Proactively Secure Cloud-Enabled Storage

Karim Eldefrawy *
Information and Systems Sciences Laboratory
HRL Laboratories
Malibu, CA
kmeldefrawy@hrl.com

Sky Faber
Computer Science Department
UC Irvine
Irvine, CA
fabers@uci.edu

Tyler Kaczmarek
Computer Science Department
UC Irvine
Irvine, CA
tkaczmar@uci.edu

Abstract

Attacking cloud-enabled storage is becoming increasingly lucrative as more personal and enterprise data moves to the cloud. Traditional security mechanisms temporarily limit such attacks, but over a long period of time attackers will eventually find vulnerabilities; this can lead to compromising large amounts of valuable data and lead to large-scale privacy breaches. This paper addresses this problem by incorporating *proactive security* guarantees into cloud-enabled storage. Proactive security deals with an adversary's ability to eventually compromise *all involved servers* in a distributed storage or computation system. While there are several proactively secure secret sharing protocols that can be used to improve confidentiality of data stored in the cloud, their high overhead has traditionally limited them to less than ten parties and to only 100s of bytes typical for cryptographic keys. Realizing proactively secure cloud storage for larger data (e.g, MBs) requires careful design and calibration of system parameters, and faces several challenges. In this paper we design, implement and assess performance of the first system for Proactively Secure Cloud-Enabled Storage (PiSCES) of data larger than cryptographic keys. Based on our practical performance results we advocate that the high level of resilience and long-term security and confidentiality guarantees enabled by *proactive security* should be considered in future distributed and cloud-based storage and computing services.

I. Introduction

The number of Cloud Storage Providers (CSPs)[1] has grown significantly in recent years. According to the Cloud Security Alliance (CSA) [4], the two top threats to a CSP are *data breaches and data loss*. CSA [3] acknowledges that lack of reported incidents does not preclude occurrence and existence of such large breaches, as publicizing a successful compromise would damage consumer confidence in the CSP, leading to a decreased user base and significant financial loss. *Datalossdb.org* estimates that 28% of all lost data on the web is due to successful hacks, and 57% of these incidents are due to attacks from outside the target organizations.

In order to mitigate attacks, CSPs generally store the data in encrypted form, and use TLS/SSL to secure transmissions from/to storage application's client [12]; more secure CSPs and privacy-enabling add-ons allow encryption to take place on the client side [5]. We argue that standard security measures fall short of completely eliminating all risks of (large-scale) security and privacy breaches, especially in the long-term and by resourceful organizations and adversaries. If an attacker is able to obtain the encrypted data – either through physical access to the CSP's infrastructure, or by remotely exploiting vulnerabilities in any layer of such systems – then users' secret keys must be guarded indefinitely. If an attacker learns secret keys at a later time either through user carelessness or other means, they will be able to decrypt and recover, possibly, valuable and sensitive data. Additionally, if an attacker is able to use either their proximity to, or knowledge of the CSP's system to alter or damage the encrypted data, or if users lose secret keys, then the data will be lost permanently. On top of this, the attacker need only hold onto the encrypted data until the cryptosystem becomes weak or computing power increases to the point that the encryption is useless. This becomes an issue for data that has to be secured for decades, e.g., personal health records, and genomic information about individuals which is becoming increasingly affordable.

Proactively secure distributed storage and computation protocols [29], [7] mitigates such attacks and ensures long-term confidentiality in distributed systems. In the operation of proactively secure protocols, time is divided into separate periods (or rounds) and an adversary wishing to view or corrupt sensitive distributed (typically via secret sharing [33]) data has to perform all attacks *in the same period*. Between periods secret data is refreshed (randomized) and old data is deleted. This means that, in addition to the division of the system's lifetime as per the proactive model, the sensitive data is shared among n parties; between successive periods, the parties engage in an update protocol where some parties are rebooted, data shares are refreshed, and old shares are discarded rendering any knowledge of old shares useless. With proactive security, a mobile adversary (one that moves

*Currently at the Computer Science Laboratory at SRI International: karim@csl.sri.com

[1]We use Cloud Storage Provider (CSP) to denote an entity that provides (some or all of) the following services: storage, backup, and synchronization of personal and/or enterprise data.

between parties and components of a system at will) cannot use knowledge of a secret learned in a previous period in order to corrupt or compromise secret information in a later round. Instead the adversary must compromise a sufficient portion of the system in-between two successive update rounds to gain access the protected data.

Contributions: In this paper we demonstrate that proactive security can practically realize long-term confidentiality and high resilience guarantees necessary for future cloud-enabled storage. Specifically, we make the following contributions: (1) we design a system (PiSCES) for proactively-secure storage that has a very large threshold of corruptions (data consists of tens of secret shares distributed among tens of parties, i.e., virtual servers that could be potentially hosted at different cloud providers) and can handle large files (orders of magnitude larger than cryptographic keys and/or certificates which has been the main focus of prior work); (2) we prototype PiSCES and assess its performance and monetary cost for a variety of parameter choices and show that it scales to handle large files and tens of servers with today's resources. Our prototype shows that practical large-scale deployment of proactive security *is feasible with existing infrastructure and off-the-shelf computing*, and that periodically updating shares may provide considerable confidentiality and security guarantees that can significantly limit large-scale privacy breaches and leakage of sensitive data from the cloud. We note that the PiSCES system is designed to work with an arbitrary underlying Proactive Secret Sharing (PSS) scheme, but, as we discuss below, the feasibility of practical implementation of such a system for large data and a large number of parties relies heavily on the nature of the choice of the underlying PSS.

Envisioned Use Cases: We envision three deployment scenarios of PiSCES: (1) deployment across a single CSP, (2) deployment across multiple CSPs, and (3) deployment across a local enterprise server together with multiple CSPs. In each of these scenarios, we aim to design a user experience mimicking that of existing CSPs. A user's client connects to the system, and uploads one or more files to each server S_i, in secret shared form. At this point, the client could disconnect, potentially maintaining no state. While the client is disconnected, the servers periodically update the shares of the files. When the client wishes to retrieve a given file, it reconnects to each S_i requesting reconstruction of the stored file.

1) *Deployment on a Single Cloud:* This is the simplest deployment case, and reflects the deployment type implemented in Section VI. In this case, as detailed by Figure 1, all of the user's secret shares are given to a single CSP. This case is the most similar to typical Cloud Storage usage today, as it does not require that the user communicate with multiple CSPs. In this scenario confidentiality of the data is guaranteed even if a large portion of the CSPs infrastructure is compromised, as long as less than $\frac{1}{3}$ of the infrastructure is ever corrupted at the same instant.
2) *Deployment across Multiple Clouds:* PiSCES can also leverage access to multiple CSPs, as shown by Figure 2. In this example, we show a total of n shares split evenly across M different CSPs. As long as $M > 3$, the involvement of multiple CSPs allows the user's data to remain confidential even if the entire infrastructure of an individual CSP is compromised for some time. The confidentiality of the user's data only comes under threat if at least $\frac{1}{3}$ of all involved secret shares, S_i are compromised at the same instant. The only difference experienced by the user is that they must reconnect to each involved CSP to retrieve that cloud's shares when they wish to reconstruct a given file. In addition, in order for shares to refresh, the CSPs must communicate with one another without active prompting from an online user.

