SURE-FIT – SecURE and Adaptive Framework for Information Hiding with Fault-Tolerance

Avinash Srinivasan and Hunter Dong

Computer and Information Sciences, Temple University,
Philadelphia PA 19122, USA
E-mail: avinash@temple.edu; hunter@temple.edu

Received 30 January 2018; Accepted 30 January 2018;
Publication 16 February 2018

Abstract

Historically, *Information Hiding* has primarily been associated with malicious intentions. However, it also has beneficial applications such digital rights management and passport control. A *"DeadDrop"* is one such method of espionage trade craft used to physically exchange items or information using a secret rendezvous point. Hiding information in digital file slack space is one such technique that has been used extensively in the modern day, which operates under significant constraints. More importantly, none of the existing work offer robust hiding in slack space with fault tolerance that guarantee recovery of the hidden secret. In this paper, we propose SURE-FIT – a novel asynchronous *"Digital Dead Drop"* robust to detection and data loss. Our proposed technique offers fault tolerance as a tunable parameter leveraging the Shamir's classic threshold secret sharing scheme $\langle n, k\rangle$ [21]. Through a working prototype implemented on a 64-bit Ubuntu Linux system, we confirm the performance and robustness of SURE-FIT We implement a simple hash-based message integrity verification into SURE-FIT framework to validate secret shares upon their retrieval, which results in significant performance improvement. SURE-FIT is also verified through secret message survivability under various operating conditions including block corruption and defragmentation. Finally, we present results confirming the performance improvement of SURE-FIT over two state-of-the-art IH techniques.

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

1 Introduction

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 [26]. 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 pre-agreed secret location. This property eliminates the need for parties involved in the covert communication to meet directly, thereby maintaining operational secrecy and security. A live drop method, in sharp contrast to the dead drop method, necessitates the physical meeting of parties involved in the covert communication. 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.

From a computing perspective, it is the art of exchanging digital information during a covert mission and the drop point in this case would constitute a *'Digital Dead Drop'*. This technique involves one individual dropping the digital information to be exchanged at a predetermined location and another individual acquiring that information from that same location at a later time. Unlike the physical dead drop, a digital dead drop can and will leave trace evidence. Unless proper precautions are taken to sanitize the digital footprint left behind during the covert information exchange. Hiding information in *slack space*[1] 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. They not only suffer due to the dynamic nature of the storage volume, especially when the target is a bootable partition, but also from file system actions such

[1]"Slack Space" and "File Slack Space" are used interchangeably in the remainder of this paper and they mean the same.

as *file modification* and *file deletion.* They are further impacted by some of the not so common OS tasks such as *defragmentation.*

In this paper, we propose "SURE-FIT" – a digital dead drop for covert asynchronous information exchange. The proposed digital dead drop is a tunable fault-tolerant IH technique that is robust to detection and data loss for various reasons including block corruption and file modification. SURE-FIT employs a storage volume's (bootable or non-bootable) allocated files' slack space as the information exchange drop point. SURE-FIT operates in asynchronous mode for covert message exchange essentially creating a "*digital dead-drop*".

Finally, SURE-FIT 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 SURE-FIT through a prototype implementation on a Ubuntu Linux machine.

1.1 Information Hiding on Secondary Storage Drives

There are several protected and hidden, yet readily available, areas on a secondary storage drive that can be exploited to hide information. More importantly, these areas are inaccessible by the OS and the typical users. Therefore, they serve as excellent information drop points for asynchronous covert information exchange. Some very popular IH techniques include – *obfuscated file names* [23], *file encryption*, *disk and volume encryption (e.g., TrueCrypt, BitLocker)*, *Encrypting File Systems (EFS)*, and *file slack space* [22]. Of these, slack space is perhaps the most popular for hiding information.

Hiding in slack space is also 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 is constituted of either unused blocks 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 pre-loaded with system utilities to this aim. Since deletion of the file only marks the corresponding block(s) as free, which can and will be used by the OS to store data at some point in the future, there is no certainty that the block(s) 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 contents in the slack space of a file is to erase/wipe the corresponding blocks.

The slack space of a file is the first to get overwritten when the cover 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. Furthermore, all these techniques are impacted by the state of the target storage volume including – block (or cluster[2]) size, total number of allocated files, and available cumulative slack space. Hiding information in file slack space has several limitations, which are summarized in Table 1. Our proposed SURE-FIT is an adaptive IH framework that is sensitive to the state of the target volume. It does have the capability to create cover files on the fly if and when needed.

Table 1 Impact of file system actions on slack space of allocated disk space

File Action	**Impact on Original Slack Space**
Grows in size by $\leq b_m^{slack}$	slack space contents are overwritten either partially or completely
Grows in size by $> b_m^{slack}$	slack space contents are overwritten completely Additional blocks are allocated.
Shrinks in size by $< b_m^{file}$	slack space b_m^{slack} increases in size; original contents are intact
Shrinks in size by $\geq b_m^{file}$	Block b$_m$ is added to "free-block list". Original contents accessible through their physical address shares cannot be accessed once the blocks corresponding to the shares are overwritten or erased/wiped
File Deletion	Original slack space contents accessible unless overwritten
Disk Defrag	Original slack space contents accessible unless overwritten
Bad Sectors & R/W Errors	slack space contents are inaccessible if any of the cover files span across these sectors
Drive Failure	If full physical backup made after secret shares were hidden to restore/replace the failed drive, then secret shares will survive drive replacement Otherwise, shares cannot be recovered since they do not exist on physical drive backup used to restore new drive

[2]Cluster and block are used interchangeably and in the remainder of this paper we will use the term block.

1.2 Summary of Contributions

Our contributions in this paper can be summarized as follows. To the best of our knowledge, SURE-FIT is the first of its kind *digital dead drop* designed extending the idea of the physical world covert communication utilizing the asynchronous "Dead Drop" of espionage trade-craft. It is unique compared to contemporary steganography techniques since it does not alter the payload of the cover file. It is designed with built-in tunable fault tolerance capabilities leveraging strong and provably secure cryptographic primitives. The framework is implemented and analyzed through a Ubuntu Linux prototype implementation. Our proposed framework is robust to information loss resulting from dynamic nature of system events that generate temporary files, cause storage volume corruption, disk de-fragmentation, etc. Recovery of secret shares is based on physical address to the byte-offset of secret shares. This property enables recovery of secret even if the cover file is deleted or the storage volume is de-fragmented. Furthermore, as long as the drive is not sanitized, secret can be extracted using the physical address of shares from the map-file. The framework hides secret files in system areas whose presence is normal and does not trigger suspicion. Therefore it easily evades detection by tools and techniques that rely on payload analysis. SURE-FIT has an open architecture to enable integration of other tools and techniques. One such tool includes a slack space generator plug-in module. Such a module will enable SURE-FIT to generate required amount of slack space on-the-fly if and when the target volume does not have sufficient slack space. Finally, SURE-FIT provides all three core security properties – *confidentiality*, *integrity*, and *availability*.

