Custos: Increasing Security with Secret Storage as a Service

Andy Sayler
University of Colorado, Boulder

Dirk Grunwald
University of Colorado, Boulder

Abstract

In the age of cloud computing, securely storing, tracking, and controlling access to digital "secrets" (e.g. private cryptographic keys, hashed passwords, etc) is a major challenge for developers, administrators, and end-users alike. Yet, the ability to securely store such secrets is critical to the security of the web-connected applications on which we rely. We believe many of the traditional challenges to the secure storage of digital secrets can be overcome through the creation of a dedicated "Secret Storage as a Service" (SSaaS) interface. Such an interface allows us to separate secure secret storage and access control from the applications that require such services. We present Custos: an SSaaS prototype. We describe the Custos design principles and architecture. We also discuss a range of applications in which Custos can be leveraged to store secrets such as cryptographic keys. We compare Custos-backed versions of such applications to the existing alternatives and discuss how Custos and the SSaaS model can improve the security of such applications while still supporting the wide range of features (e.g. multi-device syncing, multi-user sharing, etc) we have come to expect in the age of the Cloud.

1 Introduction

Security is hard, but it is also critical. As we continue to pour every facet of our day-to-day lives into a growing pot of cloud-based digital service, the security of these services becomes ever more important. Indeed, security may be the linchpin controlling the success or failure of "The Cloud" business model as a whole. We need look no further than the recent NSA leaks or Heartbleed bug to see the risks security failures pose. Many of the challenges underlying the creation of secure systems can be directly connected to the question of secure secret storage. From personal information to cryptographic keys, how can we securely store sensitive "secrets"? Furthermore, any secure secret storage system must support the wide range of use cases common today: multi-device syncing, multi-user sharing, etc. Unfortunately, users find existing "secure" storage systems difficult to use [31], developers find "secure" storage systems challenging to build [2], and using third party "secure" storage providers raises numerous questions of privacy and trust [11].

Take, for example, a cloud-backed file locker service like Dropbox [4]. Such services are extremely popular since they provide easy access to desirable features like cloud-backed multi-device syncing or multi-user sharing. But these features come at the expense of control over who can access your data: any data stored via Dropbox is accessible to the Dropbox corporation, anyone with subpoena power over the corporation (e.g. the US government), or anyone who gains unauthorized access to the corporation's infrastructure (e.g. malicious crackers). While Dropbox provides server-side encryption of data at rest and SSL encryption of data in transit [5], these methods deprive the user of exclusive control over and knowledge of the associated encryption keys, forcing them to trust Dropbox itself to safeguard their data. We could mitigate these risks by encrypting all our data locally (i.e. client-side) before sending it to Dropbox, but doing so would break many of the useful features Dropbox provides: e.g. sharing data with other users or syncing it across multiple devices now requires manual out-of-band encryption key exchange. As in many related situations, the challenge is one of secure secret storage and access control: how do we securely store and control access to the secret encryption keys used to encrypt data atop Dropbox while still enabling features like multi-user sharing or multi-device syncing?

The creation of a standard, dedicated "Secret Storage as a Service" (SSaaS) interface can ease many of the existing issues associated with securely storing secrets: it relieves developers of the burden of building ad-hoc secure storage systems from scratch, it provides users with the flexibility to pick SSaaS providers they trust, and it isolates the most sensitive components of our applications behind a dedicated, vetted, and centrally administered interface. In this paper, we present Custos[1]: our prototype SSaaS

[1] Custos is Latin for "Guard".

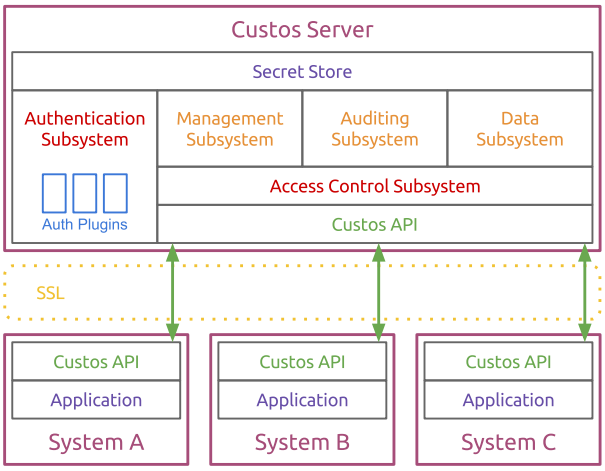


Figure 1: Custos Platform Overview

platform [26]. Custos extends traditional object storage semantics to add flexible, fine-grained access control and robust access auditing functionality to each stored object. These features make Custos an ideal system for implementing a secret storage service well suited for storing cryptographic keys (i.e. "Key Storage as a Service").

2 Goals

Custos (Figure 1) is designed around three core goals:

Decoupled: Existing applications are generally bundled with their own ad-hoc secret storage and access control components. This practice unnecessarily couples the general problem of securely storing and controlling access to various secrets with the specific solution a particular application implements. Such coupling has many downsides: developers are forced to re-implement non-trivial secret storage schemes for each application they create, these ad-hoc implementations are often un-vetted, and users or administrators are rarely able to flexibly modify the system semantics to accommodate situations the original developers failed to anticipate (e.g. new authentication protocol, etc). Custos breaks this unnecessary coupling and resolved these downsides by providing a dedicated service to store and control access to any secret an application requires.

Flexibility: Not all secrets are created equal, and neither are the means by which we must protect them. Some secrets must be shared with third parties (e.g. SSNs) while others should only be accessible to the original creator (e.g. GPG keys). Some data must be accessible to automated processes (e.g. file system encryption keys must be accessed by backup apps) while others must be kept in sync across a range of devices (e.g. private ssh authentication keys must be synced across a user's laptop and desktop). Any secret storage system that forces a single authentication and authorization model on all secrets is inherently limiting. Custos strives to overcome this issue by providing a flexible and extensible authentication and authorization framework capable of supporting separate security and usage models for each secret it stores. Furthermore, the standardization of the Custos interface provides users the flexibility to select from a range of Custos providers, shard their secrets across multiple providers, or even host their own personal Custos server. This provides the user with control over the degree to which they trust third parties with various secrets.

