Tutamen: A Next Generation Secret Storage Platform

Andy Sayler, Taylor Andrews, Matt Monaco, and Dirk Grunwald

University of Colorado

Abstract

The storage and management of secrets (encryption keys, passwords, etc) are significant open problems in the modern age of cloud-based, third-party hosted, ephemeral computing infrastructure. How do we store and control access to the secrets necessary to configure and operate a range of modern technologies without sacrificing security and privacy requirements or significantly curtailing the desirable capabilities of our applications? To answer this question, we propose Tutamen – a next generation secret storage service. Tutamen offers a number of desirable properties not present in existing secret storage solutions. These include the ability to operate atop minimally trusted infrastructure and the ability to control secret access based on contextual, multi-factor, and alternate-band authentication parameters. These properties have allowed us to leverage Tutamen to support a variety of use cases not easily realizable using existing systems, including supporting full-disk encryption on headless servers and providing fully-featured client-side encryption to cloud-based file lockers. We present the importance of the secret storage problem, Tutamen's design and architecture, the implementation of our Tutamen prototype, and several of the applications in which we have implemented Tutamen-based secret storage to achieve previously unattainable goals while still meeting a variety of security and privacy requirements.

1 Introduction

How best to store and manage secrets – the bits of tightly controlled data necessary to ensure or bootstrap the security of computing systems and services – has always been a non-trivial problem. As we continue to move toward computing and storage platforms controlled by third parties, and embrace modern trends toward ephemeral infrastructure, the secret storage problem only becomes more prevalent and critical to solve.

Tutamen[1] is our attempt to solve the secret storage problem in a manner that allows the user to adhere to a range of security and privacy requirements without sacrificing functionality in the process. Tutamen is a next generation secret storage platform. It builds on our previous secret storage efforts [54] and strives to offer features not provided by the various secret storage systems available today. In this paper, we offer several contributions:

- The design, implementation, and evaluation of a secret storage system with support for several novel features, including:
  - Modular authentication modules designed to add flexibility through the use of contextual, multi-factor, and alternate-band (e.g., SMS text messages) authentication mechanisms.
  - The ability to operate atop minimally trusted infrastructure by leveraging multiple storage and access control providers to achieve both redundancy and to mitigate trust.
- Several practical demonstrations of how a secret storage system with such features can be integrated with real-world applications to offer desirable features in a secure and easy-to-use manner.

1.1 The Need for Secret Storage

Computing systems today invade every contour of our lives – from the fitness trackers on our wrists, to our "smart" home appliances, to the server infrastructure required to support the range of web sites and services we interact with every day. With this explosion of computing systems has come an equally large explosion in the amount of data stored by and about us. While some of this data is designed to be public (i.e., the entries on

[1]Latin – A means of protection or defense.

Wikipedia), much of it is not, requiring the enforcement of various privacy and security guarantees with respect to its handling and storage. The basis of providing such guarantees relies on our ability to store and selectively share secrets ranging from the keys used to encrypt our data to the passwords used to protect our online accounts. How best to store and manage these secrets is thus a critical question – the answer to which forms the foundation to all of computing's higher level security and privacy guarantees.

Beyond the need to bootstrap a variety of security guarantees, there are several other factors driving the need for robust secrete storage solutions. On the system administration front, the trend toward ephemeral infrastructure cable of rapidly scaling up or down is driving the adoption of configuration management systems such as Puppet [47] or Chef [45]. Such systems, however, do not tend to have suitable mechanisms for enforcing the security and privacy requirements inherent to storing secrets. Despite this, configuration data often contains a variety of secrets such as SSH keys, TLS/SSL keys, file encryption keys, and the tokens or credentials necessary to authenticate with external APIs and services.

Similarly, on the end user front, the need for suitable secret storage systems is being driven by a rapid expansion of the number of sites and services to which users must authenticate themselves, as well as by the growing expanse of digital data users wish to protect. Indeed, the popularity of password management systems such as LastPass [34] or 1Password [1] as well as user demands for encryption support on their devices [41] demonstrate the importance of secret storage, and the applications it enables, to end users.

1.2 The Ideal Secret Storage System

Unlike standard configuration management systems, or even specific secret storage systems such as password managers, a general purpose secret storage presents a number of unique requirements, including the following capabilities:

- Store a wide-range of arbitrary secret data
- Store secret data in a secure manner
- Enforce fine-grained access control requirements
- Support a range of authentication sources/methods
- Provide audit logs tracking secret access history

In response to these needs, a number of general purpose secret storage systems have been recently developed by industry, including HashiCorp's Vault [27], Lyft's Confidant [38], and Square's Keywhiz [57]. These systems exist to fulfill some or all of the requirements listed above. We believe, however, that such systems are hindered by several key limitations. First, they generally require at least one fully trusted server as the basis of their security model, making them unsuitable for operation atop untrusted infrastructure. Second, they tend to lack support for use cases requiring autonomous or remote access to secret material in a secure manner. These deficiencies give rise to a few more secret storage requirements:

- Avoidance of the need to place a high degree of trust in any single system outside the application that wishes to store a secret.
- Ability to support a range of secret access use cases, including use cases where automatic or remote access to secrets is required.

It is toward these final two requirements that Tutamen attempts to advance the state of the art over existing secret storage systems. In particular, Tutamen supports operational modes where no single entity other than the application must be trusted. This allows users to leverage third party secret storage providers running Tutamen servers without having to place high degrees of trust in any single provider. Tutamen also provides support for a modular authentication interface. This interface makes Tutamen suitable for use in situations where it is desirable to leverage external environmental information to automatically evaluate the authenticity of a secret request or where it is necessary to keep a human in the authentication loop without actually requiring that the human be physically present. For example, Tutamen can be used to store the disk encryption keys required to boot a headless server, and only release these keys to the server requesting them when a human responds to a text message confirming the boot request.

2 The Tutamen Platform

The Tutamen Secret Storage Platform is designed to handle the storage of arbitrary secret material from a range of applications. In this section, we present the Tutamen architecture and our reference Tutamen server implementations.

2.1 Architecture

Tutamen has three discreet architectural components:

Access Control Servers (ACS): The systems responsible for storing and enforcing secret access control requirements and for authenticating secret requests.

Storage Servers (SS): The systems responsible for storing secrets (or parts of secrets).

Applications: The systems leveraging the Tutamen platform to store and retrieve secrets.

The bulk of all Tutamen communication occurs between an application and one or more of each type of server. Inter-server communication is kept to a minimum to support scalability. All communication in Tutamen takes place via TLS [18] HTTPS connections, and in some cases leverages mutual TLS to provide both client and server authentication. Both access control and storage servers are designed to be used individually or in sets. E.g., an application may store its secrets on a single storage server and delegate access control to a single access control server, or the application may shard its secrets across multiple storage servers and delegate access control to multiple access control servers, or any combination thereof.

