Defeating Encrypted and Deniable File Systems: TrueCrypt v5.1a and the Case of the Tattling OS and Applications

Alexei Czeskis[*] *David J. St. Hilaire*[*] *Karl Koscher*[*] *Steven D. Gribble*[*]

Tadayoshi Kohno[*] *Bruce Schneier*[†]

Abstract

We examine the security requirements for creating a Deniable File System (DFS), and the efficacy with which the TrueCrypt disk-encryption software meets those requirements. We find that the Windows Vista operating system itself, Microsoft Word, and Google Desktop all compromise the deniability of a TrueCrypt DFS. While staged in the context of TrueCrypt, our research highlights several fundamental challenges to the creation and use of any DFS: even when the file system may be deniable in the pure, mathematical sense, we find that the environment surrounding that file system can undermine its deniability, as well as its contents. We hypothesize some extensions of our discoveries to regular (non-deniable) encrypted file systems. Finally, we suggest approaches for overcoming these challenges on modern operating systems like Windows. We analyzed TrueCrypt version 5.1a (latest available version during the writing of the paper); Truecrypt v6 introduces new features, including the ability to create deniable operating systems, which we have not studied.

1 Introduction

A *deniable file system* (DFS) is one where the existence of a portion of the file system can be hidden from view. This is different from an encrypted file system, where files and directories are visible yet unintelligible. In a DFS, the very existence of certain files and directories cannot be ascertained by the attacker.

Deniable File Systems are receiving increasing attention both in popular media and in academia due to the current political environment in which personal laptops are being searched, sometimes even seized, at international border crossings. In response, the EFF (Electronic Frontier Foundation) and CNET have published guides on securely crossing the border with your laptop [3, 5]. Both guides recommend, as one of the most secure options, using a DFS to hide the existence of data. The suggested tool is an open-source disk-encryption software package called TrueCrypt, hence we use it as the focus of our case study.

The TrueCrypt package for Microsoft Windows[1] includes the ability to make a portion of the disk deniable. While TrueCrypt allows for a large number of configurations, a typical configuration might be as follows: Alice creates a regular (non-deniable) encrypted file system on her laptop, protected by some password she knows. Inside that encrypted file system, she has the *option* to also create a deniable file system, protected by a second password. TrueCrypt refers to these inner, deniable file systems as *hidden volumes*. These hidden volumes are claimed to be deniable since, unless she reveals that second password to an adversary, it should be impossible for that adversary to determine whether Alice's regular encrypted file system contains an encrypted hidden volume or not.

We examine both the security requirements for creating a DFS, and how well TrueCrypt's solution meets those requirements. Our results show that deniability, even under a very weak model, is fundamentally challenging. The natural processes of the Windows operating system, as well as applications like Microsoft Word or Google Desktop, can leak significant information outside of the deniable volume. For example, lists of recently changed documents, audit logs of recent file actions, and data saved by application programs can all serve to inhibit deniability. As our results suggest, any DFS will not only have to encrypt and hide data — as file systems like TrueCrypt do — but must also erase any traces of that data left by the operating system through normal operation.

The rest of the paper is organized as follows. Section 2 discusses relevant threat models for deniable file systems. Section 3 describes the TrueCrypt approach in more detail. Section 4 describes our principal information leakage attack vectors against deniable file systems, which we experimentally evaluate in the context of TrueCrypt hidden volumes. We propose potential defensive directions in Section 5, as well as discuss the potential applicability of our results to non-deniable (regular) encrypted file systems. We summarize some related works in Section 6, and then close in Section 7.