Auditing: One of the major challenges to using and sharing secrets like cryptographic keys is how to properly handle secret leaks or access revocation. Once a user has accessed a secret, there is no reliable[2] way to force them to "un-see" what they have "seen" (and potentially copied, photographed, etc). Thus, the effect of revoking a user's access to a secret is often dependent on the secret's access history. Since Custos provides a logically centralized point through which all secret requests flow, it is in an ideal position to provide an audit trail for any secret it stores. To facilitate this use case, Custos provides a framework for detailed secret usage auditing. Using the Custos audit data, an administrator can compute exactly what effect and guarantees a revocation operation will have: e.g. determining cases where guaranteed revocation is impossible because a user has already accessed and potentially copied a given secret. These auditing features are also extremely beneficial if a Custos-backed application is ever cracked: Custos's auditing functionality provides the information necessary to place bounds around the potential side effects a secret leak might incur.

3 Applications

The Custos architecture enables a range of use cases not readily available today. For example, Custos-based encryption systems can transparently encrypt files atop third party cloud services (e.g. Dropbox) while still supporting the multi-device sync, multi-user sharing features these services provide. Likewise, Custos can be used as the backing-store for a key-based authentication systems (e.g. SSH) by storing a user's private authentication keys for access from multitude of devices. In both cases, Custos

[2] At least within the range of ethically acceptable possibilities.

can make cryptographic applications more accessible and usable from both end user and a management perspectives. It can also ease the development burden of such systems by providing a dedicated key management service that relieves developers from the effort required to "roll their own" ad-hoc secret management. Custos allows users to isolate the trust they must place in a system to one or more dedicated SSaaS providers in charge of storing and regulating access to their secrets (e.g. cryptographic keys). These users may then leverage a variety of untrusted services relying on Custos-backed cryptography layers to sit between them and the untrusted service.

3.1 Encrypted File Locker

Cloud-based file locker services are one of the most common ways to sync files across devices and share them with others. Service like Dropbox [4] or Drive [9] are common in both the home and office. While convenient, such services have a significant downside: the cloud provider backing each service has full and unfettered access to each user's files. This makes such service inadequate for use in privacy-minded, high-security, or tightly-regulated domains. Services like SpiderOak [28] have been created to address the privacy and security concerns associated with traditional cloud file locker services, but even these "secure" file lockers suffer from privacy issues since the service provider generally retains access to the user's encryption keys in order to enable support for features like sharing data with other users [32]. An end user could manually encrypt all of their data or leverage a layered encrypted file system like eCryptfs [12] before pushing it up to a cloud file locker service, mitigating the privacy risk by personally maintaining control of all encryption keys, but such efforts would break the sharing and syncing use cases that make file locker services desirable in the first place.

To counter these issued, Custos can be used to build a client-side encryption layer atop traditional cloud-backed file lockers like Dropbox. Figure 2 shows the basic design of such a system. Similar to a service like SpiderOak, a Custos-backed encrypted file locker transparently encrypts all user data client-side before sending it to the cloud-based storage provider. This ensures that the storage provider never has access to any unencrypted user data. Unlike existing services, a Custos-backed system then stores the client encryption keys with a separate, dedicated SSaaS provider, ensuring that no single third party ever posses both the encrypted data and the keys needed to decrypt it. When a user wishes to access their files, they request the necessary encryption keys

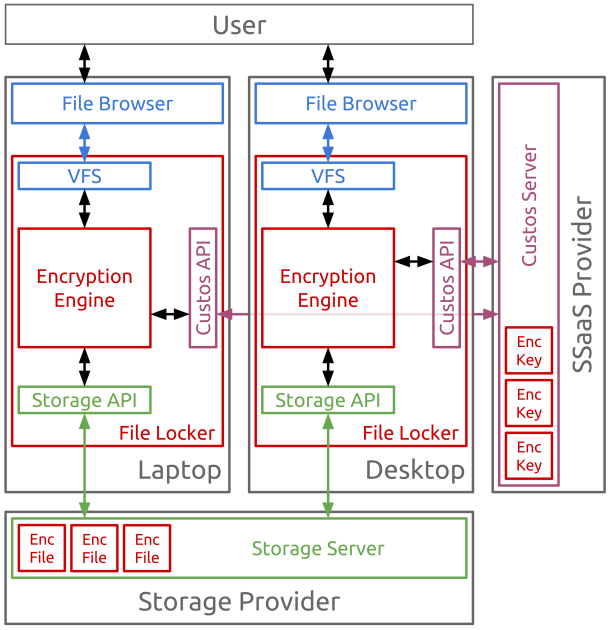


Figure 2: Custos-Backed Encrypted Cloud File Locker

from their SSaaS provider and the encrypted data from their storage provider, allowing the data to be decrypted locally and returned to the user. This model supports traditional file locker use cases like multi-device syncing: the user must simply grant each of their devices access to the necessary encryption keys via the SSaaS provider's management interface. Likewise, to share data with other users, the user must simply grant another user access to the appropriate per-file keys. This model allows existing cloud-based storage providers to continue providing basic data storage services while relieving end users of the need to trust such providers with the contents of their data, all while continuing to support modern day usage demands.

3.2 Multi-Device, Managed SSH Agent

The management of private cryptographic keys has long been a challenge for users. To help mitigate this challenge the systems community has developed the concept of an "agent" program. Agent programs sit between the user and an authentication system, providing the required cryptographic keys on the user's behalf to the authentication system when required [3]. Agents are commonly used with popular computing utilities like SSH [33] and GnuPG [15]. Unfortunately, existing agent solutions are designed for legacy usage models: single-device, non-portable desktop environments. They do not provide a mechanism for managing private keys across multiple de-

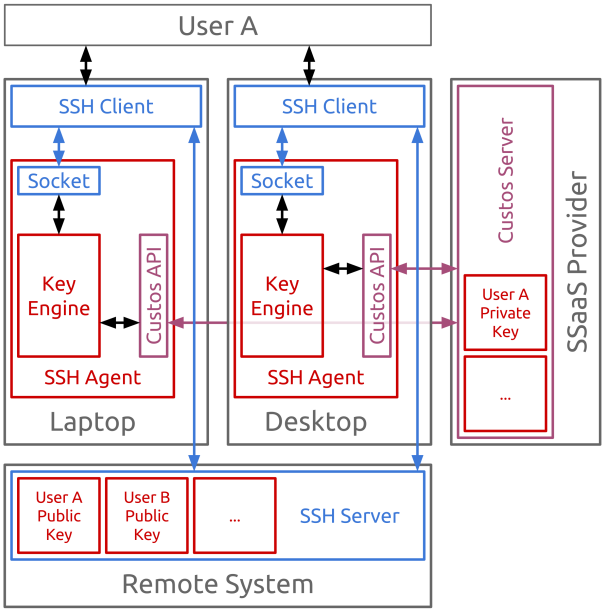


Figure 3: Custos-Backed SSH Agent

vices, securing keys if the associated device is lost or stolen, or managing keys for a large group of users across an organization.