2.1.1 Access Control Servers

Tutamen access control servers (ACS) are responsible for authenticating Tutamen requests as well as storing and enforcing all Tutamen access control requirements. Access control servers expose a number of core data structures that reflect the manner in which they operate. Figure 1 shows these structures.

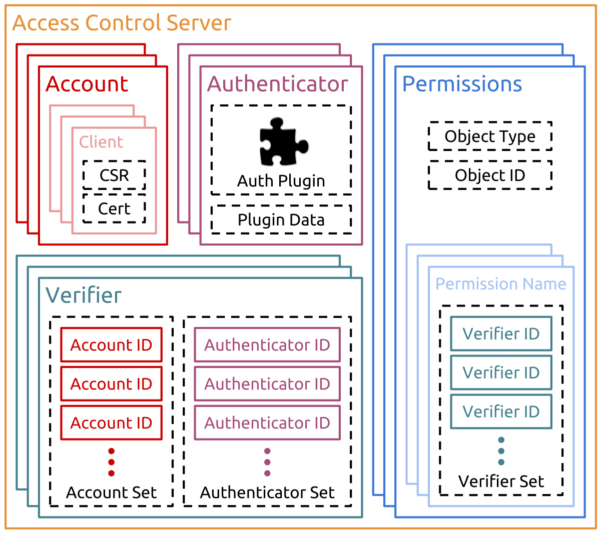

Figure 1: Access Control Server Data Structures

In order to track and control access from specific actors, the access control server uses per-actor accounts. These accounts are generally designed to map to individual end users, but they can be used to track any entity to which one wishes to assign specific access control privileges. Accounts thus form the basis of controlling and sharing access to secrets via Tutamen. Within each account are one or more clients. While accounts represent logically singular entities, clients represent specific devices controlled by such entities. Each account has one or more clients. For example, Jane Coworker may have a single account with three clients: one for her laptop, one for her desktop, and one for her phone.

Each client is associated with a single x509 [13] TLS key/cert-pair used to authenticate the client to the access control server. The access control server acts as the Certificate Authority (CA) administering these certificates. When a new client is created it generates a local private key and uses this key to generate an X509 Certificate Signing Request (CSR). This request is then sent to the access control server where it awaits approval from an existing client in the account. If approved, the CSR is used to generate a signed certificate that is sent back to the new client for use in future ACS communication. To facilitate bootstrapping new accounts, client CSRs are also generated and sent during new account creation. These are automatically approved and associated with the new account – i.e., the initial client is created in tandem with a new account while all subsequent clients are approved by previously approved clients.

In addition to accounts, the Tutamen access control server also uses "authenticators". Authenticators are modular mechanisms used to implement contextual access control requirements [28] such as only allowing access during specific times of day or from specific IP addresses. Authenticators can also be used to implement multi-factor and/or alternate-band authentication mechanisms such as confirming approval for a specific request from a user via text message, or otherwise interfacing with external services to gain approval.

Accounts and authenticators are combined via verifiers. A verifier consists of a set of accounts and a set of authenticators. In order to satisfy a verifier, a request must originate from a client associated with one of the member accounts and must satisfy all of the member authenticators. A verifier may contain no authenticators, in which case authorization is granted solely on the basis of accounts.

The final component of the Tutamen access control server is the permissions group. Each permissions group corresponds to a specific object (identified via the combination of an object type and an object ID) within the Tutamen ecosystem. A permissions group contains one or more permissions, each corresponding to a specific class of actions that can be performed on the corresponding object. Each permission is associated with a set of verifiers. In order to be granted a given permission, a request must satisfy at least one of the verifiers in this set.

2.1.2 Storage Servers

Tutamen storage servers (SS) are responsible for storing all or part of each Tutamen secret. Figure 2 shows the core storage server data structures.

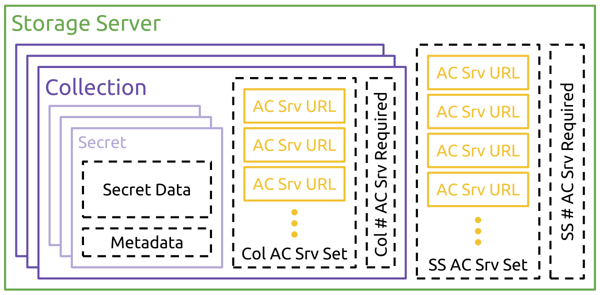

Figure 2: Storage Server Data Structures

The top-level data structure employed by storage servers is the "collection". A collection represents a logical grouping of one or more secrets (or parts of secrets). Associated with each collection is a list of one or more access control servers delegated with enforcing the access control requirements for the collection. Access control granularity is thus set at the per-collection, not per-secret level. A collection is also capable of storing user-provided metadata to aid in the mapping of collections to the objects for which they store secrets.

Each collection stores one or more secrets or secret shards. These secrets consist of the actual secret data the applications leveraging Tutamen wishes to store as well as any associated user-provided metadata. Since access control is set at the per-collection level, secrets inherit the access control characteristics of the corresponding collection.

How best to map secret data to collections is left up to each application. This decision is primarily driven by the fact that access control is performed on the per-collection level. Thus, if an application requires that a set of secrets always have a common set of access control requirements (e.g., per-sector encryption keys for an encrypted block device), it become efficient to group these secrets into a single collection. Doing so minimizes the complexity of trying to keep access control requirements synced across multiple secrets, and increases performance by minimizing the number of requests that the applications must make to secure tokens from the access control server. In cases where each secret requires its own access control requirements (e.g., per-file encryption keys), it is appropriate for the corresponding application to store only a single secret per collection.

2.1.3 Access Control Protocol

Access control servers control access related to both internal (i.e., access control server) and external (i.e., storage server) objects by providing signed authorization tokens in response to valid requests. Similar to previously proposed distributed and federated access control systems [11, 37], each authorization token grants the bearer a specific permission related to a specific object. Unlike previous systems, however, Tutamen is designed to avoid needing to trust any single access control provider (see § 2.1.4). Figure 3 shows the basic communication involved in the Tutamen access control process.

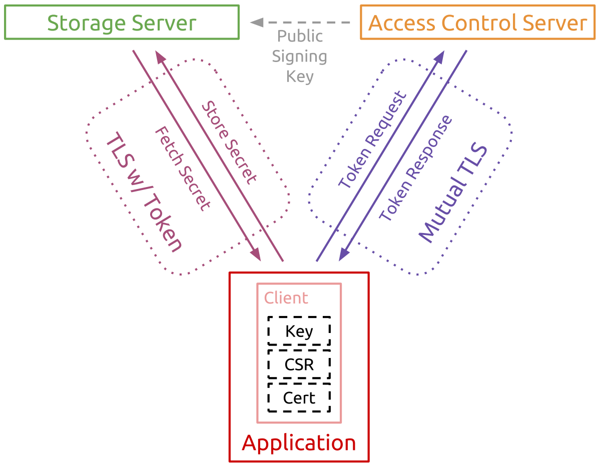

Figure 3: Access Control Communication

Each access control server generates authorization tokens in response to a client sending an authorization request. Each authorization request (and each corresponding token) includes two claims binding it to a specific object: the object type and the object ID. Each token request also contains a claim that binds it to a specific permission (e.g., read, write, delete, modify) for the corresponding object. Authorization requests are further bound to the specific client making the request (authenticated via mutual-TLS), and to an expiration time after which the token is no longer valid.

