Volume 4, Issue 9, September 2014 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Cascade Architecture for Self Destructing Data System Using Vanishing Data Object

C. Saravanan[*]
Lecturer, Hi-tech Polytechnic College
Tirunelveli, Tamilnadu, India

E. Nisha
Research Scholar
Tirunelveli, Tamilnadu, India

P. G. Karunya
Research Scholar
Tirunelveli, Tamilnadu, India

Abstract:** There are an increasing number of services offering 'cloud storage' where you can upload documents, photos, videos and other files to a website to share with others or to act as a backup copy. These files can then be accessed from any location or any type of device (laptop, mobile phone, tablet etc.* ***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.

Keywords— Cloud Storage, Self destructing, Cascade, Tide

I. INTRODUCTION

As Cloud computing and mobile Internet are getting more and more popularized, In people's life cloud services are becoming important because of this. People are now and then requested to submit or post their personal private information through the internet on the cloud. When people post their data, on the cloud they hope that the security will be provided by the service providers to protect their data from leaking, so that invading of privacy will not be done by other people. Security of their privacy is on more risks as people rely more on internet and cloud technology. The systems or network must cache or copy the data that is being processed, transformed and stored. Because these copies are essential for systems and the network.. On the other hand, Cloud Service Providers negligence, hackers' intrusion or some legal actions by all this also privacy can be leaked. To protect people's privacy all these problems present formidable challenges.

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.

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 ApacheWeb servers. Like Vanish, Tide relies on multiple, 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.

Therefore, we performed an extensive measurement-based study of the Vuze DHT and designed a set of defenses to resist the Sybil dataharvesting 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.

II. SELF DESTRUCTING SYSTEM

Self Destructing data refers to the deletion of data after a particular period of time. Self-destructing data mainly aims at protecting the user's data privacy. The entire user's data and their copies become destructed after a specified time, without any user communications. In addition, the decryption key is also destructed after the user-specified time. Self destructing the data in a local system is easy. Destruction includes deletion of a file from a directory. In a cloud environment, one has to delete all the copies of data and it is difficult to know where these copies are stored, as cloud environments are managed and controlled by external entities.

A. THREAT MODELING

Threat modeling has a structured approach that is far **more** cost efficient and effective than applying security features in a haphazard manner without knowing precisely what threats each feature is supposed to address. With a random, "shotgun" approach to security, how do you know when your application is "secure enough," and how do you know the areas where your application is still vulnerable? In short, until you know your threats, you cannot secure your system.

While you can mitigate the risk of an attack, you do not mitigate or eliminate the actual threat. Threats still exist regardless of the security actions you take and the countermeasures you apply. Threat modeling can help you manage and communicate security risks across your team. Treat threat modeling as an iterative process. Your threat model should be a dynamic item that changes over time to cater to new types of threats and attacks as they are discovered. It should also be capable of adapting to follow the natural evolution of your application as it is enhanced and modified to accommodate changing business requirements.

B. THE VANISH 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 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.

III. CASCADE ARCHITECTURE

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.

A. DESIGN PRINCIPLES

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

1.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.

2.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.

B. CASCADE ARCHITECTURE

Figure 1(a) shows the high-level architecture of the Cascade multibackend 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.

The encryption key K is split into three shares (K1, K2, and K3), all of which are needed to reconstruct key K. Each share is then itself split into multiple sub-shares (K11, K12, 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 (K1, K2, and K3) 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.

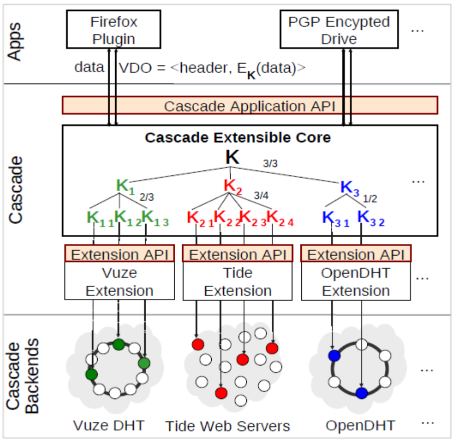
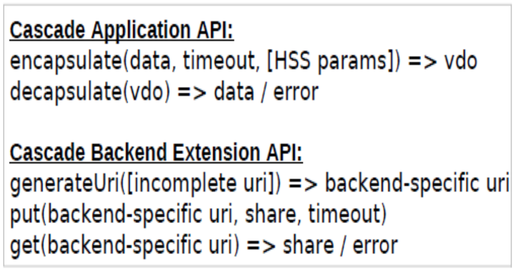


Fig.1 (a) Cascade Architecture

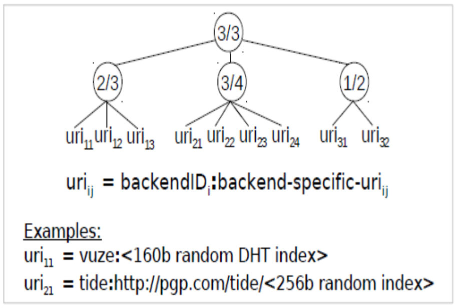
1. Cascade Key Storage Systems. The Cascade architecture is agnostic to these key storage backend choices; each backend simply stores (index,value) pairs and can retrieve each pair up to its specified timeout, after which that value can never be recovered. Cascade is designed to support timeouts ranging 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.


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

Fig.1 (b) Cascade APIs

2. 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.


