New Directions for Self-Destructing Data Systems

Roxana Geambasu, Tadayoshi Kohno, Arvind Krishnamurthy, Amit Levy, Henry Levy
University of Washington

Paul Gardner
Vuze, Inc.

Vinnie Moscaritolo
PGP Corporation

ABSTRACT

This paper seeks to advance the state of the art in practical self-destructing data systems that secure sensitive data from disclosure in our highly mobile, social-networked, cloud-computing world. Our work facilitates the automatic, timed, and simultaneous destruction of *all* copies of a self-destructing data object (such as a message or file) without any explicit action by the user and without relying on any single trusted third party.

We make three contributions to the study of self-destructing data. First, we present *Cascade*, an extensible framework for integrating multiple key-storage mechanisms into a single self-destructing data system. Cascade enhances resistance to attack by combining the security advantages of a diverse set of key-storage approaches. Second, we introduce *Tide*, a new key-storage system for self-destructing data that leverages the ubiquity and easy deployment of Apache Web servers throughout the Internet. Third, based on our earlier work on Vanish and in light of recent attacks against the Vuze DHT, we demonstrate how to significantly harden Vuze and other DHTs against Sybil data-harvesting attacks, making DHTs applicable as key-storage systems under Cascade.

To validate our approach, we designed, implemented, deployed, and measured these systems. We prototyped the extensible Cascade system with support for Tide, Vuze, and OpenDHT. We prototyped the Tide key-storage system on Apache, deployed it on over 400 PlanetLab nodes in the Internet, and demonstrated that the structure is highly immune to attack. Finally, we designed and deployed a set of defenses to Sybil data-harvesting attacks in the live Vuze P2P system and measured them at full scale in the million-node DHT; our results demonstrate that these defenses provide a three-order-of-magnitude improvement over the original Vuze DHT, rendering data-harvesting attacks extremely impractical.

1. INTRODUCTION

Storage is cheap. Data lives forever. The Internet never forgets. These truths profoundly affect the interactions between people and modern computing systems. Some computer users, aware of these properties, adjust their behavior accordingly: for example, they avoid using email for personal communications, preferring the phone instead. Others do not store personal information on mobile computing devices, such as laptops, when crossing international borders for fear of international corporate espionage [46]. An unfortunate few, however, are suddenly surprised when their lack of caution or understanding of the permanent nature of data wrecks havoc with their personal or professional lives. Well-known examples involve the exposure of sensitive communications of celebrities and politicians [5, 47], although such oversights can affect us all [33]. For example, the hacking of Google China exposed emails belonging to reporters and activists to still unknown parties [25].

Self-destructing data systems are designed to address these concerns. Their goal is to destroy data after a pre-specified timeout, regardless of where the data is stored or archived and despite technology that may make such deletion challenging. As a result, such systems prevent the exposure of "old" data that is past its useful life. Self destruction is implemented by encrypting data with a key and then escrowing the information needed to reconstruct the decryption key with one or more third parties. Assuming that the key-reconstruction information disappears from the escrowing third parties at the intended time, encrypted data will become permanently unreadable: (1) even if an attacker obtains a copy of the encrypted data and the user's cryptographic keys and passphrases after the timeout, (2) without the user or user's agent taking any explicit action to delete it, (3) without needing to modify any stored or archived copies of that data, and (4) without the user relying on secure hardware. Once the key-reconstruction information disappears, data owners can be confident that their data will remain inaccessible to powerful attacks, whether from hackers who obtain copies of backup archives and passphrases or through legal means.

This paper strives to advance the state of the art in self-destructing data systems. Past research has explored two ends of a spectrum. At one end is the centralized approach, as exemplified by the Ephemerizer [30, 34] or the revocable backup system [8]. Centralized systems are difficult to scan or crawl externally, but a centralized structure is open to legal exposure and internal hacking attacks (such as Google China). A centralized system must also be trusted, but that trust may be unwarranted, as was the case for Hushmail [37].

At the other end of the spectrum is Vanish, based on a highly decentralized system (the Vuze DHT) with millions of privately owned, autonomous computers and no single trusted party. The DHT-based system is difficult to attack after data timeout; system churn makes it difficult or impossible to learn which nodes held data in the past, and legal attacks are challenging due to geographical dispersion (e.g., Vuze DHT nodes are distributed across 200 countries). However, the openness of most DHTs and their security-insensitive design makes Sybil data-harvesting attacks easily mountable, as was recently demonstrated [49].

Our work makes three contributions, building on Vanish concepts but advancing them in important new directions. First, we introduce *Cascade*, which takes a *hybrid approach* to self-destructing data. Cascade combines the best properties of the range of alternatives described above. It integrates multiple key-storage systems in a single framework so that the system as a whole is stronger than any individual component. That is, an attacker must compromise *all* of its key-storage components in order to violate the privacy properties we target. By providing this framework we raise the bar against attack significantly, forcing the attacker to use multiple, disparate means to break the system (e.g., both technical and legal attacks, or perhaps both legal and illegal attacks).

Second, we introduce *Tide*, a new key-storage system for self-destructing data that is positioned between Vanish and the Ephemerizer on the design spectrum. Tide harnesses the ubiquity and easy deployment of Apache Web servers. Like Vanish, Tide relies on mul-

tiple, autonomous, distributed systems; like the Ephemerizer, each system is a (possibly well-known) centralized server. Tide leverages strengths of both types of systems: it cannot easily be crawled or subverted by a single-site attack.

Third, we discuss in significant depth our security analysis of the Vuze DHT. In light of the recent Sybil data-harvesting attacks on Vuze and Vanish, and general skepticism about DHTs, we believe it is necessary to assess: (1) the extent to which those attacks exploited particular design weaknesses in Vuze, and (2) how simple security-sensitive DHT design changes can thwart such attacks. Therefore, we performed an extensive measurement-based study of the Vuze DHT and designed a set of defenses to resist the Sybil data-harvesting attack. We believe that our results should inform the design of all current and future DHTs.

To evaluate our contributions, we implemented a proof-of-concept Cascade prototype and added extensions for Tide, Vuze, and OpenDHT. We also implemented a Tide prototype based on the highly popular Apache Web server, deployed it on over 400 nodes on the PlanetLab global distributed system, and measured its reliability, performance, and security, showing that it is highly immune to infiltration attacks. Further, we designed practical defenses and deployed them in the Vuze DHT. While our study was not intended to investigate all possible attacks on open DHTs, our results show that these defenses increase the complexity of a Sybil data-harvesting attack by over *three orders of magnitude*, making such attacks extremely impractical for all but the most serious and highly provisioned attackers. Moreover, to the best of our knowledge, this is the first experimental study of new, deployed security mechanisms *at scale*, in a *live* million-node DHT with *real users*.

Overall, by strengthening the Vuze-based Vanish system, introducing a new key-storage system with complementary strengths and weaknesses (Tide), and creating an encompassing architecture to integrate these and other components (Cascade), we have provided further evidence for the feasibility of self-destructing data systems. Informed by these new research results, we are now integrating Cascade into PGP Virtual Disk and preparing an Internet-Draft to extend RFC-4880 (OpenPGP Message Format) to formally specify the interaction between Cascade and PGP (so that OpenPGP messages can be encapsulated with both Cascade and PGP public keys). Finally, we are planning to make all source code for Cascade and Tide available in the near future; source code for our deployed Vuze defenses is already available in Vuze's CVS.

2. SELF-DESTRUCTING DATA SYSTEMS

Control over data lifetime will become increasingly important as more public and private activities are captured in digital form, whether in the cloud or on personal devices. Self-destructing data systems can help users preserve some control, by ensuring that data becomes permanently unavailable after a pre-specified timeout. We now describe key properties of our threat model for self-destructing data systems and review how Vanish addressed these properties as background for understanding our current work.

2.1 Threat Model

Self-destructing data systems [8, 24, 34] seek to prevent *retroactive disclosure attacks* against sensitive data. In these systems, users identify certain data as *ephemeral*, encapsulate that data in a *vanishing data object* (VDO), and specify a timeout for it. For example, a user can create a VDO that contains a private message for a friend. The VDO might then be stored, copied, or transmitted over a network. The self-destructing data system ensures that *all* copies of the VDO become permanently unreadable after the timeout. This includes copies that may be cached or archived, online or offline, by both end systems and intermediate systems (Web servers, backup systems, etc.). Self-destructing data systems target scenarios in which users would prefer that their sensitive data disappear early rather than have it fall into the wrong hands after the timeout.

The concept of a retroactive attack is key. Specifically, we seek to protect against an adversary attempting to learn the contents of a specific VDO *after* its timeout. For example, a hacker might try to access communications – including files, emails, or messages – that were exchanged between certain parties in the past, or a legal entity might subpoena old data or communications, or an opportunistic corporate espionage attacker might try to steal secrets from a laptop. Self-destructing data does not protect against attackers who target a *specific* VDO *prior to* its timeout. For example, if a hacker breaks into a user's email account, self-destructing emails that are still readable (as well as future emails the hacker might see) are not protected, while those that have timed out are protected.

In a retroactive attack, the attacker targets a specific VDO days, weeks, months, or even years *after* it has timed out. Before that, we assume that the attacker does not know that a specific user or VDO is of interest. Once the attacker identifies a VDO as a target, however, we assume that he can obtain a copy of the VDO, as well as the user's secret keys and passphrases, from prior to timeout. Old data can be obtained from replicas, backup tapes, storage residues, etc. Keys can be obtained by legal means (e.g., judges ordering people to produce their passwords or keys) or by breaking or stealing users' passphrases (e.g., a user writes his passphrase on an easily accessible sticker).

Although the attacker identifies a specific VDO as a target only after its timeout, the attacker may run *untargeted precomputations* at any point in time, in preparation for future targeted attacks.[1] Untargeted precomputations are mounted against the self-destructing data system *as a whole*, not against any specific user or VDO. For example, if the self-destructing data system uses third-party servers to store critical information for a limited period of time, an attacker could try to opportunistically harvest that information, in the hope that it *might* prove useful in the future. The attacker might compromise some of the servers, for instance, or infiltrate the system with his own servers. Note that harvesting the necessary data at a single point in time is insufficient; because the attacker does not know which VDOs may be of interest in the future, he must *continuously* harvest critical information for as many VDOs as possible, 24 hours a day, 365 days a year.

Finally, we assume that users communicating via self-destructing data objects are trustworthy. For example, users exchanging VDO-encapsulated sensitive emails trust each other to view those emails; they also trust each other never to save cleartext copies of a VDO's sensitive data. This assumption is realistic for self-destructing data systems and distinguishes our threat model from that of digital rights management (DRM) systems, which assume user untrustworthiness.