Upon receiving an authorization request from a client, the access control server looks up the permission group for the corresponding object (identified via the combination of object type and object ID) and then loads the verifier set corresponding to the requested permission. The server then traverses each verifier in this set, verifying both client membership in one of the accounts listed in the verifier as well as executing any authenticator modules required by the verifier until it finds (or fails to find) a verifier that is satisfied by the request. If the server is able to verify compliance with at least one verifier, it grants the authorization request and returns a signed authorization token that includes the object type, object ID, granted permission, and expiration time. The bearer of this token can then present it in conjunction with a request to either the access control server or a storage server in order to be granted the right to perform an approved action on the corresponding object.

Other than the bootstrapping operations and the token request operations themselves, all requests to either storage or access control servers must be accompanied by a valid token. The receiving server validates this token using the public signing key of the associated AC server; for requests to the AC sever itself, this key is available internally. For requests to external storage servers, the signing key is downloaded by the storage server from the access control server and cached for future use. In this manner, access control servers are responsible both for granting and verifying authorization requests and signing the corresponding tokens, as well as for verifying tokens accompanying requests to perform actions on ACS objects (e.g., to create or modify verifiers or accounts). Storage servers are responsible only for verifying tokens accompanying requests to perform actions on SS objects (e.g., to create a collection or read a secret).

2.1.4 Distributed Usage

Tutamen is designed to be used in both centralized and distributed use cases. The simplest Tutamen arrangement (e.g., as shown in Figure 3) involves leveraging a single Tutamen access control server and a single storage server. In this arrangement, the storage server stores a complete copy of each secret while the access control server is solely charged with enforcing access to these secrets. While this use case is easy to deploy, it has two notable downsides. First, it forces the user to place a high degree of trust in the operator of the access control server (who has complete control over whether or not the access control rules for a given secret are being faithfully enforced), as well as in the operator of the storage server (who, likewise, must faithfully verify incoming tokens and avoid otherwise leaking secret data). Second, it lacks any form of redundancy – if either the access control server or the storage server is unavailable, applications will be unable to retrieve any secrets.

A variety of systems have been proposed with the goal of minimizing trust requirements for cloud infrastructure [7, 31, 33, 39, 62]. Tutamen applies similar "minimal-trust" goals to the secret storage problem by offering support for a distributed operation mode as an alternative to single-server operation. Operating Tutamen in a distributed manner is largely a task that is pushed down to the application (or client library). With the exception of offering the necessary primitives to support such operation, both Tutamen storage and access control servers are designed to be largely agnostic as to whether they are being used in a centralized or a distributed manner. This design has the benefit of avoiding server-side scaling challenges, allowing the extra overhead required for distributed operation to be supported by each application that requires it.

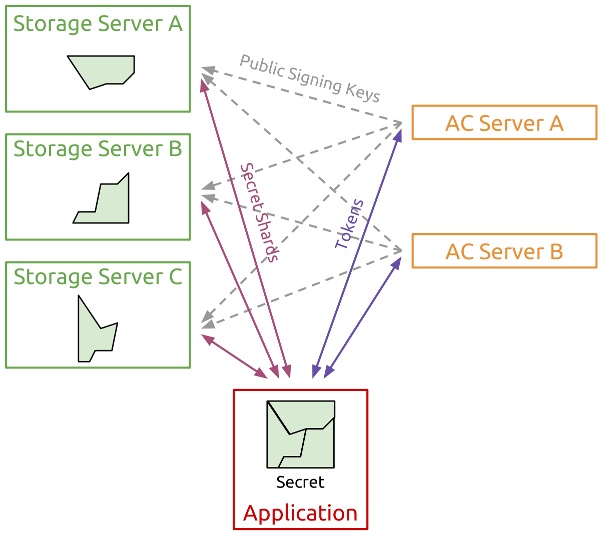

Figure 4: Distributed Operation

Figure 4 shows the basic layout of a distributed Tutamen setup. In such a setup, the Tutamen application first shards its secret using a (k,n) threshold scheme [32, 56], similar to other proposed "minimal trust" cloud systems [7] as well as previously proposed "key escrow" systems [9, 17]. The application chooses the value of n based on the number of storage servers it wishes to utilize. The value of k is then chosen to control how many of the servers must be responsive in order to retrieve the secret; i.e., the difference between n and k controls how much storage redundancy the system has by dictating the number of storage servers that can be unavailable before access to the secret itself is lost. The application then pushes each shard to the n storage servers. If the application is merely concerned about storage redundancy, or about their ability to trust a storage server operator, it can delegate the access control for each secret shard to a single access control server. To retrieve such a secret the application would request the necessary token from the access control server and include it in its request to each storage server for their respective shard of the secret. When the application receives a response from k of the storage servers, it is able to reconstruct the original secret.

In most cases, however, we imagine that in addition to wishing to mitigate storage server trust and reliability failures, the application will also wish to protect itself against access control server trust and reliability failures. This is accomplished via storage server support for the specification of two pieces of access control metadata corresponding to each stored collection: a list of AC servers approved to provide access control tokens for the collection and a minimum number of servers from which valid tokens must be received. These parameters form the basis of a novel (k,n) threshold scheme for access control servers – e.g., a collection may delegate a list of n access control servers from which an application must acquire at least k valid tokens in order to gain access. Thus, if the user does not wish to trust a single access control server, they may require tokens from at least k different AC servers in order to access the data stored in a given collection. Likewise, if the application wishes to withstand the failure of one or more AC servers, it can specify n possible AC servers where $n > k$.

In order to facilitate ease of management when operating in a distributed mode, Tutamen also supports allowing applications to request specific UUIDs [36] for each object at creation. This allows clients to use the same object ID across multiple servers, alleviating the burden of maintaining a mapping between object IDs and the servers to which they correspond. Using the same object IDs across multiple servers also allows for more efficient token management – e.g., if an application uses the same collection ID on three separate storage serves, all of which delegate a common set of access control servers, it's possible (and desirable) for the application to use a single token granting access to the relevant collection ID on all three servers. Without this capability, an application would be forced to request multiple tokens from each access control server corresponding to the collection ID used on each storage server.[2]

2.2 Implementation

In order to demonstrate and test the Tutamen platform we've created reference implementations for the storage server, access control server, and several client libraries.