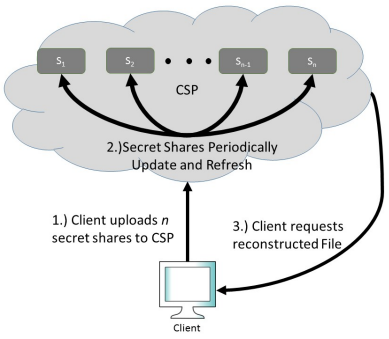

Fig. 1. A sample deployment across a single CSP

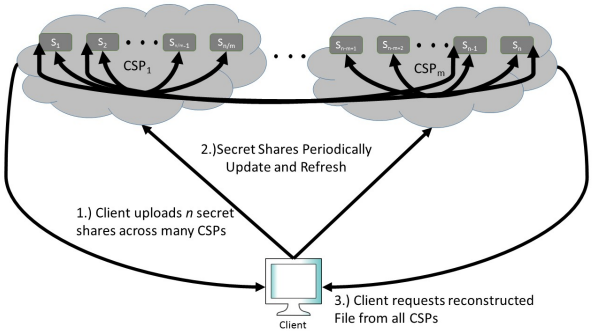

Fig. 2. A sample deployment across multiple CSPs

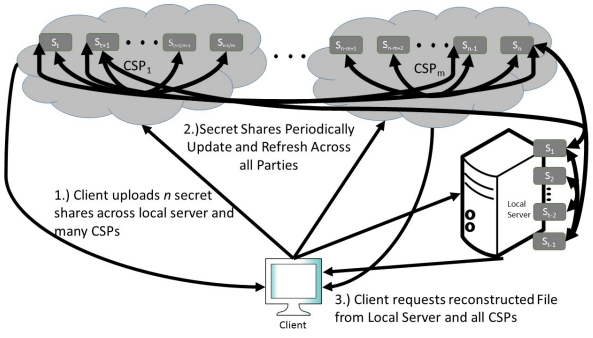

Fig. 3. A sample hybrid deployment with a local component and several CSPs

3) *Hybrid Deployment:* In addition to deployments that make use of a single CSP or multiple CSPs, a user can use local hardware in concert with any number of CSPs as detailed in Figure 3. Since the local server is physically accessible, we assume that it can be more easily hardened, or customized in general and is more trusted by the user. Because of this inequitable trust between the local system and the CSPs, the local server is given a disproportionately large number of secret shares. So now, instead of equally distributing n secret shares across M CSPs, the local server is given $\frac{n}{3}$ shares, and each CSP is given $\frac{2n}{3M}$ shares. This makes it so that the confidentiality of the user's data is only threatened if the entirety of the local server's infrastructure becomes compromised, as well as at least one secret share stored on one of the remote CSPs. In addition, just like in the other two deployment cases, data confidentiality is threatened if more than $\frac{1}{3}$ of all involved S_i are compromised across both the trusted local server and the remote CSPs. Assuming that the more trusted local server remains uncompromised, this equates to just over half the shares stored with the remote CSPs. The user experience in this deployment case is identical to a deployment across multiple clouds.

Paper Outline: The rest of the paper is organized as follows, Section II discusses related work, Section III overviews required preliminaries, Section IV presents the (PiSCES) system design, and Section VI the implementation; Section VII provides a detailed performance analysis and the results of our experiments, Section V contains the security analysis while Section IX concludes the paper.

II. Related Work

Cloud Storage Security: A comprehensive security analysis and comparison of commercial cloud storage providers (CSPs) can be found in [12]. Most CSPs encrypt stored data using symmetric encryption, e.g., AES-256, and secure transmission(s) between users and the cloud with TLS/SSL. Some CSP add-ons [5] allow users to encrypt their data, typically also with AES-256, before uploading it to the cloud. None of the above guarantees long-term confidentiality of data stored in the cloud. An adversary that slowly corrupts cloud nodes and eventually also obtains the symmetric encryption keys can collect all the data and decrypt it. Our main goal is to prevent such attacks, this is achieved by ensuring that as the secret shares of the data are refreshed old ones (possibly obtained by an adversary) become obsolete and useless. In addition, if a cloud storage node becomes compromised, the adversary will eventually be expunged once the node is rebooted and reloaded from a pristine image of the operating system during the next update step.

Secret Sharing and Proactively Secure Cryptographic Protocols: Secret sharing was introduced independently by Shamir and Blakeley in [33] and [10]. Our work builds on a packed variant [22] of Shamir's basic scheme to improve storage efficiency. The packed scheme shares multiple secrets using one polynomial by storing them as values of the polynomial at certain pre-set evaluation points, as opposed to at the free term as in the basic schemes.

The proactive security model and the first proactive secret sharing (PSS) scheme were introduced in [29]. Several other proactive secret sharing schemes that vary in assumptions and performance have been proposed since then [25], [13], [7], [8]. Section III-C discusses in details the roadblocks facing adoption of PSS for cloud-enabled storage. PSS is an essential building block for storage of large documents, but the performance has been the limiting factor for practical adoption.

Our implementation and experiments of PiSCES utilize the most efficient proactive secret sharing scheme in [7]. [7] has $O(1)$ (amortized) communication complexity per secret share and tolerates up to $< 1/3$ with perfect security guarantees, i.e., does not require any cryptographic assumptions[2]. In addition to proactive secret sharing, there has also been substantial research on proactively secure threshold encryption and signature schemes (e.g., [19], [20], [31], [14], [21], [11], [27], [6]); these are out of scope of this paper, combining such signature schemes with our system is an interesting avenue for future work.

III. Preliminaries

This section contains preliminaries for the rest of the paper.

A. Adversary Model

Our adversary model consists of an outside (to the cloud infrastructure) passive mobile adversary, also called honest-but-curious (HBC), or semi-honest in some of the literature. This adversary is seeking to obtain or violate confidentiality and privacy of the stored data and information through compromise of the CSP. This adversary is powerful and is able to corrupt up to $1/3$ of the servers constituting the cloud infrastructure in every round. The adversary is not restricted in which servers it is allowed to corrupt, and across many rounds every server may be assumed to have been corrupted at some point it time. If a server is corrupted during a update phase in between two rounds, it is considered corrupted during both the round immediately preceding and the round immediately after that update phase. Despite these strengths, the adversary is unable to monitor all traffic over the network of servers, i.e., it does not have global view, and is unable to delay packets that it did not send. This means that the adversary is further unable to corrupt internal routers or block communication between uncorrupted nodes without resorting to a traditional denial-of-service (DoS) of the entire system. This adversary can not spawn new server or nodes into the network, or corrupt multiple hypervisors. The adversary is also unable to cause Internet failure or orchestrate a DoS. Finally, while it is not a strict requirement, the adversary desires to remain stealthy.