3/3
2/3
3/4
1/2
uri$_{11}$ uri$_{12}$ uri$_{13}$
uri$_{21}$ uri$_{22}$ uri$_{23}$ uri$_{24}$
uri$_{31}$ uri$_{32}$

uri$_{ij}$ = backendID$_i$:backend-specific-uri$_{ij}$

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

Fig.1(c) Hierarichal Secret Sharing Tree in VDO Header

To be able to retrieve and compose key shares back into the original key K upon decapsulation, Cascade saves placement and secretsharing 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.

IV. 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 but are also more vulnerable to legal attacks and hacking. 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.

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

A. The Tide Apache Module

To maximize deployability, we designed and implemented a Tide prototype that leverages the widespread adoption of the ApacheWeb server and its modular structure. Our prototype is a small, lightweight, and dynamically loadable Apache module that allows clients to temporarily use an ApacheWeb 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.

1. Tide Apache Module Design. Our goal of simple and lightweight implementation requires us to reject complicated, heavy, or stateful protocolsOur 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 It is unlikely that a large proportion of key shares on disparate servers preserve residues for weeksor months after the timeout However, threshold secret sharing helps mitigate such losses probabilistically. Finally, since Tide relies on volunteer opt-in, we must ensure that our system remains unobtrusive to the server's general functioning.

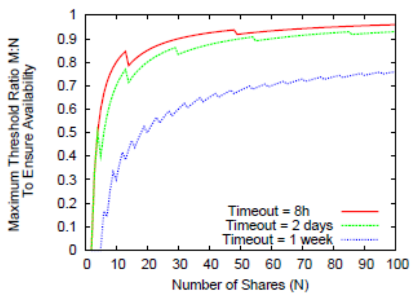
2. Integration with Cascade. Due to Cascade's extensibility, integration of the Apache-based Tide system is trivial. We wrote a 151-line Cascade extension in Java for the Tide backend that simply relays puts and gets from the Cascade core to the appropriate Tide Apache server. The server is directly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

V. EVALUATION AND PERFORMANCE

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.

A. 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. However, our goal is to find the ratio that provides the optimal tradeoff between security and availability. Higher ratios provide better security but result in lower availability (a smaller number of failures makes the VDO unavailable).



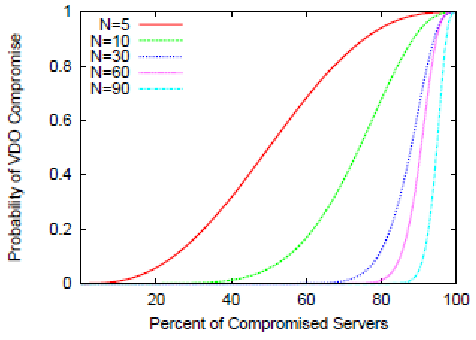
(a) Availability.

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.

B. 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 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 (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 (b), third curve from the left).

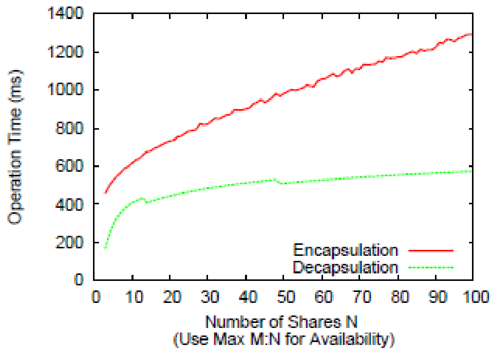


(b) Security.

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.

C. 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 (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.



(c) Performance.

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.

VI. CONCLUSION

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.

REFERENCES

[1] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy, "*Vanish: Increasing data privacy with self-destructing data*," in Proc. USENIX Security Symp., Montreal, Canada, Aug. 2009, pp. 299–315.

[2] A. Shamir, "*How to share a secret*," Commun. ACM, vol. 22, no. 11,pp. 612–613, 1979.

[3] S. Wolchok, O. S. Hofmann, N. Heninger, E. W. Felten, J. A. Halderman, C. J. Rossbach, B. Waters, and E. Witchel, "*Defeating vanish with low-cost sybil attacks against large DHEs*," in Proc. Network and Distributed System Security Symp., 2010.

[4] L. Zeng, Z. Shi, S. Xu, and D. Feng, "*Safevanish: An improved data self-destruction for protecting data privacy*," in Proc. Second Int. Conf. Cloud Computing Technology and Science (CloudCom), Indianapolis, IN, USA, Dec. 2010, pp. 521–528.

[5] L. Qin and D. Feng, "*Active storage framework for object-based storage device*," in Proc. IEEE 20th Int. Conf. Advanced Information Networking and Applications (AINA), 2006.

[6] [6] Y. Zhang and D. Feng, "*An active storage system for high performance computing*," in Proc. 22nd Int. Conf. Advanced Information Networking and Applications (AINA), 2008, pp. 644–651.

[7] T. M. John, A. T. Ramani, and J. A. Chandy, "*Active storage using object-based devices*," in Proc. IEEE Int. Conf. Cluster Computing, 2008, pp. 472–478.

[8] A. Devulapalli, I. T. Murugandi, D. Xu, and P. Wyckoff, 2009, *Design of an intelligent object-based storage device* [Online]. Available: http://www.osc.edu/research/network_file/projects/object/ papers/istor-tr.pdf

[9] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, W.-K. Liao, and A. Choudhary, "*Enabling active storage on parallel I/O software stacks*," in Proc. IEEE 26th Symp. Mass Storage Systems and Technologies (MSST), 2010.

[10] Y. Xie, K.-K. Muniswamy-Reddy, D. Feng, D. D. E. Long, Y. Kang, Z. Niu, and Z. Tan, "*Design and evaluation of oasis:An active storage framework based on t10 osd standard*," in Proc. 27th IEEE Symp.Massive Storage Systems and Technologies (MSST), 2011.