1.3 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 our proposed fault-tolerant IH technique in Section 3. We present the implementation details of a working prototype of our proposed IH mechanism in Section 4. This section also presents the results from evaluation of our prototype on real-world systems. In Section 5, we provide detailed analysis of our proposed work'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 the proposed fault-tolerant IH technique along with directions for future research in Section 7.

2 Background and Preliminaries

It is a common practice for vendors of secondary storage devices to create protected reserved areas when they format new secondary storage disks before shipping them to the consumers. These reserved areas are referred to as *Host Protected Area (HPA)* and *Device Configuration Overlay (DCO).* An HPA is created for the purpose of storing recovery tools, proprietary software and data. The DCO is created for the purpose of storing meta-data relating to the storage disk itself. Most users are unaware of even the existence of these protected areas and further more not even the OS is allowed to access an HPA or a DCO. Another area that can be readily used for IH is the *volume slack,* which is the unused and inaccessible space from end of file system volume[3] to end of the partition.

While HPAs and DCOs and even encrypted volumes are vendor/user controlled since they can be created to be of any arbitrary size and destroyed, slack space is the byproduct of normal, benign and approved system tasks. Consequently, slack space inevitably exists on all secondary storage volumes. However, the mere presence of slack space on a volume is not an indicator of any foul play or information hiding.

The amount of slack space that gets created every time a file is stored on the target drive is dependent of several factors. More importantly, the amount of slack space on the target volume increases as more files are stored on the target volume. On the contrary, HPAs and DCOs once created remain fixed in size until they are either destroyed or re-sized. Similarly, once the drive is partitioned and formatted, the volume slack remain fixed unless user re-sizes the partition(s).

Volume slack is significantly easier to access when compared to accessing file slack space. More importantly, if volume slack exist on a storage drive, then typically it's minimum size far exceeds the maximum file slack possible. Nonetheless, there may be isolated instances where in this may not be the case.

Tools and techniques for detecting IH have also evolved significantly. Today, there is a wide range of digital forensics suites that can detect HPAs, DCOs, and encrypted volumes readily. Even if the contents of these regions may not be accessible, it is easy to get rid of such contents by simply destroying them by overwriting. In particular, detection of HPAs and DCOs are extremely simple since they are not part of a file system volume.

[3]Throughout this paper "file system volume", "storage volume" and "volume" are used interchangeably and mean the same thing.

Consequently, hiding information in slack space offers better odds of both evading detection and surviving events such as de-fragmentation. Finally, information hiding in slack space is undetectable by file integrity checkers that primarily rely on *file checksum* or *file meta-data.* This is also true with regards to HPA, DCO, and volume slack.

File systems can be categorized based on numerous features, and detailed discussions on this topic is beyond the scope and key objectives of this paper. However, a classification of file systems based on *block suballoca-tion* (Definition 2.1) feature is important and relevant to our research presented in this paper. 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 sub-allocation* include the two very popular and widely used file systems – *FreeBSD UFS2* and *Brtfs.* However, contemporary file systems predominantly used on consumer and business class computers and servers include *NTSF, ext3, ext4,* and *HFS+* that do not support *block sub-allocation.*

Definition 2.1. *Block suballocation* is a feature on some file systems that allows large blocks to be used while making efficient use of *slack space* at the end of large files.

2.1 File Systems Internal Fragmentation

When a file is created or copied on to a secondary drive, the file system generates and stores meta-data for that file on the disk along with the file. However, meta-data is stored in protected system area that can be accessed only by the OS or through physical access to the drive with appropriate tools. On the other hand, the contents (aka payload) of the file is stored by the file system in fixed-size logical units referred to as *Blocks* (Unix/Linux) or *Clusters* (Windows)[4]. The size of the blocks depends on several aspects and is different for each file system volume. However, the information is stored within the *Volume Boot Record* of the file system volume.

For discussions, let ν be the target storage volume, and ν can be bootable or storage-only. Also, ν can be internal or external, and available either locally or remotely, when required. Note that availability is also a critical requirement under the SURE-FIT framework. It is a known fact that files come is all sizes

[4]Cluster and block are used interchangeably throughout this paper.

and formats, but a file system volume ν has fixed sized blocks denoted as b_{ν}^{dize}. Once formatted, the block size is fixed unless the volume ν re-sized. As a result, when variable-sized files are stored on volumes with fixed-sized blocks, some amount of space will inevitable remain unused. This phenomenon where in space gets wasted within a block assigned to a file and is unusable is known as *Internal Fragmentation* (Definition 2.2). Note that the slack space resulting from internal fragmentation always occurs in the last allocated block of the file.

Definition 2.2. *Internal Fragmentation* is the result of assigning one or more fixed sized allocation units (blocks) to a file or folder and some amount of space remains unused inside the last assigned block. Based on Property 2.1 the unused space inside a block that is already assigned to a file is unusable.

Let the set of all files on ν be denoted as:

$$F = \{f_x | x \in [1, 2, \ldots, y]\} \tag{1}$$

Then, let the set F be divided into two subsets F_{nor} and F_{cov}, and denoted as follows:

$$F = F_{nor} \bigcup F_{cov} \tag{2}$$

F_{nor} is the set of files that are not used as cover files as shown in Equation 3:

$$F_{nor} = \{f_p | p \in [1, 2, \ldots, q]\} \tag{3}$$

F_{cov} is the set of all files that are used as cover files (Definition 2.3) as shown in Equation 4:

$$F_{cov} = \{f_s | s \in [1, 2, \ldots, t]\} \tag{4}$$

Once the secret shares are hidden in the slack space of cover files, let the set of files that are modified, specifically increase in size be denoted as follows:

$$F_{modified} = \{f_u | u \in [1, 2, \ldots, v]\} \tag{5}$$

For discussion, let us consider a file f_x which is assigned blocks $b_1, b_2, \ldots, b_m$. Let b_m be the last allocated block of f_x. Now, without any information about the target volume, the average-case slack space for any given will be half (50%) the size of a block (Equation 6). The cumulative slack space for the entire volume can be computed as shown in Equation 7. Therefore, the bigger the block size of a volume, the larger the slack space resulting from internal fragmentation. This is also captured in Figure 1 for various block sizes ranging from 2–128 sectors per block.

