mathcal{V}$ that are modified. Since the files are selected randomly, we believe the representation of files from both F_{nor} and F_{cov} are proportional to their ratio. The survivability of the original secret file is measured based on the percentage of the files on the volumes that are manipulated that also happen to be the cover files, which we see in figure 1.

This can then be used to fine-tune the threshold in accordance with the percentage of files grown. We measure the intersection of the set of manipulated files that are also cover files. As long as the resulting set of elements has a cardinality less than threshold parameter t, the secret message can be successfully recovered.

4.4 Disk Block Corruption & Recoverability

We evaluate the robustness of FROST in the face of block corruption on the target storage volume which could be erratic and random or due to wear. Additionally, blocks can go bad either in a specific locality or randomly across the disk. We evaluate both scenarios through our prototype. To this aim, we first hide the secret file on the target disk and generate the map file. Subsequently, we evaluate the above two scenarios of block corruption as follows and corrupt random blocks of the disk by marking blocks as "corrupt". Then, we execute our "recovery" algorithm using the map file generated during hiding.

If the algorithm encounters a secret share within a "corrupt" block, that share is deemed no longer usable and is discarded. With this, we determine the percentage of the n shares that were lost due to block corruption if our mechanisms survive with at least t valid recoverable shares. In figure 2, we can see that the percentage of surviving shares is the complement of the percentage of corruption despite varying values of n and number of blocks.

4.5 Disk De-fragmentation & Recoverability

Similar to internal fragmentation, file systems can also suffer from *external fragmentation*. *External Fragmentation* is the creation of holes of one or more contiguous blocks on the file system due to the creation, modification, or deletion of files. The contiguous blocks of storage space are not large enough to fit a complete file, and due to efficiency reasons, the OS will look for contiguous blocks when storing files.

The effect of external fragmentation is more severe when fragmented files are deleted, because this leaves numerous small regions of free spaces (or holes). This is eventually countered by the system by coalescing all free spaces using the process of *de-fragmentation*.

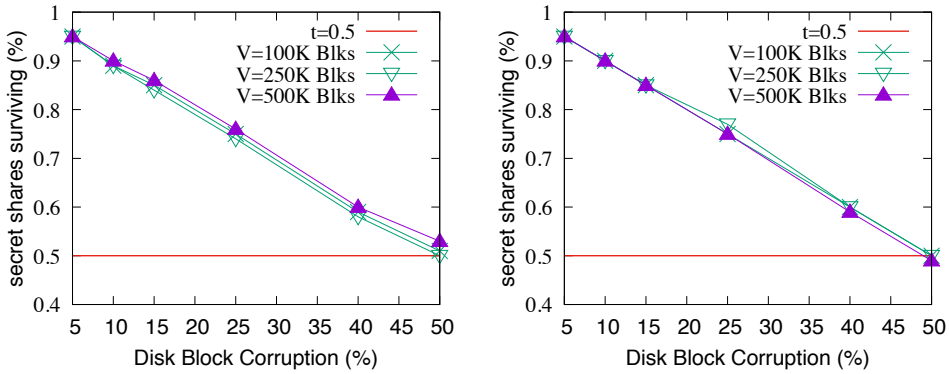
We have evaluated the performance of our proposed IH mechanisms and their robustness in the face of a de-fragmentation on the target disk. To this aim, we take drives from real-world that exhibit external fragmentation and hide a secret file using our IH mechanisms within the slack space of allocated files. Then, we run the system de-fragmentation routine followed by our "recovery" algorithm, with the corresponding map file, to recover the original secret file in its entirety.

The results from these experiments are presented in the figure 3. For greater percentages of blocks in use, the percentage of secret shares that survive drops drastically. However, despite the relatively large drop, the percentage of shares that survive is still comfortably above the threshold, and a sufficient amount of valid shares are present to reconstruct the secret message.