The locality and management challenges associated with traditional cryptographic agent programs can be overcome by using a Custos-backed agent. Figure 3 shows the basic design for a Custos backed SSH-agent. Instead of storing private keys locally, such an agent would defer private key storage to a dedicated Custos SSaaS provider. When the agent requires the user's keys, it requests them from the SSaaS provider. Thus, the user and any associated agent programs are able to securely access the necessary private keys from multiple devices (e.g. laptop, desktop, tablet). Furthermore, if the user ever loses one of their devices, they can greatly reduce the risk of exposing any of their private keys but revoking the lost device's access to the off-site SSaaS data (e.g. similar to [8] and [30]). Such a system would also allow large organizations to manage SSH or other cryptographic keys for all their users from a centralized Custos management applications. Divorcing private key storage from local devices opens up a range of use case possibilities, increases security by keeping keys off of frequently lost or stolen portable devices, and relieves the user of the usability overhead required to manually manage their private keys.

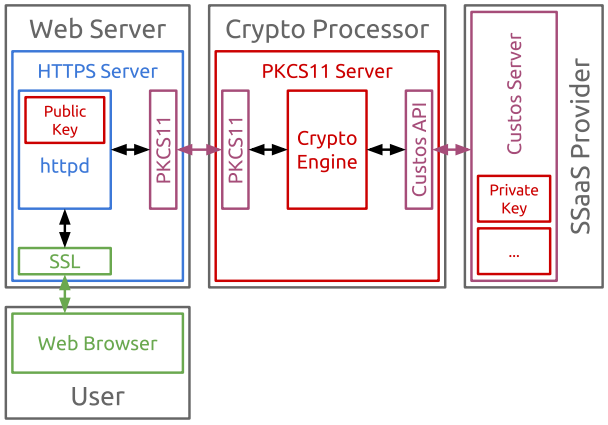


Figure 4: Custos-Backed Dedicated Crypto Processor

3.3 Dedicated Crypto Processor

The recent Heartbleed bug [2] exposed one of the main risks of embedding cryptographic secret storage within the applications requiring access to these secrets: when applications break, they also risk exposing access to the private keys stored in the same memory segments. The Heartbleed fallout has forced everyone to reevaluate whether or not giving public-facing services direct access to cryptographic keys is a good idea. There has long existed an alternative: using a hardware security module (HSM) to perform dedicated crypto processing and key storage on behalf of other services. Such systems ensure that cryptographic keys are never exposed outside of the secure hardware module. Programs communicate with the HSM via a standard protocol like PKCS#11 [7], sending the HSM clear-text data to encrypt and getting back the encrypted ciphertext in return. Unfortunately, most existing HSM solutions don't scale to the levels required for high-volume services. This has led some to suggest moving to a software-based "HSM" model [18]. Such softHSM systems would still ensure cryptographic keys remain stored in separate isolated memory spaces while also out-performing traditional HSM systems.

A software-based dedicated crypto processing system is an ideal use case for a SSaaS system like Custos. Figure 4 shows the potential design of such a system. Here, a Custos SSaaS provider supplies the back-end key storage for a dedicated crypto processing server which performs cryptographic operations (e.g. SSL) on behalf of a web-server. In this setup, the web-server never accesses any private cryptographic keys directly, mitigating one of the major risks Heartbleed exposed. Furthermore, the logically centralized nature of an SSaaS interface like Custos allows dedicated crypto processing servers to scale hor-

izontally (e.g. multiple load-balancing instances) as demand requires. Storing all keys via a Custos provider allows new crypto processing instances to immediately access these keys without the need to utilize ad-hoc key-syncing or configuration management interfaces. Furthermore, Custos's auditing functionality ensures that you always know which keys have been accessed by which systems, placing hard bounds on what an attacker may or may not have had access too: a luxury that Heartbleed-prone servers did not have.

4 Threat Model and Mitigation

The SSaaS model is designed to operate securely as long as certain assumptions are met. The primary assumption is that SSaaS (e.g. Custos) service providers are discharging their SSaaS duties faithfully. Namely, they must only grant access to the secrets they store as per the user-provided access control specifications (§ 5). We realize, however, that fully placing one's trust in a single third party SSaaS provider is not always practical nor desirable. Thus, Custos provides several mechanisms for limiting and managing service provider trust. In particular, the Custos architecture allows user to securely shard their secrets across multiple, non-cooperating SSaaS providers. A variety of secure secret-sharing (e.g. [27, 10]) systems have been proposed and these system can be leveraged by Custos-backed applications to split Custos data between multiple non-cooperating SSaaS providers (Figure 5). Thus, even if a single provider proves to be untrustworthy, their share of each secret is useless unless they are able to collude with other providers to obtain access to the requisite K shards. Such a strategy has additional benefits beyond limiting trust: it also increases availability. Shamir-like secret sharing systems offer redundancy through the over-subscription of shares, allowing reassembly of a secret using only K of a set of N total shards.

Beyond formal secret sharing schemes, we believe the SSaaS model itself provides incentives for trustworthy provider behavior: an open market of competing SSaaS service providers would tie trustworthiness to free market competition [1]. If a user finds that a specific Custos provider is prone to misbehavior, they can take their secrets elsewhere. A competitive ecosystem makes provider misbehavior only desirable in cases where the gain from dishonestly disclosing one customer's secrets is worth more than the cost of losing all of your other customers. In most situations, the economics favor SSaaS providers striving to provide the strictest adherence to the SSaaS provider trustworthiness assumptions: aligning the SSaaS consumer's goals with those of the SSaaS provider.

