BurnBox: Self-Revocable Encryption in a World Of Compelled Access

Nirvan Tyagi, ***Cornell University;*** **Muhammad Haris Mughees,** ***UIUC;***
Thomas Ristenpart and Ian Miers, ***Cornell Tech***

https://www.usenix.org/conference/usenixsecurity18/presentation/tyagi

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-04-5

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

BurnBox: Self-Revocable Encryption in a World of Compelled Access

Nirvan Tyagi
Cornell University

Muhammad Haris Mughees
UIUC

Thomas Ristenpart
Cornell Tech

Ian Miers
Cornell Tech

Abstract

Dissidents, journalists, and others require technical means to protect their privacy in the face of compelled access to their digital devices (smartphones, laptops, tablets, etc.). For example, authorities increasingly force disclosure of all secrets, including passwords, to search devices upon national border crossings. We therefore present the design, implementation, and evaluation of a new system to help victims of compelled searches. Our system, called BurnBox, provides self-revocable encryption: the user can temporarily disable their access to specific files stored remotely, without revealing which files were revoked during compelled searches, even if the adversary also compromises the cloud storage service. They can later restore access. We formalize the threat model and provide a construction that uses an erasable index, secure erasure of keys, and standard cryptographic tools in order to provide security supported by our formal analysis. We report on a prototype implementation, which showcases the practicality of BurnBox.

1 Introduction

More and more of our digital lives are stored on, or remotely accessible by, our laptops, smartphones, and other personal devices. In turn, authorities increasingly target these devices for warranted or unwarranted searches. Often this arises via *compelled access*, meaning the physically-present authority requires disclosure (or use) of passwords or biometrics to make data on the device temporarily accessible to them. Nowhere is this more acute than in the context of border crossings, where, for example, the United States authorities searched 158% more devices in 2017 than 2016 [5]. This represents a severe privacy concern for general users [62], but in some contexts, searches are used to arrest (or worse) dissidents, journalists, and humanitarian aid workers.

Proposals for privacy-enhancing tools to combat compelled access are not new, but typically do not consider the range of technical skills and preparedness of the increasingly broad population of targeted users, nor the frequently cursory nature of these searches. Take for example, deniable encryption [9,18,52], in which a user lies to authorities by providing fake access credentials. Deniable encryption has not proved particularly practical, both because it puts a high burden on users to be willing and able to successfully lie to authorities (which could, itself, have legal consequences) and because it fundamentally relies on realistic "dummy" content which users must construct with some care.

We explore a new approach that we call *self-revocable encryption*. The idea is simple: build applications that can temporarily remove access to selected content at the user's request. This functionality could then be invoked right before a border crossing or other situation with risk of compelled access. Should the user's device be searched, there is no way for them to give the authority access to the sensitive content. Because revealing metadata (e.g., filenames), whether a file was revoked or deleted, or when revocation happened could be dangerous, we want self-revocable encryption to hide all this from searches. The user should be able to later restore access to their content.

In this work, we focus specifically on the design, implementation, and evaluation of a cloud file storage application. Here we target self-revocable encryption in a strong threat model in which the adversary monitors all communication with the cloud storage system and can at some point compel disclosure of all user-accessible secrets (including passwords) and application state stored on the device. This means we target privacy not only for cursory searches of the device, but also for targets of more thorough surveillance. To be able to later restore access, we assume the user can store a secret restoration key in a safe place (e.g., with a friend or in their home) that the adversary cannot access. Should that not be available, only secure deletion is possible.

The first challenge we face is refining and formalizing

this threat model, as it is unclear a priori what privacy goals are even achievable. For example, no efficient system can hide that there exist cloud-stored ciphertexts that are no longer accessible by the client, because the adversary can, during a search, enumerate all accessible files and compare to the total amount of (encrypted) content that has been uploaded to the cloud service. Hiding this would require prohibitive bandwidth usage to obfuscate the amount of storage used. Instead, we target that the adversary, at least, cannot distinguish between regular deletion of data and temporary revocation. One of our main technical contributions is a formal security notion that captures exactly what is leaked to the adversary, a notion we call *compelled access security* (CAS). It uses a simulation-based definition (similar to that used for searchable encryption [21, 23]).

To achieve CAS, we design an encrypted cloud storage scheme. It combines standard encryption tools with techniques from the literature on cryptographic erasure [16, 22, 57] and use of data structures in a careful way to avoid their state revealing private information. The latter is conceptually related to history-independent data structures [31, 47, 48], though we target stronger security properties than they provide.

The proof of our construction turns out to be more challenging than expected, because it requires dealing with a form of selective opening attack in the symmetric setting [13, 19, 54]. Briefly, our approach associates to individual files distinct encryption keys, and in the security game the adversary can adaptively choose which files to cryptographically erase by deleting the key. The remaining files have their keys exposed at the end of the game. Ultimately this means we must have symmetric encryption that is non-committing [19]. We achieve this using an idealized model, which is sufficient for practical purposes. We leave open the theoretical question of whether one can build self-revocable encryption from weaker assumptions.

We bring all the above together to realize BurnBox, the first encrypted cloud file storage application with self-revocation achieving our CAS privacy target. We provide a prototype client implementation that works on top of Dropbox. BurnBox can revoke content in under 0.03 seconds, even when storing on the order of 10,000 files.

Summary. In this paper, we investigate the problem of compelled access to user's digital devices.

- We propose a new approach called self-revocable encryption that improves privacy in the face of compelled access and should be easier to use than previous approaches such as deniable encryption.
- We provide formal security definitions for compelled access in the context of cloud storage applications. Meeting this notion means that a scheme leaks nothing about private data beyond some well-defined leakage.
- We design a self-revocable encryption scheme for cloud storage that provably meets our new definition of security.
- We provide a prototype implementation of our design in the form of BurnBox, the first self-revocable encrypted cloud storage application.

We also discuss the limitations of BurnBox. In particular, in implementations, the operating system and applications may unintentionally leak information about revoked files. While our prototype mitigates this in various ways, being comprehensive would seem to require changes to operating systems and applications. Our work therefore also surfaces a number of open problems, including: how to build operating systems that better support privacy for self-revocable encryption, improvements to our cryptographic constructions, what level of security can be achieved when cloud providers actively modify ciphertexts, and more. We discuss these questions more throughout the body.

2 The Compelled Access Setting

We start by taking a deeper dive into the setting of compelled access. To be concrete, we focus our discussion on cloud storage applications. Consider a user who stores files both in the cloud and on a device such as a smart phone or laptop that they carry with them. The cloud store may be used simply to backup a copy of some or all files on their device or it may be used to outsource storage off of the device for increased capacity. We assume the files include some that are sensitive, such as intimate photos, videos, or text messages, or perhaps politically sensitive media such as a journalist's photos of war zones. As such the user will not want this data accessible by the cloud provider, and will want to use client-side encryption.

We consider settings in which the user may be subjected to a *compelled access search*. After using their application for some time, a physically present authority forces the user to disclose or use their access credentials (passwords, biometric, pin code, etc.) to allow the adversary access to the device and, in particular, the state of the storage application's client. Thus all secrets the person knows or has access to at that time will be revealed to the authority. We will assume that the user has advanced warning that they may be searched, but we will target ensuring the window between warning and search need not be large (e.g., just a few minutes).

As mentioned in the introduction, compelled access searches are on the rise. Border crossings are an obvious example, but they occur in other contexts as well. Protesters are frequently detained by the police and have

their devices searched [27]. Even random police stops in some countries have led to compelled access searches, so much so that people reportedly carry decoy devices [17]. In these settings, standard client-side encryption proves insufficient: because the user is compelled to give access to their device and all passwords they have, the authority gains both the credentials to access the cloud and the keys necessary to perform decryption.

Surveilled cloud storage. At first glance, one apparent way to resist compelled access searches would be to simply use a client-side encryption tool, and have the cloud storage delete ciphertexts associated to sensitive data. This wouldn't allow temporary revocation, just cryptographic deletion. But more fundamentally, it will not work should the cloud storage fail to act upon delete requests. Such ciphertext retention can occur either unintentionally, e.g., Dropbox's accidental retention of deleted files for 8 years [50], or through collusion with an adversary such as a nation-state intelligence service. For example, at the time the United States' National Security Agency's PRISM surveillance program was disclosed, Dropbox, Google, and Microsoft were either active participants or slated for inclusion [39].

Beyond existing systems, ciphertext retention seems unavoidable in newly emerging models of cloud storage that use public peer-to-peer networks. These approaches range from systems such as Resilio Sync (formerly BitTorrent Sync) built on top of distributed hash tables, to commercial startups using blockchain-based storage [40, 44, 68, 71]. In such peer-to-peer settings ciphertexts are widely distributed and it is impossible to either assure that copies were not accidentally retained or deliberately harvested via, e.g., a Sybil attack [72].

In either case, we will want solutions that do not rely on data written to the cloud being properly deleted.

Potential solutions to compelled access. One common approach, used widely in practice for boarder searches, is simply to wipe the device of all information (perhaps by destroying it). However, this does not provide any granularity and forces users to discard every file. This would deprive them of contacts numbers, travel documents, and most of the functionality of their device.

Another approach is that of feigned compliance, e.g., via tools such as deniable encryption [4, 10, 18, 29, 34, 46, 52, 55, 65] or so-called "rubber hose crypto." These require the user to purposefully lie to the authorities, and manage "dummy" cover data that must be realistic looking. We believe such feigned compliance approaches have severe limitations in terms of both psychological acceptability due to the requirement to actively deceive, and on usability because users must manage cover data. Given that most users do not really understand basic encryption [60, 63, 70], this seems a significant barrier to useful deployment.

Our goal will instead be for the user to genuinely comply with demands for access to the device and everything they know, and not force them to manage cover data or lie to achieve any security. Of course the user may face specific questions abut what they deleted or if they can restore access to files. In this case, the user can choose to lie or admit to having deleted or (temporarily) revoked files. But unlike deniable encryption, either choice still preserves the security of deleted or revoked files. In short, deception should not be *inherent* to security.

