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 age of ephemeral, cloud-based 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 systems? 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 across administrative domain boundaries and atop minimally trusted infrastructure. Tutamen also supports access control 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 for cloud-based file-storage services. In this paper, we present an overview of the secret-storage challenge, Tutamen's design and architecture, the implementation of our Tutamen prototype, and several of the applications we have built atop Tutamen. We conclude that Tutamen effectively eases the secret-storage burden and allows developers and systems administrators to achieve previously unattainable security-oriented goals while still supporting a wide range of feature-oriented requirements.

1 Introduction

How best to store and manage secrets – the bits of sensitive 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 [53] as well as various secret-storage systems available today [24, 35, 57].

In this paper, we present the design, implementation, and evaluation of a secret-storage system with support for several novel features. We also provide 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. Tutaman's primary selling points include:

- A modular authentication system designed to support 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 redundancy and mitigate trust.
- The ability to share and manage secrets beyond the boundaries of a single administrative domain.

1.1 The Need for Secret-Storage

Computing systems today invade every facet 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 websites 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 (e.g., the entries on 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

[1] Latin for "A means of protection or defense."

manage these secrets is thus a critical question, the answer to which forms the foundation for 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 secret-storage solutions. On the systems administration front, the trend toward ephemeral infrastructure capable of rapidly scaling up or down is driving the adoption of configuration management systems such as Puppet [43] or Chef [41]. Such systems, however, do not tend to have suitable mechanisms for enforcing the security and privacy requirements inherent to storing secrets. Nonetheless, 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 and the growing expanse of digital data and computing devices users wish to protect. Indeed, the popularity of password management systems such as LastPass [31] or 1Password [1] and the increasing trend toward "on-by-default" device-encryption 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 arbitrary secret data.
- Secure the manner in which such data is stored.
- 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 recently been developed by industry, including HashiCorp's Vault [24], Lyft's Confidant [35], 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 are designed for use within the boundaries of a single administrative domain, and require the user to trust the administrators of that domain. Finally, 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 or administrative domain.
- 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 client 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 is also designed to scale beyond a single administrative domain, and does not require centralized control or administration of each server in a Tutamen deployment. 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 to 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. Tutamen expands on Custos [53] our previous secret-storage attempt. It aims to simplify some of the concepts Custos proposed and to add better support for distributing secrets across multiple servers.

2.1 Architecture

Tutamen has three discrete architectural components:

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

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 as the number of servers grows. All communication in Tutamen takes place via TLS [16] 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. For example, 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 and for 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.

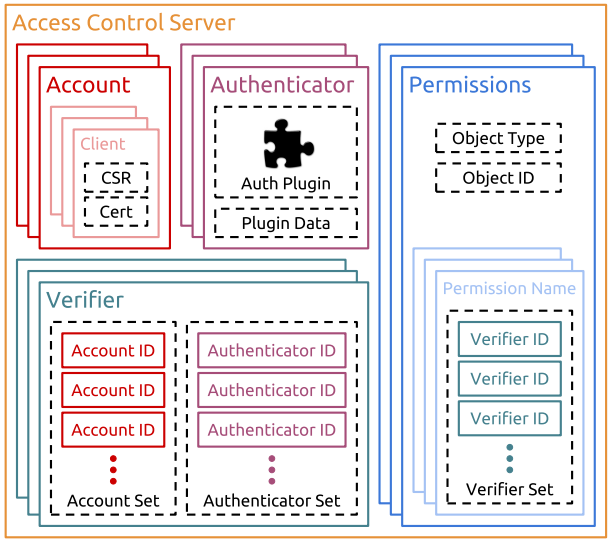


Figure 1: Access Control Server Data Structures

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 [12] TLS key/certificate-pair used to authenticate the client to the access control server. Each access control server acts as a Certificate Authority (CA) administering these certificates. When a new client is created it generates a private key and uses this key to generate an 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 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. Client certificates are also designed to be revocable either by other clients within an account, or by the administrator of a given Tutamen AC server.[2] Account holders can use this functionality to transition from old clients to new ones, first using the old client to approve the CSR for new client, and then using the new client to revoke the certificate of the old client.

In addition to accounts, the Tutamen access control server also uses *authenticators*. Authenticators are modular plugins (similar to PAM [46]) used to implement contextual access control requirements [25] such as only allowing access during specific times of day or from specific IP addresses. Authenticators can also be used to implement multi-factor 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. 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). A permissions

[2] While not yet fully implemented in our prototype, certificate revocation is handled in the typical manner whereby the Tutamen AC Server maintains and advertises a signed certificate revocation list that is consulted any time a client certificate needs to be verified to confirm validity.

group contains one or more permissions (e.g., create, read, modify, delete,), 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. To be granted a given permission, a request must satisfy at least one of the verifiers in this set. In this manner, Tutamen allows the fine-grained control of each permission for each object on the basis of account membership, authenticator satisfaction, or a combination of both.