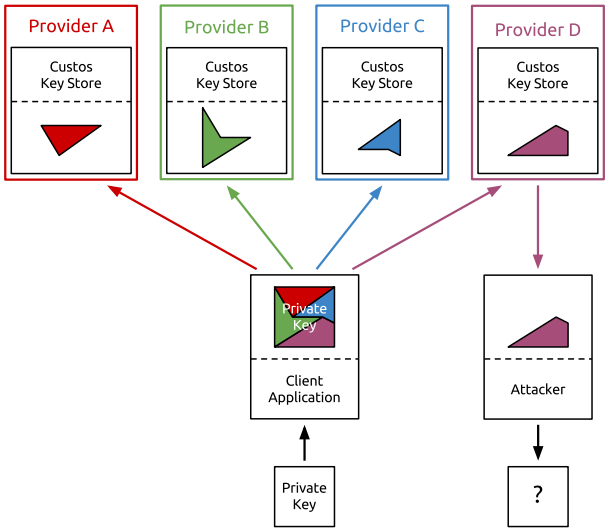


Figure 5: Custos Multi-Provider Sharding

The threat model for a given SSaaS-backed application is largely a function of the application's implementation. For example, a Custos-backed file system may opt to maintain a local cache of encryption keys to allow offline file access. Such behavior, however, would open an additional attack vector whereby an adversary must only compromise the local key cache without ever having to compromise a Custos server itself. Custos, however, is designed to be flexible, which means leaving such trade-offs up to each individual application.

As previously mentioned, secret access via Custos is a one-shot game: once a user has been granted access to a secret, it must be assumed that the user will always have access to that secret and any data protected with it. This "inability to revoke access to previously accessed data" problem is not unique to Custos. As such, Custos can mitigate this issue in the same manner other systems have: through versioning, key rotation, and lazy revocation [13]. While Custos can not guarantee access revocation to data that has already been decrypted and read, Custos can revoke access to all future versions of that data. For example, a Custos-backed file encryption application can re-encrypt a file with a new Custos-stored key each time the file is updated. When a user revokes access to a Custos object, Custos blocks all access to any versions of that object uploaded after the revocation occurs. In the file system example, this prevents users from viewing updates to a file after their access has been revoked.

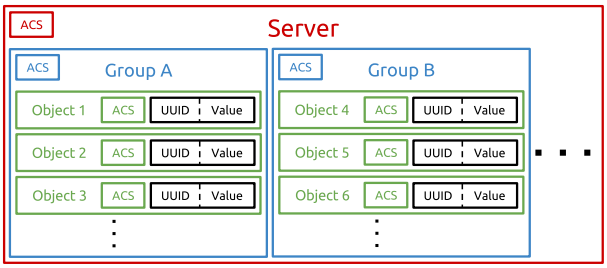


Figure 6: Custos's Organizational Units

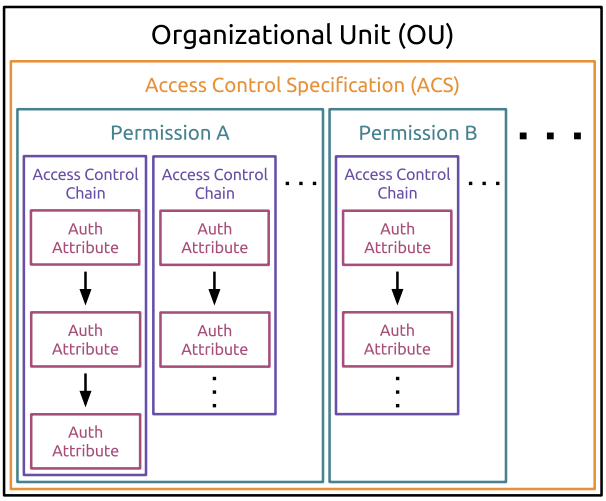


Figure 7: Access Control Specification Components

5 Design

Figure 1 shows the core Custos components. The bulk of Custos functionality is handled on the server side. The Custos server implements the following components:

API Handles all Custos requests, including requests for key:value objects, requests for audit data, and requests to modify access control parameters. The API is designed to promote a variety of Custos-compliant server implementations.

Access Control Compares the set of provided authentication attributes (calling into the authentication system to verify them) to the set of required authentication attributes to determine if a Custos request should be allowed or denied.

Authentication Verifies the validity of any authentication attributes associated with a given Custos request via a pluggable authentication module interface capable of supporting a variety of authentication primitives.

Data Handles data API requests (e.g. get, set, create, and delete of key:value objects).

Auditing Handles audit API requests and logs all Custos requests and their corresponding responses.

Management Handles management API requests (e.g. the manipulation of access control parameters).

Key-Value Secret Store Stores persistent data such as end-user secrets (e.g. encryption keys) as well as internal state (e.g. access control requirements).

5.1 Custos Access Control

The Custos access control abstraction lies at the core of Custos's flexible semantics.

In order to discuss the Custos access control system, we must first explain the Custos *organizational units* (OUs): the core Custos data structures. The Custos architecture specifies three organizational units (Figure 6): a server, a group, and a key:value object. The server unit is used to specify server-wide configuration. A server has one or more groups associated with it. A group is used to slice a server between a variety of administrative domains (e.g. separate customers). A group, in turn, has an arbitrary number of key:value objects associated with it. Each OU is responsible for the creation of OU instances beneath it: i.e. servers create groups and groups create objects.

The Custos access control abstraction revolves around designating an *Access Control Specification* (ACS) for each OU in the Custos architecture. An ACS consists of three components (Figure 7). Each ACS contains a full list of the applicable *permissions* for the given OU. Associated with each permission is one or more *access control chains* (ACCs). Each ACC consists of an ordered list of *authentication attributes*.

5.1.1 Permissions

Each Custos ACS contains a list of permissions: rights to perform specific Custos actions. Custos defines permissions for each OU: i.e. per-server permissions, per-group permissions, and per-object permissions (Table 1). Unlike many systems, Custos has no notion of object ownership. Instead, it relies on explicitly granting each right an owner would traditionally hold via explicit permissioning. Custos permissions are initially set when the associated OU is created. After creation, each ACS can be updated by anyone granted the necessary `acs_set` permission for the specific OU instance. Custos group and server ACSs also include an "override" permission. This permission can be used to override the permissions of a lower-level OU's ACS. For example, any-