Given this objective, the next logical straw proposal is to just selectively delete files. Cryptographic erasure has been studied in a number of works [16, 22, 57] that primarily focus on deleting files from local storage. However, standard cryptographic erasure as a primitive is insufficient for two reasons. First, without embellishment it does not allow users to later recover their files. Second, and more subtly, it does not protect privacy-sensitive metadata such as filenames: for efficient retrieval from cloud storage, the client must store some index enumerating all files by name.

Self-revocable encryption. We therefore introduce a new approach that we call *self-revocable encryption*. Here the user renders selected information on the device temporarily unreadable, but retains some means to later regain access. How? The user cannot store material on the device or memorize a password, as these will be disclosed. Instead, we leverage the fact that a compelled access attack is limited to what information and devices a user has on their person: data stored at their home or with a friend is not accessible. We refer to this storage location, generically, as a restoration cache and have the user store a token tok_{res} in it that enables restoration of revoked ciphertexts. A diagram appears in Figure 1.

We believe self-revocable encryption, should it be achievable, has attractive properties. It's conceptually simple and doesn't require lying to authorities. Moreover, the user does not have to manage dummy data.

Threat model. We now review our threat model in more detail. Our goal is to protect the confidentiality of a client device and encrypted cloud store in the presence of an adversary who can compel the user to give the adversary access to the device. The user stores sensitive files encrypted in the cloud and on their device which has the ability to add, retrieve and decrypt files from the cloud. The adversary can force a user to unlock their device, disclose account passwords, and may fully interact with the device and clone it. Furthermore, we assume they are a passive adversary with respect to the cloud: obtaining access logs as well as all versions of any (encrypted) files the user uploaded (including subsequently deleted files) but not actively manipulating files. While we will pro-

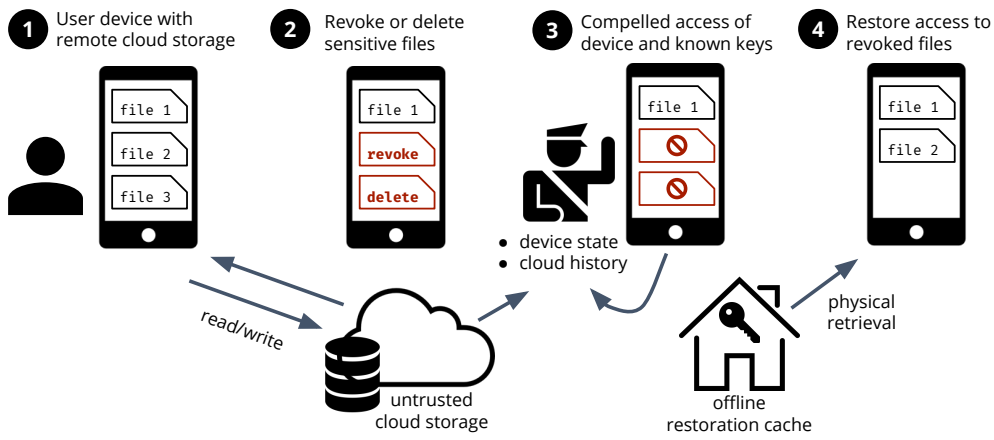

Figure 1: Self-revocable encryption for cloud storage. A user stores data on their device and in the cloud. Anticipating their device will be inspected, the user temporarily revokes access to file 2, a sensitive file they will need access to later, and deletes file 3. When the device is searched, file contents and filenames of deleted or revoked files are hidden. After the search, the user can restore access to revoked files using their device and the restoration cache—key material stored at their home, office, or with friends.

vide some mechanisms against tampering in the concrete construction, our formal analysis does not consider active attacks on the cloud store.

In this context, we now describe the properties we want of our system for deleted and revoked files in the presence of compelled access.

File content privacy. The content of deleted or revoked file should be protected post compromise. File contents may include intimate details of a user's life such as photo or videos, politically controversial content such as banned books or newspapers, or sensitive business information.

File name privacy. The names of deleted or revoked files should be protected post compromise. File names can reveal information about the content of the file. Moreover, it allows the adversary to check if a user owns a flagged file from a list of, e.g., politically "subversive" works.

We next describe two secondary goals to support the (optional) ability of a person to equivocate about revocation and deletion history. These properties are not necessary for BurnBox to be useful, but may be desirable in some instances.

File revocation obliviousness. Whether a file was deleted or revoked should remain hidden. If the adversary determines access to files was self-revoked, then she has learned the user explicitly has files he wants to hide. Revocation is done precisely to avoid compelled disclosure. In contrast, deletions can be done for many reasons.

Deletion and revocation timing privacy. The timings of file deletions and revocations should, to the extent possible, be concealed. If the adversary has reason to believe a user deleted or revoked data specifically to avoid compelled access, the user could face consequences. As we discuss in Section 8 this is not fully realizable without certain forensic guarantees on persistent storage.

Threats not modeled. We do restrict the threat model in several ways. First, the adversary cannot force the user to retrieve keys from other locations such as their home or office, i.e., the restoration cache. If that were possible, then one can only provide privacy via secure deletion (which is supported by BurnBox). Second, the adversary cannot implant malware on the device that persists after the compelled access search ends. In that case, files will be exposed when later restored and the only solution would be to never use the device again with those sensitive files. Third, we assume the adversary does not get access to system memory, i.e., the device is turned off prior to compelled access (at which point it may be turned on again). Fourth, we assume the adversary only has passive access to the cloud.

Finally, although we hide the individual number of deleted or revoked files, we will not target hiding the sum of these values, meaning the total number of files that have been either revoked or deleted. Similarly, we will not hide some forms of access patterns. We will hide whether a delete or revoke occurs, but we will reveal to the cloud storage adds and accesses. We discuss the implications of this leakage in Section 8.

3 Overview and Approach

In this section we give some intuition about our approach to realizing self-revocable encryption in the context of cloud storage systems. Section 4 presents the details.

From encrypted files to erasable files. Consider a cloud storage provider that offers a simple key value store mapping a human-readable filename ℓ to its contents m via a $\mathsf{Put}(\ell, m), \mathsf{Get}(\ell)$ interface. We start with the simpler problem of permanently deleting files from the cloud store and then extend the system to support temporary self-revocation and to protect metadata. To enable secure deletion of encrypted files, we generate a random per file key k_f which is stored locally, and store $\mathsf{Enc}_{k_f}(m)$ instead of m in the cloud under label ℓ. Here Enc is a symmetric encryption scheme (technically, one should use an authenticated-encryption scheme). Erasing the local copy of k_f erases the file contents.

While cryptographic erasure securely deletes the file contents, it failures to provide filename privacy: there is still an *index*, i.e., a mapping from filename ℓ to an (undecryptable) ciphertext. This index must be preserved to enable file retrieval. Thus cryptographic erasure does not provide a full solution to the problem.

Following the approach of many searchable encryption schemes [23], one could create a "PRF index" that replaces ℓ with a filename pseudonym $t = F_k(\ell)$ where F is a secure pseudorandom function (PRF). This hides the human readable filename but still enables efficient retrieval of the file given its name. It does not completely fulfill our goals, however. On compromise, knowledge of the PRF key k and a previously stored value t would allow an attacker to enumerate the filename space and learn filenames, essentially mounting a brute-force dictionary attack like those used for password cracking. If the PRF is also used to generate encryption keys, they can learn these as well.

From erasable files to erasable index entries. Puncturable PRFs [30] would appear to resolve the issue of leaking label to filename pseudonym mappings by providing an algorithm, *puncture*, that converts the PRF key k to a key k' for which one cannot evaluate the PRF on a particular point v. If the key is punctured on the filename, an attacker with access to k' cannot enumerate filenames by testing evaluations of the PRF on candidate filenames. Unfortunately, puncturable PRFs do not hide the points the key is punctured on: while an attacker would not be able to identify the mapping from filename to ciphertext, they would be able to identify the punctured filenames themselves. This can be resolved with a private puncturable PRF [15] which hides the points the key is punctured on. Unfortunately, these are not currently practical and thus not (yet) suitable for BurnBox.

Instead, we construct an *erasable index* using a simple table to store a mapping from filename to a randomly sampled value. This can be viewed as a form of stateful, private puncturable PRF. While extremely simple in concept, secure implementation is complicated by the requirement that the table is persisted to disk.

In the compelled access setting, an attacker gets full access both to the on-disk representation of the table and the physical state of the disk. This raises two distinct problems: first any data that has been overwritten or deleted from the table may still be be retained by the file system (e.g., in a journaled file system) or physically extractable from the drive (e.g., due to ware-leveling for SSDs or the hysteresis of magnetic storage media). Second, even if we can ensure old data is erased, the current state of the backing data-structure may reveal operations even if the data itself is gone. Were we to use a simple hash table, for example, the location of a particular entry depends on whether collisions occurred with other entries at insertion time. This lack of history independence leaks the past presence of other colliding entries even if the entries themselves are removed and physically erased.

We are thus left with two questions: how to ensure individual entries in the table can be removed without leaving forensic evidence, and how to structure the table so no trace is left when they are.

Erasing index entries securely. To remove or overwrite entries from the table without accidentally leaving old values accessible via forensics, we follow the approach of previous cryptographic erasure techniques [58]. We assume a small (e.g., 256-bit) securely erasable "effaceable storage" in which to store a *master key*. Naively, we could encrypt the entire table under this key and update or remove a row by overwriting the effaceable storage with a new key and writing an updated version of the table encrypted under the new key to disk. However, this means operations on a single entry require work linear in the size of the table.

Instead, we adopt a tree-based approach [58] for key management. Each entry in the table is encrypted with a unique key. Keys are stored as leaves of a key tree; sibling nodes are encrypted with a new key, which is stored as their parent. The root of the tree is encrypted under the master key stored in effaceable storage. Thus, an update (1) re-encrypts the updated row under a new key and (2) updates the key tree by sampling new keys for the tree path corresponding to that row and re-encrypting the tree path and path siblings. In summary, the erasable index consists of an encrypted table with encryption per entry and corresponding key tree, depicted in Figure 2.