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.

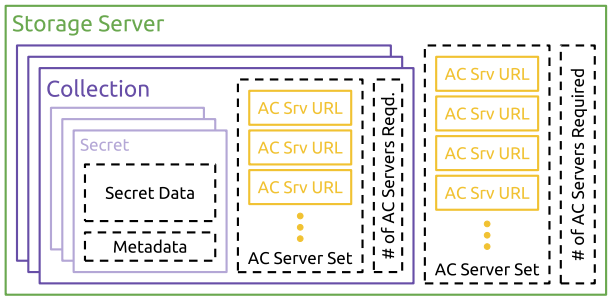


Figure 2: Storage Server Data Structures

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 secret data an application wishes to store and any associated user-provided metadata.[3]

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 [10, 34, 40], 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.

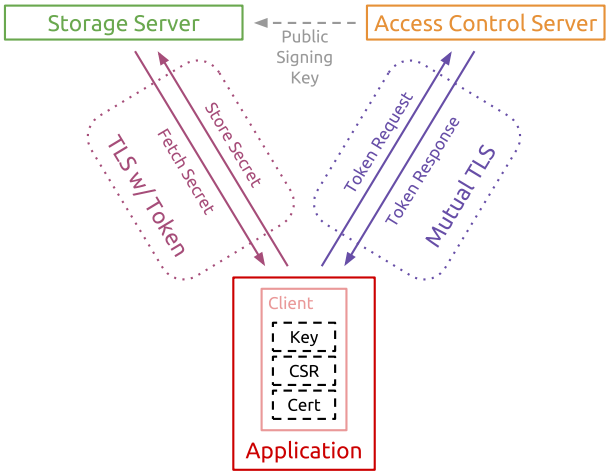


Figure 3: Access Control Communication

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 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. Tutamen allows the client to request a specific expiration time for each token, facilitating the reuse of a single token to repeat a specific action on a given object without needing to re-authenticate each time. The access control server considers the request expiration time when deciding whether or not to approve an authentication exist, and may return an authorization valid for less time than originally requested or deny a request for an overly long duration altogether.

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, checking for client membership in one of the accounts listed in the verifier and 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 com-

[3] 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, it is efficient to group these secrets into a single collection. 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.

pliance 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 presents it in conjunction with a request to either the access control server or a storage server to act 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 access control server. For requests to the access control server itself, this key is available internally. For requests to storage servers, the storage server downloads the signing key from each delegated access control server. In this manner, access control servers are responsible for granting and verifying authorization requests, signing the corresponding tokens, and 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 storage 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, a single storage server stores a complete copy of each secret while a single access control server is 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 both the operator of the access control server and the operator of the storage server. 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 [6, 28, 30, 36, 60]. Tutamen applies similar "minimal-trust" goals to the secret-storage problem by offering support for sharding secret storage and access control duties across multiple servers. 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 sharded 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. This design also helps avoid leaking unnecessary metadata to each Tutamen server, making it harder for one server to identify (and thus attempt to attack) the other servers involved in storing a single secret.

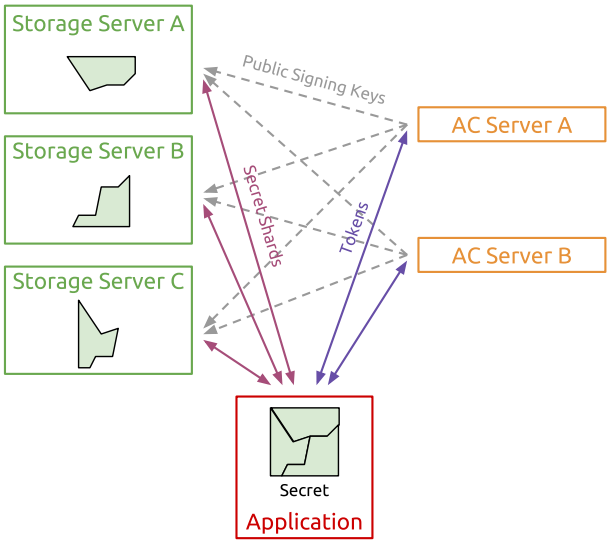


Figure 4: Distributed Operation

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 [29, 56], similar to previously proposed systems [6, 8, 15]. The application chooses the value of n based on the number of storage servers it wishes to use. The value of k is then chosen to control how many of the servers must be available to retrieve the secret, or inversely, how many server operators must collude to access a user's secret. The application then pushes each shard to the n storage servers. If the application is merely concerned about storage redundancy, or about its 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, 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 metadata corresponding to each stored collection: a set 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 yet simple (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 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 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 $n > k$.