Permission	OU	Rights
srv_grp_create	Server	create groups on a Custos server
srv_grp_list	Server	list groups on a Custos server
srv_grp_override	Server	escalate to any group-level permission, overriding the per-group ACS
srv_audit	Server	read all server-level audit information
srv_clean	Server	delete all server-level audit information
srv_acs_get	Server	view the server-level ACS controlling the permissions in this list
srv_acs_set	Server	update the server-level ACS controlling the permissions in this list
grp_obj_create	Group	create a key:value objects within the given group
grp_obj_list	Group	list key:value objects within the given group
grp_obj_override	Group	escalate to any object-level permission, overriding the per-object ACS
grp_delete	Group	delete the given group on a Custos server
grp_audit	Group	read all group-level audit information
grp_clean	Group	delete all group-level audit information
grp_acs_get	Group	view the group-level ACS controlling the permissions in this list
grp_acs_set	Group	update the group-level ACS controlling the permissions in this list
obj_delete	Object	delete the given key:value object within the given group
obj_read	Object	read the given key:value object within the given group
obj_update	Object	create a new version of the given key:value object within the given group
obj_audit	Object	read all object-level audit information
obj_clean	Object	delete all object-level audit information
obj_acs_get	Object	view the object-level ACS controlling the permissions in this list
obj_acs_set	Object	update the object-level ACS controlling the permissions in this list

Table 1: Custos Permissions

one gaining the srv_grp_override permission can use it to gain any of the rights normally granted via a group-level permission. Likewise, anyone gaining the grp_obj_override permission can use it to gain any of the rights normally granted via an object-level permission. These overrides exist for administrative tasks: allowing server admins to manipulate group data, and allowing group admins to manipulate object data.

5.1.2 Access Control Chains

Each ACS permission has one or more associated access control chains (ACCs). An access control chain is an ordered list of authentication attributes (discussed in § 5.1.3). In order for a request to be granted a specific permission, it must be able to provide authentication attributes satisfying at least one of the ACCs associated with that permission. If a user wishes to disable access to a permission, they can do so by associating the null ACC with that permission. If the user wants to provide unrestricted access to a permission, they may do so by associating an empty ACC with the permission. For example, consider a key:value object whose obj_read permission has the following ACC set:

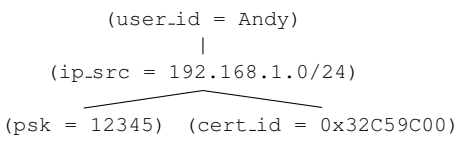
```
[ (user_id = Andy),
  (ip_src = 192.168.1.0/24),
  (psk = 12345) ]
[ (user_id = Andy),
  (ip_src = 192.168.1.0/24),
  (cert_id = 0x32C59C00) ]
[ (user_id = John),
  (psk = Swordfish) ]
```

In order for our read request for the associated key:value object to succeed, we would have to make sure that our request contained all the authentication attributes in at least one of the lists above. In the case of the first ACC, that would mean attaching the 'user_id' attribute with a value of 'Andy', as well as attaching the 'psk' attribute with a value of '12345'. The 'ip_src' attribute is an implicit attribute (see § 5.1.3) and will be automatically appended to our request when received by the Custos server. In order to satisfy it, we would have to send the request from the 192.168.1.0 subnet. In the case of the second ACC, we still need the 'Andy' username and must satisfy the IP restriction, but this time we must prove that we have access to the private key associated with the

specified authentication certificate instead of providing a password. In the third ACC, we have granted access to an additional user, John, with his own password. As long as we can satisfy at least one ACC in a set of ACCs for a given permission, we are granted the right to perform actions associated with the permission.

This system is highly flexible. Take, for example, the lack of explicit username support anywhere in the Custos specification. As was done above, usernames simply become another authentication attribute. Often a username will be the first attribute in a ACC to allow for all following attributes to be specified relative to a given username (as shown in the example above). But there's nothing special about usernames. We could just have easily started each ACC with an IP attribute, requiring a separate password based upon the location a user is making their request from. The combination of simple ordered attribute lists and a wide range of flexible attributes makes for powerful access control semantics.

Another point worth noting is that sets of ACCs can be converted into ACC trees, often simplifying the understanding or verification of their semantic intent. ACC lists are converted into ACC trees by combining common attributes across multiple ACC lists into single nodes in an ACC tree. For example, the first two ACCs in the previous set of ACCs could also be represented as:



Finally, where desired [3], the Custos API can continue to prompt the user for the next N missing attribute types in a chain. When in use, this feature allows a Custos server to engage in a back-and-forth message exchange with a client to prompt the client through all required attribute types in an ACC. For example, in the case where N is equal to 1 and the previously mentioned ACCs are in effect, the following set of transactions would occur:

1. The user sends a read request with no attributes
2. The server respond that a username is required
3. The user resubmits the request with an attached username attribute equal to 'Andy'
4. The server responds that a password or a certificate is required (the IP attribute is implicit and is thus not prompted for)

[3]Custos's attribute prompting feature is a form of information leakage, so its use, and the associated trade-offs, are optional.

Type	Class	Description
ip_src	implicit	Request source IP
time_utc	implicit	Request arrival time
user_id	explicit	Arbitrary user ID
psk	explicit	Arbitrary pre-shared key

Table 2: Example Authentication Attributes

5. The user resubmits the response with a password equal to '12345'
6. As long as the user's request originates from the specified IP range, the server will grant the request.

5.1.3 Authentication Attributes

Each Access Control Chain contains one or more Authentication Attributes (AAs). An authentication attribute is a generic container for authentication data. AAs contain the following information:

Class The top level classification property of an AA. It is used to designate the nature of a given AA. Currently, Custos specifies two possible values for class: "implicit" and "explicit". Implicit attributes are those that are automatically associated with a request (like an IP address). Explicit attributes are those that the user provides directly to Custos (like a username).

Type Within a given class, the AA type specifies which authentication plugin should handle a specific attribute.

Value The value contains the arbitrary data associated with a given attribute.

The Custos specification supports a flexible set of authentication types. Each AA is processed by a specific AA plugin module allowing for extensible authentication primitive support similar to systems like PAM [22]. Examples of potential Custos AA types are shown in Table 2.

5.1.4 Access Example

To demonstrate the full access control process, consider a Custos-backed encrypted file system application. In our example, two users of this application are attempting to access an encrypted file. In order to decrypt the file and provide access, the encrypted file system must query Custos for the necessary encryption key.