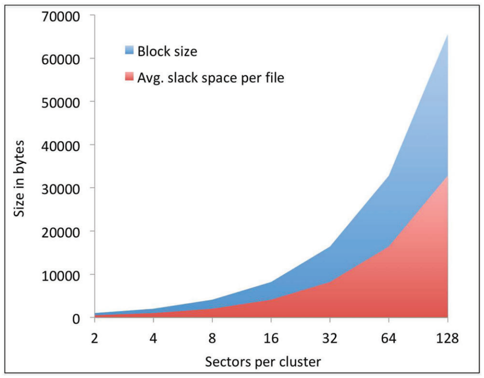


Figure 1 Hiding a secret share along its *hash*.

$$f_x^{ss} = \frac{1}{2} \times b_\nu^{size} \tag{6}$$

$$V_x^{ss} = |F| \times \frac{1}{2} \times b_\nu^{size} \tag{7}$$

Definition 2.3. A cover file is a innocuous looking file on the target volume ν whose file slack space is used for hiding a secret share.

Property 2.1. There in only one possible pair $\langle b_i, f_x \rangle$ for a given (i, x). That is, if there are two pairs $\langle b_i, f_x \rangle$ and $\langle b_i, f_y \rangle$, then either $(x == y)$ or $i \neq j$ has to be true. Therefore, all pairs $\langle b_i, f_x \rangle$ exhibit a *one-to-one* relationship.

Every file f_x on ν, irrespective of the underlying file system, has two associated sizes: $i)$ $logical\ size$ (f_x^{ls}) (Definition 2.4); and $ii)$ $physical\ size$ (f_x^{ps}) (Definition 2.5). This is another way of looking at the phenomenon of internal fragmentation. The simplest way to determine the slack space of file f_x is to compute the difference between its *logical* and *physical* size, as shown in Equation 8.

$$f_x^{ss} = f_x^{ls} - f_x^{ps} \tag{8}$$

Note that while the *logical size* of a file is almost always greater than its *physical size,* the two sizes can be equal (Equation 9).

$$f_x^{ls} \geq f_x^{ps} \tag{9}$$

Definition 2.4. The logical size of a file is the sum total of all the blocks assigned to that file. Logical size of a file is the product of the block size on that volume and the number of blocks assigned to the file.

Definition 2.5. Physical size of a file is the size of the actual contents (aka payload) of the file.

Furthermore, each file is allocated one or more blocks where as any given block can be allocated to one and only one file. Therefore, each new file created or copied to ν starts at the beginning of a new block (also due to Property 2.1). This simplifies their organization and makes file tracking easier as they grow. However, the mapping of file-to-block denoted by the pair $\langle f_x, b_y \rangle_{(\forall_y \in \{1,2,\ldots,\})}$ exhibits a *one-to-many* relationship. Note that a pair $\langle f_x, b_y \rangle$ can indeed exhibit a *one-to-one* mapping if size of f_x is less than the size of a block.

Finally, file slack space has two parts: *i) RAM slack*; and *ii) Drive slack.* RAM slack is the unused space from the end of the physical file to the end of the sector in which the physical file ends. Drive slack is the unused sector(s), full or partial, within the last block assigned to the file.

2.2 Impact of System Events on IH in Slack Space

While altering some of the file meta-data such as its name and copying it to a different location within the volume will not impact the information hidden within the file slack space. However, actions that alter the payload – both direct such a content modification, and indirect such a changing the extension that forces payload alteration, will have varying degrees of impact on the information hidden in the file slack space. In this paper, we have focused primarily on file system actions that directly impact a file's payload. Two key actions that will alter the file payload are – *adding content to the file* and *deleting content from the file*.

Specifically, deleting content from a cover file is not as much of a serious threat as adding content to the cover file. 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 is intact. Therefore, as long as the freed blocks of a cover file are not overwritten, the corresponding secret share can be retrieved if the physical address of the start of the slack space is known.

On the other hand, when content is added to a file, is the file's the slack space this is first utilized. If the contents being added to the file exceeds the file's slack space, only then are additional blocks assigned. Hence, when a cover file grows in size, the contents in its slack space are the first to be overwritten. Hidden information can also be lost due to numerous other causes. However, a detailed discussion on this topic is beyond the scope of this paper. Interested readers can refer to [15, 19, 20, 18, 5] for additional details.

3 Information Hiding with Tunable Fault Tolerance – SURE-FIT

3.1 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* [2, 3], *proactive secret sharing* [7], *secure key management* [13, 9], and *Byzantine agreement among participants* [16]. 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 in [21].

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 [21]. The broad idea of (t, n)-threshold secret sharing is presented in Figure 5(a).

3.2 IH in Slack Space – Need for Fault Tolerance

Assume a secret file is split into *n* parts and hidden by distributing them across the slack space of *n* different cover files. This secret, without the concept of threshold sharing, will need all *n* parts in order to reconstruct the original secret file. Even if a single cover file grows in size overwriting the contents in the slack space, then the that "part" of the secret file in the slack space will be overwritten, either partially or completely. Whatever the case may be, with just $(n - 1)$ valid parts, the original secret file cannot be reconstructed. This idea is captured intuitively in Figure 2. This problem can be overcome by leveraging the threshold secret sharing scheme discussed in Subection 3.1. Our proposed SURE-FIT framework is one such solution, which is an adaptive and secure IH framework. One of the key advantages of this framework is its fault tolerance utilizing threshold-based information hiding.