Self-destructing data systems let users create ephemeral data (such as files) and ephemeral communications (such as emails or text messages) with a critical guarantee: once that data has timed out, it will be expunged and forever safe from discovery and abuse.

2.2 The Vanish System

This research extends our previous work on the Vanish [24] system. Vanish encapsulates data with a pre-specified timeout by: (1) encrypting the data with a random symmetric key that is never revealed to the user, (2) splitting that key into multiple pieces (shares) using threshold secret sharing [36], (3) scattering key pieces across

[1]The notion of untargeted precomputation is not unique to our threat model. For example, to prepare for a password-breaking attack, the attacker can precompute an untargeted dictionary with hashes of common passwords.

randomly chosen nodes in a global-scale, distributed peer-to-peer (P2P) system, and (4) bundling information necessary to retrieve key pieces with the encrypted data. This resulting bundle, or VDO, can be stored on the user's computer, sent in an email, or stored on a remote server. To reconstruct the message before the pre-specified timeout, someone with access to the VDO uses the bundled information to retrieve the key shares, reconstruct the key, and decrypt the data. The VDO can be further encapsulated using PGP to deny access to unauthorized parties, such as the email provider, prior to the timeout (recall that a retroactive attacker may learn the user's PGP keys/passphrases after the timeout).

The Vanish prototype used the commercially supported, global-scale Vuze DHT, which is part of the Vuze P2P system.[2] Vanish relies on two properties of the DHT to prevent key pieces from being recovered well after their specified timeouts. First, the DHT system supports a standard timeout mechanism that makes a key piece programmatically unrecoverable following its timeout. Second, DHTs have significant churn of various types, including nodes coming and going, nodes receiving new dynamic IP addresses, nodes assuming ID ranges previously controlled by other nodes, etc. This churn makes it difficult to determine which physical nodes stored given pieces of the key in the distant past. The use of global-scale distributed systems further deters recoverability, since key pieces may be stored across different legal jurisdictions.

We chose the Vuze DHT as a key storage backend for Vanish because of its popularity, global distribution, and large scale, which was estimated at over 1M nodes.[3] In addition, like other DHTs, Vuze is self managing, i.e., there is no single trusted entity in charge. It has recently been shown that the original Vuze DHT used by Vanish was highly vulnerable to data-harvesting attacks.[4] Specifically, Wolchok et al. [49] show that an efficient Sybil [21] attack could be mounted against Vuze; in so doing, attackers could harvest during the non-targeted precomputation phase a large fraction of the key pieces stored by Vanish in the DHT. This work shows the extent to which the Vuze DHT was *not* designed to support systems like Vanish; hence (and in retrospect), it is not surprising that a Vuze-based Vanish system is not sufficiently secure. Noting that the Vanish concept is much broader than the single data point of its initial prototype, the question remains: is it possible to build a Vanish-like self-destructing data system that makes such attacks impractical?

This paper reconsiders self-destructing data systems based on highly distributed components. We examine several questions. First, can we define an extensible high-level architecture that combines the strengths of multiple, diverse key-storage systems to substantially increase security? Second, can we design new types of key-storage systems that share the benefits of DHTs, but mitigate their weaknesses? Third, in light of the Sybil attack on Vuze, can we strengthen the security of a deployed DHT so that it can play an important role in self-destructing data systems? We answer these questions beginning with the next section.

3. CASCADE: MULTI-BACKEND ARCHITECTURE FOR SELF-DESTRUCTING DATA

This section presents Cascade, an extensible framework for integrating multiple key-storage mechanisms into a single self-destructing data system. An attack against Cascade succeeds only if the attacker can compromise *all* of the diverse components upon which the system is built. We now describe Cascade's design principles and architecture.

3.1 Design Principles

Cascade's architecture is guided by three key design principles:

Combine diverse components with different strengths. It is often said that a system is only as secure as its weakest link. In Cascade, on the other hand, we seek to build a system that is as secure as the union of its defenses. Cascade is a unified self-destructing data framework for multiple key-storage systems, or backends. Adding new key-storage components to Cascade should strengthen the system against confidentiality attacks; if not, it should never weaken the system. Combining different defenses with orthogonal security properties under different adversarial models can significantly increase the cost of an attack and take the possibility of a mountable attack outside of the reach of potential adversaries. A successful attack must subvert *all* of the combined system's backend components.

Apply both defense-in-depth and redundancy. Related to the preceding principle, Cascade provides defense-in-depth under a single adversarial model. While the greatest value for Cascade is obtained when it combines multiple backends that are secure under different adversarial assumptions, it can also combine multiple backends believed to be secure under the *same* adversarial model. This provides redundancy if one of those assumptions proves to be incorrect.

Support future innovation. Cascade's design must be extensible to allow the inclusion and incremental deployment of new key storage backends. Therefore, Cascade provides an environment within which experimental, often unproven approaches can be deployed while simultaneously benefiting from the security offered by other, better proven approaches. For example, our currently deployed, significantly strengthened Vuze backend can foster the gradual deployment of our new Apache-based Tide backend. Without composability, deployability would be a major roadblock for Tide.

3.2 Cascade Architecture

Figure 1(a) shows the high-level architecture of the Cascade multi-backend system. At the top are self-deleting data applications, which might include email, messaging, social networking, file systems, etc. An application interacts with Cascade through the Cascade Application API, which encapsulates data into a VDO and later decapsulates that data from the VDO.[5] Encapsulation and decapsulation requests are handled by Cascade's extensible core. The core functions on the same principles as the original Vanish system: it encapsulates the data, generates a key K, splits it into key shares, scatters those shares for temporary storage on random components of a backend storage system, and then deletes all local copies of K. Cascade's core differs from Vanish in two ways. First, it supports share distribution across an arbitrary and extensible set of backend systems. Second, it uses a flexible *hierarchical secret sharing* (HSS) scheme to compose these backends for security. While the literature on hierarchical secret sharing is broad, we find that the naive approach is ideally suited for our case: simply split the key or key share at each internal node of the tree with the desired secret sharing parameters.

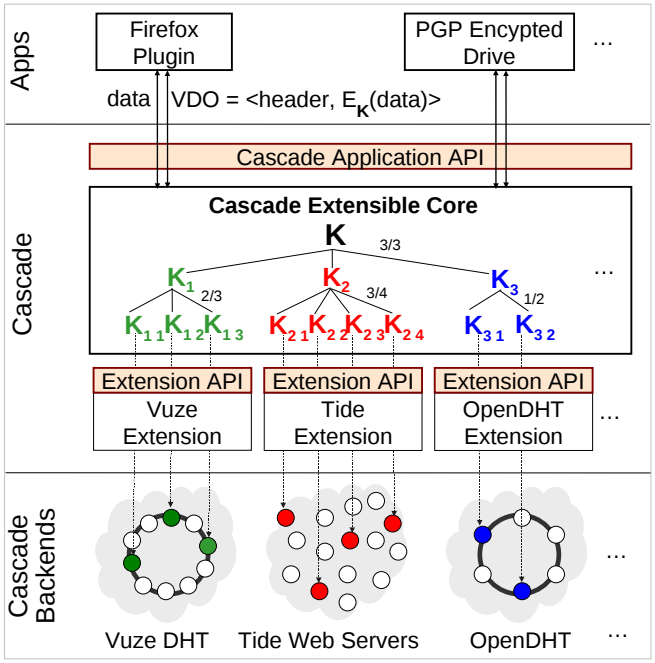
As an example, Figure 1(a) shows how an HSS scheme can be used to compose three very different backend systems: the Vuze DHT, a large set of Cascade-enabled Web servers (called Tide, which we describe in the next section), and OpenDHT (a DHT which is open to all clients but, despite its name, has strong restrictions on

[2] A DHT, or distributed hash table, implements a simple (index,value) hash table interface over a large collection of P2P nodes.

[3] DHT use has grown significantly; there are now DHTs estimated to be five- or six-million nodes.

[4] Data-harvesting attacks are related to, but different than, traditional node-crawling attacks considered in the past [41, 42, 43].

[5] For simplicity, we reuse Vanish's acronym for self-destructing objects (VDOs); however, as will become clear in this section, Cascade VDOs differ from Vanish VDOs in that their structure supports Cascade's main principles: extensibility and composability.

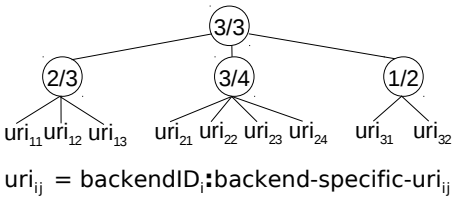


(a) Cascade Architecture.

Cascade Application API:
encapsulate(data, timeout, [HSS params]) => vdo
decapsulate(vdo) => data / error

Cascade Backend Extension API:
generateUri([incomplete uri]) => backend-specific uri
put(backend-specific uri, share, timeout)
get(backend-specific uri) => share / error

(b) Cascade APIs.



Examples:
uri_{11} = vuze:<160b random DHT index>
uri_{21} = tide:http://pgp.com/tide/<256b random index>

(c) Hierarchical Secret Sharing Tree in VDO Header.

Figure 1: The Cascade Multi-Backend Architecture, APIs, and Headers. (a) Architectural components and APIs in Cascade. (b) Cascade application and backend extension APIs. (c) Example of a hierarchical secret sharing (HSS) tree; uri_{ij} corresponds to the Cascade URI of share K_{ij} in Figure (a).

which nodes can become full DHT participants). The encryption key K is split into three shares (K_1, K_2, and K_3), all of which are needed to reconstruct key K. Each share is then itself split into multiple sub-shares (K_{11}, K_{12}, etc.) using varied, backend-specific threshold parameters. Finally, sub-shares are submitted for temporary storage to the three backends. In the example shown, the three original shares (K_1, K_2, and K_3) are all required in order to reconstruct K; this forces an attacker to compromise both Vuze and OpenDHT *and* at least three of the four selected Tide Apache servers to capture the VDO.