Our Tutamen server implementation exposes a RESTful interface [21] for both the access control and storage server APIs. This interface both accepts and responds using JSON [15] messages over the HTTPS protocol. The full access control server API specification as well as the API reference implementation source code is available online at [50]. Likewise, the storage server API specification and source code can be found at [53]. Both implementations are freely available under the terms of the AGPLv3. The prototype servers are written in Python 3 using the Flask web framework [48]. For simplicity, both servers are designed to be served via WSGI [20] using the Apache HTTP Server [4] for TLS termination and client-certificate verification.

[2]The ability to request specific object IDs does have one downside: it opens Tutamen up to a possible denial-of-service (DoS) attack where an attacker attempts to request the object IDs they know another application wishes to use for themselves. Since each server may only use each object ID once, the first application to request a given UUID gets it. Thus, if an adversary knew which object IDs a given application planned to use, they could request these object IDs on a set of AC servers for themselves, depriving the original application of the ability to use those servers. Nonetheless, we believe the convenience afforded by allowing applications to request specific object IDs outweighs the potential for DoS abuse.

Both Tutamen servers rely on a shared `pytutamen-server` python library for the implementation of their core logic. The `pytutamen-server` source is available at [52] under the terms of the LGPLv3. This library leverages the Redis [59] key-value store for persistent storage. Our Tutamen implementation adopts the JSON Web Signature (JWS) [29] and JSON Web Token (JWT) [30] specifications for exchanging cryptographically authenticated tokens between Tutamen applications, access control servers, and storage servers. We leverage the pyjwt [46] library for JWS and JWT support. These tokens are then attached to subsequent requests using a `tutamen-tokens` header field.

The access control servers expose a pluggable authenticator interface through which end users and other developers may add custom authentication functionality. This interface is similar in purpose to previous pluggable authentication interfaces such as PAM [49]. The Tutamen authenticator interface is primarily designed for providing authentication checks beyond the TLS certificate-based authentication the access control server automatically performs on every request for the purpose of associating each request with a specific client and account. As an example, we've implemented an authenticator module that allows users to approve Tutamen token requests via SMS text message using the Twilio [60] messaging platform. We also envision authenticator modules for enforcing access control rules such as only allowing requests during certain times of day or from specific network addresses. Each authenticator plugin is provided with both a set of per-instance configuration data (e.g., to whom an SMS message gets sent for approval) as well as all of the details of a specific token request including both the IDs and metadata associated with the requesting account and client (e.g., from which information such as originating IP address or time of day can be extracted).

In addition to the server and authenticator implementations, we've also created reference Tutamen client libraries for both Python [51] and Go [43]. Using the Python client library, we've created a reference Tutamen CLI through which users may directly store/retrieve secrets and control secret sharing and access control rules. The CLI is useful for managing Tutamen objects even in cases where other applications (e.g., those discussed in Section 4) are set up to interface directly with the Tutamen platform. In this manner, it's not necessary for every Tutamen application to implement all Tutamen functionality. Instead, an application might leverage only the necessary Tutamen commands to perform secret storage and retrieval, leaving the task of managing the sharing of Tutamen-stored secrets to the CLI or to another dedicated management application.

2.3 Usage Example

To illustrate the interaction of the various components of the Tutamen platform, we present an example of the steps taken by an application to store and then retrieve a secret via Tutamen. In this example, we assume the application is using three storage servers and two access control servers as shown in Figure 4. We also assume the application has already bootstrapped an account and client (i.e., it previously contacted the AC servers with a request to create a new account and associated client).

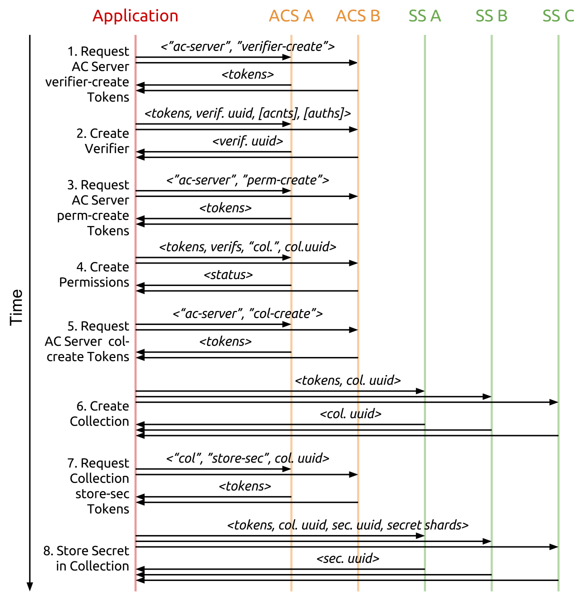
Application
ACS A
ACS B
SS A
SS B
SS C
Time
1. Request AC Server verifier-create Tokens
<"ac-server", "verifier-create">
<tokens>
2. Create Verifier
<tokens, verif. uuid, [acnts], [auths]>
<verif. uuid>
3. Request AC Server perm-create Tokens
<"ac-server", "perm-create">
<tokens>
4. Create Permissions
<tokens, verifs, "col.", col.uuid>
<status>
5. Request AC Server col-create Tokens
<"ac-server", "col-create">
<tokens>
6. Create Collection
<tokens, col. uuid>
<col. uuid>
7. Request Collection store-sec Tokens
<"col", "store-sec", col. uuid>
<tokens>
8. Store Secret in Collection
<tokens, col. uuid, sec. uuid, secret shards>
<sec. uuid>

Figure 5: Storing a New Secret

Figure 5 shows the steps required to create a new collection and store a secret within it. We assume the application has already sharded the secret into three parts – one per server.[3]

Figure 6 shows the steps required to retrieve an existing secret. This diagram also assumes that the secret in question has an SMS authenticator associated with it, requiring a user to provide a response to an SMS challenge approving access to the secret.

[3]Omitted from this diagram is the process of creating verifiers and permissions groups for the collection verifier itself. These permissions groups are necessary to control who can read, modify, or delete the corresponding verifier after creation. The process for creating said structures is similar to the process of creating the collection-related verifier and permissions group shown. To avoid the infinite recursion of needing verifiers for each verifier, it's possible for a verifier to be associated with a permissions group in which it is itself a member (i.e., a verifier can enforce its own access control specification).

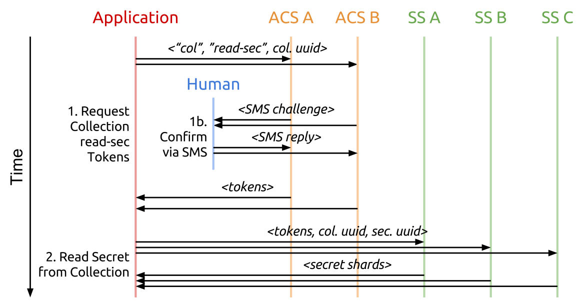
Application
ACS A
ACS B
SS A
SS B
SS C
Time
<"col", "read-sec", col. uuid>
Human
1. Request Collection read-sec Tokens
1b. Confirm via SMS
<SMS challenge>
<SMS reply>
<tokens>
<tokens, col. uuid, sec. uuid>
2. Read Secret from Collection
<secret shards>

Figure 6: Retrieving an Existing Secret (w/ SMS Auth)

3 Security and Trust

One of Tutamen's primary design goals is its ability to support a wide range of security and trust requirements. It achieves this goal through its support for both centralized and distributed operation as well as though its support for a range of authentication mechanisms.