3.3 SURE-FIT Framework Overview

SURE-FIT is a threshold-based IH mechanism that is robust to modifications to file contents, read-write errors, bad sectors, disk de-fragmentation among others, as long as *t* out of *n* shares have survived. Let us assume that a secret file

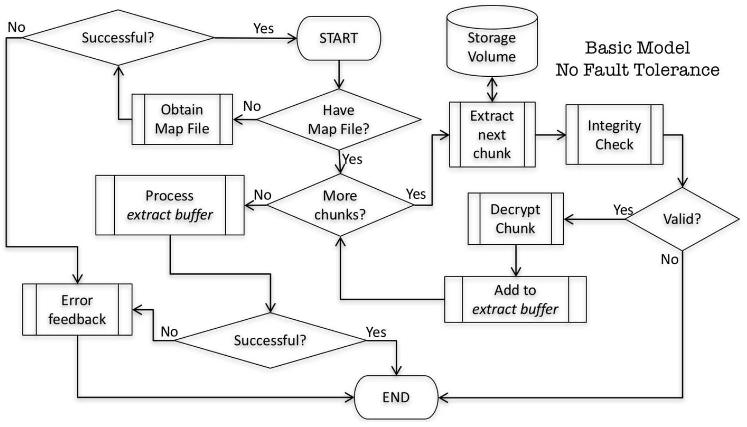


Figure 2 Flow diagram of secret file recovery with the basic model [24].

f^i_{sec} is processed into *n* secret shares. Let each secret share of f^i_{sec} be denoted as s^i_j and let S^i_j the set of all secret shares of a message f^i_{sec} (Equations 10, 11). Now, we have the following:

$$S^j_i = \left\{s^i_0, s^i_1, \ldots, s^i_{n-1}\right\} \tag{10}$$

$$S^j = \left\{s^i_0 | j = \{0, 1, \ldots, n-1\}\right\} \tag{11}$$

Let all element of the set S^i_j be hidden in the slack space of a unique file on a target volume. The high-level process flow diagram for the framework is presented in Figure 3. As can be seen, SURE-FIT accepts as input a secret file (f_{sec}) from the user. It then generates cryptographically secure secret shares from the original secret file using the polynomial method of computation. It then passes the shares to a hash computation engine. Finally, each secret share and its corresponding hash are concatenated resulting in a secret "chunk" of the form $[share \mid\mid hash]$. Each chunk is then written to the slack space of a unique cover file, and this idea is captured in Figure 5(a). Simultaneously, SURE-FIT also generates a map file for f_{sec} with an entry for each hidden chunk $[share \mid\mid hash]$, resulting in *n* entries. Each map-file entry is a pair of the form $< s_{id}, pa^{bo} >$, where pa^{bo} is the physical address of the chunk's start byte-offset.

Once hidden, the secret file f^i_{sec} can be recovered in its original form only when a minimum of *t* valid shares are input to the recovery function, where $t \leq n$. Note that any subset of *t* valid shares will suffice the need. SURE-FIT provides in-place integrity verification of extracted shares to ensure secret file recovery is attempted only when *t* valid shares are available. Therefore, each

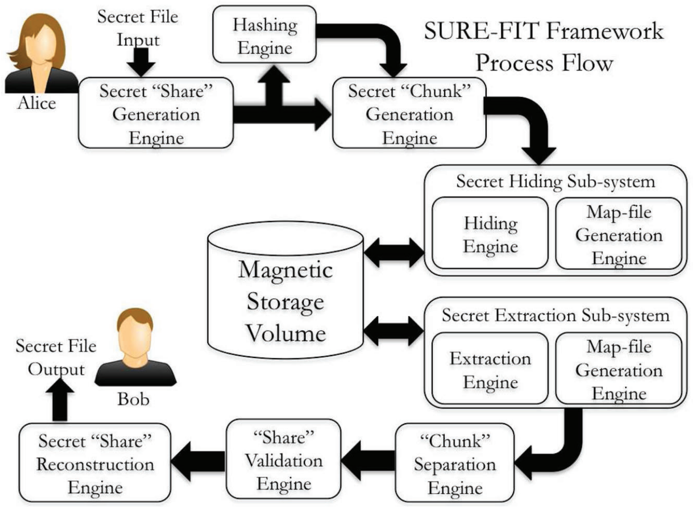


Figure 3 Hiding a secret share along its *hash*.

chunk $[share \mid\mid hash]$ upon retrieval is subject to integrity verification. Only shares that are valid are copied to a temporary recovery buffer. This process of chunk validation continues till *t* valid shares are available in the temporary recovery buffer. The *t* secret shares are then collectively processed to recover the original secret file. The whole process of IH with fault tolerance is captured in Figure 4.

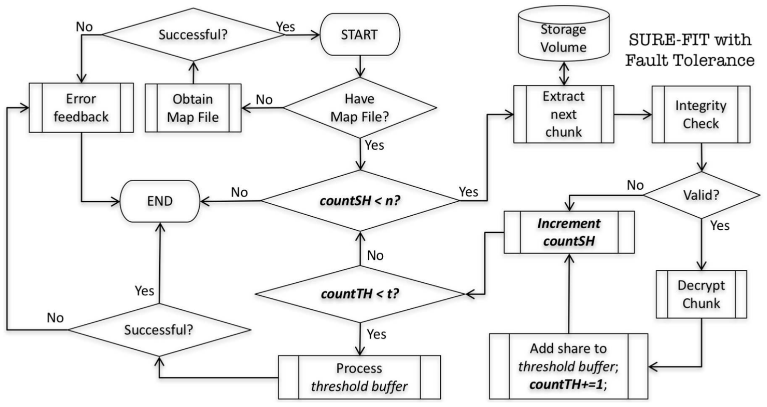


Figure 4 Flow diagram of secret file recovery with the proposed SURE-FIT Model. countSH *rightarrow* track *n*; countTH *rightarrow* track *t*.

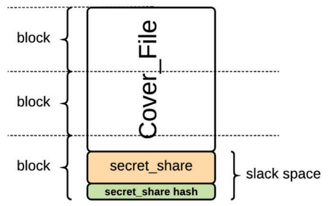
3.4 SURE-FIT Framework Working

Algorithm 1: Generate List of Available Cover files.

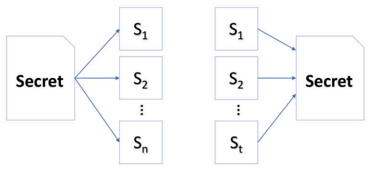
```
Input: $L_{f_{cov}}^{all}$ and $L_{f_{cov}}^{used}$
Output: $L_{f_{cov}}^{free}$
/* Add unused cover files to the temp list                */
$L_{tmp} \leftarrow L_{f_{cov}}^{all} \backslash L_{f_{cov}}^{used}$
if $L_{tmp} == \emptyset$
  then
      return $\emptyset$
end
else
      while $L_{tmp} \neq \emptyset$ do
        /* Add temp list elements to list of free
            cover files                                    */
        $L_{f_{cov}}^{free} \leftarrow L_{tmp} \bigcup L_{tmp}.getNextFile()$
      end
      return $L_{f_{cov}}^{free}$
end
```

One of the most critical aspects of SURE-FIT is that $4t$ is a tunable parameter. Therefore, by its value in the range $t \in \{2, 3, \ldots, (n-1)\}$, it is possible to empirically establish an optimal value for t specific to the IH requirements and constraints of the target volume. The values of $t = 1$ and $t = n$ present trivial boundary conditions, each of which has a security weakness.