Addendum and Document History. We analyzed the most current version of TrueCrypt available at the writing of the paper, version 5.1a. We shared a draft of our paper with the TrueCrypt development team in May 2008. TrueCrypt version 6.0 was released in July 2008. We have not analyzed version 6.0, but observe that TrueCrypt v6.0 does take new steps to improve TrueCrypt's deniability properties (e.g., via the creation of deniable operating systems, which we also recommend in Sec-

[*]Dept. of Computer Science and Engineering, Univ. of Washington.
[†]BT.
[1]`http://www.truecrypt.org/`.

tion 5). We suggest that the breadth of our results for TrueCrypt v5.1a highlight the challenges to creating deniable file systems. Given these potential challenges, we encourage the users not to blindly trust the deniability of such systems. Rather, we encourage further research evaluating the deniability of such systems, as well as research on new yet light-weight methods for improving deniability.

2 Threat Model

Alice is a human-rights worker, and keeps sensitive information on her computer. The data is encrypted, but she is concerned that the secret police will seize her computer and, noticing that part of the disk is encrypted, threaten her — or worse — for the key. She needs to protect her data in such a way that it is *deniable*: there must be *nothing* that would indicate to the secret police that there are hidden files on her computer. If she denies the existence of certain files, and the police later discover the existence of those files on her computer, she could be vulnerable to severe punishment.

Encrypted file systems such as Microsoft's BitLocker will not suffice; encryption does not hide the existence of data, it only makes the data unintelligible without the key. This is precisely the sort of scenario that requires a DFS.

However, it is exactly these restrictive security requirements that make a DFS difficult to implement. Breaking the security of a DFS does not require decrypting the data; it only requires proving that (or in some cases simply providing strong evidence that) the encrypted data exists.

There are several threat models against which a DFS could potentially be secure:

- *One-Time Access*. The attacker has a single snapshot of the disk image. An example would be when the secret police seize Alice's computer.
- *Intermittent Access*. The attacker has several snapshots of the disk image, taken at different times. An example would be border guards who make a copy of Alice's hard drive every time she enters or leaves the country.
- *Regular Access*. The attacker has many snapshots of the disk image, taken in short intervals. An example would be if the secret police break into Alice's apartment every day when she is away, and make a copy of the disk each time.

Clearly there is a point where the adversary is so powerful that even a DFS won't protect Alice. If Alice is working on files in the deniable portion of her computer when the secret police break down her door, she will not be able to deny their existence. If the secret police are able to obtain disk snapshots at close enough intervals, the existence of any deniable files will be obvious, since seemingly random bytes on the hard drive will change. Still, we would like any DFS to provide as much security as possible along the continuum of increasingly severe threat models.

In this paper, we examine attacks against a DFS under the most restrictive threat model: one-time access. Clearly, a successful attack under this threat model implies a successful attack under the less restrictive threat models.

3 TrueCrypt

TrueCrypt is a free disk encryption application that provides on-the-fly-encryption for Microsoft Windows.[2]

TrueCrypt has the ability to create deniable *hidden volumes*. These TrueCrypt hidden volumes are *optionally* – hence deniably – placed inside non-hidden, regular encrypted volumes.

Outer (Non-Hidden, Regular) Encrypted Volumes. A regular (non-hidden) TrueCrypt encrypted volume can be stored (in encrypted form) as a file on top of a regular file system. For example, a TrueCrypt encrypted volume could be stored as the file `C:\TCContainer`. Alternately, a TrueCrypt encrypted volume could occupy a dedicated partition on a disk. In either case, the encrypted volume is referred to as a TrueCrypt *container*. To decrypt a TrueCrypt container, the user must provide the password and keyfiles (if there were any) that were used when creating the volume. We do not describe the details of the TrueCrypt encryption and decryption algorithms since we (largely) treat TrueCrypt as a black box.