To facilitate ease of management when operating in a distributed fashion, Tutamen supports allowing applications to request use of specific, randomly-generated UUIDs [33] for each object at creation time. 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 servers, all of which delegate a common set of access control servers, it is possible (and desirable) for the application to use a single token on all three servers.[4]

2.2 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 an associated client.

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.[5] As shown, the client starts by setting up

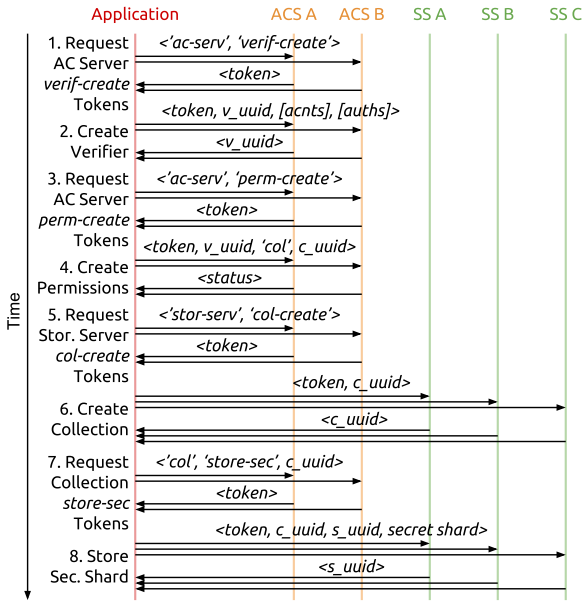


Figure 5: Storing a New Secret

the necessary access control structures (i.e., a new verifier and a corresponding set of collection permissions).[6] Once the AC structures have been created, the client creates the storage data structures: first a new collection, than a secret within the collection. Prior to each operation, the client must also request a token granting the necessary permission from the AC server, meaning that about half the interactions in the secret creation process are token requests.

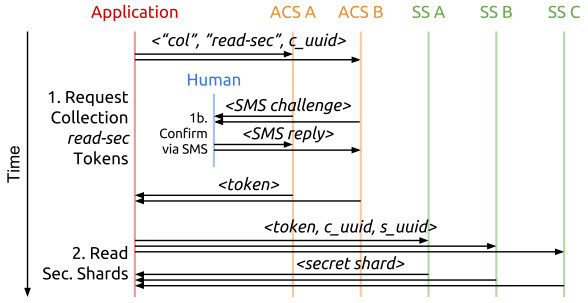


Figure 6: Retrieving an Existing Secret (w/ SMS Auth)

[4]The ability to request specific object IDs does have a 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 given access control server for themselves, depriving the original application of the ability to use those servers with that ID. At worst, however, this attack posses an inconvenience to applications, since an application is welcome to simply generate a new UUID and re-key the object with it.

[5]Omitted from this diagram is the process of creating verifiers and permissions groups for the collection verifier itself. These objects are necessary to control who can read, modify, or delete the corresponding verifier after creation. The process for creating such objects is similar to the process of creating the collection-related verifier and permissions. To avoid the infinite recursion of needing verifiers for each verifier, it is 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).

[6]Note that is is critical that the client creates the permission set corresponding to the planned UUID for the new collection prior to creating the collection itself. Failure to do so risks allowing another actor to "hijack" the collection by requesting the permission set corresponding to the new collection's UUID before the original requester. As with other Tutamen operations, the first person to request an object corresponding to a given UUID gets it. Thus, a client should only create a collection using a given UUID after they have properly secured the corresponding permissions for the UUID in question.

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. Compared to the process of storing a new secret, retrieving a secret is much simpler since it is not necessary to create the slew of new objects required when creating a new collection. Ignoring any round trips required by individual authenticator modules (e.g., the SMS verification), secret retrieval required only a single round trip to each server: one to each AC server to retrieve a token and one to each storage server to retrieve each shard of the secret. In Tutamen, secret retrieval is thus less expensive than secret storage. We foresee most secret-related workloads requiring many reads of a single stored secret, so optimizing the system for reads over writes seems appropriate.