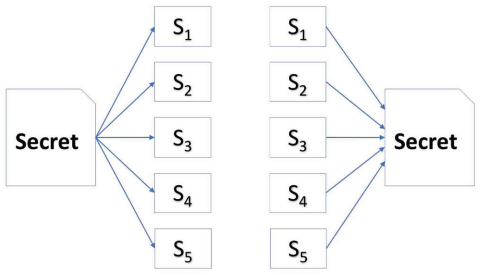
A value of $t = 1$ implies that each secret share is the complete secret message. This scenario is vulnerable to *single point failure*, since the adversary need only to obtain a single share to extract the entire secret file in its original form. On the other hand, a value of $t = n$ is very secure 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, block corruption, or defragmentation will make it impossible for the original secret message to be recovered. This scenario is presented in its generic for in Figure 5(b) and a specific case of $t = n = 5$ is presented in Figure 5(c).



(a) Hiding Chunk in Slack Space



(b) General Case (n, t)-threshold IH



(c) $n = t$- No fault tolerance

Figure 5 SURE-FIT (n, t)-threshold based IH.

Algorithm 2: Hide Secret Shares & Generate Map File.

```
Input: $|S^i| = n$, and $L^{free}_{f_{cov}}$
Output: $f^i_{map}$
/* Number of shares hidden initialized to 0 */
$S^i_{hid} \leftarrow 0$
/* Number of shares to be hidden initialized to $n$ */
$S^i_{rem} \leftarrow \{S^i\}$
$f^i_{sec} = S^i_j = \sum_{j=0}^{n-1} S^i_j$
$f^{rem}_{sec} \leftarrow f^{all}_{sec} - f^{hid}_{sec}$
$init.File(f_{map}, null)$
$init.buf_{block}(null)$
While $f^{rem}_{sec} \neq 0$ do
    if ${}^{ss}f^x_c < f^{rem}_{sec}$ then
        $buf_{block} \leftarrow [f^{rem}_{sec} - {}^{ss}\ f^{cov}_x]$
        $f^{rem}_{sec} \leftarrow [f^{rem}_{sec} - {}^{ss}\ f^{cov}_x]$
    end
    else
```

```
            $buf_{block} \leftarrow f_{sec}^{rem}$
            $^{ss}f_{x}^{cov} \leftarrow buf_{block}$
            $buf_{block} \leftarrow f_{sec}^{rem}$
            **if** $f_{sec}^{rem} \neq 0$ **then**
                $f_{map} \leftarrow gen.MapFile()$
            **end**
            **else**
                $f_{map}^{enc} \leftarrow enc.File(f_{map})$
            **end**
        **end**
    **end**
```

To achieve a balance between *fault tolerance* and *single-point failure*, we will empirically determine an optimal range for threshold *t* under our proposed SURE-FIT using the referenced real-world systems [12] as the benchmark. In the following paragraphs, we provide the step-by-step working of the proposed SURE-FIT:

- **Step-1 Process secret file into *n* secret shares.**
  The contents of the secret message to be hidden are pre-processed. During pre-processing, the secret message is mapped 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 $\{a_1, a_2, \ldots, a_{(t-1)}\}$. Additionally, the user needs to set $a_0 = f_{sec}$. The secret file to be hidden using the threshold scheme is interpreted as a binary string as shown in Equation 12, where n_p is a prime number such that $n_p \geq n$, and $d > 0$ denotes the bit-length of the secret share. In Equation 13, each of the $(n_p - 1)$ shares is interpreted as d-bit strings.

$$s \in \{0,1\}^{d(n_p-1)} \tag{12}$$

$$s = \{s_1, s_2, \ldots, s_{(n_p-1)}\} = \sum_{i-1}^{n_p-1} s_i \in \{0,1\}^d \tag{13}$$

  We have s_0, which is a zero string, as shown in Equation 14, from which we have Equation 15.

$$s_0 = 0^d \tag{14}$$

$$s_0 \oplus a = a \tag{15}$$

At this point, the secret shares can be computed as presented in Equation 16.

$$s_i = \big\{a(x_i)|i = \{0, 1, \dots, n-1\}\big\} \tag{16}$$

Additionally, an MD5 hash value is generated for each share s_i and appended to the share. The process of generating secret shares, hash value for each share, and appending each share and its corresponding hash value is performed offline by the message originator.

- **Step-2 Hide each secret share in a distinct cover file 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 ν. When hiding the secret shares, the user creates a map file f_{map} with the physical address of the byte offset for each share. The f_{map} also includes a hash of the original secret message $\mathcal{H}[f_{sec}]$. 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} offline 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 $S' \subseteq S$.

Table 2 provides a matrix summarizing the different cryprographic primitives that are available for our IH purposes. The matrix specifies the security objective(s) each identified cryptographic primitive satisfies. The matrix also notes the type of cryptographic key used by each identified primitive. In our prototype implementation, we have used the "hash" primitive, as discussed

Table 2 Summary of cryptographic primitives and the security objective(s) each satisfies

	Cryptographic Primitive		
Security Objective	Hash	MAC	Digital Signatures
Message Integrity	$\surd$	$\surd$	$\surd$
User Authentication	X	$\surd$	$\surd$
Non-Repudiation	X	X	$\surd$
Cryptographic Key	None	Symmetric	Asymmetric

above, for simplicity. However, the proposed IH framework can be easily extended to use other primitives based on the specific security objectives that need to be satisfied.

4 SURE-FIT Validation

4.1 Prototype Environment and Evaluation Parameters

With threshold secret sharing mechanisms, there are three key aspects: *i) total number of shares n, ii) threshold value t,* and *iii) survivability robustness* $(n - t)$. In addition to *n* and *t,* other parameters that impact the performance of the framework include ν_{size}, f_{sec}^{size}, and $|F|$. All these parameters were tweaked and fine tuned, one at a time, to optimize the performance of SURE-FIT on our prototype implementation on a Ubuntu Linux machine. Fault tolerance capability of SURE-FIT was measured in terms of secret shares' survivability, which was in-turn measured through secret message recover-ability. In our emperical evaluations, we construct the secret message shares using the threshold secret sharing algorithm presented in [21]. Below are key parameters we consider in our prototype implementation of SURE-FIT.