Hidden Volumes. TrueCrypt provides a DFS through a feature known as a *hidden volume*. A hidden volume is a TrueCrypt volume stored inside the container of a regular, non-hidden TrueCrypt volume. A hidden volume requires its own password, and — if the hidden volume's password is not supplied (or supplied incorrectly) — the hidden volume's data will appear as random data. Since free space in a regular (outer) TrueCrypt volume is, according to the documentation, filled with random data, this provides plausible deniability to an attacker under the one-time access threat model. Namely, such an attacker, even if given access to the password for the outer encrypted volume, should not be able to determine whether the random data at the end of the outer encrypted volume is really just random data, or whether it corresponds to an encrypted hidden volume. The True-

[2]The current version of TrueCrypt (version 5.1a) also provides Linux and OS X implementations, but neither of these implementations support deniable file systems, and hence we do not consider these implementations in this paper. Prior to a recent code rewrite, older versions of Linux TrueCrypt supported hidden volumes.

Crypt documentation concludes that a person with a hidden volume could therefore convincingly deny the hidden volume's existence.[3]

Interface to Windows. The TrueCrypt application exposes several options to users wanting to mount a regular or hidden volume, including: mount type (whether the volume should be mounted as a fixed file system or a removable file system), writability (read-only or not), and mount point (e.g., `E:`).

TrueCrypt also exposes sufficient information to the Windows operating system to allow Windows to mount and interact with the contents of TrueCrypt volumes (after the relevant passwords or other credentials are entered). We return to specifics of this exposed information later.

Information Leakage from Below. The TrueCrypt documentation already includes some recommendations and caveats for the use of hidden volumes.[4] The principal recommendations are to be cautious about the media on top of which a TrueCrypt hidden volume is stored. Namely, suppose a TrueCrypt volume (and its associated outer volume) are stored on a USB stick. Then *wear leveling* (a physical property of how data is sometimes stored on USB sticks) could reveal information about the existence of the hidden volume. The TrueCrypt documentation similarly advises against storing a TrueCrypt container on top of a journaling file system. We stress that these existing recommendations focus on being cautious about the media *underlying* a TrueCrypt container. Our research focuses on information leakage from above — i.e., information that might leak out about a TrueCrypt hidden volume when the hidden volume is mounted and the operating system and applications are interacting with the hidden files.

The TrueCrypt documentation also rightly observes that an adversary with greater than one-time access (i.e., intermittent or regular access) to the TrueCrypt container could discover the existence of a TrueCrypt hidden volume; for example, by analyzing the differences between multiple snapshots of a TrueCrypt container. Our analyses therefore focus on the weaker case in which the adversary is only given one-time access to the system.

4 Information Leakage from Above

We evaluate three general classes of information leakage vectors against deniable file systems using TrueCrypt's hidden volumes as a case study. These classes all share the following traits: the information is leaked out about the hidden volumes and the files contained therein *after* the hidden volume is mounted, and the information is not securely destroyed after the hidden volume is unmounted.

In more detail, the three classes of information leakage that we consider are:

- *Through the Operating System.* Modern operating systems are complex, with many unexpected and unintuitive behaviors. Operating systems are not designed with the goal of preserving the deniability of deniable file systems. Therefore, the natural execution of an operating system may leak information about the existence of a deniable file system, even if the deniable file system is cryptographically secure.

- *Through a Primary Application.* A deniable file system does not exist in isolation. Rather, people create deniable file systems in order to hide content that they are currently using and plan to use in the future. In order to use those files, these people must run an application (like Microsoft Word, Adobe PhotoShop, and so on). These applications, which are not designed to preserve deniability, may leak information about the existence of those files.

- *Through a Non-Primary Application.* Many computers often run non-primary applications, or applications that (at first glance) appear to be unrelated to the files stored on a hidden volume. For example, many users run applications like Google Desktop and anti-virus software that might interact poorly with the deniability of a hidden volume.

We present experiments showing that the above information leakage classes are not hypothetical. Our results highlight the subtleties and challenges to building deniable file systems.

4.1 Example Leakage Through the OS: Shortcuts

The TrueCrypt documentation observes that information about TrueCrypt is stored in the Windows Registry.[5] This information reveals the fact that a person used TrueCrypt on his or her machine, and the locations at which TrueCrypt volumes were mounted (e.g., `E:`). But this information does not reveal the container's file name, location, size, nor the type of volume that was mounted. Hence, the Windows registry does not appear to directly compromise the deniability of a TrueCrypt hidden volume.

