Inkpack: A Secure, Data-Exposure Resistant Storage System*

Oceane Bel
UC Santa Cruz
Santa Cruz, CA, United States
obel@ucsc.edu

Kenneth Chang
UC Santa Cruz
Santa Cruz, CA, United States
kchang44@ucsc.edu

Daniel Bittman
UC Santa Cruz
Santa Cruz, CA, United States
dbittman@ucsc.edu

Darrell D. E. Long
UC Santa Cruz
Santa Cruz, CA, United States
darrell@ucsc.edu

Hiroshi Isozaki
Toshiba Memory Corporation
hiroshi.isozaki@toshiba.co.jp

Ethan L. Miller
UC Santa Cruz and Pure Storage
Santa Cruz, CA, United States
elm@ucsc.edu

ABSTRACT

Removing hard drives from a data center may expose sensitive data, such as encryption keys or passwords. To prevent exposure, data centers have security policies in place to physically secure drives in the system, and securely delete data from drives that are removed. Despite advances in security technology and best practices, implementation of these security measures is often done incorrectly. We anticipate that physical security will fail, and fixing the issue after the failure is costly and ineffective.

We propose Inkpack, a protocol that prevents an attacker from reading data from a drive removed from the data center even if the attacker has the user key linked to the data. An implementation of this protocol encrypts data, and secret splits the key over a number of drives. Recovering the key requires communicating with other drives, thereby denying access to the data if a few drives have been removed. Inkpack also requires the system to verify the validity of individual drives before normal operation. A prototype created within the Ceph storage system executed individual key split, key rebuild, and drive validation operations in 100–150 μs. We also show that our protocol is sensitive to small data write overheads, demonstrating potential performance gains if implemented on smart solid state storage devices, and propose a solution to increase performance.

CCS CONCEPTS

• **Security and privacy** → *Distributed systems security*; **Management and querying of encrypted data**; Access control;

KEYWORDS

systems security, vulnerability management, access control

1 INTRODUCTION

Routine removal of drives from a data center provides opportunities for data exposure. In the ideal case, replaced drives are securely wiped for resale, such as in an online marketplace, or disposal. However, a study found that 78 percent of drives bought on eBay or Craigslist contained residual data that could be recovered [16]. To guarantee no data leaks, a drive can be zero-written several times (though this is ineffective for flash [36]), or the platter or flash chips physically crushed or melted. Unfortunately, physical destruction prevents a data center from reusing or reselling the drive and the removed drive will contain data regardless of whether it was sold, lost or stolen. Several high profile drive losses have been reported where drives containing sensitive data such as nuclear information [29], personal information [17, 25], and medical records [21] have been *improperly* removed from a data center. In such cases, the attacker can be a disgruntled employee or an intruder stealing/switching drives.

In many storage solutions, implementing encryption is an attractive and simple fix. In cases where the data is encrypted and the user has the key, an attacker can use techniques such as man-in-the-middle attacks [20], phishing attacks [13], and coercion attacks [30] to get the key from the user, a single point of failure [1]. Some storage systems choose to not

*This research was supported by the NSF under grant IIP-1266400 and by the industrial members of the NSF IUCRC Center for Research in Storage Systems.

SYSTOR '18, June 4–7, 2018, HAIFA, Israel

ACM ISBN 978-1-4503-5849-1/18/06…$15.00
https://doi.org/10.1145/3211890.3211899

use keys to secure data because of this, and instead secret split data to gain security without keys [10]. However, secret splitting has a heavy network overhead and requires large chunks to be gathered in one place to rebuild the original piece of data, thus many data centers still rely on keyed encryption.

To address this problem, we developed Inkpack, a protocol that secures data by distributing parts of an encryption key over many drives. The Inkpack protocol is represented in this paper as software that runs on individual drives within a distributed storage system. The Inkpack software generates an encryption key using information from a user, such as a user secret, and a random key generated by the drive receiving the request. This key secures data on a per data basis to reduce the impact of a lost key. The Inkpack software does not save any key data in its entirety to persistent storage to prevent exposure of data. Instead, the random key is split and saved among many drives, of which a minimum amount of pieces are required to access data.

Doing so, an attacker can have the user secret and control the drive where the encrypted data is stored, but they will not be able to gain any knowledge about the information stored on the drive. The attacker needs to take a minimum number of drives in order to rebuild the *original* encryption key, making a successful attack detectable. Each individual drive also possesses a *unique* drive secret, initialized by the systems administrator, to prove their validity to other drives in the group. Drives that do not possess a valid drive secret will not receive accesses or be permitted to run software.

The main contributions of this paper are as follows:

- We describe a protocol that will prevent the attacker from gaining any useful data from a small number of drives even if the attacker gains the user key.
- We describe a system that verifies the validity of its drives through a challenge phase, and prevents an unauthorized drive inserted into the system from operating.
- We implement a prototype of our protocol and demonstrate that this added level of security has reasonable overhead.

2 THREAT MODEL

In our threat model, an attacker has no authorization to access data on the drive, but gains control of a few drives from a data center. They can be an employee of the data center who is disgruntled, coerced, or greedy. The attacker is not a system administrator or sophisticated enough to conduct a successful network or firmware attack in a short period of time. We assume that the firmware in the drives are verified and digitally signed.

An attacker could gain access to discarded drives by buying them online or from third party vendors. This attack is possible when the drives are not properly securely erased. In such an attack, it is hard to identify the attacker since the attacker is not actively monitored.

A different attack scenario is an attacker who enters a data center with the intent of *stealing* drives. The attacker has no intention to return to the same data center, and therefore does not care if they are identified. In case an alarm is triggered, they have the ability to escape with whatever drives they managed to obtain. Because the attacker may be caught while they are taking drives, they have a limited amount of time to conduct the attack.

An insider or an outsider could also try to intercept sensitive data by inserting a number of malicious drives into the data center. In this type of attack, the attacker plans to return to the same data center to retrieve the drives, and may have the ability to view the drive's contents for a brief period of time. Therefore, this attacker has little time to conduct the attack and does not want to get caught or identified during the initial attack. This attacker will have less time to successfully conduct the initial phase of their attack compared to the previously described attack.