Other HSS constructions are possible, and their structures are dictated by the application. For example, suppose Alice (`alice@company1.com`) sends an email to Bob (`bob@company2.com`). A Cascade implementation, packaged with the email client, could check for the existence of Tide servers at `http://company1.com/tide` and `http://company2.com/tide` and store key shares on those servers. This entire structure can be further augmented with Vuze, OpenDHT, and other randomly selected Tide servers. This architecture has two principal advantages. First, it puts `company1.com` *and* `company2.com` in control of the timeout of key shares used in their communication.[6] If Alice's company fails to securely delete its shares in time, Bob's company can delete its own shares and cause their communications to self-destruct. This scenario would be useful to ensure the cleanup of sensitive communications between a large company with strict policies, like Microsoft or Google, and its possibly less well-managed contractors. Second, the storage of shares on other backends, like Vuze and random Tide servers, provides additional defense-in-depth. (Although our architecture is supportive of arbitrary hierarchies, our prototype implementation currently supports only three levels, as shown in Figure 1(a).)

Cascade Key Storage Systems. The Cascade architecture is agnostic to these key storage backend choices; each backend simply stores (index,value) pairs and can retrieve each pair up to its specified timeout, after which that value can never be recovered. Cascade is designed to support timeouts ranging from several hours up to a week. A crucial design goal in Cascade is to cleanly and elegantly support the composition of multiple key storage backends, including ones that have not yet been invented. To achieve this goal, we maintain the Cascade core completely independent of the underlying key storage backends and support dynamic loading of backend extensions that understand the specifics of the varied key storage systems. All extensions export a simple, unified API, called the Cascade Extension API, summarized in Figure 1(b). With the help of this infrastructure, inserting new backends into the system becomes trivial. For example, our Tide, Vuze, and OpenDHT extensions consist of merely 151, 827, and 386 lines of Java code, respectively, which deal almost exclusively with details of each storage system.

VDO Structure. Similar to Vanish, a Cascade VDO bundles together: (1) information describing how to retrieve the VDO's key shares (i.e., where the shares were stored), and (2) the application data encrypted under the split key K. The key information is included in the VDO's header, along with other metadata, such as specifications of the encryption and compression algorithms, the VDO's timeout, etc.

To be able to retrieve and compose key shares back into the original key K upon decapsulation, Cascade saves placement and secret-sharing information in the header of a VDO in the form of an *HSS tree*. Figure 1(c) shows the HSS tree corresponding to the example in Figure 1(a); inner nodes specify secret-sharing parameters (number of shares and threshold), while leaves specify the "locations" where each share was placed. Share locations are in the form of *share URIs*, which are composed of two fields: a *backend identifier*, which uniquely identifies a dynamically loaded backend extension, and a *backend-specific URI*, which identifies the share within the backend. Backend-specific URIs are generated and interpreted by the corresponding backend extensions (API function `generateUri`) and are opaque to the Cascade core. For example, our Vuze backend

[6] Recall from Section 2.1 that the threat model in self-destructing data systems is orthogonal to that of DRM systems. Alice and Bob only use Cascade for sensitive exchanges and, in doing so, explicitly trust each other not to save cleartext copies of the emails.

extension generates a 160-bit DHT index that indicates where in the Vuze DHT the share is stored. Our Tide backend extension, on the other hand, selects both the URL of the Cascade-enabled Web server and the index under which the share is "hidden" on that server.

3.3 Summary

Cascade is an extensible framework for composing multiple, heterogeneous backend key-storage systems. The system supports simple APIs, both for applications and backend systems, and uses a flexible hierarchical secret sharing scheme that applications can customize to achieve desired security properties. We have implemented a Cascade Java prototype along with the backend extensions shown in Figure 1(a), ported our self-destructing-web-data Firefox plugin [24] to Cascade, and plan to make all source code available.

4. THE TIDE KEY-STORAGE SYSTEM

This section presents our second major contribution to self-destructing data: *Tide*, a novel key store for Cascade that combines beneficial properties of both centralized services and decentralized P2P systems. A DHT's decentralization, global distribution, and autonomy make it hard to subpoena, while its openness and churn make it vulnerable to data-harvesting attacks. In contrast, centralized systems can deflect data-harvesting attacks (e.g., by protecting the data using ACLs or other protection mechanisms) but are also more vulnerable to legal attacks [48] and hacking [25]. Tide merges these strengths in order to overcome such limitations.

Tide leverages both the thousands of Web servers across the planet and the ease of deploying new Web servers with freely available components, such as Apache. Tide can be deployed in many ways. It is capable of scattering VDO key shares across a randomly chosen set of independent Web sites. These sites might include a global collection of organizations, companies, universities, and even individuals. Tide can also be deployed on private Web servers, or in other configurations. Slightly modified to support Tide, the Web sites maintain shares for a specified time limit; when the limit for a share expires, its server erases that share. Each share in Tide is protected by association with a random 256-bit index. To retrieve a share, a client must know its index, which makes guessing or scanning impossible (the index is effectively a password or capability). Tide can be used as the single key store in Cascade or composed with other key-storage systems to increase security, as described in Section 3.

Tide was designed to: (1) be simple, lightweight, and easily deployable, (2) avoid undeleted share residues, and (3) avail itself to deployment scenarios that are resilient to malicious infiltration. The first goal evokes our belief that a small module that is easy to understand, audit, and deploy increases our chance of adoption by many Web sites. In addition, Tide's ease of deployment facilitates hosting by end users, e.g., a group of friends could run their own private Tide Web servers to secure their communications. Our second and third goals address the primary two threats in a Tide-like system: post-timeout attacks targeted against Web sites that used to store the shares of a specific timed-out VDO, and pre-timeout untargeted key-share harvesting via infiltration in the set of Tide Web servers.

In this section, we present the design of our simple, easy-to-deploy Tide prototype based on the popular Apache Web server. We first show how our prototype's design achieves the first two goals, then we describe techniques for resisting malicious infiltration in the context of various deployment opportunities, and finally we present measurements of our early prototype against attacks.

4.1 The Tide Apache Module

To maximize deployability, we designed and implemented a Tide prototype that leverages the widespread adoption of the Apache Web server and its modular structure. Our prototype is a small, lightweight, and dynamically loadable Apache module that allows clients to temporarily use an Apache Web service as a simple (index,value) storage system. Our module contains 826 lines of C code and can be easily inspected – perhaps even model-checked – for vulnerabilities.

Tide Apache Module Design. Our goal of simple and lightweight implementation requires us to reject complicated, heavy, or stateful protocols. The Tide Apache module exposes an almost trivial REST interface that mirrors Cascade's backend API and offers two simple functions: `put` and `get`. The Tide extension running on a user's machine invokes these functions to store a key share in a specific Apache server for a specified period of time. Our module maintains little state other than a size-limited table of temporary (index,value) pairs. The module currently runs on one Web server, but larger sites can redirect all requests for the Tide URL to a specific Web server.

The Tide module explicitly seeks to avoid share residues. To minimize the risk of improper cleanup of shares at the timeout, we maintain (index,value) pairs only in primary memory and never persist them to disk. For security, we never swap server memories to disk (most Web servers avoid swapping as well for performance [3]). There are several caveats, however. If the Web server runs in a VMware ESX virtual machine, its memory might be swapped to disk during memory reallocation from one VM to another; other VMMs, like Xen, do not support memory sharing, so they never swap a VM's memory except during suspends and migrations. Thus, Amazon EC2-powered Web servers are currently safe, since EC2 is based on Xen, and no suspensions and migrations are possible. While our in-memory policy of dealing with residues is imperfect, its deficiencies are counterbalanced by our use of secret sharing and by the retroactive-attack model. It is unlikely that a large proportion of key shares on disparate servers preserve residues for weeks or months after the timeout. Pinning data in volatile memory does cause all data to be lost when an Apache server is restarted. However, threshold secret sharing helps mitigate such losses probabilistically. Section 4.3 evaluates both security and availability.

Finally, since Tide relies on volunteer opt-in, we must ensure that our system remains unobtrusive to the server's general functioning. We therefore install harsh limits on the size of each index and value, the maximum memory consumption, and the maximum timeout. Standard rate-limiting mechanisms can also be used to limit the amount of traffic serviced by our module. These limits are configurable on a per-Tide-module basis, and a single Apache server can host multiple Tide instances, each with its own URL and configuration. For example, the PGP administrator could launch a public Tide module that responds to all requests for `http://pgp.com/tide/public` and enforces a set of Draconian limits. At the same time, he could instantiate a private Tide module for PGP's employees, configure it for the URL `http://pgp.com/tide/internal`, perform `.htaccess`-based authentication, and impose more relaxed limits.

Integration with Cascade. Due to Cascade's extensibility, integration of the Apache-based Tide system is trivial. We wrote a 151-line Cascade extension in Java for the Tide backend (see Figure 1) that simply relays `puts` and `gets` from the Cascade core to the appropriate Tide Apache server. The server is directly determined from the share URI, which includes both the server's URL and the index within that server. To generate a new share URI (function `generateUri` in Figure 1(b)), the extension simply chooses a random server URL from a database of known Tide servers and concatenates it with a random 256-bit index.

4.2 Deployment Options and Infiltration Defenses

We have identified three deployment options for Tide: (1) well-known trusted servers, (2) private servers, and (3) a world-wide deployment. We now discuss each option, focusing on ways to prevent malicious infiltration in each case.

Well-Known Trusted Servers. The simplest deployment option is for several well-known and trustworthy entities to embed Tide modules into their Apache frontends. For example, we could imagine that potential early adopters might include Web sites controlled by privacy advocates (e.g., `pgp.com`), freedom-of-speech supporters (e.g., `rsf.org`, `eff.org`), open-information supporters (e.g., `kernel.org`, `sourceforge.net`, `wikipedia.org`), and academic institutions (e.g., `cs.washington.edu`). A list of such servers could be preconfigured in Cascade or published in a manner similar to Tor directory servers. As with Tor, a user must trust the list provider not to infiltrate the list with malicious Web servers. That may be acceptable, particularly if the user audits the list and selects only a subset of servers that he or she most trusts (e.g., one might select only `.edu` domains). Such an infiltration defense will not be effective against an attacker who compromises individual trusted servers, however the use of secret sharing provides some defense. Section 4.3 quantifies the percentage of servers that an attacker must infiltrate or compromise in order to compromise Tide in a realistic deployment scenario.

Private Tide Servers. The communicants themselves – or their employers – might also host Tide Web servers for the timely destruction of their communications. For example, when Alice sends a self-destructing email from her account (`alice@company1.com`) to Bob (`bob@company2.com`), the Cascade software on Alice's computer could automatically search for Tide servers at `http://www.company1.com/tide` and `http://www.company2.com/tide` and, if present, incorporate those Web servers into the secret-sharing hierarchy. Similarly, groups of individuals can host private servers for their friends; private servers are likely to be safe from malicious infiltration unless human engineering attacks are used.