We do not consider the Windows registry further, but instead turn to another artifact of the Windows operating

[3] We have not verified that the free space at the end of a regular TrueCrypt volume is in fact filled with random data. Such an analysis would be orthogonal to our principle focus of studying TrueCrypt's black-box interaction with the operating system and surrounding applications. A failure to fill the free space with random data would, however, allow for a simple attack against the deniability of the hidden volume.

[4] `http://www.truecrypt.org/docs/?s=hidden-volume-precautions`.

[5] `http://www.truecrypt.org/docs/?s=windows-registry`.

system: *shortcuts*.[6] A shortcut, or `.lnk` file, is a link to another file. For example, the shortcut `C:\shortcut.lnk` might link to the file `D:\realfile.doc`; double-clicking on `C:\shortcut.lnk` would cause Microsoft Word to open `D:\realfile.doc`. These links provides a convenient way to access files and folders.

Shortcuts can be created in multiple ways, including by users and by programs. Perhaps surprisingly, Windows *automatically* creates shortcuts to files as they are used, and in Vista these shortcuts are stored in a folder called `Recent Items` that is located in `C:\User\UserName\AppData\Roaming\Microsoft\Windows\`. For example, if a user recently opened the files `E:\File1.doc` and `E:\File2.doc`, shortcuts to these two files would be in the `Recent Items` directory. A wealth of information can be obtained from a `.lnk` file, including the real file's file name, location, attributes, length, access time, creation time, modification time, volume type, and volume serial number of the file system on which the real files are stored [4].

Suppose Alice stores the file `BadStuff.doc` on a hidden volume, edits that document while on a plane, and then closes Word and unmounts the hidden volume before passing through customs. If the customs officer inspects the Alice's disk, he or she will quickly discover that Alice was editing this file — which alone might be significantly compromising information for Alice. Worse, even if the file had an innocuous name like `GoodStuff.doc`, the customs officer can exploit the volume serial number field in the `.lnk` file in the `Recent Items` directory to garner significant evidence that Alice was using a hidden volume.

As additional setup for the latter, recall that in the U.S. it is a crime to lie to a federal law enforcement officer [3]. This means that if Alice chooses to answer a custom agent's question about whether she has a TrueCrypt hidden volume, Alice must answer truthfully. Suppose first that a customs agent asks if Alice uses TrueCrypt. The only answer Alice can supply is "yes," since evidence of TrueCrypt's usage is stored in the Windows registry (and cannot be safely or reliably deleted, according to the TrueCrypt documentation). Next, suppose that the customs agent asks Alice to identify and mount all the TrueCrypt volumes on her machine, as well as all her other mountable file systems and USB sticks, and that she does so except for the hidden volume.

The critical observation here is that each of these mounted volumes has a unique volume serial number. While this volume serial number is not available in the registry, it *is* available to applications after the volumes are mounted and *is* also stored in the shortcut `.lnk` files. The customs agent can now compare the volume serial numbers in the relevant `.lnk` files to the volume serial numbers for all the volumes that Alice mounted and, if there is discrepancy, he knows that there exists or existed a file system that Alice is not disclosing.

While the above scenario is described in real-time; that is, happening while Alice is in the presence of the customs agent, the customs agent could collect similar evidence by simply cloning Alice's hard drive and, either at that time or later, asking her to supply all her passwords. We view the above scenario as plausible given the current political environment [3, 5, 8], and even more likely in other countries. Even if such a scenario were unlikely, we view the potential for such a scenario — coupled with the well-known fact that users have difficultly following security protocols — to be sufficiently risky to advocate not trusting in the deniability of TrueCrypt hidden volumes.