Using data structures privately. While we have ensured individual entries in the table can be erased without leaving direct forensic evidence, we now need to en-

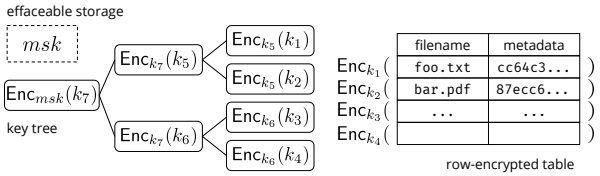

Figure 2: An erasable index for four items consisting of a key tree where each leaf encrypts a separate row of the table. The root of the key tree is encrypted by a master key stored in effaceable storage such as a hardware keystore.

sure the data structures as persisted to disk do not reveal past (erased) content. History independent data structures [47, 48] are a natural candidate for structuring the index and avoiding such leakage. Strongly history independent hash tables [47] achieve privacy for a particular update to the data structure even if an attacker has access to a snapshot both before and after a series of updates.

In the compelled access setting, however, due to the previously stated non-assumption of persistent storage deletion (e.g., journaling or hardware forensics), the attacker may get snapshots at each and every update. While cryptographic erasure ensures the actual content of the update is opaque, the timing, location, and size of individual writes needed to make the update is not. Although some schemes consider this type of storage leakage in the context of PROM for voting machines [47], we are aware of no general approaches. Indeed, eliminating all such leakage in the presence of an arbitrary file system and storage medium is problematic: even heavyweight techniques like ORAM leak the size of writes. Thus, these kinds of generic history-independent data structure techniques do not seem suitable for our setting.

We therefore take an application-specific approach, arranging that our data structures are used in a way that is independent of our application's privacy-sensitive information. Here we take that to be filenames, and so our data structures cannot be dependent on filename. Our key tree is already independent of filenames. To ensure the table is independent of filenames, we maintain it sorted in insertion order. While this means we leak some information about insertion order, we deem this acceptable (see Section 5). Looking ahead to the performance evaluation (Section 7), this ordering makes it harder to do efficient filename search, but appears to be necessary for our desired privacy properties.

From permanent erasure to self revocation. The above approach does not support self-revocation—it can only permanently delete files. To solve this, we use a form of key escrow. We generate a asymmetric key pair $(pk_{\text{res}}, sk_{\text{res}})$ and store sk_{res} only in the secure restoration cache (and not on the device). When adding a file to the storage, we generate a *restoration* ciphertext of the form $\mathsf{Enc}_{pk_{\text{res}}}(\ell \,||\, k)$ which contains both the key k for a given file and its filename ℓ. The restoration ciphertext is only stored locally on the device.

To revoke access to the file, the entry in the erasable index is deleted. To delete the file, we must also erase the restoration information. Deleting the restoration ciphertext itself would violate deletion-revocation obliviousness upon compromise. Instead, we overwrite the ciphertext with an encryption of a random value. For the same reason, the ciphertext must be stored only on the device: if the adversary can observe accesses to the restoration ciphertext, this would violate both deletion-revocation obliviousness and deletion timing privacy.

Enabling backup and recovery. The approach so far does not support recovery of files should the device be lost or damaged. If BurnBox is used for cloud backup, rather than just to extend a device's storage capacity, this is a major limitation. One option would be to create a backup key and augment our approach to ensure all files are decryptable with that key. However, such a key would be able to decrypt any file, including deleted ones. A safer way to enable recovery would be to sync key state between multiple devices over a secure channel. The choice of channel must be made carefully as an adversary could observe the channel to learn the timings of operations or block sync messages to prevent deletes.

4 Construction

We now provide a detailed description of the cryptographic primitives underlying BurnBox.

Syntax and semantics. We start by defining self-revocable encrypted cloud storage (SR-ECS). In the following we use $y \leftarrow\!\$\, \mathsf{Alg}(x)$ to denote running a randomized algorithm Alg with fresh coins on some input x and letting y be assigned the resulting output.

An SR-ECS scheme consists of seven algorithms: $\mathsf{SR\text{-}ECS} = (\mathsf{Init}, \mathsf{Add}, \mathsf{Access}, \mathsf{Delete}, \mathsf{Revoke}, \mathsf{Restore})$.

- $st_0, tok_{\text{res}} \leftarrow\!\$\, \mathsf{Init}()$: The initialization algorithm returns an initial local client state and a secret restoration token to be hidden off the local client.
- $st_{i+1} \leftarrow\!\$\, \mathsf{Add}(st_i, \ell, m)$: The add algorithm takes as input the current state st_i, filename ℓ, and file contents m and outputs a new local client state.
- $st_{i+1}, m \leftarrow\!\$\, \mathsf{Access}(st_i, \ell)$: The access algorithm takes a state and a filename, and returns a new state and file contents for that filename, or an error.
- $st_{i+1} \leftarrow\!\$\, \mathsf{Delete}(st_i, \ell)$: The delete algorithm takes as input a state and filename, and outputs a new state. The filename and associated content should be permanently deleted.

- $st_{i+1} \leftarrow\!\!\$\ \mathsf{Revoke}(st_i, \ell)$: The revoke algorithm takes as input a state and filename, and outputs a new state with filename and associated content temporarily deleted.
- $st_{i+1}, tok_{\text{res}} \leftarrow\!\!\$\ \mathsf{Restore}(st_i, tok_{\text{res}})$: The restore algorithm takes as input a state and secret restoration token, and outputs a new state with all self-revoked files restored along with a (potentially new) restoration token.

We require our schemes to be correct. Informally, that means that encrypted files that are not currently revoked or deleted should be accessible and correctly decryptable. Accesses on filenames not added to the system or that were revoked/deleted, that return a special error symbol $\perp$. As a consequence, the set of all filenames and, by extension, file contents that are not revoked or deleted are learnable by an adversary with control of the device, e.g., by mounting a brute force search. Hiding the set of active files is not a goal of SR-ECS as it is in related deniable encryption schemes.

ECS algorithms will use access to a remote storage server, which we abstract as a key-value (KV) store with operations $\mathsf{Put}(K,V)$ and $\mathsf{Get}(K)$ that put and retrieve entries from the store. Both Put and Get are available as oracles to all ECS scheme algorithms, though we omit their explicit mention from the notation for simplicity. Looking ahead, we will be interested in the transcript of calls to the KV store, representing the state of the server. For example, if an ECS algorithm made the call $\mathsf{Put}(2, \mathsf{foo})$, the transcript would include the tuple $(\mathsf{Put}, 2, \mathsf{foo})$. We later will use implicitly defined transcript-extended versions of ECS algorithms that add an extra return value, the transcript τ, consisting of calls to the oracle made during algorithm execution.

Our construction. We detail our construction in pseudocode in Figure 3. Enc, Dec represent authenticated symmetric encryption operations while PKEnc, PKDec represent IND-CCA secure public key encryption and decryption operations. System state is represented by *st* and is assumed to be stored persistently by the calling program.

We abstract our erasable index data structure as Tbl. We will make use of an initialize operation ($T \leftarrow \mathsf{Tbl.Init}()$), insert and lookup key operations notated by brackets ($T[k]$), and a delete key operation ($\mathsf{Tbl.Delete}(T,k)$). For all tables we assume that $T[k] = \perp$ if k is not currently in the table. Furthermore, we define a random mapping operation on a key that checks if the key is in the table, and if not, randomly samples a value of length $2n$ to store with the key, returning the stored value ($v \leftarrow\!\!\$\ \mathsf{Tbl.RandMap}(T,k)$). This operation acts to lazily construct a random function and is used in the protocol to map filenames to random values used for key derivation,

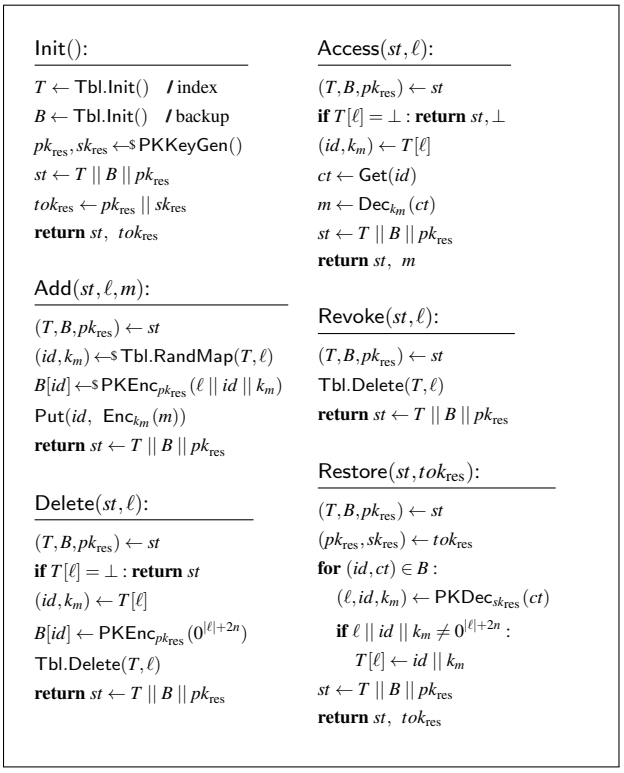

Figure 3: BurnBox algorithms for self-revocable encrypted cloud storage.

where length n corresponds to length of derived symmetric keys. To iterate over table T, the notation "$(x,y) \in T$" treats T as the set $\{(x,y) \mid T[x] = y\}$ where $x, y \neq \perp$.

5 Compelled Access Security

We formalize *compelled access security* (CAS) for SR-ECS schemes. Our treatment most closely resembles the simulation-based notions used in the symmetric searchable encryption literature [21, 23]. Our definition is parameterized by a leakage regime. One can prove security relative to a leakage regime, but the actual level of security achieved will then depend on (1) what can be learned from the leakage; and (2) how well the leakage regime abstracts the resources of a real world attacker.