All of these attacks share common characteristics. The attacker gains access to a limited set of drives containing data and key information and has unlimited time to try to recover user data. While the system can control the form of the data (encrypted vs. unencrypted) and the type of key information, it cannot control access to them by the attacker. This may be a problem for self encrypting drives since an attacker could gain the key that secures each drive. Thus our system must defend against such attacks.

Denial of service attacks by compromising drives and deleting shares are inherently inefficient. If an attacker already has physical access to a data center's drives and intends to destroy enough shares to ensure that data cannot be read, then a simpler attack would be total deletion by physical destruction. Doomsday scenarios where the entire data center is destroyed, the attacker physically destroys a majority of the drives, or a rogue administrator deletes data are outside of the scope of which Inkpack protects. We also assume that an attacker cannot simply clone a significant amount of the datacenter's data onto another drive within a short amount of time, and walk off with the other drive. Finally, in the event a data center's operating system drives are targeted for theft, we assume that the drives protect their sensitive information, such as network settings and private keys, with a security method such as UEFI Secure Boot [5]. Inkpack is not designed to protect against denial of service attacks.

We assume the network is trusted, and an attacker cannot get useful data from observing the internal network. Modern networks are point to point, therefore a message sent by a

sender will only be read by the sender's intended receiver. This network uses unicast messages that are not normally viewable by everyone. Finally, we assume that network hardware is trusted. Inkpack does not secure the network that the drives are running on and assumes that messages are not intercepted, diverted or modified during transit. We also assume that drives that pass the challenge behave properly and will not corrupt other drives.

3 DESIGN

In many storage systems, data in drive regions are fully encrypted with a large number of keys, and those keys are kept safe and hidden. To facilitate sharing data between users, many systems employ a *lockbox* style of encryption. Here, the real key is encrypted with a "lockbox key", and the lockbox key is widely distributed in the system. Lockboxes allow for easy re-keying of the lockbox, but in the case the actual key is lost, it will be expensive to re-key the entire system. According to Kallahalla *et al.* [15], a lockbox approach to securing keys allows for a more efficient and secure way of distributing sensitive data than simply encrypting data. However, improperly removed drives can contain the encrypted encryption key, making the lockbox key a single point of failure.

A straightforward approach to securing data without keys is secret splitting any data within a data center. Each individual piece of data, created from the original user data, does not reveal anything about the original data. Saving each piece on different drives provides security, as an attacker needs to obtain enough pieces of the original data in order to rebuild it. However, this approach adds significant storage and network overhead. AONT-RS [28] allows low-overhead splitting, but large chunks must be read from multiple drives. In POTSHARDS [34], each piece of data is split into N shares of the same size. Therefore, if a user wishes to save a 1 GB file, a secret split data store will actually store and transfer N GB of data, which is usually not an acceptable amount of overhead for security purposes, and thus many data centers rely on keyed encryption.

3.1 Inkpack

The Inkpack protocol is implemented as software, also referred to as an **Inkpack agent**, that runs on each drive. Each individual Inkpack agent is initialized with a unique **drive secret**, added by the systems administrator before the drive (with the agent) is inserted into the system. This software runs at the drive level and accepts incoming read and write requests from a client. It is also responsible for conducting the challenges between each drive to verify that each drive contains a properly initialized drive secret. Initializing the Inkpack agents with their drive secret is described in Section 3.6. We present a high level overview of the protocol in Figure 1.

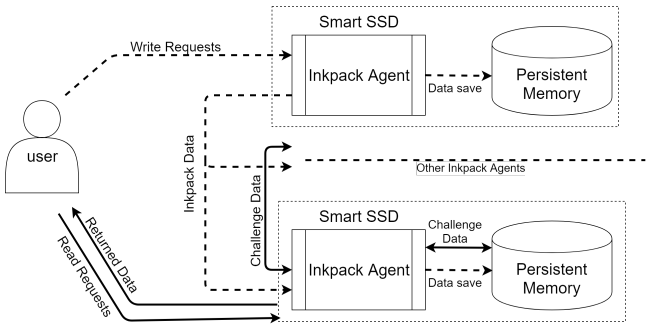

Figure 1: High level overview of the Inkpack protocol

During normal operation, an Inkpack agent receives a hash of the user secret and file ID when the system is accessed by a user. The user accesses the system using client software that hashes the user's information using a secure hash, such as a SHA-256, to preserve the security of the user's secret. Using a randomly generated key, called the **random key**, and the hashed user secret, the Inkpack agent creates a **user key**, which is used to encrypt and decrypt the user's data.

Inkpack reduces the vulnerability of the user key by secret splitting the random key into shares, and deleting the user key and random key from memory. Only the shares of the random key and the encrypted data are written to other drives in the system. The encrypted data is labeled by a hash of the file ID and the drive number that processed the data.

When encrypted data and shares of the random key are sent out to another drive, they will have a label appended to the end of the name stating that they have already been processed by an Inkpack agent. Once the encrypted data and shares are received, they are flushed to persistent storage. We take advantage of upcoming smart SSDs [32] to implement the Inkpack protocol outside of the host system the drives are mounted on, which lowers the network overhead of any security task done by Inkpack. Because the heaviest overhead in the design of the Inkpack agents is sending data to each other, having smart SSDs would allow each Inkpack agent to communicate with each other with less network overhead, thus improving performance.

3.2 Information within Inkpack

During a write request, the user submits data that they want to write to the system and this data is labeled with a hash of the file ID and drive number. This label is referred to as the **Object ID**, which links the data to a specific drive and user. Reading data requires the user to contact the original

drive that encrypted the data, and provide the file ID and their secret. Therefore, an attacker must not only control the drives that contain the encrypted data and the shares of the random key, but also the original drive that encrypted the data. Secret splitting the random key immediately after use, and never saving the random and user key, removes the vulnerability of losing any key within the system, as described in Section 3.4.