World-Wide Deployment. While the preceding solutions can resist infiltration, they exhibit little geographical diversity, scale poorly with an increased user population, and may be heavyweight for users. For example, imagine that all 500M Facebook users wanted to send and receive self-destructing messages every 10 minutes; a small set of well-known, heavily rate-limited Tide servers would likely not scale to this task.

As a more scalable, lightweight, and exciting deployment opportunity, imagine a world where most Apache web servers (half of all Web servers in the world [31]) have enabled the Tide module. In this world, different VDOs created by a range of users would choose different Apache servers on which to store their VDO key shares. Users might even choose specific amounts of geographical diversity. Such a deployment would be truly scalable. The biggest challenge with world-wide deployments is the possibility of infiltration attacks. As a defense, users might employ a PGP-like *web-of-trust* model in order to incorporate new Tide servers into their databases. For example, if Bob receives an email from Alice, and Bob trusts Alice, then he might incorporate Alice's list of trusted servers into his own list of servers. We leave the full investigation of this solution for future work.

Each deployment scenario has advantages and disadvantages. Cascade's compositional architecture allows them to be used individually and in combination. We note once again that for all preceding cases, the use of threshold secret sharing provides resistance against small numbers of server failures. Similarly, a user need not trust all servers since no single server has sufficient knowledge to reconstruct a decryption key. Finally, the VDO is never stored in Cascade or any of the Tide key servers. We next evaluate the security, reliability, and performance of our prototype Tide system in the context of the first deployment model, the only scenario our current prototype fully implements.

4.3 Evaluation

To evaluate our Tide prototype in a realistic global setting, we deployed Tide-enabled Apache Web servers on 462 PlanetLab nodes. These nodes are servers scattered all around the world and should approximate a realistic deployment. The goal of our study was to quantify these tradeoffs: (1) VDO availability, (2) VDO security, and (3) VDO operation performance. We examine availability in the context of server crashes or reboots. We derive secret sharing parameters that guarantee VDO availability throughout their lifetime. And we study VDO security when some of the servers are compromised. Finally, we evaluate the performance of VDO encapsulation and decapsulation operations on globally distributed Tide servers.

In the experiments, a VDO's encryption key, K, is split into N shares, each of which is stored at a randomly chosen Tide Planetlab node. The critical parameter of the secret sharing scheme is the *threshold ratio*, which determines the percentage of the N shares that are required to decapsulate the VDO. We use interchangeably the terms "threshold ratio" and "$M:N$" ratio, where M out of N shares are required for decryption. For example, if 40 out of 50 shares are required, the threshold ratio is 80%.

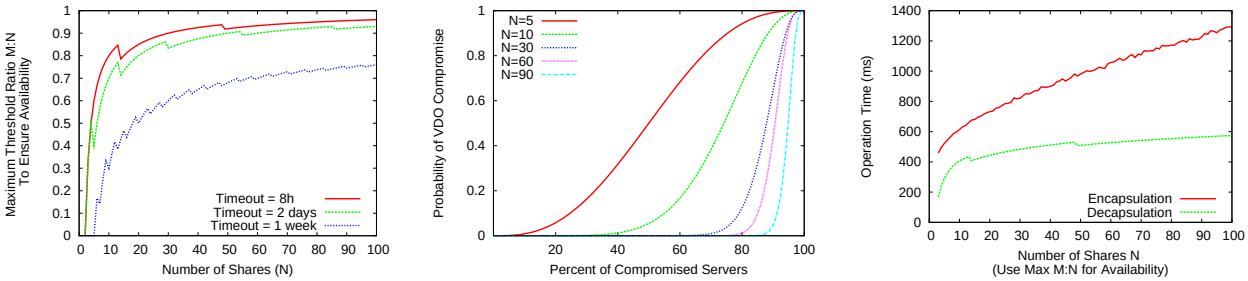
VDO Availability. In the absence of real Web server uptime and reboot data, we leveraged the uptime information obtained from our 462 Planetlab servers.[7] To estimate VDO availability, we simulate VDOs under various numbers of shares and threshold ratios and compute the probability that any given VDO would remain available until its timeout, given crashes and reboots.

Server churn is typically small (e.g., the median node lifetime in our Planetlab trace is 59 days). Therefore, it is not difficult to achieve good availability guarantees for VDOs with timeouts of up to a week. In particular, for most values of N ($N \geq 5$), we can identify a range of threshold ratios that ensure VDO availability with high probability. However, our goal is to find the ratio that provides the optimal tradeoff between security and availability. Higher ratios provide better security (an attacker must infiltrate a larger number of servers) but result in lower availability (a smaller number of failures makes the VDO unavailable).

Figure 2(a) graphs the maximum threshold ratio ($M:N$) necessary to guarantee VDO availability with a probability > 0.99999 for various numbers of shares and three timeouts (8 hours, 2 days, and 1 week). For an 8-hour timeout (the Cascade default), scattering shares across 30 servers yields a maximum threshold ratio of 90%; i.e., with very high confidence, at least 90% of those 30 servers will remain up during the VDO's 8-hour lifetime. However, for larger timeouts, such as one week, the required threshold ratios are much lower. In this case, again with 30 shares, the maximum ratio Tide can support is around 60% (i.e., requiring more than 18 of the 30 shares to reconstruct the VDO or risks losing the VDO prematurely). The reason is intuitive: during one week, there is a higher chance that servers will have rebooted or crashed than during any 8-hour period, and we must therefore adjust the secret sharing threshold ratio to account for that.

VDO Security. We evaluate security in the context of an adversary who controls a fraction f of the Tide servers. The adversary can achieve control either by infiltrating into or compromising some of

[7] We believe that our approach is conservative and that our results likely underestimate VDO availability.



(a) Availability. (b) Security. (c) Performance.

Figure 2: Evaluation of Planetlab Tide Deployment. (a) Maximum threshold ratio required to ensure VDO availability with probability > 0.99999 for increasing number of shares and various timeouts. (b) VDO capture probability with percentage of compromised servers, for various numbers of shares. (c) Encapsulation and decapsulation times with the number of shares. In (b) and (c), for each number of shares we use the maximum threshold ratio for 8-h availability.

the Web servers in a non-targeted precomputation attack, or by compromising the specific Web servers that used to store key shares for a specific VDO in a post-timeout targeted attack.

Given that VDO shares are placed at random on the servers, capturing VDOs is probabilistic. To assess the probability that an attacker captures a VDO, we use a simple combinatorial model that takes f and the VDO parameters (N, threshold ratio $M:N$) as inputs. Figure 2(b) shows the probability of VDO compromise as the fraction of compromised servers increases for various numbers of shares. For each value of N, we use the maximum allowable threshold ratio $M:N$ to ensure availability for the 8-hour default Cascade timeout. Using $N = 30$ again as an example, we see that an attacker who has compromised 80% of those servers (24 servers) will capture only 15% of the VDOs given the 90% threshold ratio (Figure 2(b), third curve from the left). Hence, in the context of a real-world deployment like our Planetlab Tide experimental setup, we conclude that using $N = 30$ shares and a threshold ratio of 90% provides both good availability for 8-hour timeouts and good security.

VDO Operation Performance. We evaluate Tide performance by measuring encapsulation and decapsulation times against our PlanetLab deployment. Each encapsulation/decapsulation involves requests to N servers in parallel and we performed $10,000$ operations of each type. Figure 2(c) shows the encapsulation and decapsulation runtimes for various numbers of shares (N); for each value of N, we again use the maximum threshold that ensures VDO availability for 8 hours. As the number of shares increases, VDO encapsulation times increase close to linearly while VDO decapsulation times quickly level off. This is because encapsulation needs to await responses from all N servers, while decapsulation waits only for the fastest M shares to arrive.

Overall, encapsulation and decapsulation times remain under $1.3s$ and $600ms$, respectively. In the case of the recommended parameters for our Planetlab deployment, using $N = 30$ shares and a threshold ratio of 90% leads to $484ms$ decapsulation times and $820ms$ encapsulation times. As we demonstrated in our original Vanish paper, much of the encapsulation time can be hidden from users via simple prepush mechanisms, which proactively split random encryption keys and store their shares in preparation for a user's encapsulation request [24]. Similarly, decapsulation times can be hidden via simple prefetch mechanisms.

4.4 Summary

Tide is a novel key-storage backend that leverages the advantages of both distributed and centralized key-storage systems. We have implemented Tide as a simple, lightweight Apache-based Cascade module and deployed and measured it on PlanetLab. Our measurements demonstrate the viability of this approach in real-world settings.

5. STRENGTHENING A DEPLOYED DHT

As noted, the properties of communal DHTs make them tempting components in our composable architecture. Yet it is precisely these properties that also make DHTs vulnerable to *Sybil data-harvesting attacks*. An attacker who joins the DHT with enough nodes can potentially harvest most or all of the data stored in the DHT. This weakness was recently shown to be a serious concern for Vanish running on the Vuze DHT [49].

This section examines simple, practical measures that can *significantly* strengthen DHTs in the face of Sybil data-harvesting attacks. We deployed and experimentally evaluated these new defenses in Vuze. Our results show that they can raise the cost of a harvesting continuous attack on Vuze by three orders of magnitude. To the best of our knowledge, defenses against such attacks have never been deployed and evaluated experimentally in a large, live DHT at scale.

This section's contribution is the design, deployment, and measurement-based evaluation of practical DHT defenses to Sybil data-harvesting attacks in the live, million-node Vuze DHT. The results of our work are much broader than Vuze. Relative to other contemporary DHTs, Vuze is comparatively small, e.g., approximately one million nodes compared to over five million for the uTorrent DHT.[8] We postulate that our defenses would be even more potent if applied to larger DHTs.

Finally, we recognize that other attacks could be mounted against deployed DHTs. Our goal, however, is to focus on a particular attack – the Sybil data-harvesting attack – and advance the state of the art in understanding and defending against this attack from a practical, measurement-driven perspective.

5.1 The Data-Harvesting Attack

The data-harvesting attack is an untargeted precomputation-phase attack that aims at capturing as many key shares from the DHT as possible under the assumption that those key shares might be useful in decapsulating (currently unknown) VDOs in the future. Two aspects of the original Vuze design made it particularly vulnerable to data-harvesting attacks.

1. *Overly provisioned replication.* The Vuze design ensures data availability in the face of churn in two ways. First, when a new node joins the DHT, its neighbors quickly push copies of

[8] Estimation is based on measurements from a uTorrent plugin.

all their contents to it (called *push-on-join replication*). This allows a malicious node to obtain all the data in its ID-space vicinity within a few minutes. Second, every node replicates each of its values to its 19 closest neighbors every 30 minutes. This replication level is a large security loophole and, as we will show, is unnecessary for availability.