To address the first concern, our cryptographic analysis (Section 5.3) will not only reduce to a leakage regime, but then also evaluate the implications of our chosen leakage regime by formally analyzing the implications of leakage using property-based security games. The second concern manifests when considering the device state leaked upon compelled access. Our abstraction necessarily dispenses with all but the cryptographic state of the SR-ECS scheme. We defer discussion of the limitations of this abstraction with respect to other device state, such as operating sys-

tem state, to Section 8.

5.1 Simulation-based Security Definition

We use two pseudocode games, shown in Figure 4. In the real game, the adversary has access to a number of oracles, which we denote by $\mathcal{A}^{\mathcal{O}}$. The adversary can adaptively make queries to an SR-ECS protocol Π using oracles **Add**, **Access**, **Delete**, **Revoke**, **Restore**. At each query, a transcript τ is returned to the adversary, representing the adversary's view of a query execution. In our setting where the storage used by the scheme is a key-value store, the transcript τ consists of tuples of the form (Put, K, V) for puts and (Get, K) for gets. Finally, the adversary may also query a **Compromise** oracle which returns the client state *st*. This models the search during compelled access.

The ideal world is parameterized by a *leakage regime* $\mathcal{L}$ and a *simulator* $\mathcal{S}$. A leakage regime $\mathcal{L} = \{\mathcal{L}_{\text{init}}, \mathcal{L}_{\text{add}}, \mathcal{L}_{\text{acc}}, \mathcal{L}_{\text{del}}, \mathcal{L}_{\text{rev}}, \mathcal{L}_{\text{res}}, \mathcal{L}_{\text{com}}\}$ consists of a sequence of leakage algorithms, one for each oracle. Each leakage algorithm takes as input a shared leakage state, $st^{\mathcal{L}}$, along with the arguments to the corresponding oracle call. The leakage algorithm acts as a filter on these inputs and returns a leakage value, σ, that is passed to the simulator. The leakage algorithm may also alter the shared leakage state, $st^{\mathcal{L}}$. The leakage regime therefore forms a kind of whitelist for what information about queries can be leaked by a scheme.

A simulator $\mathcal{S}$ attempts to use the leakage to effectively "simulate" the transcript τ and compromised state using only the leakage values σ output by $\mathcal{L}$. In other words, security is achieved if an adversary cannot tell if they are in the real world viewing the actual protocol transcript or in the ideal world viewing the transcript simulated given just the leakage. Intuitively, if the adversary view can be simulated from $\mathcal{L}$, then the adversary view in the real world reveals *no more* information than what $\mathcal{L}$ specifies.

Notice that the simulator does not get executed on **Delete**, **Restore**, and **Revoke** queries. This reflects the fact that we demand *no* leakage in response to these queries, and our scheme can achieve this because we do not interact with the cloud for these operations.

Formally, the advantage of an adaptive adversary $\mathcal{A}$ over an SR-ECS scheme Π is defined with respect to a simulator $\mathcal{S}$ and leakage function $\mathcal{L}$ by

$$\mathbf{Adv}^{\text{cas}}_{\Pi,\mathcal{S},\mathcal{L}}(\mathcal{A}) = \left|\mathbb{P}\left[\text{REAL}^{\mathcal{A},\Pi}_{\text{SR-ECS}} = 1\right] - \mathbb{P}\left[\text{IDEAL}^{\mathcal{A},\mathcal{S},\mathcal{L}}_{\text{SR-ECS}} = 1\right]\right|$$

where the probabilities are over the random coins used in the course of executing the games. We will not provide asymptotic definitions of security, but instead measure concretely the advantage of adversaries given certain running time and query budgets.

We restrict attention to adversaries that do not query **Add** on the same ℓ more than once. We believe one can relax this by changing the scheme and formalizations to handle sets of values associated to filename labels.

Ideal encryption model. Looking ahead, we will prove security in an *ideal encryption model* (IEM) which is an idealized abstraction of symmetric encryption. In the IEM model, the real world is augmented with two additional oracles, an encryption oracle **Encrypt** and a decryption oracle **Decrypt**. The former allows queries on an arbitrary symmetric key k and message m, and returns a random bit string ct of the appropriate length. We let clen be a function of the message length $|m|$ to an integer that represents the length in bits of the ciphertext. The oracle also stores m in a table indexed by $k \,||\, ct$. The oracle **Decrypt** can be queried on a key k and ciphertext string ct, and it returns the table entry at $k \,||\, ct$. We assume all table entries that are not set have initial value $\perp$. The adversary can make queries to **Encrypt**, **Decrypt** at any point in the games, including after the **Compromise** query is made.

In the ideal world the **Encrypt** and **Decrypt** oracles are implemented by the simulator $\mathcal{S}$. This means, importantly, that they can "program" the encryption, which seems necessary in our context since we require non-committing encryption [19]; the simulator must commit to an encryption of a message on **Add** before learning the contents of the message on **Compromise**. It is known that one requires programmability to achieve non-committing encryption (when secret keys are short) [51].

The IEM model can be viewed as a lifting of the ideal cipher model (ICM) or random oracle model (ROM) [14] to randomized authenticated encryption. Formally, one can replace ideal encryption with an indifferentiable authenticated-encryption scheme [12], applying the composition theorem of [45]. Those schemes are, however, not as efficient as standard ones, and we conjecture that one can directly prove our CAS scheme secure using standard authenticated encryption schemes while modeling their underlying components as ideal ciphers and/or random oracles.

5.2 Pseudonymous Operation History Leakage

We now introduce the leakage regime we will target, what we call the pseudonymous operation history leakage regime, denoted $\mathcal{L}^{\text{POH}}$. See Figure 5 for pseudocode.

Simply put, the leakage algorithms of $\mathcal{L}^{\text{POH}}$ reveal the operation name along with a pseudonym identifier for the operation target. For example, on a call to the add leakage algorithm, $\mathcal{L}_{\text{add}}(st^{\mathcal{L}}, \ell, m)$, a new random pseudonym p is sampled (without replacement) and returned along with the operation name, specifying an Add

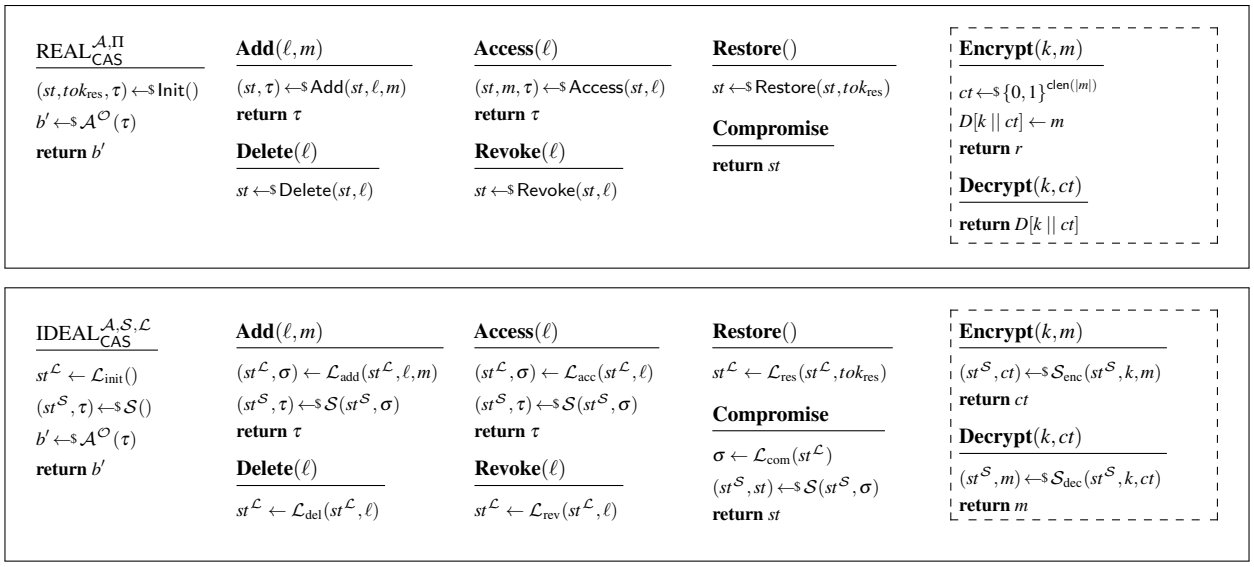

Figure 4: Games used in defining CAS security. The adversary has access to oracles $\mathcal{O} = \{$**Add**, **Access**, **Delete**, **Revoke**, **Restore**, **Compromise**, **Encrypt**, **Decrypt**$\}$ and is tasked with distinguishing between the "real"world and the simulated "ideal" world.

has occurred ($\sigma = (\mathsf{Add}, p, \mathsf{clen})$). The length of the content is also leaked upon Add. The pseudonym is saved within $st^{\mathcal{L}}$, so that on future operations involving that file, e.g., Access, the same pseudonym can be returned. Note that in the pseudonymous operation history neither the filename ℓ nor the file contents m are leaked.

The compromise leakage algorithm, $\mathcal{L}_{\text{com}}$, leaks pseudonyms of all currently available files along with their associated label and contents. Operations that do not interact with the remote server, $\mathcal{L}_{\text{del}}, \mathcal{L}_{\text{rev}}, \mathcal{L}_{\text{res}}$, do not leak anything when first called, but do update the leakage state to change the set of files that are leaked upon compromise.

Pseudonymous operation history leakage fits the SR-ECS setting with an adversary-controlled remote server processing Add and Access operations for individual files. The adversary may not learn the underlying contents or file name, but can trivially link the upload of a file ciphertext to when it is served back to the client. While techniques that add, access, and permute batches of messages can attempt to obscure these links, e.g. ORAM [53], they remain impractical in the near term. We discuss implications of access pattern leakage in Section 8.