4.2 Example Leakage Through the Primary Application: Word Auto Saves

In order to dampen data loss in the case of a crash, many applications use backup or recovery files. When editing a file, the application will create a local copy of the file and record in it all changes made to the original file. When the application successfully closes and saves all modifications, the backup file is removed. It makes sense that these applications store backup files locally in case an external volume is prematurely unmounted (such as removing a USB stick before it finishes syncing, or an emergency power-off). For example, Microsoft Word 2007[7] by default creates auto-recover files in `C:\Users\UserName\AppData\Roaming\Microsoft\Word\`.

The consequence of this auto-recover mechanism is that a file that was originally stored in a hidden volume has now been copied to the local hard disk. Although Word removes the backup file after the user closes the primary file, Word does not invoke secure delete. We were repeatedly able to recover previously edited files that were stored on the hidden volume by running a simple, free data recovery tool.

Specifically, we first ran `cipher /w:c:\`[8] to clean up any dirty free space. We then mounted a hidden TrueCrypt volume on which we had previously stored seven copies of the Declaration of Independence of the United States of America titled as variants of `OverthrowGovernment.doc`. We then opened and edited these files with Microsoft Word, saved the changes to the original files, exited Word, and unmounted the hidden volume — all things an ordinary user might be reasonably expected to do while using a TrueCrypt volume.

[6]Our experiments were with Windows Vista Business edition with Service Pack 1, though our results apply more broadly.

[7]Word 2007 (12.0.6311.5000) SP1 MSO (12.0.6213.1000) Part of Microsoft Office Home and Student 2007.

[8]A Windows command that writes 0x00, 0xFF, and then random data to all free space blocks on a disk.

Without rebooting, we ran a simple, free data recovery tool.[9] The number of files we recovered and their quality was not consistent on every experiment, since the free space on the `C:` drive can be changed at any moment. Nevertheless, we were able to fully recover most of the documents in their entirety from the Word auto-recovery folder. After a fresh experiment that included a reboot after the hidden volume was unmounted, half of the files were still recoverable. We hypothesize that we could have recovered even more data if we had performed more sophisticated techniques such as examining the individual file blocks on disk.

Furthermore, in cases when an application is prematurely terminated — for example, during a crash or by a kill command — the auto-recover file will persist on the local disk in clear view and will be recovered by Microsoft Word without the TrueCrypt volume being mounted. This means that just pulling the power on one's computer at the sight of authorities will not safeguard a file if it was open or in the process of being edited.

While Microsoft Word presents users with the ability to prevent auto-saves from occurring through the preferences dialog, other applications may not. An attacker can use information gleamed from these files — as well as other information leakage from the primary application — to not only infer that a hidden volume exists, but also recover some of its contents.

4.3 Example Leakage Through A Non-Primary Application: Google Desktop

Non-primary applications may also access the files stored on a hidden volume. For example, desktop search applications are becoming more prevalent, allowing users to quickly navigate their computers. However, in addition to merely indexing the files on a computer to aid in searching for files, at least one desktop search application — Google Desktop[10] — includes an additional feature that presents problems for a DFS. This feature is the ability to view previous states of websites and files such as Microsoft Word documents. According to Google, the purpose of this caching feature is to recover accidentally deleted files or to simply view an old version of a file or web page. The default installation of Google Desktop does not provide for the indexing or caching of files; however, this basic mode of operation limits Google Desktop's ability to assist the user. When installing Google Desktop, the user merely needs to select the "Enhanced Search" option to enable both the indexing and caching of certain files types.

To protect privacy and to optimize searched areas, the user is provided with the ability to choose which folders to include and exclude from indexing. However, Google Desktop claims that all fixed drives will be indexed by default. As a user opens documents, modifies them, and saves them, Google Desktop caches snapshots of the files and stores them for later viewing. Google Desktop provides not only the latest cached version of a file but also multiple versions of a file cached at different times. These cached files could provide an attacker with not only the current state of a document but also a view of its evolution over time.