3.3 User information

When accessing the system, the user needs to provide to the system their secret, such as a password or a public key, and the ID of the data that they want to access. By default, each piece of data that the user submits is encrypted with a unique key, and the user's data is indexed using a hash of the file's ID. However, the user can also submit other information, such as their user ID, to index shares of the random key at a per user level, allowing the user to encrypt all their data with one key. This gives the system a performance and storage gain, since the system needs to create, manage and save less security data, however a determined attacker who steals enough drives to compromise a random key can gain an entire user's worth of data.

This information must be hashed client-side before it is sent to the storage system to prevent the existence of any plaintext user information in the storage system or Inkpack agents. In write accesses, the user needs to also give the system the data itself which will be encrypted by the Inkpack agent.

3.4 Key management

Inkpack uses keyed encryption to secure user data. The user keys are a combination of a **user's secret**, such as a password, and a **random key**, a randomly generated key. Requiring the user's secret and the random key to generate an encryption key prevents an attacker from gaining any useful information from a drive even if they possess the user's secret. When a random key is created, N shares are generated from the key, with each share the same size as the random key. m of these shares contain random data, and k of these shares are parity data generated from the random key and the m random data shares. Each individual share of the random key is sent to a unique drive. To rebuild the key, $m + 1$ drives, containing shares created from the *same* random key, are needed. Having fewer than $m+1$ drives does not reveal any information about the key, preventing the loss of a few drives from exposing data. Therefore an attacker would not only have to gain possession of $m + 1$ drives containing shares from the same random key, but also the corresponding user secret that was used to create the user key initially.

The same share creation technique is also used in a *trusted computing block* (TCB) [38] to generate the drive secrets from the cluster secret, a chunk of data initially created by the systems administrator in the TCB. The TCB is a different, closely monitored, computer system than the storage system the Inkpack protocol protects. It does not participate in normal storage operations, but is contacted when a challenge is conducted. When drive secrets are created, the TCB generates x random shares and y parity shares from the cluster secret. Since y represents the number of tolerable drive failures, y must be less than $m + 1$, since we do not want an attacker to go unnoticed in case enough drives are taken to rebuild the key. This is advantageous over public key authentication since shares of the key appear as encrypted data rather than as plaintext.

Inkpack uses Shamir Secret Sharing [33] as the technique for creating shares of the random key and cluster secret. This method is used to generate chunks of data from an original piece of data, of which all or some are needed in order to rebuild the original. Shamir Secret Sharing does not require every share to return to rebuild the original piece of data, allowing Inkpack to tolerate distributed system failures such as drive failure, network partition, and slow hardware. When generating the k chunks of parity data, Inkpack ensures that any share of the random key does not reveal anything about the original key.

Inkpack only secret splits the random key, not the data, in order to lower network congestion and storage overhead [3]. In a secret split data store, the system needs to store and transfer many chunks of data across many drives. By avoiding this, we lessen the storage overhead of security and reduce network load. Our protocol also separates key management from data storage in order to handle data and keys in parallel as proposed by Mazieres *et al.* [18].

When the shares of the random key are created, each is given a **share ID** to index the share. The share ID allows the protocol to find the shares of the random key once they are distributed. The share ID can be a hash of a file ID concatenated with the share number, allowing an implementation of this protocol to encrypt data on a per data basis. However, securing the key on a per data basis means that the system will have to keep track of a larger number of keys compared to securing the random key shares on a per user level, which can be done by using the hashed user's ID instead of the hashed file ID. During distribution, the shares are dispersed to many drives in the system. If there are more shares than there are drives, the protocol ensures that no one drive receives a majority of the shares. If there are fewer shares than drives, the shares are randomly dispersed in a way that not one drive gets a majority of the shares.

If an attacker wishes to steal data from a system of j drives, an attacker needs to steal at least $m + 1$ drives out of n drives

that contain a share needed to rebuild a random key. The probability of successful attack follows,

$$\frac{\binom{n}{m+1}}{\binom{j}{m+1}}$$

with j representing the number of drives and $m + 1$ the minimum number of drives needed to potentially get enough shares to reconstruct the key. The chances of a successful attack decreases dramatically the more drives are in the system.

3.5 Cluster Secret and TCB

The cluster secret is a piece of data generated by the systems administrator with a high entropy random number generator in the TCB. The cluster secret itself should never be saved or reconstructed by an Inkpack agent, and is only created to be secret split into shares, which are used as the drive secret. It must be large enough to allow our challenge phase enough room to choose a reasonable subset of the bytes created from the drive secret to verify it. This also allows the challenge phase to vary where the requested subset is taken from the drive secret, and how large the subset is. Since the challenging drive will not ask for the same subset every challenge, a malicious drive cannot replay previous responses to fool the requesting drive.

3.6 Drive secret

A **drive secret** is a share generated from the cluster secret, and used by a drive to prove its validity to other drives. Drive secrets are the same size as the cluster secret, and are loaded into unique drives before the drives are added to the system. Each drive uses that disk secret in order to get validated by other drives in the group. Drive validation prevents an unauthorized drive from intercepting data. Once a drive has been validated with a *challenge* phase, it is allowed to receive user data and validate other drives in the group. The locations of the drive secrets are kept in the TCB's persistent memory, preventing an attacker from taking a drive and locating the drive secret. We assume that the TCB is protected and more available than the system Inkpack protects. Because the drive secret is created from secret splitting, a visual inspection of the data would make a drive secret appear the same as encrypted data. The TCB must be contacted by a drive to find the drive secret during a challenge phase.

A challenge phase is only conducted when a drive is added or a new random key is written to the system. A previously validated drive, the challenging drive, queries the TCB to give back a minimum subset of the system's drive secrets. The challenging drive uses Store Forget and Check [31] to prove that any subsets of drive secrets that return are valid; other provable data possession techniques such as the one described by Ateniese *et al.* [4] could be used instead. Store Forget and Check is used as a proof-of-possession check, to prove that all drives in the data center contain the right drive secret initialized by the systems administrator. To verify that all drives possess valid drive secrets, our protocol verifies that the signature of the parities and the parity of the signature of the drive secret subsets that are returned match.