5 ANALYSIS AND OBSERVATIONS

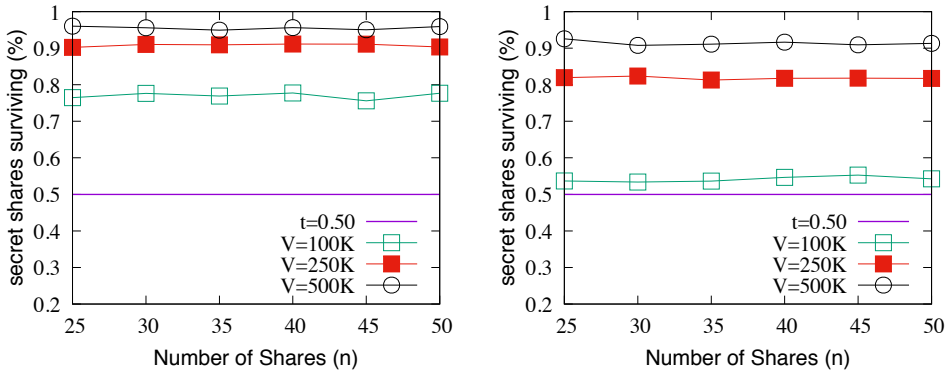
Since the secret file f_{sec} is processed into n secret shares, any t of which can be pooled to recover the secret file f_{sec} in its entirety, it is critical to extract and process the least number of valid shares, the threshold number t. However, as discussed in Section 2, cover files can get modified or deleted and storage volume can get de-fragmented among other things which will result in potential loss of the secret message shares. Therefore, it is imperative that there be some mechanism to detect corrupt shares before proceeding to reconstruct the original secret message.

FROST computes a hash for each secret share generated, appends the hash to the end of the corresponding share, and then



(a) $n = 25; t \leq \frac{n}{2}$ (b) $n = 50; t \leq \frac{n}{2}$

Figure 2: FROST survivability when blocks on $\mathcal{V}$ get corrupt after hiding the shares.



(a) $F = 10K$ files (b) $F = 20K$ files

Figure 3: FROST survivability when $\mathcal{V}$ is de-fragmented.

writes it in the slack space of a unique cover file alongside the secret share itself. While inclusion of the in-place integrity verification mechanism enables FROST to detect corrupt shares instantly, it also adds to the computation and storage overhead. Nevertheless, without such an integrity verification mechanism, corrupt shares would not be detected until much further in the process, and reconstruction of the original secret message would fail.

The inclusion of integrity verification mechanism saves significant amount of time that would otherwise be wasted for computations associated with processing corrupt shares in an effort to reconstruct the original secret message. Furthermore, lack of an integrity verification mechanism would make it extremely hard, if not impossible, to isolate the shares(s) among the t shares that are corrupt. FROST's additional storage overhead for achieving the integrity objective and minimizing wasted computation involving corrupt shares is, in our opinion, a necessary trade-off.

Finally, analysis of cover files associated with corrupt shares can be leveraged to further optimize the performance by avoiding file types that are highly likely to get modified. FROST satisfies the

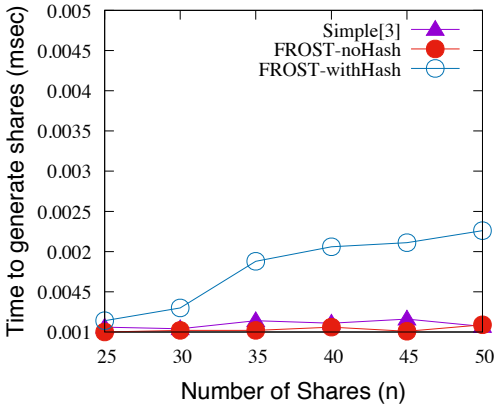


Figure 4: Time to generate the secret shares.

three core security requirements, namely *Confidentiality*, *Integrity*, and *Availability*. The small increase in hash computation time is

computed and results are presented in figure 4 comparing FROST with and without hash and the simple technique presented in [9].

(1) **Confidentiality.** FROST provides confidentiality of the secret message through $(t-1)$-recovery resistance. Consequently, knowledge of any $t' < t$ shares of the secret message is not sufficient to reconstruct the secret message in its entirety. This security property is enforced across the board and thus applicable to legitimate colluding members and unauthorized third party users.