- $n = \{25, 30, 35, 40, 45, 50\}$
- $t \leq \lceil \frac{n}{2} \rceil$
- ν_{size}: 100K, 250K, and 500K blocks
- (b_{ν}^{size}): 2048 bytes

Below, we present two scenarios discussing the impact of the three parameters on the robustness and fault tolerance capabilities of SURE-FIT.

- **Scenario-1:** Lower threshold value with higher $(n - t)$ augments resilience of SURE-FIT 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* low, irrespective of *n*, increases the risk of guessing or brute-force attacks.

- **Scenario-2:** Keeping both n and t high increases the space and time complexity of recovering the secret file in its entirety. However, such a system will have very robust fault-tolerance and can recover the secret file even if a significant number of secret get corrupt over time.

During message reconstruction, we divide the set of *n* extracted of secret chunks into two subsets: i) $n_{survive}$ – shares that survive and are intact; and ii) $n_{corrupt}$ – shares that get corrupted and are now unusable. If all chunks are extracted validated for integrity, then we have:

$$n = n_{survive} \bigcup n_{corrupt} \tag{17}$$

4.2 Cover File Modifications & Recoverability

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*. To accomplish this, we randomly select files on target disk ν and modify them by growing them by a few bytes. Then, we measure the percentage of the *n* shares were lost due to growth in cover file size and if our mechanisms survive with at least *t* valid recoverable shares. During our evaluation, we did test adding new files to ν after deleting existing files. However, none of the newly added files were written to those blocks that were freed previously when files were deleted. For empirical analysis, we vary the percentage of files on ν 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. We measure the intersection of the set of modified files and cover files. As long as the resulting set intersection has a cardinality less than threshold parameter *t*, the secret message can be successfully recovered. From the results presented in Figure 6, we see that when as few as 5% of files from the set *F* grow in size, varying *n* has very little impact on survivability under the SURE-FIT framework. Approximately 95% of the chunks survive. Even when 25% of files from *F* grow in size, on average 75% of shares survive. This can then be used to fine-tune the threshold in accordance with the percentage of files grown.

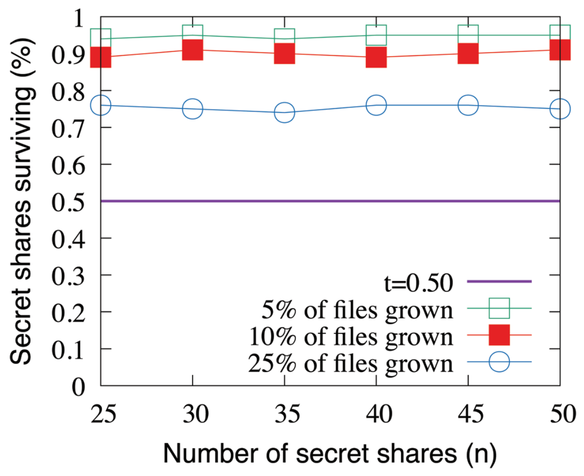
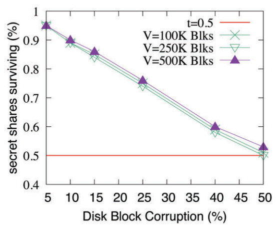


Figure 6 SURE-FIT survivability when cover files grow.

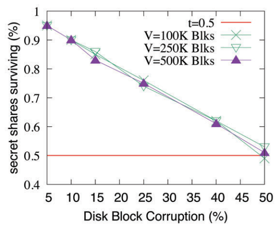
4.3 Disk Block Corruption & Recoverability

We evaluate the robustness of SURE-FIT 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 what percentage of the *n* shares were lost due to block corruption if our mechanisms survive with at least *t* valid recoverable shares. In our empirical studies, we have varied the following parameters: ν_{size} was set to 100, 250, and 500 thousand blocks; *n* was varied from 25 to 50 in steps of 5; disk corruption rate was varied in 5% increments from 5 to 50%, and *t* upto $\frac{n}{2}$. From the results presented in Figure 7, we see that in almost all scenarios, secret chunks greater than *t* survive the corruption. For those scenarios in which the surviving shares are close to *t*, it was determined that at $t = 45\%$, SURE-FIT will have enough tolerance to survive.

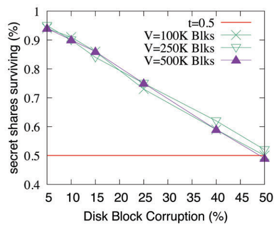
We take this a step further to determine the survivability of our mechanisms by gradually increasing the percentage of corruption and observing the number



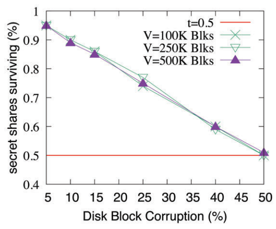
(a) $n = 25; t \leq \frac{n}{2}$



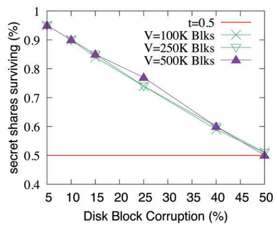
(b) $n = 30; t \leq \frac{n}{2}$



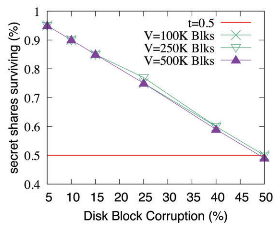
(c) $n = 35; t \leq \frac{n}{2}$



(d) $n = 40; t \leq \frac{n}{2}$



(e) $n = 45; t \leq \frac{n}{2}$



(f) $n = 50; t \leq \frac{n}{2}$

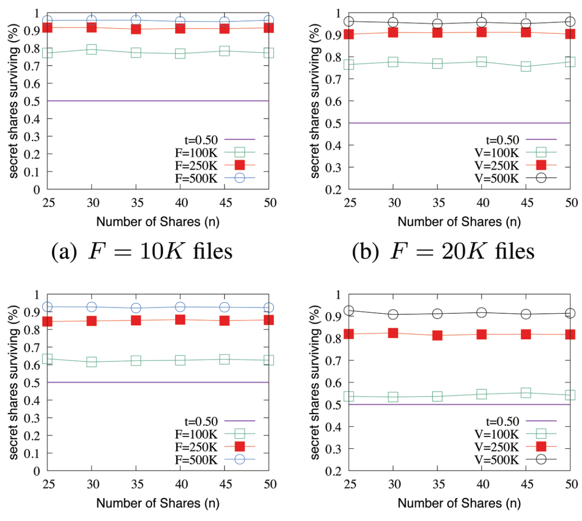
Figure 7 SURE-FIT survivability when blocks on ν get corrupt after hiding the shares.

of valid secret shares that survive. This has two benefits: i) it helps evaluate the robustness of our mechanisms, and ii) it determines the empirical value for the threshold corruption rate that our mechanisms can withstand.