If both the signature of the parities and the parity of the signature match, then the drives have been validated. In the case a drive returns junk data or no data, it will not be validated and marked as dead, preventing an attacker from inserting malicious drives into the data center with hopes to gather key shares or encrypted data. Since the attacker cannot know beforehand where the drive secret is located in the drive and the firmware cannot be corrupted in the allotted time, the attacker is also prevented from taking a valid drive and moving the drive secret to a malicious drive to be inserted. In case more drives are added, the cluster secret needs to be re-computed and redistributed. Re-computing the drive secrets is not costly, as discussed in Section 5.1, and we believe that growing the system is an uncommon enough event that the recalculation overhead is minimal.

The systems administrator is able to toggle the number of incorrect or missing drive secrets the system can tolerate by raising or lowering the minimum threshold of drive secrets. This threshold allows the system to tolerate normal drive failures, and raises alarms in case more drives fail then the set threshold. The challenge phase has the added benefit of helping the system identify which drives are down, giving the system a better understanding of what drives may have been compromised.

3.7 Writes

When a user attempts to write data to the system, they will submit their user secret, file ID (and user ID in case the shares of the random key are indexed on a per user level), and the data that they wish to write. The Inkpack agent that receives the write request will find the shares of the user's random key, and rebuild the random key. The random key is then hashed with the hashed user secret to recreate the user key, which is used to encrypt the user's data. The users data is then written to the storage system, and the user key and random key are deleted from memory.

If shares of the random key for that user are not found, our protocol will run the challenge phase to verify the validity of the drives in the system. If the challenge phase determines that the system is healthy, it will generate a random key. Our challenge operation algorithm works as described in Algorithm 1.

When the system finishes verifying its drives, a new random key is generated and used to create the user key. After

Algorithm 1: Challenge operation

```
ChallengingDrive.GetSecretList(TCB)
forall the Drives in System do
    SecretList.append(TCB.GetSecret(Drive))
end
TCB.SendSecretList(ChallengingDrive)
PassList ← ChallengingDrive.Verify(SecretList)
if PassList.contains(Fail) = False then
    Continue
else
    Throw ChallengeFailureError
    return PassList
end
```

the user key is used, the random key is secret split and the shares of the random key are stored into the storage system. The write operation works as shown in Algorithm 2.

Algorithm 2: Write operation

```
HUS ← User.Hash(UserSecret)
HFID ← User.Hash(FileID)
User.SendWriteReq(System,HUS,HFID,Data)
System.AssignAgent(User.Request)
DataLabel ← InkAgent.Hash(HFID,InkAgent.Drivenum)
if InkAgent.FindRandKeyShares(DataLabel) = False then
    RandKey ← InkAgent.GenRandKey()
    UserKey ← InkAgent.Hash(RandKey,HUS)
    EncData ← InkAgent.Encrypt(Data, UserKey)
    EncData.Label(DataLabel)
    RandKeyList ← InkAgent.SecretSplit(RandKey)
    Delete HUS,HFID,UserKey,RandKey,Data
    forall the Shares in RandKeyList do
        TargetAgent ← InkAgent.ChooseRandAgent()
        TargetAgent.SendShare(Share)
        if Share.IsLastShare = True) then
            TargetAgent.SendData(EncData)
        end
    end
else
    RandKeyList ← InkAgent.RetrieveShares(DataLabel)
    RandKey ← InkAgent.RebuildKey(RandKeyList)
    UserKey ← InkAgent.Hash(RandKey,HUS)
    EncData ← InkAgent.Encrypt(Data, UserKey)
    EncData.Label(DataLabel)
    Delete HUS,HFID,UserKey,RandKey,Data
    TargetAgent ← InkAgent.ChooseRandAgent()
    TargetAgent.SendData(EncData)
end
```

The random key is hashed with the user's hashed information to create a **user key**, which is used to encrypt the user's data. After the encryption is completed, the random key is secret split into N shares, of which $m + 1$ need to return to rebuild the original random key. Each random key share is labeled with the hash of the user's information and disk number. The storage system uses this label to index the shares and save them to unique drives. Once all shares have been written to persistent storage, the encrypted data is written, and the random and user keys are deleted once the encrypted data is saved. In the event that the user writes the same data, such as when updating the file, the write operation will not create a new key, but rebuild the old one and encrypt with the existing key.

3.8 Reads

In a read access, the user provides the file ID of the data they wish to access, their user ID in case the shares of the random key are indexed on a per user level, and their user secret. The user's secret is hashed in their client software, and sent to an Inkpack agent. The Inkpack agent will communicate with the storage system to find the shares of the random key and the encrypted user data. Our read operation is described in Algorithm 3.

Algorithm 3: Read operation

```
HUS ← User.Hash(UserSecret)
HFID ← User.Hash(FileID)
User.SendReadReq(System,HUS,HFID,Data)
System.AssignAgent(User.Request)
DataLabel ← InkAgent.Hash(HFID,InkAgent.Drivenum)
if InkAgent.FindRandKeyShares(DataLabel) = True then
    RandKeyList ← InkAgent.RetrieveShares(DataLabel)
    EncData ← InkAgent.RetrieveData(DataLabel)
    RandKey ← InkAgent.RebuildKey(RandKeyList)
    UserKey ← InkAgent.Hash(RandKey,HUS)
    Data ← InkAgent.Decrypt(EncData, UserKey)
    Delete HUS,HFID,UserKey,RandKey,EncData
    InkAgent.ReturnData(Data)
else
    throw DataNotFound
end
```

Once the storage system returns the encrypted data and the shares of the random key, the Inkpack agent rebuilds the random key. Next, the Inkpack agent hashes the random key with the hashed user secret to rebuild the user key. The user key is used to decrypt the user data, and the decrypted user data is returned to the user. Once the decrypted data is returned, both the random key and the user key are deleted from memory.