Additionally, the threshold parameter t can be fine-tuned to be robust to different degrees of collusion. When the colluding users' group leader has specifics of the capability of a rogue insider or an outside party, the system configuration, and the system limitations, the threshold parameter t can be tuned to appropriate levels preventing collusion among users who have turned malicious or have been taken over by external malicious actors.

(2) **Integrity.** FROST provides robust security against message integrity attacks. Hash values using a function such as MD5 are computed for each of the n secret share. The hash values are then appended to their corresponding secret share and finally written to the slack space of files on $\mathcal{V}$. When a colluding partner retrieves the shares to construct the secret message, each secret share is verified with its hash, only when the integrity check succeeds, the corresponding secret share is counted and included towards the required t valid shares.

Optionally, a hash value of the entire secret message can be computed and stored in map file, which can be used to verify the message upon reconstruction. However, this is optional and redundant since individual shares are verified prior to reconstructing the secret message.

(3) **Availability.** Availability is perhaps the most complex security property to achieve. A popular solution to enforcing availability of information or any other resource is through redundancy. In our proposed FROST, user's knowledge of any $t' \geq t$ is sufficient and required to recover the original secret message in its entirety. This security property provides reliability of the technique by assuring recoverability of the secret message in the face of accidental or intentional data loss.

Such loss could occur due to numerous reasons such as *cover file modification*, *disk block corruption*, *cover file deletion and block' reallocation*, *erasure of free space*, etc. To this aim, it is imperative that the user optimize the number of shares n, and the threshold parameter t relative to n.

6 RELATED WORK

In classical data hiding, data is hidden in places that tools don't typically look. Metasploit's Slacker[2] hides data in the slack space in both FAT and NTFS systems. *FragFS* [17] hides data throughout an NTFS system's MFT. *RuneFS* [4] hides data in bad blocks, since most tools will simply ignore bad blocks. *WaffenFS* [3] will hide data in the ext3 journal. *KY FS* [4] hides data in directory entries. *Data Mule FS* [4] hides data in the reserved i-node space.

[2]https://www.bishopfox.com/resources/tools/other-free-tools/mafia/

Data can be hidden in unallocated pages of Microsoft office files, so it may appear to be a regular word document. Data can also be hidden outside of the file system in the device configuration overlay. This way, the data can be extracted by certain tools, but it will not be recognized by the file system, therefore leaving no trace of it. Users can limit the data that is left behind by using live CDs, bootable USBs, and virtual machines.

In [6], authors have presented the following as the primary goals of Anti-forensics – *Avoiding detection*, *Disrupting information collection*, *Increasing the examiner's time*, and *Casting doubt on a forensic report or testimony*. Two additional goals identified further include – *Subverting the forensics tool* and *Leaving no evidence that an anti-forensic tool/process has been run*.

Marcus Rogers has identified the following as the broad areas of Anti-Forensics in [13] – *Data Hiding*, *Artefact Wiping*, *Trail Obfuscation*, and *Attacks against the CF Process and tools*. Our proposed FROST falls under "Data Hiding" and "Trail Obfuscation" categories presented by Marcus Rogers.

Thompson and Monroe [17] have categorized information hiding into the following three broad categories - 1) *Out-of-Band*, 2) *In-Band*, and 3) *Application Layer* and our proposed FROST falls under the category of "In-Band" according to [17]. Srinivasan and Wu [15] have proposed a novel steganography technique for data hiding using duplicate file names, which exploits a subtle yet serious file system vulnerability. This is the only other work that, unlike contemporary steganographic techniques, uses merely the cover file name for information hiding and not actually modify data in the cover file.

StegFS [8] is one other steganography related work that works similar to our technique in that it does not modify the contents of the cover file. However, it is important that we draw the distinction clearly at this point. *StegFS* is a modified ext2 file system that hides encrypted data in unused blocks of the file system. Additionally, it renders the hidden data to look like a partition in which unused blocks have recently been overwritten with random bytes using some disk wiping tool.

In [16], Srinivasan et al. have presented a technique for hiding information in the slack space of files. Their proposed technique once again suffers from the lack of fault tolerance capabilities. Like all the other techniques, this method is also vulnerable to loss of even a single byte of information hidden in the slack space of a file.