[2]The work in [7] also describes a statistically secure version of the PSS scheme with $O(1)$ communication complexity. In this paper and in our system we only consider the perfectly secure version which is simpler to implement and provides the highest level of security, i.e., it does not assume any specific computational hardness assumptions.

B. Cryptographic Building Blocks

Secret Sharing (SS): We build on a long line of work in secret sharing [33], [9], [22] where polynomials are used to distribute shares of a secret among n parties. In order to share a block $\mathbf{s} = (s_1, \ldots, s_\ell)$ of ℓ secrets, a dealer chooses a polynomial $f(x)$ of degree d satisfying $f(\beta_j) = s_j$ and send $f(\alpha_i)$ to P_i. We say that the $f(\alpha_i)$ are *shares*, and that $f(x)$ *evaluates to* s_j at the secret values β_j. Here α_i and β_j are distinct public elements of a finite field. Our protocol will have perfect security, i.e., no computational hardness assumption, as long as $d \geq t + \ell$ where an adversary corrupts up to t parties.

Verifiable Secret Sharing (VSS): A secret sharing scheme is *verifiable* if parties can check that their shares are of a well formed secret. Some proactive SS schemes [26] use the VSS schemes of Feldman or Pedersen [17], [30] as subprotocols in the share update process. We use the more recent VSS techniques of [16], [15] where vectors of shares are multiplied by a hyperinvertible matrix (e.g., a Vandermonde matrix [16]). This is more amenable to performing computation on blocks of shares allowing us to greatly reduce the amortized complexity in our system.

Proactive Secret Sharing (PSS): The update of shares required by the proactive model provides two main technical challenges, namely refreshing old shares, and reconstructing lost shares to newly rebooted parties. The main PSS scheme we rely on [7] updates blocks of $\mathcal{O}(n^2)$ shares at a time, thus decreasing the amortized computation/communication complexity from $\mathcal{O}(n^2)$ (which is the best overhead in existing schemes, i.e., [25]) to $\mathcal{O}(1)$ per secret. In the refresh protocol of [7] each player deals temporary shares of a zero sharing (i.e., a polynomial which evaluates to zero at the secret values) using a modified form of the Vandermonde VSS. Players verify that the secrets are indeed zero and then add these temporary shares to their old share obtaining a new share of the same secret. By deleting their old share, they render knowledge of old shares useless.

Reconstructing lost shares is more complicated, and is the main technical challenge for any PSS scheme, as the parties need a way to reconstruct the shares of newly rebooted parties without reconstructing the entire secret and compromising privacy. The parties who hold shares of the secret polynomials use a modified form the VSS scheme to deal temporary shares of their own shares to other parties. After verifying that the shares are well formed, each party interpolates their temporary shares to obtain a polynomial which is a linear combination of the secret polynomials. To reconstruct the share of P_r, the parties each evaluate their polynomial at α_r and send the value to P_r who inverts the system of linear equations obtaining his share of the original secret polynomials. The share reconstruct protocol requires $\mathcal{O}(n^3)$ but reconstructs $\mathcal{O}(n^3)$ secrets at once, thus obtaining constant amortized complexity. A detailed explanation of the share reconstruct protocol can be found in [7]

Setting the Parameters: The parameters of interest are n, the number of parties, t the number of corrupt parties the system tolerates, ℓ the number of secrets we pack into each sharing, and d the degree of the polynomials used in the sharing. We must have $\ell + t \leq d$ for privacy and $3t + \ell < n$ for correctness and robustness. This makes $(t, \ell) = \left(\frac{n}{4}, \frac{n}{4} - 1\right)$ a natural choice of parameters. Increasing the threshold to $\frac{3n}{10}$ requires decreasing the packing parameter to $\frac{n}{10} - 1$. The performance of our scheme for a variety of parameter choices is shown in Section VI.

C. Roadblocks Facing Proactively Secure Cloud-Enabled Storage

One may think that between the Proactive Secret Sharing (PSS) schemes developed for constructing a proactively secure Certification Authority (CA) [25] and encryption (and signature) schemes [19], [20], [31], [14], [21], [11], [27], [6], the adaptation to the storage of larger data/files would be trivial. We argue that this is not the case. While many PSS schemes exist in theory [29], [25], [13], [7], to the best of our knowledge none have yet been tested with a large number of parties, i.e., tens of parties, and with large files, i.e., of sizes in the MBs compared to hundreds of bytes typical of cryptographic keys. In this paper we present the first scalable implementation of not only PSS, but of a deployable proactively secure cloud-enabled storage system, and perform experiments to demonstrate its practicality. We implement the PSS scheme of [7], as it is the only PSS scheme in the literature whose update protocol has constant amortized complexity in the number of parties/servers n (compared to quadratic in [25]). This allows us to let n be relatively large; we test $n = 30$ which gives us an appealing security threshold of $t = 9$. This means that, for an adversary to successfully compromise the system, it would need to compromise more than 9 nodes in the system in a single round. Finally, in passing from the theoretical results to an implementation, many of the underlying assumptions require special attention, such as:

1) *Secure Adversary Removal and Secure Disassociation:* All cryptographic primitives in the proactive model rely on the ability to completely remove the adversary from a server once it has been detected. Against honest-but-curious adversaries "restart" can be satisfactorily accomplished through the termination and immediate renewal of a virtual node.
2) *Secure Broadcast and Synchronous Networks:* Existence of a secure and synchronous broadcast channel is often assumed in secret sharing and MPC protocols [9] seeking to obtain security against a large fraction of corrupt parties. We assume only point-to-point communication, and simulate a synchronous broadcast channel following the work of [28] that shows that loosely synchronized clocks and assured message delivery with bounded delay can be used to simulate a synchronous network. Clock synchronization is easily achieved with minimal communication between the CSPs, and bounded delay is enforced by instantiating a time-out.
3) *Key Secrecy:* Our protocol requires authenticity and secrecy of messages sent between parties. In order to provide these properties servers must maintain private keying material so that when an adversary invades a server in round i he cannot use his access to forge or decrypt messages from rounds $j > i$. There are two typical methods for providing this type of key secrecy.

a) Each server can leverage a sophisticated trusted platform module (TPM) wich provides key secrecy even if an adversary has access to encryption and decryption oracles. This also requires the use of a synchronized clock to prevent future message forgery.
b) Securely replace encryption keys after every reboot. This requires that the hypervisor has the ability, after a machine restarts, to install fresh keys onto the server, and distribute these keys to the rest of the system efficiently.

Our design and implementation assumes the existence of a TPM, though it can be easily adapted to utilize a sufficiently powerful hypervisor.