3.1 Security of Individual Servers

The security of each individual access control server rests on several requirements. Failure to uphold these requirements will result in the failure of any security guarantees provided by the AC server.

Certificate Authority Role: Each access control server acts as a CA delegated with issuing and verifying client certificates. Thus, each AC server must store its CA keys in a secure manner and faithfully verify the certificate presented by each client connection.

Token Issuance and Verification: Each access control server is responsible for verifying the access control requirements bound to specific object/permission combinations, issuing signed tokens attesting to such verification, and verifying the signatures of the tokens it receives from clients wishing to operate on access control objects. Thus, each AC server must store private token signing key in a secure manner and faithfully verify both the access control requirements governing specific token requests as well as the signatures on all incoming tokens.

The storage servers must uphold the following security requirements. Failure to do so results in a failure of the security of the storage server.

Token Verification: Each storage server must securely (via HTTPS) obtain the public token signing key from each AC server delegated with providing access control for a given storage object. The storage server must then use these keys to faithfully verify the signatures on all tokens it receives. Assuming the token signature is valid, the storage server must faithfully enforce the claims asserted in a given token; e.g., by only allowing actions granted by the permission contained in the token on the object the token identifies prior to the expiration time specified by the token.

Secure Storage: Each storage server must take steps to store user-provided secrets in a secure manner, releasing them only to requests accompanied by the requisite number of valid tokens granting such release.

Since the tokens the storage server must verify are provided by the AC servers, the security of the storage server with respect to a given collection is dependent on the security of any designated AC servers associated with said collection. If these AC servers are insecure, the objects that delegate access to them will also be insecure.

3.2 Security of Multiple Servers

Unlike existing secret management systems [27, 38, 57], the Tutamen architecture is capable of remaining secure even when individual storage or access control servers fail to meet their security requirements. Such failures may result from physical server compromise, software bugs, malicious intent or incompetence on the part of the server operator, or compelled failures.[4]

To work around security failures of individual server, Tutamen applications can leverage Tutamen's distributed operation modes. In these modes, the security of the system as a whole is diffused, no longer relying on the security of any specific access control or storage server in order to keep an application's secrets secure. As described in Section 2.1.4, each application can distribute both secret storage and access control delegations using n choose k schemes. In such setups, the difference between n and k represents the degree to which a Tutamen applications can withstand security failures of the associated AC and storage servers. For example, an application which chooses to shard its secrets across five storage servers where any three shards are sufficient to recreate the secret ($n = 5$, $k = 3$) will continue to remain secure even if two of the storage servers fail to meet their security obligations. Similarly, if each secret shard delegates five possible AC servers, tokens from three of which are required to grant secret access, the applications can withstand the failure of two AC servers to uphold their security guarantees.

[4]For example, being forced to turn over stored secrets in response to legal or governmental pressure.

3.3 Trust Model

Trust in Tutamen follows from the security models of both individual Tutamen servers and of the distributed Tutamen deployment architectures. If a Tutamen application is leveraging only a single storage and AC server, the application is placing a high degree of trust in both servers (and by proxy, the operators of both servers). This level of trust may be appropriate for some use cases (e.g., when a user is operating their own Tutamen's servers), but is inappropriate in many other cases (e.g., when using third party hosted Tutamen servers). Fortunately, Tutamen allows applications to avoid placing a high degree of trust in any single server by leveraging multiple servers and picking k and n in a manner commensurate with the degree to which each server is trusted.

Beyond minimizing the amount of trust placed in each individual Tutamen server by leveraging multiple servers, we also envision economic incentives helping to ensure Tutamen server trustworthiness [2]. The Tutamen protocol is standardized and designed to support a range of interchangeable ACS and SS providers. Such a design allows for the development of a Tutamen server marketplace where both ACS and SS providers can compete against each other on the basis of trustworthiness, features (e.g., what types of authenticators they support), and cost. In such an ecosystem, Tutamen service providers who fail to uphold the Tutamen security requirements on their servers will suffer a negative economic effect, disincentivizing such behavior. It is also likely that storage providers who take additional steps to protect the secrets they store (e.g., by using systems such as Trusted Platform Modules (TPMs) to encrypt the secrets they hold and harden server security) would be able to command a higher price in the marketplace, incentivizing such best practices.

Thus, unlike other third-party cloud services where trustworthiness and economic incentives are in direct competition (as is the case on many "free" third part services that depend on selling user data in order to generate revenue) [22], Tutamen encourages a system where economic incentives are well aligned with user trust. That fact, coupled with the high degree of control over third-party trust Tutamen grants to each application by allowing each application to select how many servers to diffuse trust across, make Tutamen a robust system in the face of both security and trustworthiness failures. Such robustness is a critical component of any successful secrete storage system.

4 Applications

Tutamen is designed to support a wide range of applications. We have integrated our reference Tutamen design with a set of common applications for the purpose of demonstrating the value derived from using a secure storage system such as Tutamen. These applications all leverage Tutamen's flexibility to achieve functionality that would have been difficult or impossible to achieve without using a system like Tutamen.

4.1 Block Device Encryption

Block device encryption systems are a popular means of protecting the data stored on computing systems in the event that the system is lost, stolen, or otherwise physically compromised. Block-level encryption systems such as dm-crypt [10] (generally coupled with the Linux Unified Key Setup (LUKS) [23] container) or the QEMU [5] qcow2 encryption system provide methods for securing the data stored on laptops, desktops, and VMs. Such systems traditionally bootstrap security by requiring the user to enter a password at boot-time to unlock a locally stored encryption key which is then used to decrypt the block device in question. Unfortunately, the "human-at-keyboard" security root make such systems difficult or impossible to use atop headless servers or in related situations where no human can be expected to be present at boot-time. We've leveraged Tutamen to overcome this barrier.

To add Tutamen-support to LUKS/dm-crypt we've created a Tutamen-aware implementation [42] of the `systemd` Password Agent Specification [58]. This specification is used by LUKS/dm-crypt `cryptsetup` utility to request the necessary decryption secret. At boot time, `cryptsetup` will send out a request for this secret. Normally this triggers a "human-at-keyboard" prompt for a boot pass-phrase. Our Tutamen-aware password agent can instead respond to such requests by retrieving the necessary decryption secret from a Tutamen storage server (after first retrieving the necessary tokens from the corresponding Tutamen AC server). In addition to modifying the ask-password utility, we made several modifications to the `initrd` creation process to add Tutamen networking support, the necessary Tutamen client TLS key pair, and a config file specifying which Tutamen servers to use and the UUIDs of the relevant Tutamen collection and secrets.