If the system does not find enough shares of the random key, the system returns an error even if the encrypted data

is found. This is done to prevent an attacker from taking a small number of drives, and rigging them to act as a new storage system. If the user requests data not found in the system, they will also receive an error.

3.9 Distributed system failure protection

Distributed systems are vulnerable to drive failure, network partition, and byzantine faults. We account for drive failure by setting a reasonable threshold of data chunks that need to return before a key can be rebuilt. For example, if a group key is split into eight shares, with six random chunks and two parity symbols, seven out of eight data chunks must return, as discussed in Section 3.4. Our protocol leaves it up to the storage system to back up any encrypted data written to drives. In situations where a network partition occurs, the set threshold also allows for a number of drives to leave the network. We do not account for byzantine faults caused by an attacker in the software, on the network, or by faulty hardware.

3.10 Smart SSDs and Ceph

Since smart SSDs are not yet commercially available and our main focus was to implementing a version of this protocol, we have chosen to implement our approach in the open source Ceph [37] object storage system. Ceph stores data on virtual drives called Object Storage Devices (OSDs), and we use this functionality as an analogue to smart SSDs since OSDs provide computation at a level close to the drive. In the most recent release of Ceph (Luminous), the storage system offers three different services for key management and automated encryption. However, our design does not intend to secure just the Ceph storage system.

We use Ceph as a generic storage system, and treat it as if it did not have any security capabilities. Instead of relying on the internal cryptographic functionality Ceph provides, Inkpack uses SHA-256, AES-256 [7] and gferasure [24] as tools to efficiently perform the calculations it requires. Anything done in our Inkpack system prototype can be readily ported to other systems as long as the same or similar tools are used and the storage system our protocol is implemented on can index and store data. In order to benefit from hardware support of these tools, we implemented libraries that gave Ceph easy access to the tools.

Current research suggests that future SSDs will have the capability for users to program their own software onto the drives themselves [32]. Smart SSDs provide a performance gain if the Inkpack agents are implemented as software on disk. Kinetic drives today offer a network interface to a storage system, allowing a user to directly communicate with the drive. Jin *et al.* proposed a specialization on this idea, where an SSD was used (with hardware modifications) as a key value store to give applications a level of indirection between keys and data [14]. By separating the need to secure user information from the central processing units, a smart SSD based storage system can allocate computing resources to other tasks with no loss in security. However, malicious software can also be programmed into the smart SSD, which can expose data or be used to conduct directed attacks on users. Enabling smart SSDs to work together independent of the host system allows the drives to verify that other drives are not malicious.

4 IMPLEMENTATION

Our setup consists of six OSDs, two monitors, and two managers. We have four physical nodes, each one equipped with three hard drives. Two nodes are dedicated monitor nodes, and the monitors write their data to the hard drive. The last two nodes are equipped with three 500 GB 7200 RPM hard drives. The nodes communicate with each other on a 10 GiB switch, and are located in the same physical room to reduce network latency. These nodes are designated as the OSD nodes. Each OSD node runs three OSD daemons, and each OSD writes its data to its own unique hard drive, for a total of six OSDs. Journaling is also done in the same partition as the OSD writes to due to hardware limitations. Though it may be possible to run many OSDs on an OSD node, and have multiple OSDs write to a single drive, doing so may cause performance issues [27]. Ceph managers increase the availability of a Ceph cluster [26], and in our experimental setup also increased stability. Each OSD journals its data to a 1 GB journal in the same partition as the OSD writes to due to hardware limitations.

5 EVALUATION AND PERFORMANCE

To demonstrate the performance of our protocol, we created an automated client that writes 500 KB to our Inkpack prototype system and reads the same data. The amount of data generated by the client does not affect the latency of saving keys or Inkpack computation. Varying the size of the data to write only affects the performance of Ceph's native write operation and the time it takes to do cryptography, which are two benchmarks outside of the scope of our security protocol. Thus, we chose to save 500 KB as an example. Each time a drive in the system receives a new piece of data, it generates a new random key. The random key is hashed with the user secret to create the encryption key. The random key is split after encryption, and each piece of data created from the random key is saved to separate drives in the system. Write actions, when repeated many times, cause the largest latency overhead from our protocol implementation in Ceph. We also restarted the system several times to gather performance data on the challenge phase that runs during system

boot up. We measured execution time for sections of the Inkpack protocol as well as normal Ceph operations.

5.1 Secret split, rebuild and challenges

When the system initializes, the primary monitor creates the cluster secret, generates the six shares from it, and distributes one share to each of the six OSDs. Only the primary monitor does this step in order to avoid dueling monitors overwriting each other's cluster secret shares. The share distribution is done in memory to simulate a systems administrator manually loading a share of the cluster secret onto the drive before it is inserted in the system. Challenging the system does not significantly increase the overhead since, in our prototype, the drive secrets are stored in memory to simulate how smart SSDs would cache their secret for fast retrieval by other smart SSDs.

During a challenge phase, our system requires at least five of the drives to return a correct proof-of-possession before the system can proceed. If it does not return correctly, no data is written or read from the cluster. The challenges prevent a user from writing sensitive data to a compromised system and will alert the system administrators about possible missing drives. Since generation of shares affects performance minimally, having more shares than drives demonstrates no performance loss if more shares are generated than drives. These measurements have been taken during read and write operations done by our prototype. Inkpack will always need to split, rebuild, generate, encrypt or decrypt some data, requiring our implementation to incorporate high performance math libraries [35]. We simulated having the Inkpack agents communicating with the TCB by having it communicate to the Monitoring Agent. Then the Monitoring agent communicates with the other agents to gather the each drive secret. Once gathered the list of drive secret is sent to the challenging drive.

Once the challenge is done, we found that generating eight shares from the key and rebuilding it takes minimal overhead, as demonstrated in Table 1. Additional overhead is caused by transforming the share data into a format that Ceph can ingest. This overhead occurs when formatting the shares as strings that can be sent to the Ceph file system or converting the received share strings into galois field array. Secret splitting the random key into eight shares or rebuilding the random key takes 42.39–61.77 μs.