The first user is a daemon process running on a headless server (IP = 1.2.3.4). The encryption key for the file the daemon wishes to read has an ACS associated with it

that grants the obj_read permission on the basis of two implicit attributes: the host IP and the time:

```
{
  obj_read:
    [
      [ (ip_src = '1.2.3.4'),
        (time_utc = '1300 +/- 5') ]
      ...
    ]
  ...
}
```

When the daemon reads the file, the encrypted file system requests the associated encryption key from the server. The request passes through the access control module, which looks up the Access Control Chains associated with the obj_read permission for the requested encryption key object. The request is then passed to each of the necessary Authentication Attribute modules in the order they appear in the ACC. Because the request is coming from an allowed IP, it passes the source IP verification module. Next, as long as the request is being made within 5 minutes of 1300 hours UTC, the request will also pass the time verification module. After satisfying both attributes specified in the ACC, the request is granted the obj_read permission and passed to the audit module for logging. Finally, the server looks up the requested object (in this case the encryption key for the corresponding file), generates a response, and returns it to the encrypted file system. The file system uses the returned encryption key to decrypt the file and returns the contents to the daemon that originally made the read request. All of this is done without requiring any interactive input on the part of the user, overcoming one of the traditional obstacles to using encryption with automated processes.

The second user is a human named Eric who is also trying to read a file on the encrypted file system. The encryption key for the file the user wishes to read has an ACS associated with it that contains the obj_read permission and grants access to this permission on the basis of the user ID and a password:

```
{
  obj_read:
    [
      [ (user_id = 'Eric'),
        (psk = 'password') ]
      ...
    ]
  ...
}
```

When the user reads the file, the encrypted file system requests the associated encryption key from the server, attaching the current user's ID of 'Eric' to the request (but excluding the password). The request passes through the access control module, which, as before, looks up the Access Control Chains associated with the obj_read permission for the requested key:value pair. The request is then passed to the user ID verification authentication plugin, which confirms that the user ID of 'Eric' is present, next the request is passed to the PSK module for password verification. Unfortunately, the request lacks the necessary password, so the server responds to the request informing the encrypted file system that a password is required for user 'Eric'. The encrypted file system prompts the user for their password, and reissues the request, appending the newly provided password of 'password'. This time the request clears both AA verification modules, passes through the auditing system, and finally hits the actual object store. Here the server looks up the requested encryption key, generates a response, and returns it to the file system. The file system decrypts the requested file and allows the user's read operation to proceed on the resulting clear text.

5.2 API

The Custos API is the primary interface for interacting with a Custos server. The API handles data, management, and auditing requests through a common interface. All API requests provide authentication attributes as a means of attaining the necessary permission level for a requested operation. The order in which these authentication attributes are passed in each request is not relevant. Custos treats them as a heap of attributes and attempts to extract attributes from the heap in an order that will satisfy the requirements of a specific ACS. The API is RESTful and stateless (although individual authentication modules may maintain state if required).

The Custos API is secured via SSL/HTTPS. Custos servers are authenticated over SSL via the standard public key infrastructure (PKI) mechanisms (e.g. certificate authorities, etc)[4]. API requests are made to specific server HTTPS endpoints. The standard HTTP verbs (GET, PUT, POST, and DELETE) are used to multiplex related operations atop a specific endpoint. Each combination of endpoint and verb defines a specific API method. Each method requires a specific permission to complete. All API message formats are composed in JSON. Binary data is encoded as Base64 ASCII text. Authentication attributes are passed via query string as URL encoded JSON. Custos uses UUIDs [17] as keys, each associated

[4] In situations were the traditional PKI mechanisms are deemed undesirable, we are also exploring authenticating Custos servers via self-certifying mechanisms [6, 19].

with an arbitrary object for values. The full API specification, including detailed message formats, example messages, and a full list of endpoints and methods is available in [26].

6 Prototypes and Evaluation

We have built prototype implementations for both a Custos server and an example Custos application. These implementations demonstrate the benefits a "Secret Storage as a Service" architecture can provide over traditional ad-hoc secret storage systems. Our Custos prototypes are available via git at [23, 24, 25].

6.1 Custos Server

Our prototype server implementation [23] is written in Python adhering to the architecture discussed in § 5. The prototype server implementation can interface with a variety of off-the-shelf backing stores from local files to SQL and NoSql databases. The bulk of server code is spent performing the necessary Access Control regulations: reasonable given that the bulk of the Custos server exists for the purpose of performing access control. The use of a web-app friendly language and an HTTP-based interface clearly reduces the amount of code required by allowing most of the complexity associated with networking and message exchange to be handed off to existing libraries. Using an off-the-shelf backing store also simplifies the Custos code base by avoiding the need to build an entirely new object storage database from scratch. Overall, we found the creation of a Custos-compliant server a relatively straightforward and manageable undertaking, suggesting that Custos-compliant interfaces would not be difficult for other SSaaS server developers to adopt.

6.2 `EncFS`: A Custos-backed Encrypted FS

On the application front, we began by creating a reference client library appropriate for use with C-based Custos applications: `libcustos` [25]. `libcustos` deals with translating Custos JSON messages into C data structures and providing primitives for communicating with Custos HTTPS servers. It exposes a series of functions for dealing with Custos data types, handling data type memory management, making Custos requests, and processing the resulting response. The library makes it easy to interface C applications with the Custos architecture. We have leveraged `libcustos` to build a Custos-backed encrypted file system.

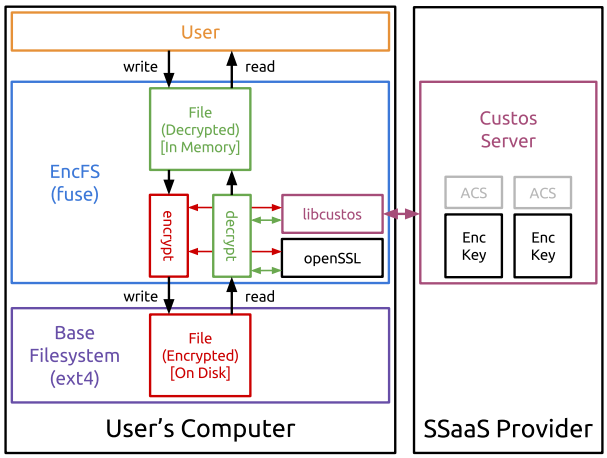
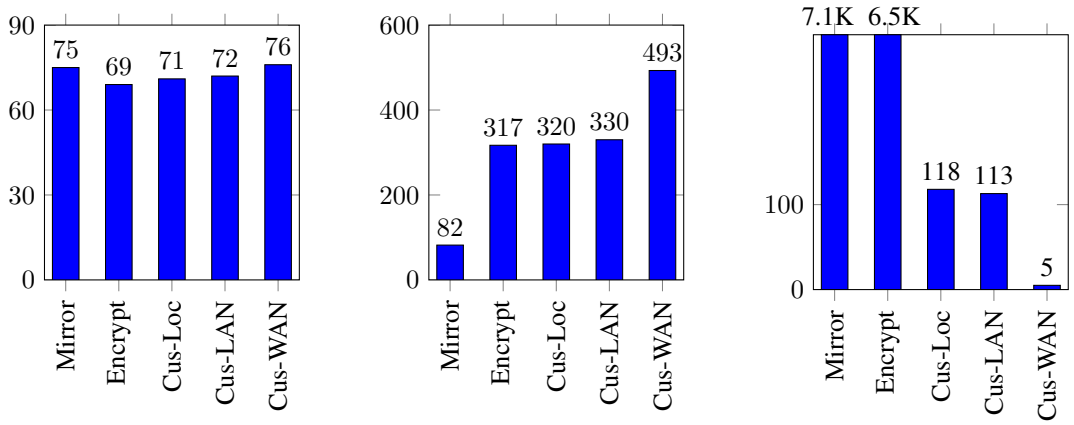