4.4 Disk Defragmentation & 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 effect of external fragmentation is compounded when fragmented files are deleted as this leaves numerous small regions of free spaces (or holes). Eventually, the system counters this by coalescing all free spaces using the *defragmentation* process. Defragmentation is advantageous and relevant only to file systems on electromechanical disk drives. They are unnecessary and rather counter-productive on solid-state drives with random access technology.

We have evaluated the performance of SURE-FIT and its robustness in the face of a defragmentation on the target disk. To this aim, we take drives from real-world that exhibit external fragmentation and hide a secret file using SURE-FIT framework within the slack space of allocated files. Then, we run the system defragmentation routine followed by our "recovery" algorithm,



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

(c) $F = 30K$ files (d) $F = 50K$ files

Figure 8 SURE-FIT survivability when ν is defragmented.

Table 3 Impact of n, t, and $(n - t)$ on fault tolerance

n	t	$n - t$	Fault-Tolerance
High	High	Low/Medium	Low/Medium
Low	High	Low	Low
High	Low	High	High
Low	Low	Low	Low

with the corresponding map file, to recover the original secret file in its entirety. The results from these experiments are presented in the Figure 8. For greater percentages of a system's blocks that are 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 defragmented 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. SURE-FIT computes a hash for each secret share generated, appends the hash to the end of the corresponding share, and then writes it in the slack space of a unique cover file alongside the secret share itself.

SURE-FIT indeed appears to have key characteristics of a steganographic technique. However, on a closer look, unlike steganography-based IH, SURE-FIT never alters the actual payload of the cover file. Instead it relies on effectively utilizing the unused space within the last allocated block of the cover file, which is otherwise inaccessible to both the user and the OS. While inclusion of the in-place integrity verification mechanism enables SURE-FIT 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.

More importantly, detection of failed attempts to recover the secret file would be of little help without knowing which of the t shares used are corrupt. The inclusion of in-place integrity verification mechanism saves significant amount of time that would otherwise be wasted in processing corrupt shares in an effort to reconstruct the original secret message. Therefore, in our opinion, the small amount of latency incurred by SURE-FIT for achieving integrity and minimizing unwanted computation of corrupt shares is a necessary trade-off. Results comparing SURE-FIT with and without hash and the simple technique presented in [11] are presented in Figure 9. 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. The proposed information hiding technique SURE-FIT satisfies the three core security requirements, namely *Confidentiality*, *Integrity*, and *Availability*. The small increase in hash computation time is computed and results are presented in Figure 9.

5.1 Confidentiality

SURE-FIT 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 hence applicable

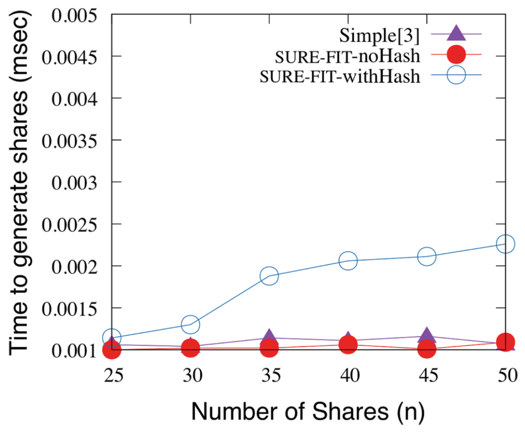


Figure 9 SURE-FIT (with and without hash) vs. Simple [24].

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.

5.2 Integrity

Hashing the secret-shares and hiding them along with the hash value enables the user to verify the integrity of each and every secret share retrieved before counting a share toward the required t shares. This not only helps with verifying the integrity of shares as and when they are retrieved, but also helps minimize computation in successfully reconstructing a secret from valid shares.

SURE-FIT 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 ν. 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. Additionally, a hash value is computed for the secret message as a whole and stored in map file. entire secret file.

5.3 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 SURE-FIT, 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 for 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 [1] hides data in the slack space in both FAT and NTFS systems. *FragFS* [25] hides data throughout an NTFS system's MFT. *RuneFS* [6] hides data in bad blocks, since most tools will simply ignore bad blocks. *WaffenFS* [4] will hide data in the ext3 journal. *KY FS* [6] hides data in directory entries. *Data Mule FS* [6] hides data in the reserved i-node space. Data can be hidden in unallocated pages of Microsoft office files, so it may appear to be a regular word document.

In [8], 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 [17] – *Data Hiding*, *Artifact Wiping*, *Trail Obfuscation*, and *Attacks against the CF Process and tools*. Our proposed SURE-FIT falls under "Data Hiding" and "Trail Obfuscation" categories presented by Marcus Rogers. Thompson and Monroe [25] have categorized information hiding into the following three broad categories – 1) *Out-of-Band*, 2) *In-Band*, and 3) *Application Layer* and our proposed SURE-FIT falls under the category of "In-Band" according to [25]. Srinivasan and Wu [23] 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 [10] 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 [24], 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. The reason for the blocks to appear as overwritten with random data is because both encrypted data and random data have very high entropy values ranging between 7.5–8 bits-per-byte. Entropy is the measure of randomness of data, and higher the entropy value of given data the more random that data is and hence less predictable.

In [24], authors have tested the following two scenarios and measured the resulting entropy of the blocks – 1) entropy of an encrypted data file with ASCII characters – which had an entropy value of 7.89, and 2) entropy of a blank encrypted volume – *TrueCrypt* – which had an entropy of 7.99. Note that the higher the entropy value, the more random the data, where as the lower the value, the more uniform the data. Interested readers may refer to [14] for further details on taxonomy and applications of IH techniques.

7 Conclusion and Future Work

In this paper we have presented SURE-FIT, a novel information hiding framework. The proposed framework enables fault tolerant information hiding in slack space of files on secondary storage drives. Our proposed SURE-FIT is an asynchronous and first of its kind "Digital Dead Drop" robust to detection and data loss with tunable fault tolerance. SURE-FIT leverages fundamental cryptographic primitives that are provably secure, a variation of Shamir's threshold secret sharing scheme [21], to achieve its robust fault tolerance. This augmented technique provides reliability when saving secret data in the slack space of existing allocated files that serve as the cover file. Unlike existing IH techniques, SURE-FIT achieves robust security and stealth of hidden information. Most importantly, it is the reliability and fault tolerance capabilities of SURE-FIT that clearly differentiate it from other state-of-the-art IH tools and techniques. This is a critical requirement given that the operating environment is highly dynamic and can potentially overwrite the contents in slack space. The framework computes the hash of each secret share and saves it along with the corresponding share in the slack space. During recovery process, SURE-FIT makes use of in-place integrity verification mechanism to validate secret chunks immediately upon retrieval. Any chunk that is corrupt while at rest will be discarded. This ensures that reconstruction of the original secret message is not attempted until t valid shares are retrieved into the temporary recovery buffer. Finally, through a prototype implementation on a Ubuntu system, we have validated the robust fault tolerance and secret survivability under the SURE-FIT framework.