IV. System Design

PiSCES is designed with the goal of addressing roadblocks outlined in section III-C, while remaining scalable. The system has three main components: the hypervisor, the networking stack, and the share storage hosts (S_i). For deployment on a single CSP , all of these components are managed the provider. In the case of hybrid or multiple cloud based deployments, each CSP and any local agent must maintain its own hypervisor and subset of the S_i. However, since all storage hosts are required to communicate with every other host, the entirety of the system must be connected such that each host is able to reach every other host.

A. Requirements of the Hypervisor

The hypervisor is the chief organizing agent of virtual machines on each CSP, and manages the creation, resource allocation, task execution and shutdown of virtual machine hosts. In order to achieve an efficient, cost-effective solution for the cloud provider, the hypervisor has a minimal set of functionalities required to maintain security. These functionalities must be implemented by the cloud service providers, and are requirements for PiSCES to function properly as an application running on top of such a hypervisor. Each functionality is easily realizable with little to no computation from the hypervisor. The hypervisor must support extensions to correctly manage periodic secure disassociation and restart, in addition the hypervisor to the additional security requirements described in SectionV.

Public Key Installation The hypervisor generates, signs, and installs a new key pair (public key/secret key, PK/SK) on each server immediately after it is rebooted. The freshly rebooted server then broadcasts its new, signed public key in order to rejoin the network. This is done to ensure PiSCES's *Key Secrecy*. This functionality requires little computation on behalf of the hypervisor, and many CSPs already provide a form of this service. For example, Amazon's Elastic Compute Cloud (EC2) provides public key generation and installation for newly created virtual machines (VMs) via calls to the API [1] . We point out that this functionality could be taken completely away from the CSP by requiring the user to be online during the update phase. However, we believe that for most settings, a practical implementation should not require the periodic appearance of the user, and so we push this simple task down to the hypervisor. Furthermore, hypervisor computation can be further reduced by the user uploading a list of signed PK/SK pairs ahead of time. These key pairs could then be refreshed periodically by the client (C) or a system administrator. In some environments (e.g. an enterprise) the use of a TPM could also be a viable option. However, this would be quite expensive on both the Cloud Provider and C, as providing dedicated access to a TPM from a VM could be costly.

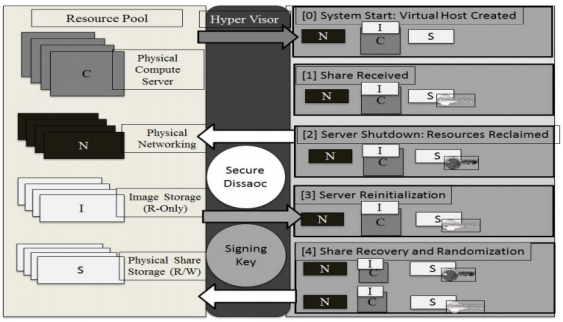

Fig. 4. The components used in each stage of execution. The Hypervisor allocates Physical Compute Servers (C), Physical Networking (N), Read-Only Images (I) and Read-Write Enabled Share Storage(S) in 5 steps: Host Creation, Share allocation, Shutdown / Resource Reclamation, Reinitialization and Share Recovery/Randomization.

Secure Reboot Reintegrating a reset host is a challenge in PSS. To solve this, a rebooted host must both be able to prove its authenticity and negotiate share recovery for all lost shares. This can be troubling as we need to ensure that an adversary cannot forge any step of the bootup procedure. If this is overlooked an adversary may compete with a new node for network acceptance, or worse, create an arbitrary number of corrupted servers replacing honest nodes in the process.

PiSCES employs the following safeguards in the hypervisor to aid in server reboot: First, when a new host is brought online all code needed for host's operation is loaded onto bare metal from a read-only image . Ideally, this read-only image can only be modified with physical access, however restricting modifications by newly rebooted host, S_r is sufficient. Second, the hypervisor will install a new signed key pair – using a hypervisor specific key – onto the server immediately after bootup. This key pair is then broadcast to the other S_i in the system, who in turn verify its authenticity and continue with protocol execution. The entire server lifecycle along with the boot process is shown in Figure 4.

Restart Schedule Rather than leveraging adversary detection mechanisms PiSCES reboots S_i on a predetermined schedule. The specifics of the schedule selection are discussed in Section VI-D. Due to this, the hypervisor must have the capacity to reboot instances according to said schedule. However, this does not require intimate knowledge of the system or protocol. Any restart schedule can be expressed in terms of absolute time, e.g., each S_i must be rebooted every day at hour i mod 24.

Secure Disassociation Since no Cloud Provider available today provides the ability to guarantee complete removal of

data, we assume the weaker property of "Secure Disassociation" instead. This property ensures that S_i cannot access short or long term storage from previous time steps in the protocol. For many Cloud Providers this is provided (but not guaranteed) simply by the fact that new virtual machines run on bare metal in a non predictable fashion. However, this may not be enough for a persistent adversary. Instead, we assume the hypervisor is capable of masking any previous memory usage patterns via virtualization. We acknowledge that current SaaS systems may not provide this today, but argue that it would not be difficult to provide in the near future. In addition, an adversary with physical access to the bare memory medium may still be able to infer sensitive data; however, such an adversary can be considered as corrupting the entire Cloud Provider.

Finally, we note that Secure Reboot completely disassociates data from the operating system and all of it's potential caches. Thus, any attempts at accessing stale data must go through the hypervisor.

B. Network Stack

PiSCES relies on the existence of a reliable point to point communication channel in between each S_i. Further, we assume a global bounded delay on all network packets. In our deployment the Internet connects each of the S_i. Each Cloud Provider supplies networking architecture (i.e. routers, gateways, etc.) connecting the S_i under its control. While the networking architecture could itself be compromised, an attack of this magnitude is out of the scope of this paper.

Reliability of each channel is provided by the Transmission Control Protocol. For low latency links, like those provided by Cloud Providers (close to the Internet backbone), 99.99% of packets are delivered in 31ms [18]. We assume a liberal maximum delay of one second (ten times the worst delay identified in [18]). For better performance this delay could be periodically adjusted to reflect actual round trip times between each S_i. Note, if all S_i are honest this parameter does not affect protocol delay. We use this delay to detect when a S_i has become unresponsive (due to failure or compromise).

C. Share Storage Hosts

Unlike traditional cloud storage systems, PiSCES requires periodic computation in order to refresh/randomize stored data, and protect against a mobile adversary. With refresh periods on the order of days, the S_i need not have quick access to all shares all the time. Instead, hosts store inactive shares on secondary storage. These shares are retrieved as needed and stored in RAM. We require that the secondary storage and the RAM from previous instantiations are no longer accessible after S_i reboots. A mechanism for secure deletion of individual memory locations is often used in PSS schemes to satisfy this requirement. Instead we rely on *secure disassociation*, or the property of a Cloud Provider that new VM's will (with significant probability) not have physical access to the same disk as VM's from other rounds. While "assured deletion" systems exist [34], [32] these systems still allow the recovery of encrypted data and such functionality is not readily available.

V. Security Analysis

In this section we analyze security of the design of PiSCES.