We've also integrated Tutamen support with QEMU to provide qcow2 encryption keys when a VM boots [55]. Similar to the dm-crypt setup, QEMU normally requires the user to provide the encryption key via the QEMU console when a VM launches. Our system replaces this "human-at-keyboard" process with Tutamen-based secret retrieval. In addition, QEMU currently requires the user to provide the full encryption key, not just a pass-phrase to unlock a pre-stored key [6]. This has negative security repercussions in the common case where users pick short password-like keys. Using Tutamen, we can overcome this barrier since Tutamen servers have no qualms about needing to store or remember sufficiently long encryption keys. Our system thus increases both QEMU's security and its ease of use.

Using these setups, we're able to boot servers and VM images with encrypted disks without requiring a human to be physically present at the machine. In cases where we still desire human approval of the boot process, we can leverage our SMS authenticator module to get an on-demand confirmation from a designated human as a prerequisite to Tutamen releasing the correct key. This allows us to gain the same level of human-in-the-loop security provided by a typed pass-phrase, but without actually requiring a human to go to the datacenter to type one in. In situations where we don't desire a human-in-the-loop at all, we envision automating the approval process via the use of time-of-day and IP-source authenticators.

4.2 Encrypted Cloud File Locker

Cloud-based file lockers such as Dropbox [19] are extremely popular today. Unfortunately, these systems require users to trust the cloud provider with full access to their (generally unencrypted) data [61]. Users wishing to overcome this deficiency can optionally encrypt all of their data on the client before syncing it to the file locker provider, but doing so does not generally interact well with such services' sharing and multi-device use cases, requiring users to employ manual, out-of-band key exchange mechanisms to share or sync their encrypted files. We don't believe file locker users should have to choose between easily syncing or sharing their files and using encryption to protect their data.

Tutamen provides a solution to this problem by offering a secure key-sharing mechanism. Instead of manually distributing or sharing encryption keys, the user can store their key as a Tutamen secret and leverage Tutamen's access control features to share the secret with the accounts of their friends. This entire process could even be automated such that when a user shares a file via Dropbox, the corresponding encryption key is automatically shared via Tutamen.

Toward this end, we have created FuseBox: an alternate Dropbox client that performs client-side encryption of all Dropbox files, storing the corresponding encryption keys on our reference Tutamen server. FuseBox achieves goals similar to those achieved by [25], but without requiring out-of-band key management. Similar to other file-system-level encryption systems [8, 12, 26], FuseBox provides transparent file encryption to end users. In order to avoid the storage space and security challenges presented by locally caching all Dropbox data (i.e., the operation mode for the official Dropbox client), FuseBox uses AES [16, 44] as a stream cipher to transparently stream and encrypted data to/from Dropbox's servers on demand. The source code for our FuseBox implementation is available at [3].

Since FuseBox leverages Tutamen to store each per-file encryption key, it becomes possible to share an encrypted file via Dropbox, share its encryption key via Tutamen, and achieve the same level of functionality traditional Dropbox users have without having to expose one's data to Dropbox. While the key sharing process in FuseBox is not yet directly synced with Dropbox's file sharing system, the Tutamen CLI can be used to quickly share the encryption keys between users. In this manner, we've used FuseBox to store and share encrypted files with nearly the same ease with which one might use the traditional unencrypted Dropbox client. By leveraging Tutamen, FuseBox also gains the ability to remotely revoke file access, e.g., in the case a device is lost or stolen, similar to systems such as [24]. We also have plans to add cryptographic file authentication to FuseBox's streaming architecture using techniques such as those described at [40]. FuseBox, via Tutamen's distributed operation mode, also avoids the sharing pitfalls associated many existing "secure cloud storage" providers [63] by avoiding reliance on a single trusted party to facilitate sharing operations.

4.3 Other

Since our Tutamen-capable ask-password port speaks the standard systemd Password Agent protocol, it can also be used to provide Tutamen-backed pass-phrase storage to any applications leveraging this protocol, including OpenVPN and various password storage utilities. We have not yet thoroughly explored these use cases, but we envision a Tutamen-passed systemd password agent being useful in a wide range of situations beyond just full disk encryption. Our experience integrating the systemd password agent with Tutamen also suggests that Tutamen would provide a useful backend for a variety of other "agent" protocols (e.g., [14, 64]).

5 Performance

We've evaluated Tutamen in a variety of scenarios using the applications described in § 4. These scenarios have proven Tutamen's usefulness as an enabler of previously unattainable functionally. While Tutamen is still a prototype, we feel it provides a well-designed architecture capable of supporting a wide range of practical secret storage applications. While the server software has not yet been optimized for performance, we have performed a number of performance measurements in order to better understand the relative computational load and bottlenecks of the various parts of the Tutamen system.

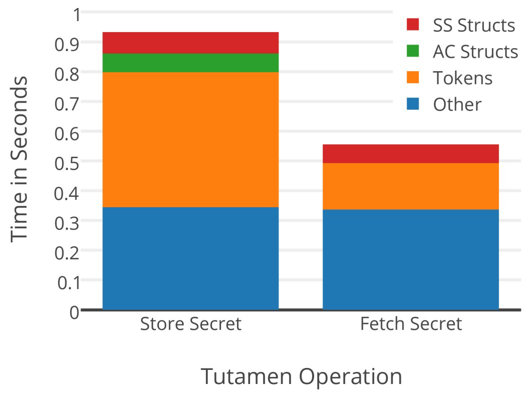

Figure 7: Timings for Tutamen Operations

Figure 7 shows the time required to complete two of the most common Tutamen operations: storing a new secret and retrieving a previously stored secret. We profiled the amount of time the Tutamen CLI application spent performing various parts of each of these two Tutamen operations. In both operations, the bulk of the wait-on-server-related run time is spent requesting and retrieving the authorization tokens required to complete the associated operations. In the secret creation case, five tokens are required. In the secret read case, only a single token is required. The remainder of the server-related time is spent either creating AC and storage data structures (as in the store secret case), or reading existing data structures (as in the retrieve secret case). The "other" time is spent reading the Tutamen config files, loading the necessary client certificates, and dealing with the overhead required to interpret the python-based CLI.

It is not unexpected that the client must spend the bulk of its time requesting tokens and waiting for them to be approved – token verification is the primary role the access control server must perform, and depending on the complexity of the verifiers associated with the permission the token is requesting, verification can be a fairly complex task. When performing these measurements, we employed a simple verifier that only required client membership in a specific account. Verifiers that include human-in-the-loop authenticators (e.g., SMS approval) would increase the token turnaround time by the amount of time the human requires to provide approval. Thus, it is important that Tutamen applications treat token approval as an operation that can take anywhere from under a second to human-scale times (e.g., 10s of seconds). To help alleviate these waits on applications that must perform a high number of Tutamen requests, Tutamen tokens may be reused up until their expiration time. Thus, it is possible for an application to request a long-lived token and to reuse this token to access multiple secrets within the collection to which the token grants read access.