2. *Lack of protection against Sybil attacks.* A node's location in the Vuze DHT (viz., its identity) is determined as a function of its IP address and port. In the Vuze design, a single physical machine can fabricate up to $64K$ identities (for a given IP address), which gives it huge freedom to place itself in the DHT. Restricting this ability is crucial to defending against Sybil attacks.

In an example scenario, the adversary infiltrates the original Vuze DHT with a large number of nodes (Sybils), places them at distinct locations in the DHT (i.e., assumes different IDs), and archives all of the shares that the nodes receive. The Clearview paper [49] showed that an adversary can mount this attack using a small number of physical machines, each hosting a large number of Sybils that communicate through distinct ports. Each Sybil remains at a DHT location only for a short period of time, collects data replicated by the nearby nodes, and then *hops* to a different location to repeat the process.[9] Because self-destructing data is intended to protect against retroactive attacks, a data-harvesting attack must be performed *continuously* – 24 hours a day, 365 days a year – since the attacker lacks a-priori knowledge of what shares it will need in the future.

5.2 Increasing Security against Sybil Data-Harvesting Attacks

Guided by the above observations, we modified Vuze's design as follows:

1. *Limit data dissemination* by altering the replication algorithm and drastically tuning its parameters to significantly increase security while preserving availability.
2. *Limit DHT ID fabrication* by imposing harsh limits on the set of IDs that can be fabricated from one physical machine and from various network IP *prefixes* (e.g., from within a corporation's or other organization's network).

Table 1 summarizes our defenses and their intended effects. We describe these defenses in detail below, but we foreshadow our results with this high-level summary.

1. *Defenses significantly increase attack costs*. Our modifications increase the cost of mounting a data-harvesting attack by over three orders of magnitude. For instance, if the attack were to be performed from an EC2-priced cloud computing infrastructure, the attacker would have to invest $\$7M$/year as opposed to the $\$5K$/year required to compromise Vuze prior to our modifications.
2. *Data-harvesting becomes infeasible for most attackers.* Our modifications require the attacker to exhibit high levels of IP-prefix diversity, such as control over 24 distinct /16 IP address blocks. This makes the data-harvesting attack impractical for all but a few multi-national companies or ISPs.
3. *Defenses have little or no negative impact on the DHT.* We show that our changes have a negligible impact on system properties such as availability and DHT size. In fact, some of our measures (e.g., disabling push-on-join and reducing replication) lower DHT load. Our defenses are also simple and practical; they were implemented by one developer in two days and $\approx 1,500$ lines of code.

[9] The hopping aspect is an innovation of the Clearview paper along with their efficient Sybil infrastructure.

While our defenses are specifically designed to frustrate data-harvesting attacks, some of them may have wider applicability. In particular, our limited ID-fabrication scheme may be valuable for limiting the impact of other Sybil-driven attacks, such as routing attacks [39]. By being the first live DHT to deploy these defenses, we also hope that our hardened Vuze deployment will become a testbed for other researchers seeking to study DHT security at scale.

5.2.1 Limiting Data Dissemination

To limit data dissemination, we alter Vuze's replication mechanism in three ways. First and most obvious, we disable push-on-join-replication, which allowed an attacker to join with an extremely low number of simultaneous nodes and still capture a majority of the shares during an 8-hour period [49].

Second, we designed and deployed a new replication algorithm, called *conditional replication*. Conditional replication follows four principles: replicate only when needed, replicate only by the amount needed, ensure a minimum time between consecutive replications, and allow some data loss. The first two principles avoid creating a new replica unless the number of replicas has dropped below a specified threshold. The third principle lets us protect against attacks where colluding nodes might attempt to force a node into replicating prematurely (a variant of the cuddling attack described Clearview [49]). Finally, Cascade tolerates some data loss and prefers it to over-replication.

With conditional replication, a Vuze node considers replicating a value only when a specified *minimum replication interval* has passed since the value was last replicated or stored. More important, a node first checks to see how many replicas exist for the value before replicating. If the number of existing replicas is at or above a specified *replication factor*, no replication is performed; otherwise, the node bumps the number of value replicas back to the replication factor. We were careful to implement the replica survey so that no new attacks are exposed (e.g., during the replica survey, neither the surveying node nor the responding node reveals full indexes of stored values).

Our final change to Vuze's replication consists of a measurement-driven configuration of replication parameters based on real churn conditions. We stress that this is not a mere fine-tuning of parameters. Rather, we show that today's default replication parameters grossly over-estimate the churn in the Vuze DHT.[10] For example, a coarse replication interval of 4 hours and a small replication factor of 5 are sufficient. Compared to the Vuze default of 30 minute, 20-way replication, these modifications represent dramatic changes to the Vuze replication mechanism.

5.2.2 Limiting DHT ID Fabrication

Our next defense seeks to limit an attacker's ability to infiltrate the DHT at very large scale. Many techniques for this have been proposed in the literature and are reviewed in Section 6, e.g., [9, 21, 28, 50]. As previously noted, the Vuze design allowed an attacker to create large numbers of Vuze virtual nodes (Sybils) on a single IP address: one node for every allocatable port, or close to $64K$ nodes/IP. To limit the number of nodes that an attacker can emulate, we introduce a new lightweight, yet effective formula for computing DHT IDs. The revised node ID calculation caps the number of nodes that an attacker with limited IP diversification can create in a DHT. Previous work has proposed relying on IP diversity to detect routing attacks [23]. However to the best of our knowledge no one has incorporated IP diversity requirements in DHT ID calculations.

In Vuze and many other DHTs, a joining node's ID is generated by computing the SHA1 hash of the node's publicly visible address (IP) and port number (P), i.e., $H(IP,P) = \text{SHA1}(IP \mid\mid P)$, where $\mid\mid$ is the

[10] Results from two-year-old studies also suggest relatively low churn [22].

Defense	Effect	Status
Disable on-join-replication	Limits data dissemination	Deployed, used by all Vuze apps
Conditional replication	Limits data dissemination	Deployed, used by Cascade
Reduce replication factor (3x impact)	Limits data dissemination	Deployed, used by Cascade
Increase min. replication interval (80x impact)	Limits data dissemination	Deployed, used by Cascade
Prefix-based ID calculation	Raises bar for Sybil	Implemented, to be deployed and enforced in future release*
NAT traversal (2x impact on direct puts)	Raises bar for Sybil	Implemented, not deployed
Port to larger DHTs (up to 6x impact)	Raises bar for Sybil	Not implemented

Table 1: Vuze Data-Harvesting Defenses, Effects, and Deployment Status. *We delayed deploying the prefix defense because it would prevent us from conducting the large-scale attacks needed for measurements in this section. We intend to deploy it in the near future.

bitstring concatenation. One can restrict to k the number of identities that a given node can utilize by using the modified hash function, $H(IP,P) = \text{SHA1}(IP \mid\mid (P\%k))$, where % denotes the modulus operation. A node can generate at most k distinct DHT identities by using different ports from the same IP address.

We modify this hash function to also limit the number of nodes that can participate from a given *IP prefix*. Let $IP[1],\ldots,IP[4]$ be the first through fourth bytes of an IP address, with $IP[1]$ being the most significant (e.g., 128 in the case of the IP 128.18.15.3). The following function generates IDs for nodes joining the DHT and determines their "locations" in the DHT:

$$H(IP,P) = \text{SHA1}(IP[1] \mid\mid (IP[2] \mid\mid (IP[3] \mid\mid (IP[4] \mid\mid (P \,\%\, k_4)) \,\%\, k_3) \,\%\, k_2) \,\%k_1)$$

This function $H(\cdot)$ limits an IP to at most k_4 identities and also caps the number of identities that can be generated by /8, /16, and /24 prefixes to k_1, k_2, and k_3, respectively. As a concrete example, the University of Washington (UW) uniquely controls a 16-bit IP prefix (128.208) and can generate IP addresses 128.208.0.0 through 128.208.255.255.[11] UW can therefore create up to 64K unique IP addresses that could be deployed in a Vuze Sybil attack if it were malicious. However, by setting k_2 to 2K, for example, we reduce by a factor of 32 the number of DHT positions that UW (and all other /16 owners) can occupy in the DHT – from 64K positions to 2K positions. If a successful Sybil attack requires placement at, say, 64K positions, then UW would need to co-opt at least another thirty-one /16 networks to collaborate in the attack. Moreover, assuming that we also set $k_4 = 4$ nodes, a user or hacker who controls one or a few IP addresses in each /16 would not be able to mount the attack. Rather, an attacker must either control the routers of all thirty-two /16 networks or 500 IP addresses in each /16 network. This would be a formidable task for UW.[12]

There are two issues with our scheme. First, this technique prevents some nodes from operating as *full participants* in the DHT, e.g., only k_4 nodes could fully participate from behind a NAT. A node that cannot participate will not store values on behalf of other DHT nodes and will not be reached by others' lookups. However, it can still operate as a DHT *client*, i.e., the node can still perform its own stores and lookups. Such nodes *can* use BitTorrent swarms and store or read Cascade shares.[13] Second, a potential negative effect of our IP-based ID limitation is the possible reduction of the DHT's size. In particular, if some nodes were prevented from participating in the DHT's maintenance traffic (e.g., storage, replications, etc.), then the DHT would appear smaller than it actually is. We evaluate this effect in detail in Section 5.3.2.

[11] While others can spoof UW's IP addresses, spoofing has no value for a data-harvesting attack, because the attacker would not receive the return packets with data values. Route hijacking is also possible, though continuously hijacking a large number of routes for an extended period of time (years) poses an unprecedented challenge.

[12] We believe that we can be equally as effective in filtering in the large, flat IPv6 address space, however a detailed discussion is beyond the scope of this paper.

[13] This distinction between client nodes and full-participant nodes is part of the design in some DHTs. In OpenDHT, only some select nodes are allowed to participate fully, while others can be clients of the system.

5.3 Detailed Evaluation

We performed extensive experiments against the 1M-node Vuze DHT to evaluate the revised system's security against Sybil data-harvesting. To the best of our knowledge, this is the first study of deployed security defenses in a DHT *at scale*. We focus here on a small set of issues: (1) the security/availability tradeoff for conditional replication, (2) the impact of conditional replication on data-harvesting attacks, and (3) the effect of IP-prefix-based admission control. We were unable to experiment with the original push-on-join mechanism (which the Clearview paper is based upon) because Vuze has disabled that insecure mechanism altogether. While our evaluation focuses on Vuze, our findings are broadly applicable and could guide new architectures for even larger-scale DHTs (e.g., uTorrent, which has over 5M nodes).