A. Hypervisor Resilience

While we assume the adversary cannot compromise the safeguards of the hypervisor, such an event could occur. In the case of a hybrid or multiple deployment, if only a single CSPs hypervisor is breached then the secrecy of the stored secrets may still be intact. However, recovering from such a corruption would require manual intervention. We further argue that even in the event of a hypervisor breach, many of the security properties provided by the hypervisor will still be intact. For example, an attacker who breaches the hypervisor once and is thus able to access additional S_i but is not able to permanently compromise the hypervisor may still be removed via the Secure Reboot functionality, as S_i selected for reboot as selected randomly and do not rely on any detection mechanism. This is especially likely if the features are provided by unmodifiable hardware. PiSCES remains secure as long as at any given timestep the adversary only corrupts a single Cloud Provider's hypervisor, and is unable to to interfere with either the secure reboot or secure disassociation mechanisms.

B. Secure Broadcast

The ideal design of PiSCES assumes a secure broadcast protocol such as the one detailed in [28]; our prototype relies on a stronger but realistic assumption about the network stack in an effort to remove this added layer of complexity. Specifically, we assume that the adversary can only influence network traffic from corrupted S_i. Among other things, this prohibits the use of disrupting communication between uncompromised nodes, we argue that this is a realistic assumption especially in switched LANs, as different physical hosts may exist in different collisions domains, or maybe even different VLANs if used in the cloud's networking infrastructure.

C. Corruption Rate

PSS protocols typically measure security by the fraction of parties that could be corrupted while still ensuring confidentiality of the secret and correctness of the protocol steps, e.g. the fraction $\frac{t}{n}$, where n is the total number of parties participating in the protocols, and t is the corruption threshold. In practice, however, we are more concerned with the corruption rate. The corruption rate is a measure of the speed (S_i/hour) at which an adversary must corrupt S_i in order to reconstruct the secret. It is chiefly determined by the corruption threshold, t, the length of each time step, and the rate of refreshing and rebooting. In an efficient implementation we wish to fine-tune the time step length and refresh and reboot rate for a given value of t to achieve a fixed corruption rate. As such in our setting t is a much stronger indicator of security than the conventional $\frac{t}{n}$.

D. Limiting Adversarial Movement

One chief design goal of PiSCES is to limit the adversary's rate of corruption. If each S_i is built identically then when a vulnerability is found in one host it is likely applicable to all hosts. If the exploitation of the known vulnerability is quick,

then an adversary can quickly gain control of all machines in the system. In an effort to alleviate this issue, we rely on code and system diversity. Each of the Cloud Providers has differing architectures, and independent virtualization technology. Further, we advocate the use of n-version design [24] or automated binary randomization [23] in order to further diversify each S_i. In our implementation, we only evaluated performance of a single CSP with some number of identical instances running on it. This is an undesirable configuration; a "real-world" deployment would necessarily want to utilize multiple Cloud Providers, or at the very least multiple instance configuration from a single Cloud Provider. An increase in diversity of deployment decreases an adversary's ability to leverage homogeneity to quickly compromise multiple parties from any insight gained by the initial compromise. That is, ideally an adversary who has compromised c parties would have to do the same amount of additional work to compromise $c+1$ parties $\forall c < t$, minimizing the advantage gained from having compromised an arbitrary party on a given Cloud Provider.

Naturally, in order to fulfill this goal it is ideal to introduce some large number of heterogeneous parties. This leads to a situation in which enforcing an ideal level of security is again facilitated by introducing a n arbitrarily large number of parties, and is at odds with the ideal corruption threshold:cost ratio for a static corruption rate discussed in Section V-C. It should then be the topic of future implementations of PSS to identify the tradeoff between diversity, corruption rate, and cost to attain a desired security level.

VI. Prototype and Experimental Setup

We implemented a prototype of PiSCES for the single cloud deployment case. We focus on this deployment case for two reasons. First, the use of a single cloud service provider allows us a finer grain of control in the selection of homogenous machines to evaluate the impact of the tuning of various parameters. Second, a multiple cloud deployment case would require communication between the multiple cloud service providers used. This would need to be facilitated through the modification of the cloud service providers' hypervisors and such modifications are outside of the scope of this prototype. Our experiments were conducted using a single Cloud Provider, Amazon's Elastic Compute Cloud (EC2). Amazon was selected due to the availability of dedicated machines. This greatly increases the repeatability of our experiments. The use of dedicated machines ensures that measurements were taken without any additional delays incurred due to time-sharing of machines with other customers. Experiments where conducted on three different instance types offered by Amazon: "Small", "Medium" and "Large"[3]; the specifications of each instance type is detailed in Table I. In addition to the listed price, there is an additional $2.00 fee per hour incurred any hour any instance is used. A practical deployment of PiSCES would not require the more pricey dedicated instances.

[3]Specifically m1.small, c1.medium, m1.large

TABLE I. Amazon EC 2 Instance Specifications. Note: 1 Gibibyte (GiB) $= 10^9$ bytes

Instance Type	CPU	Memory (GiB)	Storage (GB)	Cost per Hour (Dedicated)	Cost per Hour (Spot)
Small	1	1.7	160	$ 0.048	$0.0071
Medium	2	1.7	350	$0.143	$0.0162
Large	2	7.5	840	$0.193	$0.025

A. Testing Parameters

In addition to varying the type of EC2 machines utilized in the experiments, we vary the following parameters:

1) n: The total number of participating parties/nodes/servers.
2) t: The number of tolerated corruptions
3) s: The size of the file that we are protecting.
4) r: The number of parties/nodes/servers restarted simultaneously.
5) ℓ: The share packing (sometimes called batching) parameter.
6) b: The number of shares that are simultaneously refreshed in each refresh period.
7) g: The size of the underlying prime field, determined by the length of the prime defining the field.

The majority of these parameters originate from the underlying PSS scheme used [7]. s, r, and b are specific to PiSCES. All servers are eventually refreshed in every time period; r measures the number of servers restarted during a single reconstruction phase. b measures the number of threads processing shares on each server. For accurate measurement we showcase parameter comparisons using $b = 1$. We explored the affect of parallelization using Medium and Large instances as these each have two virtual compute units. We hypothesize that b would have a greater affect on server's with larger compute resources, however these systems were prohibitively expensive so we did not perform experiments using them due to financial constraints. Note that g is not a security parameter. It does however directly impact the size (and number) of shares, as well as the cost of some of the PSS cryptographic operations. We test g in powers of two from 256 to 2048.

B. Testbed Setup

While all of these parameters have some affect on the overall cost to rerandomize/refresh the secret shares of a file, their direct and indirect interactions and relations are not immediately obvious. We built an automated system in order to explore the empirical affects of the many parameter configurations. Our system interfaces with Amazon Web Services (AWS) to dynamically scale the number and type of virtual server as needed for the specific measurement. Our system then uses an outside "driver" machine to initiate the various protocol phases, and collect the resulting statistics. This driver machine operates asynchronously with the PiSCES prototype, and as such does not impact overall runtime complexity, or any other measurements. For each phase, we record the total time and communication bandwidth used on each S_i. For each experiment the driver records the results in a sqlite database for easier result exploration. Where necessary the driver also simulates the responsibilities of the hypervisor that AWS does not support.