5.3 Cryptographic Security Analysis

There are two steps to our formal cryptographic security analysis. First, we show that our protocol is secure with respect to the pseudonymous operation history leakage regime $\mathcal{L}^{\text{POH}}$, by presenting a simulator $\mathcal{S}^{\text{POH}}$ (see Figure 6) that can effectively emulate the real world protocol given only access to the leakage in the ideal world. For simplicity, we define operation-specific simulators, $\mathcal{S}^{\text{POH}} = \{\mathcal{S}_{\text{add}}, \mathcal{S}_{\text{acc}}, \mathcal{S}_{\text{com}}, \mathcal{S}_{\text{enc}}, \mathcal{S}_{\text{dec}}\}$, which are invoked based on the leakage from $\mathcal{L}^{\text{POH}}$. The simulator $\mathcal{S}^{\text{POH}}$ uses programmabililty of the ideal encryption oracles, which it simulates.

This simulation-based security can be thought of as a whitelist which specifies what is revealed through the leakage regime. In many ways, this approach is desirable, as it does not require the prover to defend against specific attacks. However, complex models lead to complex leakage regimes in which the interactions between leakage algorithms can be unintuitive. In the worst case, proving simulation-based security would lead to a false sense of confidence should leakage suffice to violate security in ways explicitly targeted by scheme designers.

We therefore complement simulation-based security analysis with formalization of, and analyses of our scheme under, two relevant property-based security games. As we will see, these results end up being straightforward corollaries of the more general leakage-based security, which provides evidence that our leakage regime suffices to guarantee important security properties.

Main security result. The following theorem proves CAS security of our scheme Π (as shown in Figure 3). It upper bounds the advantage of any adversary against the scheme by the advantage of adversaries against $\mathsf{INDCPA}_{\mathsf{PKE}}$ of the underlying components, plus a birthday-bound term associated to the probability of collisions occurring in identifiers or the success of a

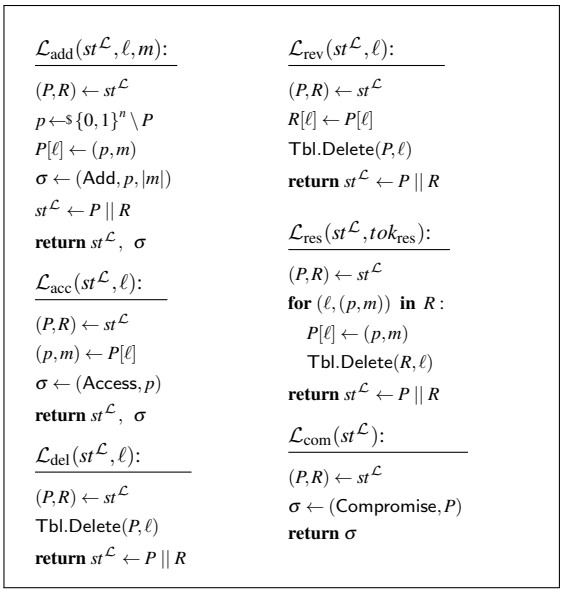

Figure 5: Leakage algorithms defining the pseudonymous operation history leakage, $\mathcal{L}^{\text{POH}}$. Table P tracks undeleted file pseudonyms and R tracks revoked file pseudonyms.

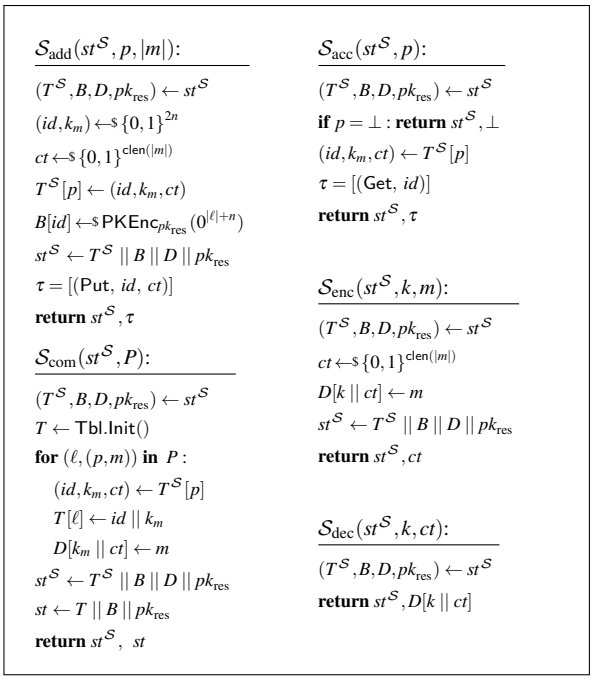

Figure 6: The simulator for the pseudonymous operation history leakage regime $\mathcal{S}^{\text{POH}}$ used in the proof of Theorem 1. Table $T^{\mathcal{S}}$ stores added file pseudonyms and committed ciphertexts, B stores restoration ciphertexts, and D is used for ideal encryption.

brute-force key recovery attack against the ideal encryption. The full proof and description of the (standard) $\mathsf{INDCPA}_{\mathsf{PKE}}$ security game are given in our extended technical report [67].

Theorem 1. *Let $\mathcal{A}$ be a CAS adversary for protocol Π and leakage regime $\mathcal{L}^{POH}$. Let $\mathcal{S}^{POH}$ be the simulator defined in Figure 6. Then we give adversary $\mathcal{B}$ such that if $\mathcal{A}$ makes at most q_{Add}, q_{Enc}, q_{Dec} queries to* **Add**, **Encrypt**, **Decrypt**, *respectively, and runs in time T then*

$$\mathbf{Adv}^{cas}_{\Pi,\mathcal{S}^{POH},\mathcal{L}^{POH}}(\mathcal{A}) \leq \mathbf{Adv}^{indcpa}_{PKE}(\mathcal{B}) + \frac{q_{\mathsf{Add}} \cdot (2q_{\mathsf{Add}} + q_{\mathsf{Dec}})}{2^n}$$

where n is the length of identifiers and symmetric keys. Moreover, $\mathcal{B}$ runs in time $T' \approx T$ and makes at most q_{Add} queries to its oracle.

Above when we say that $T' \approx T$, we mean that those adversaries run in time that of $\mathcal{A}$ plus the (small) overhead required to simulate oracle queries. A more granular accounting can be derived from the proof. Here we just briefly sketch the analysis.

Proof Sketch. We can divide the simulator's role in two: simulating the cloud transcript (on **Add** and **Access**) and simulating the client state (on **Compromise**). To simulate the cloud transcript in **Add**, the simulator must commit to a random ciphertext for file contents that are not known. To simulate client state, the simulator must provide (1) restoration ciphertexts and (2) keys and file contents that are consistent with the ciphertexts to which the simulator previously committed. The first step is a straightforward reduction to the INDCPA security of PKE. The second step is more challenging. In the IEM, the simulator can "program" the **Encrypt** and **Decrypt** responses to match the previously committed-to ciphertexts once file contents are leaked in **Compromise**. However, prior to compromise, it is possible for the adversary to brute-force decrypt ciphertexts by querying the ideal encryption oracles which, if successful, will catch the simulator in its attempt at programming. But we can show this probability is small, at most $q_{\mathsf{Add}} q_{\mathsf{Dec}}/2^n$ because the adversary has no information about these keys. The remaining part of the bound, $2q^2_{\mathsf{Add}}/2^n$, accounts for the need in the proof to switch identifiers to being chosen without replacement and then back again.

Property-based security. Recall two security goals for BurnBox in the compelled access threat model: (1) file name/content privacy — the content and name of deleted or revoked files should be hidden upon compromise; and (2) file revocation obliviousness — temporarily revoked files should be indistinguishable from securely deleted files upon compromise. We formalize these goals as adaptive security games $\mathsf{FilePrivacy}^{\mathcal{A},b}_{\Pi}$ and

$\mathsf{DelRevOblivious}_{\Pi}^{\mathcal{A},b}$ and give the following two corollaries of Theorem 1. The full description of the security games including advantage definitions and proof sketches are given in our extended technical report [67].

Corollary 2. *Let $\mathcal{A}$ be a FilePrivacy adversary for SR-ECS protocol Π. Then we give an adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}_{\Pi}^{\mathsf{FilePrivacy}}(\mathcal{A}) \leq 2 \cdot \mathbf{Adv}_{\Pi,\mathcal{S}^{POH},\mathcal{L}^{POH}}^{cas}(\mathcal{B})$$

where if $\mathcal{A}$ runs in time T and makes at most q oracle queries, $\mathcal{B}$ runs in time $T' \approx T$ and makes at most q queries to the CAS oracle defined in Figure 4.

Corollary 3. *Let $\mathcal{A}$ be a DelRevOblivious adversary for SR-ECS protocol Π. Then we give an adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}_{\Pi}^{\mathsf{DelRevOblivious}}(\mathcal{A}) \leq 2 \cdot \mathbf{Adv}_{\Pi,\mathcal{S}^{POH},\mathcal{L}^{POH}}^{cas}(\mathcal{B})$$

where if $\mathcal{A}$ runs in time T and makes at most q oracle queries, $\mathcal{B}$ runs in time $T' \approx T$ and makes at most q queries to the CAS oracle defined in Figure 4.

6 Implementation

We design and implement a prototype of BurnBox in C++ suitable for use on commodity operating systems. The system architecture is depicted in Figure 7. The prototype consists of 3,373 lines of code. The core cryptographic functionality is exposed through a file system in userspace (FUSE) [8] that can be deployed as a SR-ECS scheme by mounting it within a cloud synchronization directory, e.g., Dropbox. Add, Access, and Delete algorithms are captured and handled transparently via the file system write, read, and delete interfaces. Revoke and Restore are implemented as special FUSE commands and can be invoked through either the file system user interface or a command-line interface.