To measure availability, we stored 1,000 values in the Vuze DHT for each set of replication parameters. We attempted to retrieve the values every 10 minutes for 16 hours (a typical timeout in Cascade). To measure the effectiveness of a data-harvesting attack, we join the Vuze DHT with nodes from various locations at UW and Amazon EC2 (distributed over different /16s). We use a modified version of the Clearview Sybil software [49] to create 1,000 DHT nodes per IP and up to 12,000 nodes per /16 network.[14] We ran a data-harvesting attack with 10K, 25K, 50K, and 72K simultaneous attack nodes in the Vuze DHT. The Sybil nodes hopped every replication period. We experimented with multiple replication and hopping intervals (from 1 to 8 hours). We found that a replication interval of 4 hours provides high levels of both availability and security. For each Clearview experiment, we stored at least 9,000 values per replication factor.

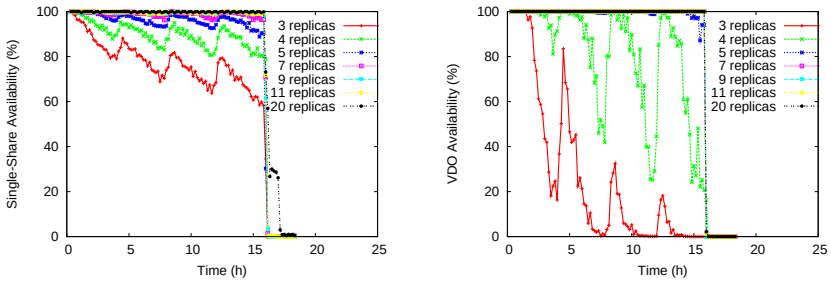
5.3.1 Evaluating Conditional Replication

This section evaluates conditional replication, focusing on the security/availability tradeoff and the impact of conditional replication on data harvesting attacks.

Availability Under Conditional Replication. Conditional replication seeks to limit data dissemination without hurting availability. Figure 3(a) shows the effect of the replication factor on single-share availability. We see that the availability of DHT values degrades with time, but some of that loss is reclaimed by periodic (once in 4 hour) conditional replication events, as shown by the graph's up-spikes. With a replication factor of three, approximately 21% of the values are permanently lost by 8 hours. However, at five replicas and above, availability stays above 95% during the 16-hour interval. These results confirm our assumption that the Vuze default replication policy is significantly over-engineered and over-provisioned for real churn conditions.

Our use of threshold secret sharing is designed to withstand some share loss. Figure 3(b), which shows VDO availability for 60 shares and a threshold ratio of 85%, illustrates the thresholding effect of secret sharing. VDO availability remains close to 1 when the ex-

[14] Our attack is more aggressive than the one used in Clearview [49]. Clearview benefits from push-on-join replication but avoids receiving direct `puts` in an effort to limit incoming traffic. As we will show, direct `puts` become more important to the attack due to our significantly increased replication interval. We therefore enabled our Sybils to receive direct `puts`.



(a) Single-Share Availability.

(b) VDO Availability.

Figure 3: Availability Under Conditional Replication. (a) Availability of a single share for conditional replication with 4h replication interval and 3–11 replicas and for the default 20-way, every-30-minutes replication in Vuze. (b) Availability of a VDO under the above replication parameters and for 60 shares and a threshold ratio of 85%.

pected availability of a single share exceeds 0.85, but it degrades significantly for lower values. The graph demonstrates that using five replicas and a minimum replication interval of 4 hours results in near-perfect availability for our parameters.

The Data-Harvesting Attack Under Conditional Replication. To evaluate the impact of conditional replication on data-harvesting attacks, we joined the DHT using a large number of Sybil attack nodes, with each performing the hopping attack every replication interval. Figure 4(a) shows the probability of capturing individual DHT values with 25,000 simultaneous attack nodes as a function of values' ages (i.e., the time since the values were stored in the DHT). Lines are shown for different replication factors, all using a 4-hour minimal-replication interval. The figure quantifies the probability of the two types of captures: those due to direct `puts` (the points at $age = 0$) and those due to replication (the points around $age = 4$, 8, and 12 hours). Conditional replication significantly reduces the attacker's share capture: the top curve shows the original Vuze 20-way replication policy, which results in nearly 100% capture at 8 hours, compared to 40% capture at 8 hours for conditional replication with a replication factor of 5.

Conditional replication changes the nature of the attack. First, our 4 hour minimal-replication interval severely limits the number of chances that an attacker has to capture a share during its lifetime. This implies that the attacker must hop less frequently and instead maintain a much larger *continuous* presence in the DHT. Second, the proportion of shares captured during any individual replication event is much smaller than the proportion of shares captured during the initial direct puts. This is partly due to a large proportion of Vuze nodes being firewalled, which makes the DHT seem much smaller from a direct `put` perspective (see Section 5.3.3 for details) and also partly due to conditional replication pushing shares to *fewer* nodes than its replication factor. For instance, if only two of five replicas have left the system since the last replication event, then conditional replication makes only two new replicas.

Compromising a VDO is much more difficult than compromising a single DHT value. Figure 4(b) illustrates the dramatic thresholding effect of secret sharing on VDO security. For 2 to 12 replicas, it shows the probability of the attacker capturing a given VDO (with $N = 60$ shares) using 25,000 simultaneous attack nodes hopping every 4 hours. For each of the replication factors, we use the maximum threshold ratio allowable to ensure VDO availability for the default 8h timeout. The graph's V shape illustrates an interesting tradeoff and the presence of an optimum replication factor. For small replication factors (e.g., two replicas), churn greatly affects the availability and persistence of the shares, requiring us to use extremely low threshold ratios. This results in poor security. From the graph, the optimal choice for replication factor is clearly five, which provides a probability of VDO compromise of approximately 10^{-10} with a threshold ratio of 0.85. As the replication factor increases above five, the small increase in allowable threshold ratio does not offset the increase in per-share capture that we saw in Figure 4(a), resulting again in poor security.

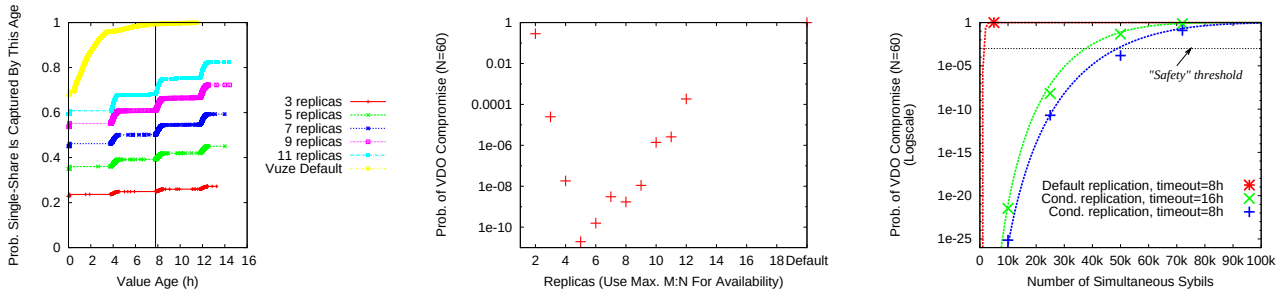
Figure 4(c) shows the probability of VDO compromise for an increasingly powerful attacker, measured by the number of simultaneous nodes it maintains in the DHT at all times. The graph compares conditional replication under our recommended parameters with the default Vuze 20-way replication policy. For conditional replication, we show results for two different timeouts: 8h (the default in Cascade) and 16h. Points on the graph indicate results directly obtained from our experiments with various simultaneous attacker nodes, while lines indicate the predictions of a simple probabilistic model seeded with the measurements from the 10,000-simultaneous-node experiment.

With conditional replication and VDO timeouts between 8 and 16 hours, attackers would require from 50,000 to 70,000 DHT nodes (Sybils) continuously in the DHT (365 days/year) to have a 10% chance of capturing any given VDO. Conservatively, we set the "safety" threshold (shown on the graph) to a much lower VDO compromise probability: 10^{-3}. Although an attacker will have little incentive to operate at such low capture probabilities, we conclude that for the timeouts that Cascade is designed for, an attacker needs at least 50,000 simultaneous nodes that maintain a *continuous* presence in the DHT. We integrate these results with our Sybil-restriction ID calculation scheme to assess the overall security of Cascade in Section 5.4.

Overall, the graph illustrates the radical improvement achievable by conditional replication relative to the default Vuze replication policy. For the default Vuze policy (after removing push-on-join replication), an attacker with only 5,000 simultaneous nodes can compromise nearly 99% of the VDOs within 8 hours. In comparison, conditional replication leads to an order-of-magnitude increase in the number of nodes an attacker must maintain in the DHT at all times. The improvement is even more dramatic if one considers push-on-join replication, which allowed an attacker to hop every few minutes. As we will discuss in Section 5.4, these increases lead to a several order-of-magnitude increase in the cost of an EC2-based data-harvesting attack relative to the original Vuze.

5.3.2 *Evaluating Prefix-Based ID Calculation*

We next estimate the extent to which our revised node ID calculation can limit Sybil attacks. As noted in Section 5.2.2, the goal of our ID calculation is to significantly reduce the ability of a particular organization to place Sybil nodes in the DHT. However, limiting DHT identity fabrication could reduce the number of nodes that can



(a) Share Compromise vs. Time. (b) VDO Compromise vs. Replicas. (c) Compromise vs. Simultaneous Nodes.

Figure 4: The Data-Harvesting Attack Under Conditional Replication. (a) Single-share compromise probability over time by attacker with 25K simultaneous nodes, conditional replication with 4-h minimum replication interval, replication factors 3–11. Points labeled "Vuze Default" correspond to 20-way, every-30-minute replication. (b) VDO compromise probability with the number of replicas, 60 shares; we use the maximum threshold ratios that guarantees VDO availability for 8h. (c) VDO compromise probability with increased attackers.

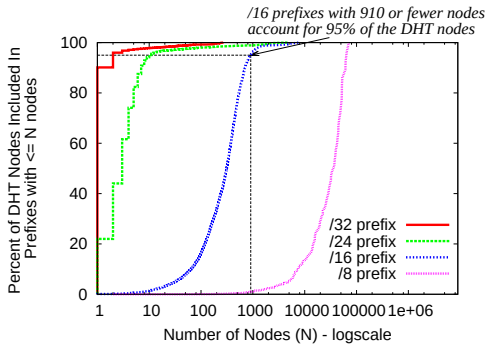


Figure 5: Evaluation of IP-Prefix-Based ID Calculation. Percentage of DHT nodes covered by prefixes (CDF). Remarkably, for /16s, prefixes with 910 nodes or fewer account for 95% of the DHT size.

participate in the DHT. If the number of DHT participants is reduced substantially, there would be too few nodes to share the storage load; as a result, the DHT could suffer and be easier to attack.