In our tests we created a Word document called `OverthrowGovernment.doc` in a TrueCrypt hidden volume and added sections of the Declaration of Independence to the document. With Google Desktop installed[11] and running, we edited and saved the document numerous times. After unmounting the TrueCrypt hidden volume, we were easily able to recover a number of snapshots of our file by merely searching for any word contained in the file. We were able to repeat these results irrespective of the volume's mount point (any drive letter F, G, H, L) or mount type (normal/fixed and removable medium). See Figure 1.

Thus, to protect oneself, the user cannot rely on only tweaking TrueCrypt's settings, but rather must understand the dangers presented by the non-primary application itself. In the case of Google Desktop, one of two choices could be made to prevent Google Desktop from leaking file contents out of TrueCrypt volumes. The user can either forgo the features of Google Desktop Enhanced Search, forcing it into a limited mode of operation, or the user can make the conscious choice to either shut down Google Desktop or pause its indexing whenever using a TrueCrypt volume. This burden of needing to understand exactly how a non-primary application interacts with a TrueCrypt volume underscores the difficulties of implementing and using a DFS.

While non-primary applications such as Google Desktop may allow the user to pause its actions at arbitrary points, other non-primary applications may not provide the user with this capability even if the user understands the dangers posed by the application. In addition, malicious applications, like botnets or viruses, could obviously compromise the deniability of a hidden volume in ways that the user cannot predict nor prevent.

5 Future Directions

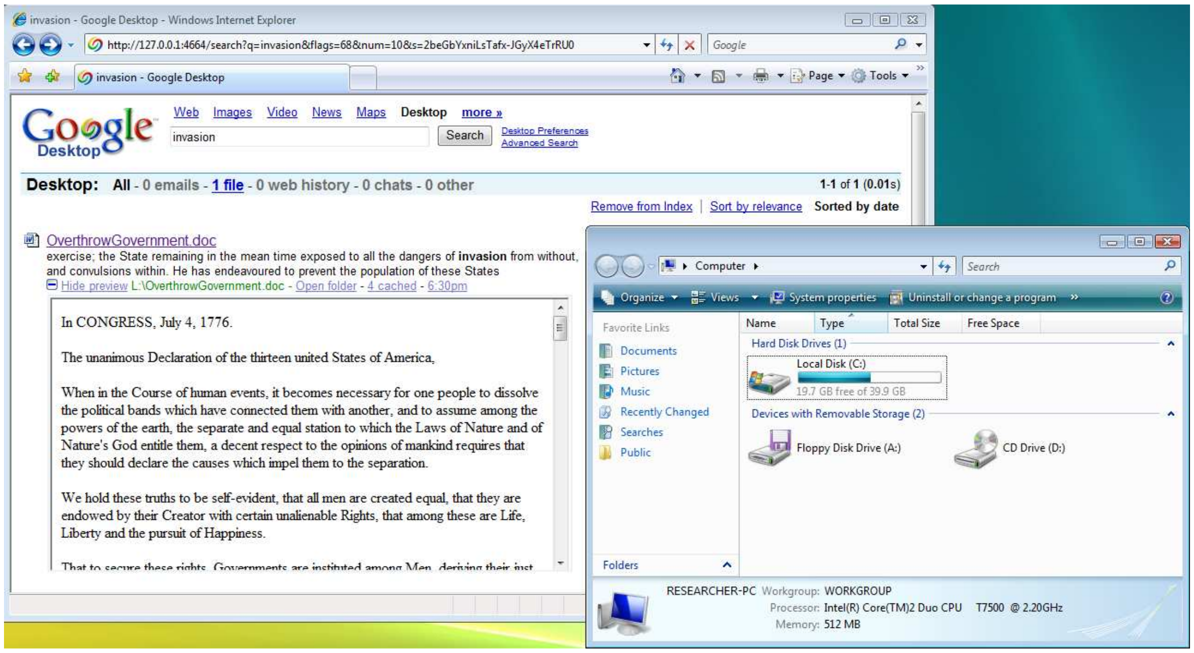
5.1 Defensive Directions

It may be possible to address each of the above-mentioned information leakage vectors in isolation. For example, to counter the Windows shortcut-based attack, TrueCrypt may consider using the same serial number for *all* volumes, serial numbers that are a function of the

[9] `http://tokiwa.qee.jp/EN/dr.html`; DataRecovery 2.4.2.