BurnBox maintains local state in an erasable index (Section 3) which stores filenames, file keys, and restoration ciphertexts. From the Crypto++ library [6], we use AES-GCM with 128-bit keys for encryption of file contents and of the erasable index key tree. We use ECIES [64] with secp256r1 for public key encryption of restoration keys. The implementation is available open source at `https://github.com/mhmughees/burnbox`.

Effaceable storage. As discussed in Section 4, to construct the erasable index, we require some mechanism that can securely store and delete symmetric keys. Both iOS [3] and Android [1] provide keystore APIs that, when backed by hardware security elements, provide this functionality. On desktops, there are no built-in mechanisms for doing so, but the functionality can be constructed from, for example, SGX [2]. For our prototype, we leverage the functionality provided by a trusted platform module (TPM) [66], and test it using IBM's software TPM [7].

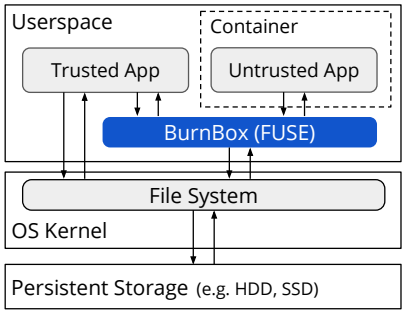

Figure 7: BurnBox is implemented as a file system in userspace (FUSE). Trusted applications that are known not to leak file information about files can interact freely with BurnBox and the rest of the file system. Untrusted applications can be run in a container with access to BurnBox and a temporary file system that can be wiped on application exit.

It is possible to use BurnBox without hardware support for secure storage of the master key of our encryption tree. In this case, the master key is stored in persistent storage. This, of course, is insecure in the threat model where hardware forensics can recover past writes to persistent storage, e.g., a previous master key and key tree pair can be recovered to learn the key material for deleted files.

Operating system leakage. BurnBox is designed specifically to address leakage from persistent storage. To restrict an adversary to this scenario, BurnBox is implemented using memory-locked pages when appropriate and prompts users to restart their device following deletes/revokes prior to compelled access. This approach eliminates many issues such as kernel state and in-memory remnants of data, however, it is not a complete solution; BurnBox is not the only program that can write to disk. Both the operating system and applications can persist data that, although outside of BurnBox's control, will expose what it wishes to hide (e.g., through recently-used lists, search indices, buffers, etc.). We discuss these limitations further in Section 8.

Application support. Our prototype provides two ways for applications to use files stored in BurnBox. Trusted apps can obtain direct access to the BurnBox file system. These apps should be carefully vetted to ensure they do not leak damaging information about deleted or revoked files, e.g., by saving temporary data to other portions of the file system. Obviously such vetting is highly non-trivial, and so our prototype also allows a sandboxing

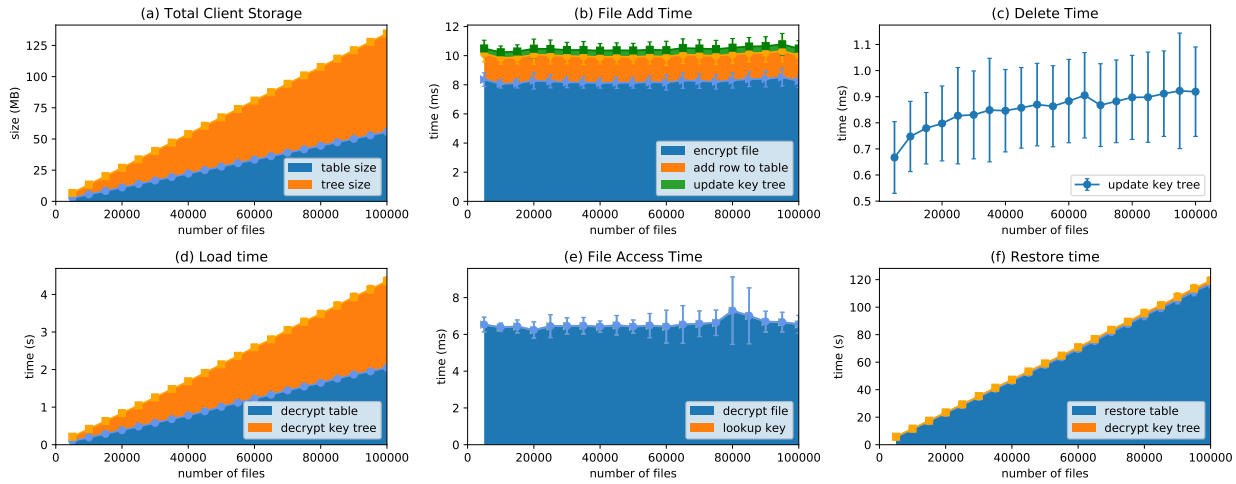

Figure 8: Evaluation of the storage and latency overheads imposed by BurnBox with respect to the number of files stored. Operation costs are plotted broken down into constituent parts and stacked to make up the total cost.

mechanism for untrusted applications. In particular, we allow running an application within a Docker container given access to BurnBox and a temporary file system that is wiped on application exit. For the latter we use a ramdisk [41].

7 Evaluation

As with a standard encrypted cloud store, the time to add and read files is primarily a function of client bandwidth and file length. BurnBox adds storage and timing overhead on top of these costs in order to maintain an erasable index and support revocation/restoration. Our evaluation answers the following questions:

(1) What is the storage overhead imposed by BurnBox on the client and cloud server?

(2) What are the latency overheads of BurnBox operations and how are they affected by the number of files (i.e. size of erasable index)?

Experimental setup. To answer the questions above, we run a series of experiments on a 2.2 GHz Intel core i7 Haswell processor with 16GB of RAM. We use a constant file size of 1 MB. File size affects the time to encrypt and decrypt files, but is a shared cost of all encrypted cloud storage schemes. We focus on measuring the additional overhead BurnBox incurs, such as maintaining the erasable index, which is not dependent on file size. In our experiments, we do not mount BurnBox within a cloud sync directory. Thus our measurements capture cryptographic and I/O costs, but not the additional network costs that would be present in a cloud setting.

Storage overhead. The erasable index on the client stores a filename (16 B), key-value store key (16 B), symmetric key (16 B), and restoration ciphertext (305 B) for each file. The key tree, whose leaves are used to encrypt individual rows of the index, grows linearly in the total number of files with new branches generated lazily. As expected, total client storage, consisting of the key tree and the encrypted rows, increases linearly with the number of files (Figure 8). This amounts to a reasonable client overhead for most use cases. For example, a device can store 10^5 files in BurnBox while incurring less than 80 MB of local storage overhead. Note that the number of files includes deleted, revoked, and active files. In order to store the restoration ciphertext, revoked files incur almost the same storage overhead as active files; and thus, to achieve deletion-revocation obliviousness, deleted files also incur the same storage overhead. Finally, there is no storage overhead for the cloud server on top of the cost of the encrypted file contents.

Operation latency. Before any operation can be performed, our design requires reading the entire erasable index (i.e., filename to key mappings) into memory. Ideally, only the relevant row corresponding to the filename specified by each operation would be loaded. However, recall in order to prevent leakage of filename information from storage patterns, the index is not ordered by filename. This makes efficient direct row level accesses to the persisted index based on filename impossible. As a result, the start-up cost is linear in the number of files (in-order traversal of the key tree and decryption of each row). Nevertheless it is not prohibitively large, e.g., requiring 4.2 seconds for 10^5 files (Figure 8), since, once loaded, the index can be stored in memory using a fast

data structure, e.g., a hash table.

Next we turn to evaluating the latency of each operation. Delete and Revoke operations simply update a row of the erasable index. Updating a row consists of sampling a new key to encrypt the row and updating the keys in the key tree path. Figure 8 shows the expected logarithmic relationship with number of files (i.e., height of key tree) and is independent of the size of files. The Add operation consists of the standard file encryption cost along with the overhead of an erasable index row update (Figure 8). The file encryption cost shown here is constant since our experiments add files of constant size (1 MB), but in general this cost will depend linearly on the size of the file. We see that the majority of the cost is from file encryption and overhead is small ($< 20\%$). The Access operation does not modify the erasable index and consists only of the file decryption cost. The Restore operation decrypts all restoration ciphertexts and updates the leaves of the key tree, executing in time linear to the number of files (Figure 8). The bulk of the cost in Restore comes from the public key decryption of a restoration ciphertext for each file ($\sim$ 1 ms / decryption).

8 Limitations

Access pattern inference. BurnBox does not hide access patterns for files stored in the cloud. In other contexts such as searchable encryption, access pattern leakage has been known to allow attacks that recover plaintext information [32, 33, 49] given some information about the underlying encrypted documents. The success of these types of attacks have so far been limited to recovering information of highly structured data types, such as columns of first names or social security numbers. It remains to be seen in what contexts attacks exist for a space as large and unstructured as files. While these issues are independent of BurnBox and instead stem from the general use of cloud storage, we consider if compelled access presents a unique problem for access pattern attacks.

By learning the plaintexts of undeleted files upon compelled access, the adversary may be able to better model the access distribution for a particular user leading to a stronger inference attack. Certainly if accesses between known plaintexts and unknown plaintexts can be correlated this would lend a strong advantage to the adversary (e.g., a set of files is known to be accessed in quick succession; if a few of the files are revealed, it can be inferred that the other deleted files accessed in succession belong to the set). However, should sensitive revoked files have little correlation with unrevoked files, the adversary will not be able to exploit the revealed files in this way.

Another consideration for leakage is file name length and file size which, for example, might uniquely identify files. Names can be padded to a maximum length with little loss as most file systems only allow 255 character names. File sizes are more challenging. If BurnBox is used with files where sizes are unique, these sizes should be padded. The granularity of such padding is dependent on the distribution of file lengths.

One final note is that access patterns after a compromise can reveal whether files were deleted or just revoked, because deleted files will never be read from or written to again. While we can preserve obliviousness during a compelled access search, access to the file after the search will inform the adversary if they are monitoring the cloud store. This appears to be unavoidable without resorting to, e.g., oblivious RAM [53], and even then the volume of accesses would leak some information.