5.2 Reads and writes in our prototype

In our prototype, reading shares is completed faster than writing shares as shown in Table 2. The write decomposition is shown in Figure 2. Shares of the random key are submitted to be written to disk even if the data has not completed its write operation. To write the shares of the cluster secret to drive, we reused the write operations provided by Ceph. This ensures that any piece of data we wanted written would always be flushed to persistent storage, and not cached in memory.

Ceph attempts to coalesce writes into one large write over the entire system. If there are not enough operations in the operation buffer, the system waits until enough operations are ready to be executed. This causes many chunks of data to be written in a single write, thus giving us write times of 461 ms. However, each share takes 55.5 ms to individually write, if the coalesced write is broken down into individual parallel writes. Because each share of the random key is 32 bytes, we experience slowdowns from small writes to hard disk, a task difficult for magnetic drives to complete well. To remedy this, one may coalesce the shares into larger chunks, and write the chunks to storage. A system that wants to implement Inkpack should consider coalescing the share writes into a readable buffer.

The system will see an increase in latency if the system generates more shares. Each share takes approximately 55.5 ms to pass by the journal, write to hard drives and signal for the next share to start writing. When a new random key is generated the overhead of writing the shares to hard drives will take $N \times 55.5$ ms, with N being the number of shares generated by the system. Reading the shares in Ceph takes approximately 837.5 μs per share, which means that the overall time taken to read N shares from the system will be $N \times 837.5$ μs.

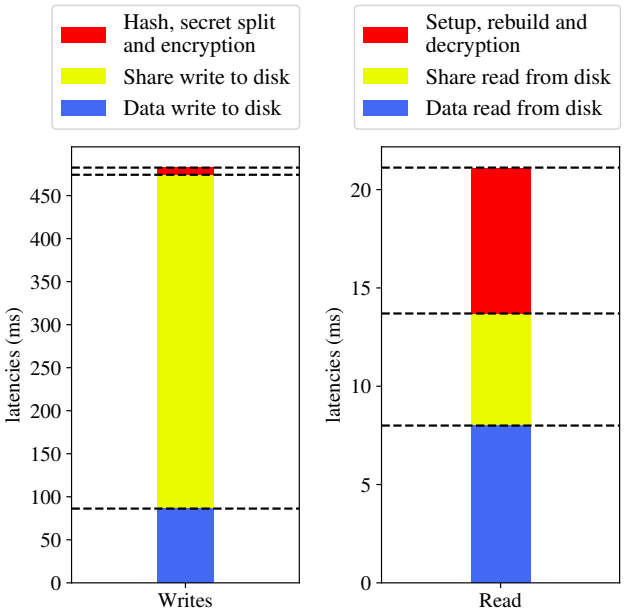

Figure 2: Breakdown of Inkpack overhead in Ceph read and write operations.

Table 1: Average execution time for Inkpack protocol components and Ceph components.

Cluster secret action	Execution time (μs)	OSD action	Execution time (μs)
Secret split	189.0 ±20.5	Challenge phase	80.52 ±2.51
Distribute	2.0 ±0.2	Secret split with share conversion	120.55 ±6.07
		Rebuild with share conversion	117.92 ±5.54

Table 2: Average time taken to transfer eight shares from persistent storage in Ceph.

Share Manipulation	Execution time (ms)
Total Write	433.2 ±47.6
Total Read	6.7 ±0.2

In the steady state, the protocol will not rewrite the shares of the random key during write accesses because the data accessed by the user already has an existing random key for that data. The approximately 433 ms latency caused by writing the shares and data in parallel to hard drives will be reduced to around 6.7 ms, thus reducing the overall write latency to around 110 ms. To illustrate this, one can expect during the steady state that write performance can be approximately the same as the read performance. During reads, the shares of the random key are read and processed before decrypted data is sent back to the user, as shown in Figure 2.

6 RELATED WORK

Many security protocols that aim to secure drives harness encryption and secret splitting. In recent work, these approaches either use one method or the other, but few have blended both of these tools into their protocol. Some approaches focus on physically securing the hard drives in the system using burglar proof casings, and assume that other software approaches are implemented to secure the data on the drive. By combining many approaches, our approach makes drive theft unproductive.

6.1 Software approaches

AONT-RS [28] is a technique to secret split data that does not require an external key in order to decrypt or encrypt data. It encrypts data on the file level of granularity with a randomly generated (and never saved) encryption key. The key is appended to the encrypted data after it is XORed with a hash of the encrypted data. This creates an AONT package, which can be secret split using an information dispersal algorithm for storage. In this manner, the keys are stored alongside the data, and can never be lost unless the data is lost along with the keys. The authors cite the fact that their current scheme may be memory intensive in comparison to other secret split data stores such as POTSHARDS.

Openstack Barbican [19] is a secure key management system for distributed file systems. For a distributed file system to decrypt data, it must request the secret from Barbican, which validates the request from a keystone. Barbican handles distribution of the keys within the system, and only decrypts the data if it deems the access to be valid. Conditions to decrypt can be set by an administrator, which includes hardware validation if the system provides the service. In comparison to Inkpack, Barbican must run within the host storage system, not on the drives, and access to the Barbican server and keystone is reliant on the credentials of the user. The keystone and Barbican server may also run on hardware external to the storage cluster, similar to how Inkpack relies on a TCB.

Oceanstore is a infrastructure to create a globally replicated and available storage system. This system can be run on untrusted hardware, and protects its data with a combination of cryptography and redundancy techniques. Specifically, access lists and encryption keys are used. This leaves the user to secure their encryption key and to securely transfer key data before accessing the Oceanstore infrastructure. Other systems deal with encryption and decryption by having centralized modules [9] that deal with decrypting the encrypted encryption key. This approach is not scalable and may cause unnecessary overhead in large distributed systems.

Using blockchain to protect personal data [40] is another approach. Zyskind modified blockchain into an automated access control manager. This system focuses on guaranteeing that only the user and the service can access the data stored in the system. The user of such a system controls the permission that the service has when accessing their data, even revoking the service's access to their personal data. Their system allows the user to create as many *identities* as they want to reinforce their privacy. Our protocol uses group based key generation, which is similar to their approach. A user can generate multiple groups of data that are encrypted by different keys generated by different drives if they wish to.