C. Server Architecture

Figure 5 details the server architecture of each S_i. At the core, each server operates two dependent loops. One for event (command) processing and one for message handling (communication). When an event is received – one of (Set, Recovery, Update, Process Message, Reconstruct) – the even its passed to the identified share object which takes over processing. During operation, S_i expect to receive incoming messages from all other nodes, as such an active connection is maintained to each peer. When share update or rerandomization is received each proactive share object (which for some PSS may itself be several blocks of a file) distributes work among a pool of b processes. This is required in order to operate on batches of shares simultaneously[4]. Event handling, connection and thread management are written in python, while share recovery and randomization/refreshing are flexible and depend on the PSS. In the case of our experiments, using the PSS of [7], share recovery and randomization is built using Cython [2].

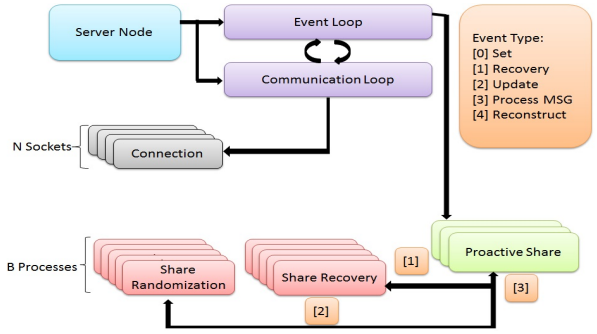

Fig. 5. Single Host's Control Flow. Messages are received from one of the N sockets and processed by the communication loop. Events are processed from this stream and passed to the event loop. Events of type [0-4] are sent to the identified Proactive Share object. Events 1 and 2 are forwarded to a process pool and processed independently. Events 0,3 and 4 are processed by the Proactive Share object internally.

D. Restart Schedule

Most PSS schemes do not strictly require that corrupted servers are removed from the system. Instead they rely on the assumption that adversaries do not corrupt more than t servers in any time step. In practice this assumption can be achieved in a variety of approaches, where each approach has its own security implications.

Existing PSS schemes [25], [7] include machinery to detect an active adversary (one that does not follow the protocol) and rely on this to detect corrupted servers. One could also rely on traditional anti-virus or attestation solutions in an effort to identify corruptions. Following this methodology, the hypervisor could focus on restarting servers that have been identified as corrupt. While this may result in fewer restarts and less work for the system as a whole (especially when no adversary exists), this would require heavy computation by the hypervisor. Additionally, many advanced adversaries are capable of avoiding detection. Therefore, such an approach would provide weak practical security.

Instead, PiSCES utilizes a restart schedule. This is simply a deterministic algorithm to determine when S_i will be rebooted. There are two types of restart schedules: randomized schedules, and complete (deterministic) schedules. Randomized schedules dictate an unpredictable set of S_i that must be rebooted each round. Due to this randomness, there are no guarantees on when, or if, a particular S_i will be rebooted. Due to this any system using such a scheme requires a stricter security assessment than the underlying PSS protocols. Complete schedules ensure that every server is rebooted every round. Many such schedules could be envisioned all with similar characteristics. In particular, we employ a round robin schedule. After every round the hypervisor iterates over the S_i initiating a secure reboot, and subsequently waiting for share recovery.

PiSCES uses complete schedules due to their stronger security guarantees. There is potential for a performance boost if randomized schedules are used, however an analysis is left for future work. Both schemes can be expedited by batching reboots together and rebooting r servers simultaneously. However, for security we require $r+\ell < n-3t$. Thus, determining the optimal tradeoff between r and ℓ becomes a very interesting direction – discussed in Section VII.

E. Lifecycle of Stored Data and Files

PSS deal in shares, however PiSCES ultimately stores and secures files. To remedy this we divide the file into shares via the following process:

1) Initially a user (the driver in our experiments) divides the file into blocks to be converted to packed shares and uploads these initial shares to each S_i.
2) The S_i then collectively update each share once within every time step. This is done by issuing a share rerandomization, followed by a series of r S_i restarts and share reconstructions.
3) Finally, the shares of the file are reconstructed upon request at the user, who then reassembles the file.

VII. Experiments and Performance Analysis

Our experiments were conducted by instantiating between 11 and 37 identical dedicated machines on Amazon's EC2 and evaluating the performance of the PSS scheme adapted from [7] as the aforementioned parameters are adjusted. Measurements are collected from all machines. When expressed as total time, we report the average time spent on each server (to represent latency). When expressed as cost we report the total cost of operating all n machines.

A. Experiments

Our results were obtained by evaluating the PiSCES's total time to refresh while modifying each of the system parameters. Due to space constraints, we showcase the most interesting relationships and configurations of parameters. In addition to examining the time to refresh under varying circumstances, we also look at the dollar cost to perform these actions. As the graphs make clear, the largest impacts on performance can be

[4]Processes are used instead of threads due to Python's global interpreter lock.

attributed to t and n, the number of allowed corrupted servers and total participants respectively.

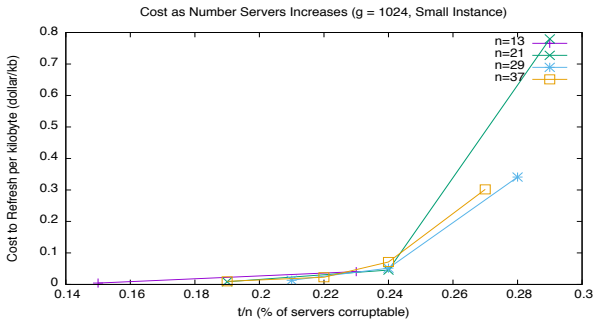

Fig. 6. Total cost to refresh for variable corruption threshold

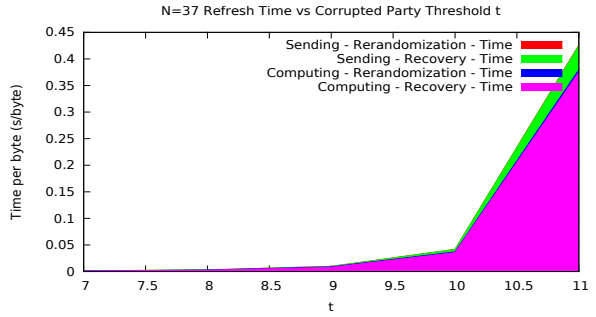

Fig. 7. Total time to refresh for variable corruption threshold with 37 Parties

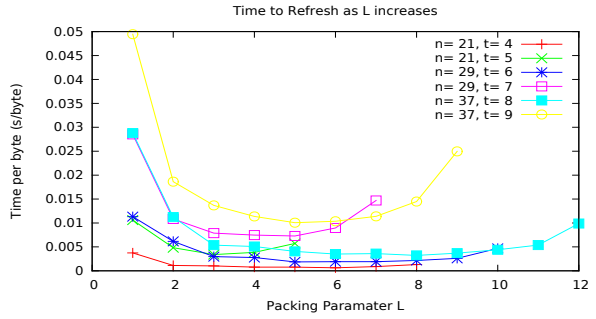

Fig. 8. Total time to refresh for variable Packing parameters

B. Performance Analysis

We examine the impact on performance of the manipulation of the 7 parameters named in Section VI-A. The largest impacts on performance can be attributed to the tradeoffs between the permissible corruption threshold $k = \frac{t}{n}$ and the packing parameters ℓ. As k approaches the cryptographic theoretical threshold, ℓ approaches 1, and performance degrades severely, causing a spike both in the fiscal cost incurred by refreshing and the total time to refresh, as clearly evidenced by Figures 6 and 7. This performance degradation continues asymptotically. The time to refresh during trials at the cryptographic threshold value $k \approx 1/3$ was prohibitively long,