Operating system leakage. BurnBox is designed to limit leakage from persistent storage following device restart in the compelled access threat model. While we have formally evaluated the security of BurnBox with respect to its cryptographic state, a complete picture of BurnBox usage includes the underlying operating system and interacting applications; both can access sensitive data and write to persistent storage. These other vectors of leakage have long been identified as a challenge for systems with similar goals to BurnBox, e.g., in deniable file systems [24].

Such concerns include: recently used file lists; indexes for OS wide search; application screen shots used for transitions[1]; file contents from BurnBox memory being paged to disk; text inputs stored either in keyboard buffers or predictive typing mechanisms; byproducts of rendering and displaying files to the user; and the volume and timing of disk operations.

Some of these issues can be handled by configuration or user action. Disabling OS-wide search and indexing for BurnBox directories prevents file names and contents from being stored in those indexes. To guard against leakage from memory being paged to disk, BurnBox uses memory locked pages where available. Users can avoid leaving applications with access to sensitive data open, which reduces the risk of leakage on suspend or resume. These approaches are somewhat unsatisfying because they require user-specific actions or at least OS-wide configuration changes (that perhaps can be handled by an installer).

BurnBox is necessary, but not sufficient, to fully protect against these issues and must be part of a larger ecosystem of techniques to achieve complete security. Applications need to take steps to prevent leakage. In some cases, as in our prototype, it may be as simple as running the application within a container with access only to a temporary file system that is erased on application exit. At the operating system level, special virtualization techniques [26], pur-

[1] Many operating systems use screen shots of the user interface when resuming either suspended applications or the OS itself.

pose built file systems [11], and write-only ORAM [59] can address many leakage issues.

Delete timing. A particular issue related to operating system leakage is revelation of timing and volume of disk accesses to forensics tools. In addition to hiding whether a file's status is revoked or deleted, BurnBox targets hiding when the status changed (deletion/revocation timing privacy). To this end, it stores all cryptographic material in two monolithic files. As a result an adversary examining timestamps learns the time of the last operation in BurnBox but nothing about the timing or volume of preceding operations or what they were.

However, the file system itself, or even the underlying physical storage medium, may leak more granular information. A journaling file system might, for example, leak when an individual entry in the erasable index was last touched. While we have carefully designed BurnBox to ensure this reveals no addition information, it does by necessity reveal when the file's status changed. Even if such fine grained information is not available, a flurry of file system activity, regardless of if it can be directly associated BurnBox, might suggest a user was revoking or deleting files immediately prior to a search, raising suspicion.

Even should such operating-system leakage reveal timing, BurnBox may provide value in terms of delete timing privacy for attackers who do not conduct low level disk forensics. We note that if one ignores the secondary goal of delete/revocation timing privacy, one could modify BurnBox to have the erasable index client state outsourced to cloud storage. Then Delete and Revoke operations would involve interactions with the cloud (revealing timing trivially), but this would arguably simplify the design.

Deleting files from the cloud. A final limitation is that BurnBox, as described, never requests the cloud storage service to delete files. This is necessary to provide deletion/revocation obliviousness. However, at some juncture it will be necessary to free up storage space and this may enable a compelled-access adversary to at that point identify that a user previously revoked files. A user might therefore do such deletions well after the compelled access search, but since it leaks information to the adversary its timing should be considered carefully.

9 Related Work

A variety of works have looked at related problems surrounding compelled access, secure erasure, and encrypted cloud storage.

Secure deletion. The problem of secure deletion for files has been explored extensively in various contexts [25, 28, 56]. These works can be divided into two distinct approaches, data overwriting [36, 69] and cryptographic erasure [16, 22, 57]. Data overwriting is not applicable to a corrupted cloud storage provider who stores snapshots. Cryptographic erasure alone doesn't provide temporary revocation. Neither approach directly solves the issue of metadata needed to locate files (in our case file names).

History independence. A line of work has examined history independent data structures [31, 47, 48] and (local) file systems [11]. As we discussion in Section 3, however, these techniques do not work when confronted with adversaries who can forensically recover fine grained past file system state, rather they ensure only that the current state is independent of its history. While the use of a history-independent file system for local storage [11] could be used to augment BurnBox to improve its ability to hide access patterns (during a forensic analysis), it does not alone suffice for the compelled access scenario as it does not protect cloud data or provide for self-revocation.

Decoy-based approaches. Several works target tricking adversaries via decoy content, revealed by providing a fake password. Deniable encryption [18, 20, 61] targets public key encrypted messages which can later be opened to some decoy message. Gasti et al. [29] use deniable public-key encryption to build a cloud-backed file system These approaches do not hide file names or provide for self-revocation, and they require choosing a decoy message at file creation time.

Honey encryption [35, 37, 38] targets ensuring decryption under wrong passwords results in decoy plaintexts, but only works for a priori known distributions of plaintext data, making it unsuitable for general use. We target CAS-secure encryption for arbitrary data.

Deniable file systems [9, 29, 34, 52, 55], also known as steganographic file systems [9], support a hidden volume that is concealed from the adversary and a decoy volume that is unlocked via a fake password. Deniable file systems require users either to a priori compartmentalize their life into a deniable and non-deniable partition or to create and maintain plausible "dummy" data for the decoy volume while conducting everything in the hidden volume. In contrast, we require users simply excise what they want to hide when compelled access is likely.

At a higher level, all decoy-based systems require the user to lie to the authority and intentionally reveal the wrong password (or cryptographic secret). In addition to requiring the user to actively not comply, lying may have legal implications in some cases. Our approach is different and does not depend on prearranged decoy content or lying.

Capture-resilient devices. A series of works [42, 43] investigated capture-resilient devices, where one uses a remote server to help encrypt data on the device so that if the device is captured, offline dictionary attacks against

user passwords does not suffice to break security. These settings, and similar, assume the user does not disclose their password, thus making it insufficient for the compelled access threat model we target here.

10 Conclusion

In this paper we explored the setting of compelled access, where physically present authorities force a user to disclose secrets in order to allow a search of their digital devices. We introduced the notion of self-revocable encryption, in which the user can, ahead of a potential search (e.g., before crossing a national border), revoke their ability to access sensitive data. We explored this approach in the context of encrypted cloud storage applications, showing that one can hide not only file contents but also whether and which files were revoked.

We detailed a new cryptographic security notion, called compelled access security, to capture the level of access pattern leakage a scheme admits. We introduced a scheme for which we can formally analyze compelled access security relative to a reasonable leakage regime. Interestingly, the analysis requires non-committing encryption.

We report on an initial prototype of the resulting tool, called BurnBox. While it has various limitations due primarily to operating system and application leakage, BurnBox provides a foundation for realizing client devices that resist compelled access searches.

Acknowledgments

This work was supported in part by Nirvan Tyagi's NSF Graduate Research Fellowship, NSF grants 1558500, 1514163, and 1330308, and a generous gift from Microsoft.

References

[1] Android keystore system. `https://developer.android.com/training/articles/keystore.html`.

[2] Intel Software Guard Extensions (Intel SGX). `https://software.intel.com/en-us/sgx`.

[3] Storing keys in the secure enclave. `https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/storing_keys_in_the_secure_enclave`.

[4] Truecrypt. `http://truecrypt.sourceforge.net/`, 2014.

[5] Cbp releases updated border search of electronic device directive and fy17 statistics. `https://www.cbp.gov/newsroom/national-media-release/cbp-releases-updated-border`, 1 2018.

[6] Crypto++ library. `https://www.cryptopp.com/`, 2018.

[7] Ibm software tpm. `http://ibmswtpm.sourceforge.net/`, 2018.

[8] Libfuse: Filesystem in userspace. `https://github.com/libfuse/libfuse`, 2018.

[9] ANDERSON, R. J., NEEDHAM, R. M., AND SHAMIR, A. The steganographic file system. In *Information Hiding, Second International Workshop, Portland, Oregon, USA, April 14-17, 1998, Proceedings* (1998), pp. 73–82.

[10] ASSANGE, J., DREYFUS, S., AND WEINMANN, R. Rubberhose, 1997. `https://web.archive.org/web/20100915130330/http://iq.org/~proff/rubberhose.org/`.

[11] BAJAJ, S., AND SION, R. HIFS: history independence for file systems. In *ACM Conference on Computer and Communications Security* (2013), ACM, pp. 1285–1296.

[12] BARBOSA, M., AND FARSHIM, P. Indifferentiable authenticated encryption. In *Advances in Cryptology – CRYPTO 2018* (2018).

[13] BELLARE, M., AND O'NEILL, A. Semantically-secure functional encryption: Possibility results, impossibility results and the quest for a general definition. In *Cryptology and Network Security - 12th International Conference, CANS 2013, Paraty, Brazil, November 20-22. 2013. Proceedings* (2013), pp. 218–234.

[14] BELLARE, M., AND ROGAWAY, P. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.* (1993), pp. 62–73.

[15] BONEH, D., LEWI, K., AND WU, D. J. Constraining pseudorandom functions privately. *IACR Cryptology ePrint Archive 2015* (2015), 1167.

[16] BONEH, D., AND LIPTON, R. J. A revocable backup system. In *Proceedings of the 6th USENIX Security Symposium, San Jose, CA, USA, July 22-25, 1996* (1996).

[17] BURGE, C., AND CHIN, J. Twelve days in Xinjiang: How China's surveillance state overwhelms daily life. `https://www.wsj.com/articles/twelve-days-in-xinjiang`, Dec 2017.

[18] CANETTI, R., DWORK, C., NAOR, M., AND OSTROVSKY, R. Deniable encryption. In *CRYPTO* (1997), vol. 1294 of *Lecture Notes in Computer Science*, Springer, pp. 90–104.

[19] CANETTI, R., FEIGE, U., GOLDREICH, O., AND NAOR, M. Adaptively secure multi-party computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996* (1996), pp. 639–648.