6.2 Hardware approaches

Burglar proof drive casings [2, 6] have been created to keep the drive safe in data centers, but these approaches fall short

when faced with insiders that may have the key to the casing that the drives are stored in. Smart card based file systems [12] utilize hardware based authentication to encrypt data inside the storage system and secure access to and from the system. Smart cards are small processors mounted on plastic cards, similar to those mounted on a credit card. Leveraging this, smart cards can be used to ensure point-to-point encryption of data on a system. This grants a storage system high amounts of security at the requirement of the use of smart cards to access the system.

Hamlin *et al.* [11] describe in their patent a secure drive that takes the client drive ID and the secure drive ID, and creates a encryption key used to encrypt the user's data. The drive also can identify the validity of the encrypted message by generating an enable signal from the authenticator. That signal is then sent to the data processing unit, which then writes the data to drive or receives the cipher text for processing. This idea is similar to our protocol, except our authenticator is in software and focuses on identifying other drives rather then the client. Key generation is similar to our protocol, since we use a secret that comes from the user and a secret stored on the drive. In the case of drive theft our protocol allows for the systems administrator to update the secret that was originally distributed among the drives.

6.3 Possible performance improvements

Colgrove *et al.* [8] present an all flash enterprise grade storage system that supports compression, deduplication, and high-availability. In comparison with other storage clusters, their system leverages the abilities of flash memory while keeping parts of data in battery backed DRAM model. In doing so, latency in persisting data to storage lowers, and there is still the advantage of caching data in DRAM. In order to mitigate the flash penalty for random writes, they use a log structure index and optimal data layouts to ensure writes are executed in large sequential blocks. To improve the write times of the shares, our protocol would benefit from using a log structured index in order to coalesce the high number of writes that are needed for the shares of the key.

Parakh *et al.* [23] have attempted to reduce the network and storage overhead of secret splitting the user's data. They do so by generating shares of size $|S|/(k-1)$, where $|S|$ is the size of the original secret and k is the amount of shares created. Our protocol reduces the overhead penalty of secret splitting the entire data by only splitting the key. All of the shares of the keys would be 32 bytes, which takes up less space then having to secret split the entire user data. We decided to split only the key since we are looking to keep the user keys secure, as opposed to the splitting the entire user data. In cases where the attacker is motivated enough to steal enough drives to gain access to user data, the system will lose a large number of drives and report itself as unhealthy.

7 FUTURE WORK

For up and coming storage class memories, keys will need to be kept safe on systems especially during system shutdown. Unlike normal storage hierarchies, part of a key or all of it might be saved and viewable by an intruder who controls such a drive. The Inkpack Agent will need to erase all the data that have been regrouped on it during system shutdown.

Currently, the system generates eight shares for each user key that an Inkpack agent generates. Each share is 256-bits, which is not ideal for saving or indexing on normal hard drives or flash SSDs. One solution could be to implement a better indexing system for small data, such as a log or LSM-Trie [39]. This will help reduce the search overhead of the system when it needs to reconstruct the group keys accessed by the user, and improve read times for the challenge phase. Also, it reduces write amplification and the number of drive accesses needed to locate a specific piece of data. Reducing amplification becomes a concern on all flash or NVM systems, and tree like data structures such as a LSM tree [22] addresses this problem.

Finally, our Ceph implementation highlights the need to design a storage system with this protocol considered from the beginning. Ceph was designed for magnetic drives, and Inkpack benefits the best from SSDs. Ceph uses journals to ensure write consistency and availability, but there are techniques to achieve these goals for small pieces of data without high write amplification and latency such as logging. In future work, we aim to design and implement a new flash file system using the Inkpack protocol as its security base.

8 CONCLUSION

The Inkpack protocol adds a layer of security within a storage system capable of preventing data leaks from drive loss. Forcing the system to pass a minimum threshold makes the drives function as a group, where the group can tolerate the loss of a few drives, but a small number of drives cannot independently reveal data. By secret splitting keys, we increase a system's defenses against drive loss. Our protocol can be applied to any storage system in general, but future storage systems can take advantage of smart SSDs to lower the impact of the protocol on the system. With smart SSDs, any drive can be an Inkpack agent, and thus spread out the required computations over many agents. Our prototype works as a proof of concept, demonstrating that little computational overhead is needed to secure the encryption key. We seek to implement a full Inkpack file system in the near future.

REFERENCES

[1] Hal Abelson, Ross Anderson, Steven M. Bellovin, Josh Benalob, Matt Blaze, Whitfield Diffie, John Gilmore, Peter G. Neumann, Ronald L. Rivest, Jeffrey I. Schiller, and Bruce Schneier. 1997. The Risks of Key Recovery, Key Escrow, and Trusted Third-party Encryption. *World Wide Web J.* 2, 3 (June 1997), 241–257. http://dl.acm.org/citation.cfm?id=275079.275104

[2] Jae-Yeon Ahn. 2005. Theft prevention device for information-stored disk. (Aug. 23 2005). US Patent 6,931,895.

[3] Beimel Amos. 2011. Secret-sharing Schemes: A Survey. In *Proceedings of the Third International Conference on Coding and Cryptology (IWCC 11)*. Springer-Verlag, Berlin, Heidelberg, 11–46. http://dl.acm.org/citation.cfm?id=2017916.2017918

[4] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, 598–609.

[5] Jeffery Jay Bobzin. 2015. Secure boot administration in a Unified Extensible Firmware Interface (UEFI)-compliant computing device. (28 April 2015). US Patent 9,021,244.

[6] Kun-Fa Chang. 2003. Anti-theft compact disk casings. (Aug. 5 2003). US Patent 6,601,414.

[7] Robert Chesebrough and Gael. 2012. Introduction to Intel AES-NI and Intel secure key instructions. (26 July 2012). https://software.intel.com/en-us/node/256280#section1

[8] John Colgrove, John Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 2015. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components.