7 CONCLUSION AND FUTURE WORK

We have presented – FROST – a novel asynchronous technique for hiding information in slack space. FROST is the first of its kind "Digital Dead Drop" robust to detection and data loss with tunable fault tolerance. We validate performance and robustness of FROST to detection and data loss through a prototype implementation on a Ubuntu Linux machine. Unlike existing IH techniques, FROST achieves robust security and stealth of hidden information. Most importantly, it is the reliability and fault tolerance capabilities of FROST that clearly differentiate it from other IH tools and techniques. This is a critical requirement given the operating environment is highly dynamic and can potentially overwrite the contents in slack space. To overcome the problem of corruption of secret shares when they are at rest, FROST writes the MD5 hash value

of each share along with the share in the slack space. This enables the user to verify the validity of each share on-the-fly upon extraction such that reconstruction of the original secret message is not attempted until t valid shares are retrieved.

REFERENCES

[1] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM, 1–10.

[2] Ronald Cramer, Matthew Franklin, Berry Schoenmakers, and Moti Yung. 1996. Multi-authority secret-ballot elections with linear work. In *Advances in Cryptology – EUROCRYPT-96*. Springer, 72–83.

[3] Knut Eckstein and Marko Jahnke. Data Hiding in Journaling File Systems.. In *Digital forensic research workshop (DFRWS)*. 1–8.

[4] Grugq. 2005. The Art of Defiling. (2005). https://tinyurl.com/my7a896

[5] Amir Herzberg, Stanis law Jarecki, Hugo Krawczyk, and Moti Yung. 1995. Proactive secret sharing or: How to cope with perpetual leakage. In *Advances in Cryptology – CRYPT0-95*. Springer, 339–352.

[6] Liu and Brown. 2006. Bleeding-Edge Anti-Forensics. *Infosec World Conference & Expo* (2006).

[7] Michael A Marsh and Fred B Schneider. 2004. CODEX: A robust and secure secret distribution system. *Dependable and Secure Computing, IEEE Transactions on* 1, 1 (2004), 34–47.

[8] Andrew D McDonald and Markus G Kuhn. 1999. StegFS: A steganographic file system for Linux. In *International Workshop on Information Hiding*. Springer, 463–477.

[9] Jeffrey Medsger and Avinash Srinivasan. 2012. ERASE- entropy-based sanitization of sensitive data for privacy preservation [Best Paper Award]. In *7th International Conference for Internet Technology and Secured Transactions, ICITST 2012, London, United Kingdom, December 10-12, 2012*. 427–432. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6470844

[10] Jeffrey Medsger, Avinash Srinivasan, and Jie Wu. 2015. Information Theoretic and Statistical Drive Sanitization Models. *Journal of Information Privacy and Security* 11, 2 (2015), 97–117.

[11] Torben Pryds Pedersen. 1991. A threshold cryptosystem without a trusted party. In *Advances in Cryptology – EUROCRYPT-91*. Springer, 522–526.

[12] Tal Rabin and Michael Ben-Or. 1989. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*. ACM, 73–85.

[13] Marcus Rogers. 2005. Anti-forensics – Presentation made at Lockheed Martin. (2005). www.cyberforensics.purdue.edu

[14] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[15] Avinash Srinivasan, Satish Kolli, and Jie Wu. 2013. Steganographic information hiding that exploits a novel file system vulnerability. *International Journal of Security and Networks* 8, 2 (2013), 82–93.

[16] Avinash Srinivasan, Srinath Thirthahalli Nagaraj, and Angelos Stavrou. 2013. HIDEINSIDE – A novel randomized & encrypted antiforensic information hiding. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*. IEEE, 626–631.

[17] Irby Thompson and Mathew Monroe. 2006. FragFS: An Advanced Data Hiding Technique. *DEFCON 2006 Presentation* (2006).

[18] Ron G Van Schyndel, Andrew Z Tirkel, and Charles F Osborne. 1994. A digital watermark. In *Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference*, Vol. 2. IEEE, 86–90.