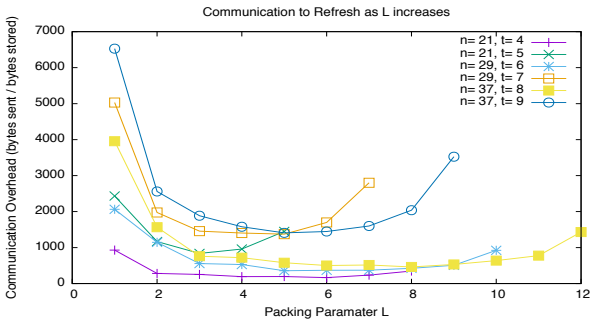

Fig. 9. Total communication overhead for variable Packing Parameter on multiple deployment configurations

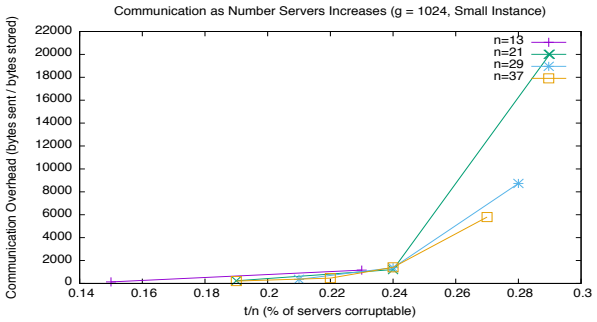

Fig. 10. Total communication overhead for variable corruption thresholds on multiple deployment configurations

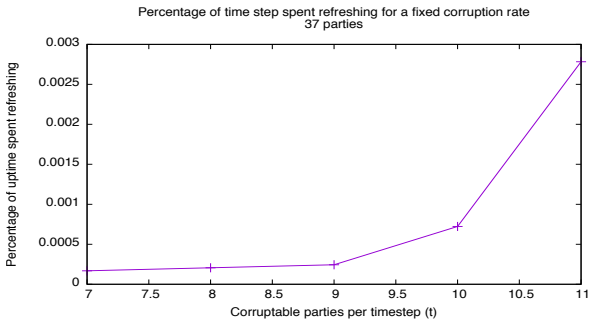

Fig. 11. Fraction of uptime spent refreshing for varying window size

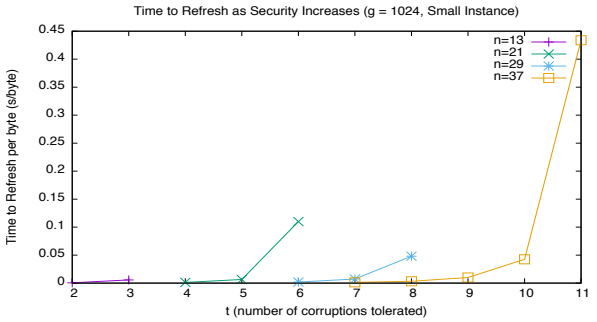

Fig. 12. Total time to refresh for varying number of tolerated corruptions

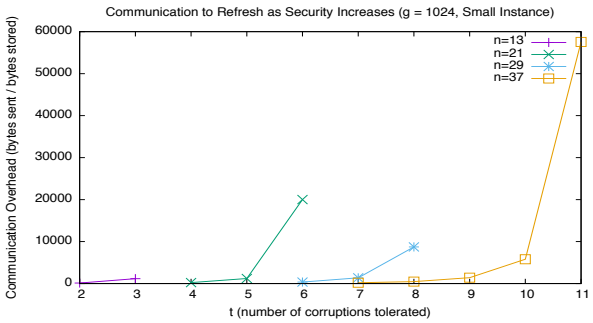

Fig. 13. Communication overhead for varying number of tolerated corruptions

and could not be included as data points in our figures. This illustrates the power of the packing parameter; a packing parameter of $\ell = 1$ fails to take advantage of the underlying PSS scheme's amortized complexity. This utilization of the underlying PSS seems to explain the trends as ℓ approaches its minimum, but interestingly, as evidenced by Figure 8, an increase of ℓ does not lead to a monotone decrease in cost, and instead raises the total time to refresh after a point. A similar trend can be seen in the total communication overhead. As Figure 9 shows, increasing the packing parameter is not a strictly beneficial thing to do, and some minima exist for different deployment configurations.

Contrary to expectations, n, the total number of servers, did not have a large impact on cost when varied with a constant corruption threshold, k. As shown by Figure 6 the corruption threshold is a much better indicator of cost than the number of servers being used. In fact, as n increased, the total time required to refresh as well as the total communication overhead are observed decreasing with a constant number of tolerable corruptions, t, as shown by Figures 12, and 13. As shown in Figure 11 even with a number of tolerable corruptions approaching its theoretical maximum, PiSCES spent under 1% of its total uptime actively refreshing.

The number of servers restarted each round, r, had a massive impact on the total time needed to refresh. Similar to the packing parameter ℓ, r was throttled when the number of tolerated threats t approached its theoretical cryptographic maximum of $\frac{n}{3}$ and helped to illustrate the power of the amortized complexity of the underlying PSS.

The size of the file being protected, s had surprisingly little effect on the overall performance across our test deployments. Increasing the file size from $100kb$ to $1mb$ resulted in a slight decrease in the time to refresh per-byte, as well as the fiscal cost per-byte. This decrease is primarily due to a reduction in padding needed when the file is packed into group elements.

When one instead considers a variable corruption rate, it becomes beneficial to utilize the maximum possible t with the largest number of corruptible parties possible. However, in a realistic implementation, a known threshold corruption rate that is feasible by an adversary is likely to be stated as a security parameter. It then becomes the case in a multiple cloud deployment that at least $\frac{n}{t}$ different Cloud Service Providers will ideally be used, as each should only be entrusted with up to t shares to provide adequate security. While this yields a reliance on more Cloud Service Providers than the theoretical upper limit on corruptible parities mandates, it facilitates a vast improvement in performance.As can be seen from the above analysis, the fine-tuning of these 7 parameters can influence the overall performance of PiSCES , especially the corruption threshold k and the packing parameter ℓ, and a one-size-fits-all approach will not work for ideal performance in arbitrary deployment scenarios.