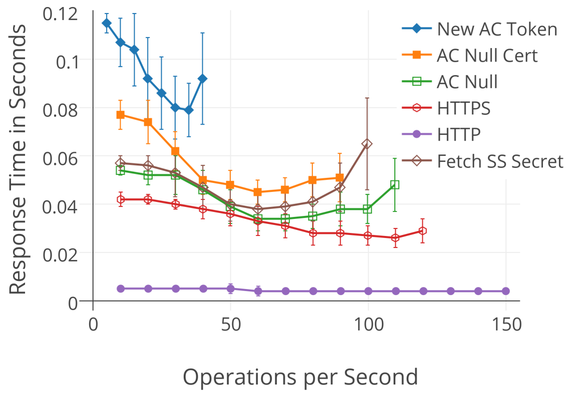

Figure 8: Throughput vs Latency Curves

Figure 8 shows the request rate vs response time (with standard deviations) of a token request operation, two “null” AC API operations (one that sends and verifies the client TLS certificate and one that does not), and a fetch secret operation. Raw Apache HTTPS and HTTP curves are also shown for comparison. As these curves show, token verification of our prototype server tops out at around 40 requests/second (rps) on a modest server (2-core, 4GB VM running atop 2011-era Intel Xeon hardware). The null AC API operation with client certificates tops out around 90 rps, and the null operation without client certificates tops out at about 110 rps. Raw HTTPS tops out around 120 rps. HTTP topped out around 5000 rps (curve truncated for viewability). The current server setup is thus primarily limited by the crypto overhead required to serve the application and verify client certificates using TLS. Token verification itself also incurs additional computational requirements - including cryptographic signing operations, but is well within the order of magnitude of the underlying server limits. Secret retrieval (after acquiring a token) is relatively quick, topping out at around 100 rps.

While these levels of performance would not likely meet the requirements of a production-level Tutamen AC server, they have been perfectly adequate for supporting the handful of Tutamen applications we’re currently using. Since most of our Tutamen applications require only a single Tutamen secret retrieval at relatively rare rates (e.g., once per server reboot or once per file open), the 40+ requests per second our AC server can provide have been more than adequate for our needs. We’ve also designed our reference server to be horizontally scalable (e.g., by spinning up multiple load-balanced API servers that share a common database). That scalability, coupled with future performance-related code optimization, leads us to believe the Tutamen server infrastructure can be adopted to meet the needs of larger installations with only moderate effort.

6 Conclusion

How best to securely store secrets is a pressing issue in today’s cloud-oriented, third-party hosted, ephemeral-infrastructure-adopting environment. This need has triggered the creation of several secret storage frameworks and systems. Unfortunately, these existing systems prove deficient in at least two key secret storage capabilities: the ability to be operate atop untrusted infrastructure and the ability to support a wide range of use cases.

We created Tutamen to demonstrate our concept for a next-generation secret storage system. Tutamen supports client-controlled secret sharding to allow applications to leverage minimally-trusted server infrastructure. Tutamen also supports a flexible and modular authentication mechanism that allows end users to specify complex access control requirements. We’ve successfully coupled Tutamen with a number of applications, including full disk encryption on headless servers and client-side encrypted file sharing between multiple parties. These use cases would be difficult (or at least burdensome on the end user) to realize without a system such as Tutamen.

We plan to continue developing the Tutamen ecosystem. On the server side, we have plans to work toward increased performance and to add support for additional authenticator modules. While the Tutamen servers currently have basic logging support, we also plan to expand this support and explore interfacing Tutamen audit logs with intrusion detection systems with an aim toward exposing more actionable intelligence to Tutamen authenticator modules. We are also considering tying Tutamen’s logging infrastructure to a public audit system similar to [35]. Such a system would help to further reduce the trust Tutamen users must place in individual Tutamen servers by exposing mechanisms by which a user could reliably audit the behavior of such providers.

We have made all of the Tutamen source code available via the previously referenced repositories. We encourage others to experiment with our Tutamen prototype and reference implementation, or to integrate Tutamen with their projects or applications. We hope that Tutamen (or similar secret storage systems) can help to ease the secret-storage burden currently imposed on administrators, developers, and end users by providing an alternative to manually managing sensitive secrets.

References

[1] AGILEBITS. 1password. `http://agilebits.com/onepassword`.

[2] ANDERSON, R. Why information security is hard - an economic perspective. In *Seventeenth Annual Computer Security Applications Conference* (2001), IEEE Comput. Soc, pp. 358–365.

[3] ANDREWS, T. Fusebox source code. `github.com/https://github.com/taylorjandrews/FuseBox`.

[4] APACHE SOFTWARE FOUNDATION, T. Apache http server project. `https://httpd.apache.org/`.

[5] BELLARD, F., AND OTHERS. QEMU: Open Source Process Emsulator. `http://wiki.qemu.org/Main_Page`.

[6] BERRANGE, D. QEMU QCow2 built-in encyrption: just say no. `https://www.berrange.com/posts/2015/03/17/`, March 2015.

[7] BESSANI, A., CORREIA, M., QUARESMA, B., ANDRE, F., AND SOUSA, P. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the sixth conference on Computer systems* (2011), pp. 31–46.

[8] BLAZE, M. A cryptographic file system for UNIX. *The 1st ACM Conference on Computer and Communication Security* (1993), 9–16.

[9] BLAZE, M. Oblivious Key Escrow. *Information Hiding 1174* (1996).

[10] BROZ, M. dm-crypt. `http://code.google.com/p/cryptsetup/wiki/DMCrypt`.

[11] CALERO, J. M. A., EDWARDS, N., KIRSCHNICK, J., WILCOCK, L., AND WRAY, M. Toward a Multi-Tenancy Authorization System for Cloud Services. *IEEE Security & Privacy Magazine 8*, 6 (nov 2010), 48–55.

[12] CATTANEO, G., CATUOGNO, L., SORBO, A. D., AND PERSIANO, P. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In *USENIX Annual Technical Conference* (2001), pp. 199–212.

[13] COOPER, D., SANTESSON, S., FARRELL, S., BOEYEN, S., HOUSLEY, R., AND POLK, W. RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Tech. rep., May 2008.

[14] COX, R., GROSSE, E., PIKE, R., PRESOTTO, D., AND QUINLAN, S. Security in Plan 9. In *USENIX Security* (2002), pp. 3–16.

[15] CROCKFORD, D. Introducing json. `http://www.json.org`.

[16] DAEMEN, J., AND RIJMEN, V. AES Proposal: Rijndael. Tech. rep., 1999.

[17] DENNING, D. E., AND BRANSTAD, D. K. A taxonomy for key escrow encryption systems. *Communications of the ACM 39*, 3 (Mar. 1996), 34–40.