References

[1] Metasploit slacker.

[2] Ben-Or, M., Goldwasser, S., and Wigderson, A. (1988). Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, 1–10. ACM.

[3] Cramer, R., Franklin, M., Schoenmakers, B., and Yung, M. (1996). Multi-authority secret-ballot elections with linear work. In *International Conference on the Theory and Applications of Cryptographic Techniques*, 72–83, Springer, Berlin, Heidelberg.

[4] Eckstein, K., and Jahnke, M. (2005). Data Hiding in Journaling File Systems. In *Digital forensic research workshop (DFRWS)*, 1–8.

[5] Fu, S., and Xu, C. Z. (2007). Exploring event correlation for failure prediction in coalitions of clusters. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing* (p. 41). ACM.

[6] Grugq. (2005). The art of defiling.

[7] Herzberg, A., Jarecki, S., Krawczyk, H., and Yung, M. (1995). Proactive secret sharing or: How to cope with perpetual leakage. In *Annual International Cryptology Conference,* 339–352. Springer, Berlin, Heidelberg.

[8] Liu, V., and Brown, F. (2006). Bleeding-Edge Anti-Forensics. Presentation at *InfoSec World Conference and Expo.*

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

[10] McDonald, A. D., and Kuhn, M. G. (1999). StegFS: A steganographic file system for Linux. In *International Workshop on Information Hiding*, 463–477. Springer, Berlin, Heidelberg.

[11] Medsger, J., and Srinivasan, A. (2012). ERASE-entropy-based sanitization of sensitive data for privacy preservation. In *International Conference Internet Technology and Secured Transactions*, 427–432. IEEE.

[12] Medsger, J., Srinivasan, A., and Wu, J. (2015). Information Theoretic and Statistical Drive Sanitization Models. *J. Info. Privacy and Sec.,* 11, 97–117.

[13] Pedersen, T. P. (1991). A threshold cryptosystem without a trusted party. In *Workshop on the Theory and Application of Cryptographic Techniques*, 522–526. Springer, Berlin, Heidelberg.

[14] Petitcolas, F. A., Anderson, R. J., and Kuhn, M. G. (1999). Information hiding-a survey. In *Proceedings of the IEEE*, 87, 1062–1078.

[15] Pinheiro, E., Weber, W. D., and Barroso, L. A. (2007). Failure Trends in a Large Disk Drive Population. In *FAST* (Vol. 7, No. 1, pp. 17–23).

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

[17] Rogers, M., and Lockheed, M. (2005). Anti-forensics. Lockheed martin. San Diego, California. Available at: http://cyberforensics.purdue.edu/documents/AntiForensics\LockheedMartin09152005.pdf

[18] Schroeder, B., and Gibson, G. (2010). A large-scale study of failures in high-performance computing systems. In *IEEE Transactions on Dependable and Secure Computing*, 7, 337–350.

[19] Schroeder, B., and Gibson, G. A. (2007). Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you? In *FAST*, 7, 1–16.

[20] Schroeder, B., and Gibson, G. A. (2007). Understanding failures in petascale computers. In *Journal of Physics: Conference Series* (Vol. 78, No. 1, p. 012022). IOP Publishing.

[21] Shamir, A. (1979). How to share a secret. *Communications of the ACM*, 22(11), 612–613.

[22] Srinivasan, A., Dong, H., and Stavrou, A. (2017). FROST: Anti-Forensics Digital-Dead-DROp Information Hiding RobuST to Detection & Data Loss with Fault tolerance. In *Proceedings of the 12th International Conference on Availability, Reliability and Security,* 1–82. ACM.

[23] Srinivasan, A., Kolli, S., and Wu, J. (2013). Steganographic information hiding that exploits a novel file system vulnerability. *Int. J. Sec. Net.,* 8, 82–93.

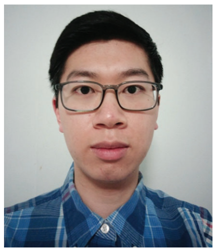
[24] Srinivasan, A., Nazaraj, S. T., and Stavrou, A. (2013). HIDEINSIDE—A novel randomized & encrypted antiforensic information hiding. In *International Conference on Computing, Networking and Communications (ICNC)*, 626–631. IEEE.

[25] Thompson, I., and Monroe, M. (2006). FragFS: An advanced data hiding technique. BlackHat Federal. Available at: http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Thompson/BH-Fed-06-Thompson-up.pdf

[26] Wikipedia. Dead drop. https://en.wikipedia.org/wiki/Dead_drop

Biographies

Avinash Srinivasan is currently an Associate Professor in the CIS department at Temple University (TU) and a Fellow of the National Cybersecurity Institute at Washington D.C. Dr. Srinivasan earned his Bachelor of Engineering in Industrial & Production Engineering (1999) from University of Mysore (India) with Honors. He also has an M.S. in Computer Science from Pace University, (NY 2003 and a Ph.D. in Computer Science from Florida Atlantic University (FL, 2008). Dr. Srinivasan's research interests broadly span the areas of Cybersecurity and Digital Forensics. He has 47-refereed publications in scholarly conferences and journals, including IEEE-INFOCOM, ACM-SAC, IEEE-ICC, IEEE-ICDCS, and IEEE-MALWARE. Since 2008, Dr. Srinivasan has been involved as PI/Co- PI on federally funded research from agencies including DoEd, DoJ, DHS, NSF, and DoD/NAVY. Dr. Srinivasan has over 400 hours of formal training in Cybersecurity and Digital Forensics.



Hunter Dong attended Temple University where he earned his Bachelor of Science in Computer Science and graduated cum laude (2017). He was accepted into and participated in National Science Foundation funded Research Experiences for Undergraduates (REU) program in summer of 2016. Dong also graduated from George Washington High School (2013) where he earned his International Baccalaureate (IB) Diploma and achieved an AP Scholar with Distinction award.