[10] `http://desktop.google.com/`; we evaluated version 5.7.0805.16405-en-pb.

[11] Default install with Enhanced Search enabled.


invasion - Google Desktop - Windows Internet Explorer

http://127.0.0.1:4664/search?q=invasion&flags=68&num=10&s=2beGbYxniLsTafx-JGyX4eTrRU0

Desktop: All - 0 emails - 1 file - 0 web history - 0 chats - 0 other   1-1 of 1 (0.01s)

Remove from Index | Sort by relevance Sorted by date

OverthrowGovernment.doc
exercise; the State remaining in the mean time exposed to all the dangers of **invasion** from without, and convulsions within. He has endeavoured to prevent the population of these States
Hide preview L:\OverthrowGovernment.doc - Open folder - 4 cached - 6:30pm

In CONGRESS, July 4, 1776.

The unanimous Declaration of the thirteen united States of America,

When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable Rights, that among these are Life, Liberty and the pursuit of Happiness.

That to secure these rights, Governments are instituted among Men, deriving their just

Hard Disk Drives (1)
Local Disk (C:) 19.7 GB free of 39.9 GB
Devices with Removable Storage (2)
Floppy Disk Drive (A:)
CD Drive (D:)

RESEARCHER-PC Workgroup: WORKGROUP
Processor: Intel(R) Core(TM)2 Duo CPU T7500 @ 2.20GHz
Memory: 512 MB

Figure 1: Information leakage through Google Desktop. The TrueCrypt hidden volume has been unmounted, and yet we can recover numerous snapshots of the hidden file's contents.

mount point, random serial numbers each time, or some combination of the above. Other *ad hoc* approaches may also be successful at addressing this particular information leakage channel.

However, such *ad hoc* approaches do not solve the fundamental problem: the operating system and applications can leak significant information about the existence of, and the files stored within, a hidden volume. We have identified three broad classes of information leakage vectors, with concrete examples for each class. However, we are sure that other examples likely exist, waiting to be discovered. The problem is therefore much more fundamental, and addressing it will require rethinking and reevaluating how to build a true DFS in the context of modern operating systems and applications.

To create a DFS, it seems inevitable that the operating system (and perhaps the underlying hardware) must assist in the deniability. New operating system architectures like HiStar [10] could help ensure that information about a DFS does not leak to other portions of a system. However, the strong information flow guarantees afforded by such architectures may be overkill to prevent the most typical information leaks. Moreover, these architectures require significant changes to the operating system.

An approach that may work well with existing operating systems is to install a file system filter that disallows a process any write access to a non-hidden volume (or the registry, under Windows) once that process reads information from a hidden volume.[12] Robust applications should accept the fact that they cannot store temporary files and either alert the user that a feature is not available (like auto-recovery) or silently fail (like saving a shortcut to a recent document). Less robust applications may not work under this new restriction. However, the fact that applications no longer work tells the user that application will leak potentially private information. The user can then weigh the risks and benefits of using the application in an unprotected manner, and manually allow the application to work outside the protection scheme.

While this does not provide 100% protection, it may work well enough to stop typical information leaks. For example, a process may leak information about a hidden volume to another process via IPC or a network connection, and that second process may write this information to a non-hidden volume. However, we hypothesize that these situations are rare in practice, and that this minimalist information flow approach will substantially improve the deniability of hidden volumes; users must still avoid being lulled into a false sense of additional security greater than what is actually afforded.