[9] Weishi Feng. 2010. Secure digital content distribution system and secure hard drive. (Jan. 12 2010). US Patent 7,647,507.

[10] Joel Frank, Shayna Frank, Lincoln Thurlow, Thomas Kroeger, Ethan L. Miller, and Darrell D. E. Long. 2015. Percival: A Searchable Secret Split Datastore. In *Proceedings of the 31st IEEE Conference on Mass Storage Systems and Technologies*. http://www.ssrc.ucsc.edu/Papers/frank-msst15.pdf

[11] Christopher L Hamlin. 2007. Secure disk drive comprising a secure drive key and a drive ID for implementing secure communication over a public network. (May 8 2007). US Patent 7,215,771.

[12] James Hughes and D Corcoran. 1999. A universal access, smart-card-based, secure file system. In *Atlanta Linux Showcase*, Vol. 10.

[13] Tom N. Jagatic, Nathaniel A. Johnson, Markus Jakobsson, and Filippo Menczer. 2007. Social Phishing. *Commun. ACM* 50, 10 (Oct. 2007), 94–100. https://doi.org/10.1145/1290958.1290968

[14] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *Proceedings of the 23rd Int'l Symposium on High Performance Computer Architecture (HPCA-23)*. IEEE, 373–384.

[15] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. 2003. Plutus: Scalable Secure File Sharing on Untrusted Storage.. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, Vol. 3. 29–42.

[16] Michael Kan. 2016. Used hard drives on eBay, Craigslist are often still ripe with leftover data. (28 June 2016). https://www.pcworld.com/article/3089343/security/resold-hard-drives-on-ebay-craigslist-are-often-still-ripe-with-leftover-data.html

[17] Erik Lacitis. 2017. WSU gets costly lesson in theft of hard drive with more than 1 million people's personal data. (July 2017). https://goo.gl/Ujr8wT

[18] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. 1999. Separating Key Management from File System Security. *SIGOPS Oper. Syst. Rev.* 33, 5 (Dec. 1999), 124–139. https://doi.org/10.1145/319344.319160

[19] Douglas MendizÃąbal, Ade Lee, Chad Lung, Dave McCowan, Fernando Diaz, John Wood, Juan Antonio Osorio Robles, Kaitlin Farr, Nathan Reller, and Steve Heyman. [n. d.]. Barbican. ([n. d.]). https://wiki.openstack.org/wiki/Barbican

[20] Ulrike Meyer and Susanne Wetzel. 2004. A Man-in-the-middle Attack on UMTS. In *Proceedings of the 3rd ACM Workshop on Wireless Security (WiSe '04)*. ACM, New York, NY, USA, 90–97. https://doi.org/10.1145/1023646.1023662

[21] LSU Health Network. 2017. Theft of external hard drive containing user information. (May 2017). http://www.lsuhn.com/healthnews/Theft-of-External-Hard-Drive-1

[22] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33 (1996), 351–385. http://www.ssrc.ucsc.edu/PaperArchive/oneil-actainformatica96.pdf

[23] Abhishek Parakh and Subhash Kak. 2011. Space efficient secret sharing for implicit data security. *Information Sciences* 181, 2 (2011), 335–341.

[24] James Plank, Kevin Greenan, and Ethan L. Miller. 2013. Screaming Fast Galois Field Arithmetic Using Intel SIMD Extensions. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*.

[25] Associated Press. 2017. 20,000+ tribal members warned of data breach. (2017).

[26] Inc Red Hat. 2016. Ceph-mgr Administrator's Guide. (2016). http://docs.ceph.com/docs/master/mgr/administrator/

[27] Inc Red Hat. 2016. Hardware Recommendations. (2016). http://docs.ceph.com/docs/kraken/start/hardware-recommendations/

[28] Jason K. Resch and James S. Plank. 2011. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST) (FAST'11)*. USENIX Association, Berkeley, CA, USA, 14–14. http://dl.acm.org/citation.cfm?id=1960475.1960489

[29] James Risen. 2000. Missing Nuclear Data Found Behind a Los Alamos Copier. (June 2000).

[30] Bruce Schneider. 1996. *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons.

[31] Thomas S. J. Schwarz and Ethan L. Miller. 2006. Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06) (ICDCS '06)*. IEEE Computer Society, Washington, DC, USA, 12–. https://doi.org/10.1109/ICDCS.2006.80

[32] Sudharsan Seshadri, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A user-programmable SSD. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*. 67–80.

[33] Adi Shamir. 1979. How To Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613. http://www.ssrc.ucsc.edu/PaperArchive/shamir-cacm79.pdf

[34] Mark W. Storer, Kevin Greenan, Ethan L. Miller, and Kaladhar Voruganti. 2006. POTSHARDS: Secure Long-Term Archival Storage Without Encryption. In *Technical Report UCSC-SSRC-06-03, Storage Systems Research Center, University of California, Santa Cruz.*

[35] Lincoln Thurlow, Andrew Kwong, Thomas J. E. Schwarz, and Ethan L. Miller. 2017. gferasure: a high performance Galois field library for erasure coding and algebraic signature computation. https://bitbucket.org/ssrc/gferasure. (2017).

[36] Michael Yung Chung Wei, Laura M Grupp, Frederick E Spada, and Steven Swanson. 2011. Reliably Erasing Data from Flash-Based Solid State Drives.. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, Vol. 11. 8.

[37] Sage Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. 2006. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*.

[38] Johannes Winter. 2008. Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms. In *Proceedings of the third ACM Workshop on Scalable Trusted Computing (STC '08)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/1456455.1456460

[39] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *Proceedings of the 2015 USENIX Annual Technical Conference*. http://www.ssrc.ucsc.edu/PaperArchive/wu-atc15.pdf

[40] Guy Zyskind, Oz Nathan, et al. 2015. Decentralizing privacy: Using blockchain to protect personal data. In *Security and Privacy Workshops (SPW), 2015 IEEE*. IEEE, 180–184.