Figure 8: The `EncFS` File System Architecture

As discussed, encrypted file systems are a core Custos use case. As such, we have written a Custos-backed layered, encrypted, pass-through file system: `EncFS` [24]. This file system leverages Custos for encrypted file key storage. In doing so, it enables use cases not normally available in other encrypted file systems. For example, since `EncFS` is a pass-through file system, it can be used atop existing Cloud storage systems like Dropbox [4], securing storage of a user's files in the cloud. Unlike existing encrypted file systems, the centralized-nature of a Custos server enables access to the encrypted files from multiple devices or by multiple users, allowing a user to use Dropbox as they normally would to sync files across multiple devises or to share files with others, all while still benefiting from client-side encryption. In addition to the cloud syncing and sharing use cases, the file system has proven useful for use on servers, where Custos's flexible authentication primitives can be programmed to support daemon-based non-interactive access. This has allowed us to encrypt server files like logs or mailboxes that normally must not be encrypted in order to support non-interactive access by background system processes.

Figure 8 shows the `EncFS` architecture. `EncFS` acts as a shim between file system operations (read, write, create, etc) and the actual realization of these operations on the underlying file system, providing transparent encryption in the process. When a user wishes to decrypt a file, `EncFS` requests the associated encryption key from the Custos server using the UUID stored with the file (either via extended attributes or in a header block appended to the encrypted file contents, depending on underlying file system's support for extended attributes). If `EncFS` posses the necessary authentication attributes (either sup-



(a) `dd` Copy Throughput (MB/s) (b) `ioping` Write Latency (ms) (c) `bonnie++` Create IOPS

Figure 9: EncFS Benchmarks

plied by the user at mount time or derived contextually), Custos returns the requested encryption key and `EncFS` proceeds to decrypt the file. The opposite operation occurs when a file is created or written. As long as the user has the necessary permissions, all encrypted file access via `EncFS` is fully transparent, allowing easy integration with other applications via the standard Linux VFS interface.

`EncFS` is implemented using the FUSE [29] user-space file system framework. We chose a FUSE-based implementation over a native Linux kernel-module implementation for `EncFS` in order to allow easy usage of a variety of user-space libraries (i.e. `libcustos`, OpenSSL, etc). All encryption in `EncFS` uses the AES symmetric encryption cipher with 256-bit keys. Encryption operations are handled by the OpenSSL [20] crypto library. `EncFS` interacts with Custos via the `libcustos` library. This allows `EncFS` to offload the complexities of the Custos API to a dedicated code base, and greatly simplifies interfacing `EncFS` with Custos. We have found that adding Custos support to EncFS actually makes the code shorter than it would be if we were to build key management and access control directly into the file system from scratch, supporting our belief that SSaaS systems ease development burdens.

Figure 9 shows the results of a set of simple benchmarks for a progression of EncFS features: a basic FUSE mirroring file system without encryption (*Mirror*), an encrypted FUSE file system with local key management (*Encrypt*), and an encrypted FUSE file system with Custos-based key management and a local Custos server (*Cus-Loc*), LAN-based Custos server (*Cus-LAN*), and WAN-based Custos server (*Cus-LAN*). As one would expect, interfacing a file system with a Custos server does incur additional latency on each initial file access due to the need to query the Custos server for the appropriate encryption keys. This added latency has very little effect of the standard read/write throughput of the file system since each read/write operation only requires a single Custos key access (Figure 9a). Instead, the added latency primarily effects the file system access latency (Figure 9b) and IOPS (Figure 9c). The nature of the connection between the Custos-backed applications and the Custos server has a large effect on this latency, ranging from relatively minimal in situations with a high speed network connection (e.g. Local LAN based server) to fairly high in situations with a low speed network (e.g. over a 4G-based WAN connection). We have found, however, that the added latency is acceptable from a user perspective for most day-to-day applications where read/write performance is more critical than IOPS throughput (e.g. playing media files). Currently, the latency overhead of high speed network connections is limited by the performance of our prototype Custos server. We have plans to continue developing our reference server implementation to increase this performance and ensure that the network is the primary limiting factor to Custos-induced latency overhead, not the server itself. In cases where the added latency of using Custos is unacceptable, we are exploring the addition of client-side caching functionality to `libcustos`, minimizing latency on subsequent access to keys. This strategy could also be used to enable offline operation in `EncFS`.

7 Related Work

A variety of encrypted file systems exist with the goal of enabling secure data storage [14]. As we have shown, however, many of these system suffer from the entanglement of key management and the underlying encryption. We are not the first to recognize the challenges this entanglement imposes. SFS [19] and Plutus [13] were designed to separate cryptographic key management from encrypted data storage, allowing for more flexible key management in the process. But both SFS and Plutus fail to fully define a standardized, generic, and flexible external system for storing and managing keys, making a generic "Secret Storage as a Service" architecture impossible to realize with either system. In particular, SFS purposely avoids specifying any key management solution, instead focusing on mechanisms that allow the user to select their own key management system (e.g. Custos). Plutus provides basic key management functionality, but it bundles these tightly with the underlying file system, forcing the user to use both or neither and preventing the user from selecting dedicated third party SSaaS providers.

Password management systems (e.g. [16] share some of the same goals as Custos and can be viewed as a subset of the more generic SSaaS model. Such systems are designed to enable users to use longer, less predictable passwords by providing a dedicated system that stores and fills password feels on a user's behalf. The user must only remember a single strong password, relying on the password manager to store and supply long random passwords for all the other services a user leverages. Custos could be used as the secret storage back end for a range of existing password manager front ends, providing more flexible password access control semantics in the process and allowing the user to select a SSaaS provider of their choice.