[20] CARO, A. D., IOVINO, V., AND O'NEILL, A. Deniable functional encryption. In *Public Key Cryptography (1)* (2016), vol. 9614 of *Lecture Notes in Computer Science*, Springer, pp. 196–222.

[21] CASH, D., JAEGER, J., JARECKI, S., JUTLA, C. S., KRAWCZYK, H., ROSU, M., AND STEINER, M. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014* (2014).

[22] CRESCENZO, G. D., FERGUSON, N., IMPAGLIAZZO, R., AND JAKOBSSON, M. How to forget a secret. In *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings* (1999), pp. 500–509.

[23] CURTMOLA, R., GARAY, J. A., KAMARA, S., AND OSTROVSKY, R. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security* (2006), ACM, pp. 79–88.

[24] CZESKIS, A., HILAIRE, D. J. S., KOSCHER, K., GRIBBLE, S. D., KOHNO, T., AND SCHNEIER, B. Defeating encrypted and deniable file systems: Truecrypt v5.1a and the case of the

tattling OS and applications. In *3rd USENIX Workshop on Hot Topics in Security, HotSec'08, San Jose, CA, USA, July 29, 2008, Proceedings* (2008).

[25] DIESBURG, S. M., AND WANG, A. A. A survey of confidential data storage and deletion methods. *ACM Comput. Surv. 43*, 1 (2010), 2:1–2:37.

[26] DUNN, A. M., LEE, M. Z., JANA, S., KIM, S., SILBERSTEIN, M., XU, Y., SHMATIKOV, V., AND WITCHEL, E. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012* (2012), pp. 61–75.

[27] FOX-BREWSTER, T. Feds have found a way to search locked phones of 100 trump protestors. `https://www.forbes.com/sites/thomasbrewster/2017/03/23/`.

[28] GARFINKEL, S. L., AND SHELAT, A. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security & Privacy 1*, 1 (2003), 17–27.

[29] GASTI, P., ATENIESE, G., AND BLANTON, M. Deniable cloud storage: sharing files via public-key deniability. In *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society* (2010), ACM, pp. 31–42.

[30] GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. How to construct random functions. *J. ACM 33*, 4 (1986), 792–807.

[31] GOODRICH, M. T., KORNAROPOULOS, E. M., MITZENMACHER, M., AND TAMASSIA, R. More practical and secure history-independent hash tables. In *ESORICS (2)* (2016), vol. 9879 of *Lecture Notes in Computer Science*, Springer, pp. 20–38.

[32] GRUBBS, P., MCPHERSON, R., NAVEED, M., RISTENPART, T., AND SHMATIKOV, V. Breaking web applications built on top of encrypted data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016* (2016), pp. 1353–1364.

[33] GRUBBS, P., SEKNIQI, K., BINDSCHAEDLER, V., NAVEED, M., AND RISTENPART, T. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017* (2017), pp. 655–672.

[34] HAN, J., PAN, M., GAO, D., AND PANG, H. A multi-user steganographic file system on untrusted shared storage. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010), ACM, pp. 317–326.

[35] JAEGER, J., RISTENPART, T., AND TANG, Q. Honey encryption beyond message recovery security. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I* (2016), M. Fischlin and J. Coron, Eds., vol. 9665 of *Lecture Notes in Computer Science*, Springer, pp. 758–788.

[36] JOUKOV, N., AND ZADOK, E. Adding secure deletion to your favorite file system. In *3rd International IEEE Security in Storage Workshop (SISW 2005), December 13, 2005, San Francisco, California, USA* (2005), pp. 63–70.

[37] JUELS, A., AND RISTENPART, T. Honey encryption: Encryption beyond the brute-force barrier. *IEEE Security & Privacy 12*, 4 (2014), 59–62.

[38] JUELS, A., AND RISTENPART, T. Honey encryption: Security beyond the brute-force bound. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings* (2014), pp. 293–310.

[39] KING, R. FBI, NSA said to be secretly mining data from nine U.S. tech giants. `http://www.zdnet.com/article/fbi-nsa-said-to-be-secretly-mining-data`.

[40] LABS, P. Filecoin: A decentralized storage network, 14 Aug. 2017.

[41] LANDLEY, R. ramfs, rootfs and initramfs. `https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt`, 2018.

[42] MACKENZIE, P. D., AND REITER, M. K. Delegation of cryptographic servers for capture-resilient devices. In *ACM Conference on Computer and Communications Security* (2001), ACM, pp. 10–19.

[43] MACKENZIE, P. D., AND REITER, M. K. Networked cryptographic devices resilient to capture. *Int. J. Inf. Sec. 2*, 1 (2003), 1–20.

[44] MAIDSAFE.NET. MaidSafe.net announces project SAFE to the community (v1.4), 14 Apr. 2014.

[45] MAURER, U., RENNER, R., AND HOLENSTEIN, C. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *Theory of cryptography conference* (2004), Springer, pp. 21–39.

[46] MCDONALD, A. D., AND KUHN, M. G. StegFS: A steganographic file system for linux. In *International Workshop on Information Hiding* (1999), Springer, pp. 463–477.

[47] MOLNAR, D., KOHNO, T., SASTRY, N., AND WAGNER, D. A. Tamper-evident, history-independent, subliminal-free data structures on PROM storage-or-how to store ballots on a voting machine (extended abstract). In *IEEE Symposium on Security and Privacy* (2006), IEEE Computer Society, pp. 365–370.

[48] NAOR, M., AND TEAGUE, V. Anti-persistence: History independent data structures. *IACR Cryptology ePrint Archive 2001* (2001), 36.

[49] NAVEED, M., KAMARA, S., AND WRIGHT, C. V. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015* (2015), pp. 644–655.

[50] NICHOLS, S. Dropbox: Oops, yeah, we didn't actually delete all your files this bug kept them in the cloud. `https://www.theregister.co.uk/2017/01/24/dropbox_brings_old_files_back_from_dead/`, January 2017.

[51] NIELSEN, J. B. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *Annual International Cryptology Conference* (2002), Springer, pp. 111–126.

[52] OLER, B., AND FRAY, I. E. Deniable file system–application of deniable storage to protection of private keys. In *6th International Conference on Computer Information Systems and Industrial Management Applications, CISIM 2007, Elk, Poland, June 28-30, 2007* (2007), pp. 225–229.

[53] OSTROVSKY, R. *Software protection and simulation on oblivious RAMs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1992.

[54] PANJWANI, S. Tackling adaptive corruptions in multicast encryption protocols. In *Theory of Cryptography Conference* (2007), Springer, pp. 21–40.

[55] PETERS, T., GONDREE, M. A., AND PETERSON, Z. N. J. DEFY: A deniable, encrypted file system for log-structured storage. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015* (2015).

[56] REARDON, J., BASIN, D. A., AND CAPKUN, S. Sok: Secure data deletion. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (2013), pp. 301–315.

[57] REARDON, J., CAPKUN, S., AND BASIN, D. A. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012* (2012), pp. 333–348.

[58] REARDON, J., RITZDORF, H., BASIN, D. A., AND CAPKUN, S. Secure data deletion from persistent media. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), pp. 271–284.

[59] ROCHE, D. S., AVIV, A. J., CHOI, S. G., AND MAYBERRY, T. Deterministic, stash-free write-only ORAM. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), pp. 507–521.

[60] RUOTI, S., ANDERSEN, J., ZAPPALA, D., AND SEAMONS, K. Why Johnny still, still can't encrypt: Evaluating the usability of a modern PGP client. *arXiv preprint arXiv:1510.08555* (2015).

[61] SAHAI, A., AND WATERS, B. How to use indistinguishability obfuscation: deniable encryption, and more. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014* (2014), pp. 475–484.

[62] SAVAGE, C., AND NIXON, R. Privacy complaints mount over phone searches at U.S. border since 2011. `https://www.nytimes.com/2017/12/22/us/politics/us-border-privacy-phone-searches.html`, 12 2017.

[63] SHENG, S., BRODERICK, L., KORANDA, C. A., AND HYLAND, J. J. Why Johnny still can't encrypt: Evaluating the usability of email encryption software. In *Symposium On Usable Privacy and Security* (2006), pp. 3–4.

[64] SHOUP, V. A proposal for an ISO standard for public key encryption. *IACR Cryptology ePrint Archive 2001* (2001), 112.

[65] SKILLEN, A., AND MANNAN, M. Mobiflage: Deniable storage encryption for mobile devices. *IEEE Transactions on Dependable and Secure Computing 11*, 3 (2014), 224–237.

[66] SUMRALL, N., AND NOVOA, M. Trusted computing group (tcg) and the tpm 1.2 specification. In *Intel Developer Forum* (2003), vol. 32.

[67] TYAGI, N., MUGHEES, M. H., RISTENPART, T., AND MIERS, I. Burnbox: Self-revocable encryption in a world of compelled access. Cryptology ePrint Archive, Report 2018/638, 2018. `https://eprint.iacr.org/2018/638`.

[68] VORICK, D., AND CHAMPINE, L. Sia: Simple decentralized storage. `https://sia.tech/sia.pdf`, 29 Nov. 2014.

[69] WEI, M. Y. C., GRUPP, L. M., SPADA, F. E., AND SWANSON, S. Reliably erasing data from flash-based solid state drives. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011* (2011), pp. 105–117.

[70] WHITTEN, A., AND TYGAR, J. D. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security Symposium* (1999), vol. 348.

[71] WILKINSON, S., BOSHEVSKI, T., BRANDOFF, J., PRESTWICH, J., HALL, G., GERBES, P., HUTCHINS, P., POLLARD, C., AND BUTERIN, V. Storj: A peer-to-peer cloud storage network (v2.0), 15 Dec. 2016.

[72] WOLCHOK, S., HOFMANN, O. S., HENINGER, N., FELTEN, E. W., HALDERMAN, J. A., ROSSBACH, C. J., WATERS, B., AND WITCHEL, E. Defeating vanish with low-cost sybil attacks against large dhts. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010* (2010).