[18] DIERKS, T., AND RESCORLA, E. RFC 5246: The Transport Layer Security (TLS) Protocol - Version 1.2. Tech. rep., 2008.

[19] DROPBOX INC. Dropbox. `http://www.dropbox.com`.

[20] EBY, P. PEP 3333 – Python Web Server Gateway Interface v1.0.1. Tech. rep., Python Software Foundation, 2010.

[21] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California Irvine, 2000.

[22] FLOWERDAY, S., AND SOLMS, R. V. Trust: An Element of Information Security. In *Security and Privacy in Dynamic Environments*, S. Fischer-Hübner, K. Rannenberg, L. Yngström, and S. Lindskog, Eds., vol. 201 of *IFIP International Federation for Information Processing*. Kluwer Academic Publishers, Boston, 2006, pp. 87–98.

[23] FRUHWIRTH, C. Luks. `https://code.google.com/p/cryptsetup`.

[24] GEAMBASU, R., JOHN, J. P., GRIBBLE, S. D., KOHNO, T., AND LEVY, H. M. Keypad: an auditing file system for theft-prone devices. In *Proceedings of EuroSys '11* (New York, New York, USA, 2011), ACM Press, pp. 1–16.

[25] GOH, E.-J., SHACHAM, H., MODADUGU, N., AND BONEH, D. SiRiUS: Securing Remote Untrusted Storage. *NDSS*, 0121481 (2003).

[26] HALCROW, M. A. eCryptfs : An Enterprise-class Cryptographic Filesystem for Linux. In *Ottawa Linux Symposium* (Ottawa, 2005), International Business Machines, Inc, pp. 201–218.

[27] HASHICORP. Vault: A tool for managing secrets. `https://www.vaultproject.io/`.

[28] HULSEBOSCH, R. J., SALDEN, A. H., BARGH, M. S., EBBEN, P. W. G., AND REITSMA, J. Context sensitive access control. In *Proceedings of the tenth ACM symposium on Access control models and technologies* (New York, New York, USA, 2005), ACM Press, p. 111.

[29] JONES, M., BRADLES, J., AND SAKIMURA, N. RFC 7515: JSON Web Signature (JWS). Tech. rep., May 2015.

[30] JONES, M., BRADLES, J., AND SAKIMURA, N. RFC 7519: JSON Web Token (JWT). Tech. rep., May 2015.

[31] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), pp. 29–42.

[32] KRAWCZYK, H. Secret Sharing Made Short. In *Advances in Cryptology-CRYPTO'93*, D. R. Stinson, Ed., vol. 773 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, July 1994, pp. 136–146.

[33] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. Oceanstore: An architecture for global-scale persistent storage. *ACM SIGPLAN Notices 35*, 11 (2000), 190–201.

[34] LASTPASS. Lastpass. `https://lastpass.com`.

[35] LAURIE, B., LANGLEY, A., AND KASPER, E. RFC 6962: Certificate Transparency. Tech. rep., 2013.

[36] LEACH, P., MEALLING, M., AND SALZ, R. RFC 4122: A universally Unique IDentifier (UUID) URN Namespace. Tech. rep., 2005.

[37] LEANDRO, M. A. P. M., NASCIMENTO, T. J., DOS SANTOS, D. R., AND WESTPHALL, C. B. C. M. Multi-tenancy authorization system with federated identity for cloud-based environments using shibboleth. *ICN 2012, The Eleventh International Conference on Networks* (2012), 88–93.

[38] LYFT. Confidant: Your secret keeper. `https://lyft.github.io/confidant/`.

[39] MAHAJAN, P., SETTY, S., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. Depot: Cloud Storage with Minimal Trust. *ACM Transactions on Computer Systems 29*, 4 (Dec. 2011), 1–38.

[40] MCGREW, D. Efficient authentication of large , dynamic data sets using Galois/Counter Mode (GCM). In *Security in Storage Workshop* (2005), IEEE.

[41] MCLAUGHLIN, J. Apples Tim Cook Lashes Out at White House Officials for Being Wishy-Washy on Encryption. *The Intercept* (Jan 2016).

[42] MONACO, M. Tutamen ask-password Source Code. `https://git.monaco.cx/matt/tutamen-ask-password`.

[43] MONACO, M. Tutamen golang Library Source Code. https://git.monaco.cx/matt/go-tutamen.

[44] NATIONAL INSTITUTE OF STANDARDS & TECHNOLOGY. Announcing the Advanced Encryption Standard (AES). *Federal Information Processing Standards Publication* (2001).

[45] OPSCODE. Chef. https://www.opscode.com/chef.

[46] PADILLA, J., AND LINDSAY, J. pyJWT: A Python implementation of RFC 7519. https://github.com/jpadilla/pyjwt.

[47] PUPPET LABS. Puppet. https://puppetlabs.com.

[48] RONACHER, A. Flask: web development one drop at a time. http://flask.pocoo.org/.

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

[50] SAYLER, A. Tutamen Access Control API Source Code. https://github.com/asayler/tutamen-api_accesscontrol.

[51] SAYLER, A. Tutamen pytutamen Client Source Code. https://github.com/asayler/tutamen-pytutamen.

[52] SAYLER, A. Tutamen pytutamen Server Source Code. https://github.com/asayler/tutamen-pytutamen_server.

[53] SAYLER, A. Tutamen Storage API Source Code. https://github.com/asayler/tutamen-api_storage.

[54] SAYLER, A., AND GRUNWALD, D. Custos: Increasing security with secret storage as a service. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)* (Broomfield, CO, Oct. 2014), USENIX Association.

[55] SAYLER, A., AND OTHERS. Tutamen-aware QEMU Port. https://github.com/asayler/qemu.

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

[57] SQUARE. Keywhiz: A system for managing and distributing secrets. https://square.github.io/keywhiz/.

[58] SYSTEMD CONTRIBUTORS. systemd Password Agents. https://wiki.freedesktop.org/www/Software/systemd/PasswordAgents/.

[59] THE REDIS TEAM. Redis: Remote dictionary server. http://redis.io/.

[60] TWILIO. Twilio: A Messaging, Voice, Video and Authentication API for every application. https://www.twilio.com/.

[61] VINTSURF. Dropbox... opening my docs? http://www.wncinfosec.com/dropbox-opening-my-docs/, Sep 2013.

[62] WILCOX-O'HEARN, Z., AND WARNER, B. Tahoe: the least-authority filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability* (New York, New York, USA, 2008), ACM Press, pp. 21–26.

[63] WILSON, D. C., AND ATENIESE, G. To Share or Not to Share in Client-Side Encrypted Clouds. *arXiv preprint arXiv:1404.2697* (2014).

[64] YLONEN, T. SSHsecure login connections over the Internet. *Proceedings of the 6th USENIX Security Symposium* (1996).