To examine this benefit-cost tradeoff, we collected data on DHT membership from a Vuze version server to which all nodes running the default Vuze application report. In a 24h period, we saw 1,842,628 nodes (IP-port pairs) that originated from 1,724,363 distinct IPs. We then quantified the change in DHT size for different prefix-based ID limiting parameters.

For the standard IP prefix lengths (/8, /16, and /24), Figure 5 shows the percentage of DHT nodes (y axis) that would be included by prefixes with a given maximum node limit (x axis). (For the /32-prefix line, we show the percent of DHT nodes that would be included by IPs with at most a certain number of ports/IDs.) The graph shows that for all prefix lengths except /8, the presence in the DHT of any specific prefix is surprisingly small. For example, /16 prefixes with 910 or fewer nodes account for 95% of the nodes in our trace. Similarly, /24 prefixes with 10 or fewer nodes and IPs with 2 or fewer ports each account for 95% of the DHT nodes.

These results suggest that harsh limits can be imposed on the number of nodes that come from each prefix. We choose our limits conservatively based on these results as follows: $k_4 = 5$ (at most 5 IDs from any IP); $k_3 = 50$ (at most 50 IDs from any /24 prefix); $k_2 = 2{,}500$ (at most 2,500 IDs from any /16 prefix); and $k_1 = unlimited$ (an unlimited number of IDs from any /8 prefix).[15] Overall, choosing these values reduces the number of distinct IP-Port pairs in our trace from 1,842,628 to 1,575,786 distinct IDs, a reduction in DHT size of only 15% (which we believe is an overestimate due to the 24-hour-long trace). Thus, harsh per-prefix restrictions will have only a marginal impact on DHT performance, but will significantly toughen the DHT's ability to withstand Sybil attacks.[16]

5.3.3 *Further DHT Improvements: Size Matters*

The size of a DHT can itself be a defense against data harvesting. In general, the larger the DHT, the more nodes and other resources (IPs, storage, etc.) required by the adversary. We describe two size-related issues.

First, the existence of firewalls and NAT devices has an impact on a DHT's apparent size. Using measurements collected by Vuze servers, we know that about half of the participating nodes sit behind firewalls or NATs. While Vuze implements a NAT traversal mechanism, it applies it only to BitTorrent traffic and *not* to DHT traffic. The absence of NAT traversal does not prevent the neighbors of a firewalled node from reaching it; the node pings its neighbors periodically, opening up holes in its firewall for two-way communication with its neighbors. However, nodes that are far away in ID space (i.e., most nodes in the DHT) will find the firewalled node inaccessible and drop it from their routing tables. As a result, the DHT "looks" half its actual size from a direct-put perspective, but will replicate data to all of its nodes during periodic replication. For the attacker, this means that direct `puts` are twice as profitable as they should be. For this reason, we believe that enabling NAT traversal would be another effective defense against data-harvesting.

More important, we note that all analysis presented here is based on Vuze, whose size is estimated to be around 1M simultaneously connected nodes [22]. Recently, other DHT systems have grown enormously in size, e.g., measurements estimate the uTorrent DHT

[15] Limits on /8s are also possible and result in only slightly larger DHT size reduction. However, given that the threat from the few companies owning /8s today is remote, we decided not to impose such limits at this time.

[16] We delayed deployment of the IP prefix restriction because it would have prevented us from mounting a large-scale attack, which was required for many of the measurements presented in this section. We have implemented and intend to deploy this defense in the near future.

to be over 5M nodes. Assuming that our availability and security results scale nearly linearly with DHT size (an assumption which may not be completely accurate, but which is not critical to our overall discussion), an attacker would require approximately $250,000$ *simultaneous DHT nodes* (Sybils) running year-round to compromise a Cascade-like system on uTorrent.

5.4 Summary and Synthesis

This section presented and evaluated a set of simple, strong defensive measures that a DHT can deploy to increase its resistance to data-harvesting attacks. The combined strategies of *limiting data dissemination* (through careful replication design) and *limiting DHT ID fabrication* (through an ID admission control mechanism) raise the bar for a DHT attacker by many orders of magnitude. In addition, the size of today's largest DHTs adds another 5-fold increase in attack complexity compared to Vuze.

We acknowledge that DHTs are open, complex structures and do not argue that they are invulnerable, especially in the context of other known or unknown attacks (described in Section 6). However, our results suggest that simple defenses work. Attackers have a number of deployment options available. First, they can *rent* machines from a cloud provider, such as Amazon EC2 or Rackspace. Second, they might *own* the machines used for the attack. In either case, one major roadblock will be the limited number of IP address spaces they can control. In particular, our results show that attackers need to fabricate in total at least $60,000$ *distinct* DHT *IDs* in order to have even a slim chance of capturing any given VDO (10^{-3}) in Vuze; this number increases significantly for uTorrent. Under our IP-limiting ID calculation policy, this means that attackers must have access to $12,000$ IPs scattered in $1,200$ distinct /24 IP prefixes and in 24 distinct /16 IP prefixes. Few attackers today have access to such resources without contracting with major international companies or ISPs. As a relevant data point, as of Feb. 2010, Amazon EC2 controls IP subblocks from only 9 distinct /16s [2], making the evaluated attack on EC2 impossible.

Also worth considering is the real dollar cost of running or renting resources, including computing, power, and networking. As shown in Section 5.3.1, attackers need to maintain $50,000$ simultaneous nodes continuously in the DHT (in addition to having access to 60,000 distinct DHT IDs); hence, they must pay for $10,000$ rented machines at all times (365 days a year, 24 hours a day) since each rented machine can generate at most five distinct DHT IDs ($k_4 = 5$). As a point of reference, if rental happens from an infrastructure provider with EC2-like pricing ($8.5c/h/machine$), then attackers' cost for the evaluated attack would exceed $7M/year. In contrast, the attack cost against the original Vuze was estimated at $5,000/year [49]. Our simple and practical measures result in a three order-of-magnitude increase in the cost of the estimated attack.

In the case of illegal rentals, such as botnets, the yearly cost will be much lower but may remain relevant given the minuscule yield one could expect from an attack against a single DHT key-storage system in Cascade. Attackers need to mount a *reliable, continuous (24x365) attack* in the hope of gaining the opportunity to blackmail or trap some individual – at the time unknown – in the future. This may well be beyond the capability of today's botnets, which are typically used for short-term, transient attacks (such as spam or DDOS attacks), and where the failure or loss of some of the botnet nodes during attack is irrelevant to the outcome. In addition, compared to using botnets for spam, the business model here is at best unclear. Finally, we note that attacking a DHT is not beyond the capability of a government, although again the value of this attack is unclear in the Cascade composition environment.

6. RELATED WORK

Protecting the Privacy of Past Data. Self-destructing data systems have been proposed before. Examples include the Ephemerizer family of solutions [30, 34], revocable backup systems [8], commercial products that support self-destructing emails [15], and Vanish [24]. Except for Vanish, all these systems require trust in one or a small set of dedicated centralized key-storage services. In contrast, Vanish shuns trust in any single centralized service and instead scatters key pieces over a decentralized P2P DHT for temporary storage. In this paper, we uniquely observe that these two approaches address complementary threats. We propose Cascade, an extensible architecture that allows combinations of these and other key-storage systems to deflect the union of these threats.

We also propose the Apache-based Tide key store, which strikes a balance between decentralized, open-membership P2P systems and centralized, closed-membership services. Tide is related to hyper-encryption, an information-theoretically secure encryption scheme proposed by Rabin [35], which leverages a decentralized collection of dedicated machines; these machines continuously serve random pages of data, where each page can be read at most twice. Tide differs from hyper-encryption in its goals (self-destructing data as opposed to information-theoretic secure communications between two parties sharing a secret key) and complexity (e.g., a recent implementation of hyper-encryption [26] describes a process through which two communicants must interactively reconcile which server pages they accessed, whereas Tide's use of threshold secret sharing creates no such need). Finally, Tide proposes and evaluates a concrete and lightweight implementation that can take advantage of the wide-spread deployment of Apache Web servers to facilitate adoption. Tide is thus a valuable component in a composite Cascade self-destructing data system.

The goals of self-destructing data systems share commonalities with many other cryptographic techniques, including forward-secure [7, 12], key-insulated [6, 18], intrusion-resilient [16, 17], and exposure-resilient [11, 19, 20] cryptography. Like self-destructing data systems, all of these techniques aim at ensuring confidentiality of past data in front of present attacks. They differ in their models and assumptions. Forward-secure and exposure-resilient schemes assume that an attacker has zero or partial visibility into past cryptographic state; our model places no such restrictions. Caching, backup archives, and the threat of legal actions might allow the attacker to either view past cryptographic state and passphrases, or force the user to decrypt his data. Key-insulated and intrusion-resilient systems also introduce new trusted agents or secure hardware, which we seek to avoid. Whereas self-destructing data systems target data that may be accessed asynchronously until the timeout, ephemeral key exchanges and recent advances like OTR [1, 10] are suited for online, interactive communications.

The threat model of self-destructing data systems may seem similar to that of DRM systems. However, the two must not be confounded: self-destructing data systems assume that end-users who have access to a VDO during its lifetime are *trusted*, whereas DRM systems target precisely untrusted users. This critical trust assumption is realistic in the context of self-destructing data (a user sending a sensitive email to another user will indeed trust that user not to save a cleartext copy of the data). DRM and self-destructing data systems should be thought of as orthogonal.

Security via Redundancy. The Cascade architecture observes that the security of a self-destructing data system can be escalated by combining two or more implementations of key-storage systems. The advantages of composing multiple systems is well-known, dating back to at least N-version programming [4], and with clear use

cases within computer security, e.g., electronic voting [32]. Threshold secret sharing [36], even by itself, has at its core the fundamental notion of distributing trust amongst multiple actors under the assumption that some – though not all – of the actors may fail or be untrustworthy. The advantages with multiple systems may be counteracted if the different systems exhibit the same mistakes or weaknesses [27]. Cascade mitigates this concern by imposing only a small generic interface (see the Cascade backend API in Section 3) and by employing enormously different, often orthogonal implementations of that interface, ranging from P2P DHTs to Apache Web servers.