Rackspace's CloudKeep [21] aims to create a standardized key management system for use across multiple applications, avoiding the need to re-implement such systems in each application. Similar to Custos, CloudKeep aims to ease developer burden while increasing the security of end-user applications by focusing security code in a centralized, carefully curated system. CloudKeep, however, lacks the generic flexibility and powerful semantics of Custos's authentication and access control mechanisms.

8 Conclusions

Secret storage is a critical requirement of almost all modern applications. A flexible, standardized, and service-oriented interface for secret storage provides numerous benefits: the separation of access control from the underlying applications, the ability to select from numerous SSaaS providers, the centralization of security-sensitive code in a dedicated library, etc. Custos provides a reference SSaaS architecture and is well suited for the storage of private encryption keys in a variety of applications. By offloading the storage of encryption keys to a dedicated service, we can allow encryption applications to focus on the task of encryption while dedicated Custos SSaaS servers focus on the task of securely storing and regulating access to cryptographic keys. Using a dedicated secret storage service like Custos expands the range of use cases supported by encryption applications by providing flexible access control semantics and globally accessible key storage. The Custos access control scheme, from the pluggable authentication modules to the arbitrary access control chains, allows for a wide range of access control intentions to be expressed in a common manner. The logically centralized nature of Custos allows applications to support traditionally challenging use cases like multi-user and multi-device access with little extra effort.

While Custos has shown promise, it is certainly not without challenges. While we believe many existing systems could be refactored to use Custos with minimal effort, the task of doing so, or convincing others to do so, in still non-trivial. The performance overhead of using a networked SSaaS system also deserves further study. For many end-user applications, raw performance is not the primary concern, and there may exist a willingness to sacrifice some (often unnoticeable) performance in the name of increased security and usability. That said, the overhead of adopting Custos across a wide range of existing applications has not been evaluated. Finally, Custos's authentication system, especially the access control chain component, is highly flexible. But it remains to be seen whether or not this flexibility will lead only to increased ease of use and breadth of deployment (our goal), or whether it risks giving the user too much freedom, making it prone to misconfiguration and errors.

Our work thus far has resulted in a prototype SSaaS platform. There is still work to be done to make Custos a fully production-ready and proven system. We plan to expand the reference Custos server implementation making it more robust, scalable, and widely deployable. This will include switching to a high performance object storage back-end, improving the Custos authentication plugin interface, producing plugins for additional authentication primitives, and improving the efficiency of the Custos's ACC verification process. We would also like to enhance the availability, redundancy, and distributed security features of the Custos server by standardizing the manner in

which SSaaS-backed applications might leverage Shamir-like secret sharing schemes. Finally, we plan to continue our build-out of a variety of Custos-backed applications. We look forward to continuing our development of the Custos ecosystem and hope to present expansions of this work in the future.

References

[1] R. Anderson. Why information security is hard - an economic perspective. In *Seventeenth Annual Computer Security Applications Conference*, pages 358–365. IEEE Comput. Soc, 2001.

[2] Codenomicon. The heartbleed bug. `heartbleed.com/`.

[3] R. Cox, E. Grosse, R. Pike, D. Presotto, and S. Quinlan. Security in Plan 9. In *USENIX Security*, pages 3–16, 2002.

[4] Dropbox. Dropbox. `www.dropbox.com/`.

[5] Dropbox. Dropbox security. `www.dropbox.com/security`.

[6] C. Ellison. Establishing identity without certification authorities. *USENIX Security Symposium*, 1996.

[7] EMC2. Pkcs #11: Cryptogrphic token interface standard. `www.emc.com`.

[8] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: an auditing file system for theft-prone devices. In *Proceedings of EuroSys '11*, pages 1–16, New York, New York, USA, 2011. ACM Press.

[9] Google. Drive. `www.google.com/drive/about.html`.

[10] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06*, page 89, New York, New York, USA, 2006. ACM Press.

[11] G. Greenwald and E. MacAskill. Nsa prism program taps in to user data of apple, google, and others. *The Guardian*, 2013.

[12] M. A. Halcrow. eCryptfs : An Enterprise-class Cryptographic Filesystem for Linux. In *Ottawa Linux Symposium*, pages 201–218, Ottawa, 2005. International Business Machines, Inc.

[13] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 29–42, 2003.

[14] V. Kher and Y. Kim. Securing distributed storage: challenges, techniques, and systems. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, page 9, New York, New York, USA, 2005. ACM Press.

[15] W. Koch. Gnupg. `www.gnupg.org/`.

[16] LastPass. Lastpass. `lastpass.com/`.

[17] P. Leach, M. Mealling, and R. Salz. RFC 4122: A universally Unique IDentifier (UUID) URN Namespace. Technical report, 2005.

[18] P. Lorier. Heartbleed and private key availability. `plus.google.com/106751305389299207737/posts/WM4i4Rqxs5n`, 2014.

[19] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. *ACM SIGOPS Operating Systems Review*, 33(5):124–139, Dec. 1999.

[20] OpenSSL. Openssl. `www.openssl.org/`.

[21] Rackspace. Cloud keep. `github.com/cloudkeep`.

[22] V. Samar. Unified login with pluggable authentication modules (PAM). In *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pages 1–10, New York, New York, USA, 1996. ACM Press.

[23] A. Sayler. Custos server repo. `github.com/asayler/custos-server`.

[24] A. Sayler. Encfs repo. `github.com/asayler/custos-client-encfs`.

[25] A. Sayler. libcustos repo. `github.com/asayler/custos-client-libcustos`.

[26] A. Sayler. Custos: A Flexibly Secure Key-Value Storage Platform. Master's thesis, University of Colorado Boulder, December 2013.

[27] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979.

[28] SpiderOak. Spideroak: Store. sync. share. privately. `spideroak.com`.

[29] M. Szeredi. Fuse: Filesystems in userspace. `fuse.sourceforge.net/`.

[30] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 77–91, 2012.

[31] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*, pages 679–702, 1999.

[32] D. C. Wilson and G. Ateniese. To Share or Not to Share in Client-Side Encrypted Clouds. *arXiv preprint arXiv:1404.2697*, 2014.

[33] T. Ylonen. SSHsecure login connections over the Internet. *Proceedings of the 6th USENIX Security Symposium*, 1996.