Another possible direction would be to create a "TrueCrypt Boot Loader" that, upon entering one password, decrypted the disk one way and booted the OS. And, upon entering a different password, decrypted the deni-

[12]It may also be desirable to mark certain applications, such as Google Desktop, as never being allowed to read a hidden volume, lest they be permanently tainted with knowledge of the forbidden data.

able portion of the disk and booted the OS in the deniable partition. (Addendum: such a boot loader is now implemented in TrueCrypt v6.0.)

We leave other specific directions as open problems.

5.2 Reflections to Regular Disk Encryption

Reflecting on the second and third classes of information leakage (Sections 4.2 and 4.3), we stress that they also seem applicable to regular (non-deniable) disk encryption systems in which only a subset of all the user's entire disks are encrypted and in which a user does not deny the existence of the encrypted regions but does refuse to divulge the passwords.

In our future work, we would like to investigate exactly how well these attack vectors apply to regular encrypted disks and volumes. We also wish to explore methods for limiting information leakage in situations when whole disk encryption is not used (for example: when using an encrypted USB drive or virtual disk with a non-encrypted system disk). In summary with regard to disk encryption, in situations where there is a need to protect the privacy of individual files, the safest strategy appears to be to encrypt the full disk with tools like PGP Whole Disk Encryption.

6 Related Works

We mention significant related work in-line above, but recall here the long history of steganographic and deniable file systems beginning with the work of Anderson *et al.* [1], and followed by both academic publications [6, 7] as well as non-academic but widely circulated applications (include TrueCrypt); see also [9]. A related thread of research, though not targeting deniability, is focused on encrypted file systems, again including both published research such Blaze's encrypted file system for Unix [2], commercial systems such as PGP Whole Disk Encryption and BitLocker, and open source systems, including TrueCrypt. There are large bodies of research focused on information leakage in other contexts.

7 Conclusion

We demonstrate three broad classes of information leakage vectors against the deniability of a TrueCrypt hidden volume: leakage from the operating system; leakage from the primary application; and leakage from non-primary applications. We believe that our work underscores a new direction for the design and analysis of deniable file systems — a direction that must include provisions for protecting against information leakage from the external environment that interacts with the deniable file system while the file system is mounted.

Addressing these issues is both timely and important, as evidenced by the current political environment in which computers are searched at international borders and the broad media discussions about the advantages of deniable file systems like TrueCrypt's, e.g., [3, 5, 8].

On the other hand, even if a DFS is secure, it might not be a good solution to Alice's secret-police problem. Just as an attacker would not be able to prove the existence of secret data under such a secure DFS, the same attacker wouldn't be able to prove the non-existence of deniable data. If the secret police continue to demand that Alice disclose the password to such a deniable file system, there is no way for her to prove that her configuration doesn't have such a volume. Deniability cuts both ways, and sometimes that's not a benefit.

References

[1] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *Information Hiding*, 1998.

[2] M. Blaze. A cryptographic file system for UNIX. In *CCS*, 1993.

[3] J. Granick. EFF answers your questions about border searches. `http://www.eff.org/deeplinks/2008/05/border-search-answers`, 2008.

[4] J. Hager. The Windows shortcut file format. `http://www.i2s-lab.com/Papers/The_Windows_Shortcut_File_Format.pdf`, 2003.

[5] D. McCullagh. Security guide to customs-proofing your laptop. `http://www.news.com/8301-13578_3-9892897-38.html`, 2008.

[6] A. D. McDonald and M. G. Kuhn. StegFS: A steganographic file system for Linux. In *Information Hiding*, 1999.

[7] B. Oler and I. E. Fray. Deniable file system: Application of deniable storage to protection of private keys. In *International Conference on Computer Information Systems and Industrial Management Applications (CISIM)*, 2007.

[8] B. Schneier. "Taking your laptop into the US? Be sure to hide all your data first". *The Guardian*, 15 May 2008.

[9] B. Schneier. Deniable file system. `http://www.schneier.com/blog/archives/2006/04/deniable_file_s.html`, 2006.

[10] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.