VIII. Lessons Learned

Our experience in deploying PiSCES across a single CSP has shown us that proactive secret sharing is practical for more than cryptographic keying material and is possible today with only minor modifications to Cloud Service Providers' extant infrastructure. Once performed, these modifications do not incur a significant cost in either computation or storage. Additionally, even without these modifications PiSCES is still realizable by deploying on a local cloud. In particular we can store $10kb$ at 0.08 cents per kilobyte for each refresh. The feasible cost of such a deployment is due to recent advances in PSS schemes. Classical results do not meet the scaling requirements needed to make a system such as PiSCES efficient.

The difficulty in an efficient file storage scheme based on PSS is parameter selection. Optimal parameters are not intuitive to find and are often deployment specific. In this paper we use our sophisticated benchmarking system to systematically explore this parameter space. We found the most impactful parameters on performance to be the security parameters t – the number of corruptible parties per time step – and w – the time between share refreshes. In particular we find t=4, l=6, r=3, g=1024 to be the best selection of parameters for n=21.

Given a fixed security parameter number of hosts an adversary can corrupt per hour adding more hosts increases the overall efficiency of the system, decreasing monetary cost. However, as we add more hosts the pragmatic complexity of the system also increases. Dealing with a large number of virtual hosts is a challenge in its own right. Additionally, overall deployment costs increase with an increase in the total number of hosts, as maintaining a large fleet of idle servers quickly becomes expensive.

IX. Conclusion

This paper details the design, implementation and performance evaluation of a cloud-enabled file storage system that provides long-term security against the continuous compromise of servers in the cloud. It was shown that with the correct utilization of recently-developed proactively-secure secret sharing schemes with constant (amortized) computation and communication complexity in the number of participating agents, the storage of large files with a Cloud Provider in a proactively-secure manner is possible. This is the first feasibility study to consider proactive security for large data files. Until this point, proactive security was considered in the context of cryptographic keys and in certificate infrastructure settings.

References

[1] Amazon elastic compute cloud api reference, version 2015-10-01. http://docs.aws.amazon.com/AWSEC2/latest/APIReference/Welcome.html. Accessed: 2016-05-23.

[2] Cython c extensions for python. http://cython.org/. Accessed: 2016-05-23.

[3] Cloud security alliance, 2014.

[4] Cloud security alliance (csa) top threats report, 2014.

[5] Sookasa: Seamless dropbox encryption and compliance, 2014.

[6] J. F. Almansa, I. Damgård, and J. B. Nielsen. Simplified threshold rsa with adaptive and proactive security. In *Proceedings of the 24th annual international conference on The Theory and Applications of Cryptographic Techniques*, EUROCRYPT'06, pages 593–611, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] J. Baron, K. El Defrawy, J. Lampkins, and R. Ostrovsky. How to withstand mobile virus attacks, revisited. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 293–302, New York, NY, USA, 2014. ACM.

[8] J. Baron, K. El Defrawy, J. Lampkins, and R. Ostrovsky. Communication-optimal proactive secret sharing for dynamic groups. In *International Conference on Applied Cryptography and Network Security*, pages 23–41. Springer, 2015.

[9] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.

[10] G. Blakley. Safeguarding cryptographic keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 313–317, Monval, NJ, USA, 1979. AFIPS Press.

[11] A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *Proceedings of the 6th International Workshop on Theory and Practice in Public Key Cryptography: Public Key Cryptography*, PKC '03, pages 31–46, London, UK, UK, 2003. Springer-Verlag.

[12] M. Borgmann, T. Hahn, M. Herfert, T. Kunz, M. Richter, U. Viebeg, and S. Vow. On the security of cloud storage services. Technical report, 2012.

[13] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *ACM Conference on Computer and Communications Security*, pages 88–97, 2002.

[14] R. Canetti, R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Adaptive security for threshold cryptosystems. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 98–115, London, UK, UK, 1999. Springer-Verlag.

[15] I. Damgård, Y. Ishai, and M. Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *EUROCRYPT*, pages 445–465, 2010.

[16] I. Damgård, Y. Ishai, M. Krøigaard, J. B. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *CRYPTO*, pages 241–261, 2008.

[17] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 427–438, Washington, DC, USA, 1987. IEEE Computer Society.

[18] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S. Diot. Packet-level traffic measurements from the sprint ip backbone. volume 17, pages 6–16. IEEE, 2003.

[19] Y. Frankel, P. Gemmell, P. D. MacKenzie, and M. Yung. Optimal-resilience proactive public-key cryptosystems. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, pages 384–, Washington, DC, USA, 1997. IEEE Computer Society.

[20] Y. Frankel, P. Gemmell, P. D. MacKenzie, and M. Yung. Proactive rsa. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '97, pages 440–454, London, UK, UK, 1997. Springer-Verlag.

[21] Y. Frankel, P. D. MacKenzie, and M. Yung. Adaptive security for the additive-sharing based proactive rsa. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, PKC '01, pages 240–263, London, UK, UK, 2001. Springer-Verlag.

[22] M. Franklin and M. Yung. Communication complexity of secure computation (extended abstract). In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, STOC '92, pages 699–710, New York, NY, USA, 1992. ACM.

[23] M. Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms*, pages 7–16. ACM, 2010.

[24] L. Hatton. N-version design versus one good version. volume 14, pages 71–76. IEEE, 1997.

[25] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO*, pages 339–352, 1995.

[26] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Advances Cryptology in Cryptology-CRYPTO'95*, pages 339–352. Springer, 1995.

[27] S. Jarecki and N. Saxena. Further simplifications in proactive rsa signatures. In *Proceedings of the Second international conference on Theory of Cryptography*, TCC'05, pages 510–528, Berlin, Heidelberg, 2005. Springer-Verlag.

[28] J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *Theory of Cryptography*, pages 477–498. Springer, 2013.

[29] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, PODC '91, pages 51–59, New York, NY, USA, 1991. ACM.

[30] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, pages 129–140, 1991.

[31] T. Rabin. A simplified approach to threshold and proactive rsa. In *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '98, pages 89–104, London, UK, UK, 1998. Springer-Verlag.

[32] A. Rahumed, H. C. Chen, Y. Tang, P. P. Lee, and J. C. Lui. A secure cloud backup system with assured deletion and version control. In *Parallel Processing Workshops (ICPPW), 2011 40th International Conference on*, pages 160–167. IEEE, 2011.

[33] A. Shamir. How to share a secret. volume 22, pages 612–613, New York, NY, USA, Nov. 1979. ACM.

[34] Y. Tang, P. P. Lee, J. C. Lui, and R. Perlman. Fade: Secure overlay cloud storage with file assured deletion. In *Security and Privacy in Communication Networks*, pages 380–397. Springer, 2010.