DHT Attacks and Defenses. There has been a tremendous body of work on malicious DHT attacks as well as defenses against them. A comprehensive survey [39] describes most known attacks: blackholing routing tables, partitioning DHTs, corrupting data, Eclipse attacks, etc. Other works provide quantitative studies of the impact of such attacks [40, 45] and propose defenses [13, 38, 29]. The proposed defenses include mechanisms for securing routing table maintenance and message forwarding [13], robust lookups through a diverse set of nodes [14], techniques to prevent a particular region of the keyspace from being hijacked by adversaries [38], and resource allocation mechanisms to protect against DoS attacks [29]. Vuze currently employs a weak version of one of these defenses [14]; it uses 20-way path redundancy, which can protect against some forms of routing attacks. Vuze lookups can be further strengthened by using routes that maximize AS diversity [23].

The common theme in these attacks is that they are aimed at disrupting the DHT's functioning by degrading its performance, availability, or integrity. Such attacks are indeed the most relevant threats for traditional applications deployed on DHTs, e.g., torrent tracking and P2P file systems. While these attacks apply to DHT-based self-destructing data systems as well, a more potent threat is to harvest the data stored in the DHT without disrupting its operations. This paper examines data harvesting attacks and provides effective defenses against them. It is also worth noting that the data harvesting attack examined in this paper is distinct from node crawling [41, 42, 43], where measurement nodes infiltrate the DHT and perform repeated random-index lookups to obtain comprehensive node membership information. Data harvesting is fundamentally different from node crawls as the target data is "hidden" within a gigantic address space (e.g., its presence is not revealed by routing tables), and the attacker has to rely on either direct puts or replication events to harvest the data. Data-harvesting attacks have only recently been shown to be feasible on deployed DHTs [49, 44]. To the best of our knowledge, until this work, no defenses against such attacks have been deployed and measured on large-scale, live DHTs.

Finally, our paper also contributes a simple yet surprisingly effective technique for limiting identity fabrication, also known as the Sybil attack [21]. Many defenses have been proposed to combat Sybil attacks. These include strong identities minted by a logically centralized authority [21], computational puzzles and bandwidth contributions to make peers prove that they are not Sybils [9], and leveraging social networks [50, 28]. Unfortunately, none of these defenses have been adopted by today's DHTs like Vuze, in part because no real need was perceived in the context of existing applications, and in part because many of them were deemed too complex or heavyweight. We instead propose simpler measures that cap the number of DHT IDs that an attacker with limited IP diversification can create in a DHT. Our proposal relies on IP addresses as weak identities and separates service nodes from client nodes, i.e., anyone can obtain service from a DHT (`get` or `put` values), but only a limited number of clients from a given IP or prefix can serve as DHT nodes.

7. CONCLUSIONS

This paper presented several contributions to the state of self-destructing data systems. We described the Cascade architecture, an extensible framework for integrating heterogeneous key-storage systems. We presented Tide, an Apache-based key-storage system that combines the advantages of DHTs, such as wide-scale distribution, with advantages of centralized systems, such as resistance to crawling attacks. And we presented our extensive experiments with Vuze, demonstrating that a security-sensitive design can significantly raise the bar for attackers of DHTs, particularly for Sybil data-harvesting attacks. Overall, we believe that this work moves practical self-destructing data systems much closer to reality.

8. REFERENCES

[1] C. Alexander and I. Goldberg. Improved user authentication in off-the-record messaging. In *WPES*, 2007.

[2] Amazon.com. Announcement: Amazon EC2 public IP ranges. `http://developer.amazonwebservices.com/connect/ann.jspa?annID`, 2010.

[3] Apache. Apache performance tuning. `http://httpd.apache.org/docs/2.0/misc/perf-tuning.html`, 2009.

[4] A. Avizienis and L. Chen. On the implementation of N version programming for software fault tolerance during program execution. In *Proc. of COMPSAC*, 1977.

[5] BBC News. US mayor charged in SMS scandal. `http://news.bbc.co.uk/2/hi/americas/7311625.stm`, 2008.

[6] M. Bellare and A. Palacio. Protecting against key exposure: Strongly key-insulated encryption with optimal threshold. *Applicable Algebra in Engineering, Communication and Computing*, 16(6), 2006.

[7] M. Bellare and B. Yee. Forward security in private key cryptography. In M. Joye, editor, *CT-RSA*, 2003.

[8] D. Boneh and R. Lipton. A revocable backup system. In *USENIX Security*, 1996.

[9] N. Borisov. Computational puzzles as Sybil defenses. In *Proc. of the Intl. Conference on Peer-to-Peer Computing*, 2006.

[10] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *WPES*, 2004.

[11] R. Canetti, Y. Dodis, S. Halevi, E. Kushilevitz, and A. Sahai. Exposure-resilient functions and all-or-nothing transforms. In B. Preneel, editor, *EUROCRYPT 2000*, 2000.

[12] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In *EUROCRYPT 2003*, 2003.

[13] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. of Symposium on Operating Systems Design and Implementation*, 2002.

[14] G. Danezis, C. Lesniewski-Laas, M. F. Kaashoek, and R. J. Anderson. Sybil-resistant DHT routing. In *ESORICS*, 2005.

[15] Disappearing Inc. Disappearing Inc. product page. `http://www.specimenbox.com/di/ab/hwdi.html`, 1999.

[16] Y. Dodis, M. K. Franklin, J. Katz, A. Miyaji, and M. Yung. Intrusion-resilient public-key encryption. In *CT-RSA*, 2003.

[17] Y. Dodis, M. K. Franklin, J. Katz, A. Miyaji, and M. Yung. A generic construction for intrusion-resilient public-key encryption. In T. Okamoto, editor, *CT-RSA*, 2004.

[18] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In *EUROCRYPT 2002*, 2002.

[19] Y. Dodis, A. Sahai, and A. Smith. On perfect and adaptive security in exposure-resilient cryptography. In *EUROCRYPT 2001*, volume 2045, 2001.

[20] Y. Dodis and M. Yung. Exposure-resilience for free: The case of hierarchical ID-based encryption. In *IEEE International Security In Storage Workshop*, 2002.

[21] J. R. Douceur. The Sybil attack. In *Proc. of IPTPS*, 2002.

[22] J. Falkner, M. Piatek, J. John, A. Krishnamurthy, and T. Anderson. Profiling a million user DHT. In *Proc. of IMC*, 2007.

[23] M. J. Freedman and R. M. Tarzan. Tarzan: A peer-to-peer anonymizing network layer. In *Proc. ACM CCS*, 2002.

[24] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proc. of Usenix Security*, 2009.

[25] Google. Google defends against large-scale Chinese attacks. `http://www.techcrunch.com/2010/01/12/google-china-attacks/`, 2010.

[26] J. K. Juang. Practical implementation and analysis of hyper-encryption. Master's Thesis, 2009.

[27] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. 12(1):96–109, Jan. 1986.

[28] C. Lesniewski-Lass and M. F. Kaashoek. Whanaungatanga: Sybil-proof distributed hash table. In *USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[29] P. Maniatis, T. Giuli, M. Roussopoulos, D. S. H. Rosenthal, and M. Baker. Impeding attrition attacks in P2P systems. In *Proc. of ACM SIGOPS European workshop*, 2004.

[30] S. K. Nair, M. T. Dashti, B. Crispo, and A. S. Tanenbaum. A hybrid PKI-IBC based ephemerizer system. In *International Information Security Conference*, 2007.

[31] Netcraft. Web server survey. `http://news.netcraft.com/archives/2010/01/07/january_2010_web_server_survey.html`, 2009.

[32] P. G. Neumann. Security criteria for electronic voting. In *National Computer Security Conference*, 1993.

[33] News 24. Think before you SMS. `http://www.news24.com/News24/Technology/News/0,,2-13-1443_1541201,00.html`, 2004.

[34] R. Perlman. The Ephemerizer: Making data disappear. *Journal of Information System Security*, 1(1), 2005.

[35] M. O. Rabin. Provably unbreakable hyper-encryption in the limited access model. In *IEEE Information Theory Workshop on Theory and Practice in Information-Theoretic Security*, 2005.

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

[37] R. Singel. Encrypted e-mail company Hushmail spills to feds. `http://blog.wired.com/27bstroke6/2007/11/encrypted-e-mai.html`, 2007.

[38] A. Singh, M. Castro, P. Druschel, and A. Rowstron. Defending against Eclipse attacks on overlay networks. In *Proc. of ACM SIGOPS European Workshop*, 2004.

[39] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proc. of IPTPS*, 2002.

[40] M. Srivatsa and L. Liu. Vulnerabilities and security threats in structured overlay networks: A quantitative analysis. In *Proc. of Annual Computer Security Applications Conference*, 2004.

[41] M. Steiner and E. W. Biersack. Where is my Peer? Evaluation of the Vivaldi Network Coordinate System in Azureus. In *Proc. of Networking*, 2009.

[42] M. Steiner, T. En-Najjary, and E. W. Biersack. A Global View of KAD. In *Proc. of IMC*, 2007.

[43] D. Stutzbach and R. Rejaie. Understanding Churn in Peer-to-Peer Networks. In *Proc. of IMC*, 2006.

[44] D. Stutzbach, R. Rejaie, and Y. Guo. Large-scale monitoring of DHT traffic. In *Proc. of IPTPS*, 2009.

[45] A. Tran, N. Hopper, and Y. Kim. Hashing it out in public: Common failure modes of DHT-based anonymity schemes. In *Proc. of WPES*, 2009.

[46] U.S. National Counterintelligence Center. Annual report to congress on foreign economic collection and industrial espionage. `http://fas.org/irp/ops/ci/docs/fecie_fy00.pdf`, 2008.

[47] washingtonpost.com. Palin's Yahoo account hacked. `http://voices.washingtonpost.com/the-trail/2008/09/17/palins_yahoo_account_hacked.html`, 2008.

[48] WebProNews. Email being used more in divorce cases. `http://www.webpronews.com/topnews/2008/02/11/email-being-used-more-in-divorce-cases`, 2008.

[49] S. Wolchok, O. S. Hofmann, E. W. Felten, J. A. Halderman, C. J. Rossbach, B. Waters, and E. Witchel. Defeating Vanish with low-cost Sybil attacks against large DHTs. In *Proc. of NDSS*, 2010.

[50] H. Yu, M. Kaminsky, P. B. Gibbons, and A. D. Flaxman. SybilGuard: defending against sybil attacks via social networks. *ACM SIGCOMM*, 2006.