2.3 Implementation

In order to demonstrate and test the Tutamen platform, we have created reference implementations for the storage server, access control server, and several client libraries. Our Tutamen server implementation exposes a RESTful interface [19] for both the access control and storage server APIs. This interface accepts and responds using JSON [13] messages over the HTTPS protocol. The full access control server API specification and reference implementation are available at [49]. Likewise, the storage server API specification and implementation can be found at [51]. The prototype servers are written in Python3 using the Flask web framework [45]. Both servers are served via WSGI [18] using the Apache HTTP Server [4] for TLS termination and client-certificate verification. Both Tutamen servers rely on a shared `pytutamen-server` [48] Python library for the implementation of their core logic. This library leverages the Redis [44] key-value store for persistent storage. Our Tutamen implementation adopts the JSON Web Signature (JWS) [26] and JSON Web Token (JWT) [27] specifications for exchanging cryptographically authenticated tokens between Tutamen applications, access control servers, and storage servers. We leverage the pyjwt [42] library for JWS and JWT support. These tokens are then attached to subsequent requests using a custom HTTP 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 primarily designed to allow users to specify authentication checks beyond the client certificate authentication automatically performed on every request. As an example, we have implemented an authenticator module that allows users to approve Tutamen token requests via SMS text message using the Twilio [59] 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) and all of the details of a specific token request (e.g., IDs and metadata associated with the requesting account and client, from which information such as originating IP address or time of day can be extracted).

In addition to the server and authenticator implementations, we have also created reference Tutamen client libraries for both Python [47] and Go [38]. Using the Python client library, we have created a Tutamen CLI application through which users may directly store and retrieve secrets and control secret sharing and access control rules [50]. The CLI is useful for managing Tutamen objects even in cases where other applications (see Section 4) are set up to interface directly with the Tutamen platform. In this manner, it is not necessary for every Tutamen application to implement all Tutamen management 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.

3 Security and Trust

One of Tutamen's primary design goals is the ability to support a wide range of security and trust requirements. In this section, we present a basic overview of the Tutamen security and trust models.

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 and 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 download 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.[7]

3.2 Security of Multiple Servers

Unlike existing secret management systems [24, 35, 57], the Tutamen architecture is designed to support users outside of a single administrative domain and 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.[8]

To work around security failures of an 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 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 value of k represents the degree to which a Tutamen applications can withstand security failures of the associated AC and storage servers, while the difference between n and k dictates the degree to which an Application can withstand availability failures. For example, an application that chooses to shard its secrets across six storage servers where any three shards are sufficient to recreate the secret ($n = 6$, $k = 3$) will remain secure as long as no more than two ($k - 1$) of the storage servers fail to meet their security obligations. Similarly, if each secret shard delegates six possible AC servers, token 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. In both cases, the system can also withstand the availability failure of up to three servers ($n - k$) while continuing to operate.

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.

When selecting Tutamen hosts for distributed Tutamen operation, it is desirable to select hosts with geospatial, geopolitical, and administrative diversity. Doing so reduces the likelihood that multiple servers will be subject to the same failure (e.g., a regional power outage), increases the cost of collusion (e.g., by avoiding the use of multiple Tutamen hosting providers controlled by a single administrative entity), and hinders compelled access (e.g., by locating secret shards across national boundaries where no single government can compel access to all shards). Should a system such as Tutamen ever become standardized, we also envision the formation of a competing market of Tutamen providers from which users may select individual hosts for their secrets. Such a market has the potential to align economic benefits with security best practices by server providers [52]. Although using Tuta-

[7] Since the security of Tutamen is derived from the security of the access control server, it is reasonable for a single host to operate both access control and storage servers. Such a deployment requires no more trust than a host who operates only an access control server. The Tutamen access control and storage roles are mainly split into separate servers to allow for independent scaling of each role and to promote separation of duties in the code base. Collocating both server types on a single host is not generally detrimental to the security of the system.

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

men will always entail risks with respect to trusted third parties, we believe these risks are lower for Tutamen than for existing centralized secret storage platforms. Furthermore, increasing the number of providers over which secrets or access control duties are split serves to arbitrarily reduce the degree of such risks.

4 Example 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 [9] (generally coupled with the Linux Unified Key Setup (LUKS) [20] 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-the-keyboard" security basis makes such systems difficult or impossible to use atop headless servers or in situations where no human can be expected to be present at boot-time. We have leveraged Tutamen to overcome this barrier.

To add Tutamen-support to LUKS/dm-crypt we have integrated Tutamen with the LUKS/dm-crypt bootstrapping process [37]. Our integration replaces the traditional "human-at-the-keyboard" boot-time password prompt with a request to a Tutamen storage server for necessary decryption secret (after first retrieving the necessary tokens from the corresponding Tutamen AC server). We have also integrated Tutamen support with QEMU to provide qcow2 encryption keys when a VM boots [54]. 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 process with Tutamen-based secret retrieval. 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 passphrase, 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 Storage

Cloud-based file storage systems such as Dropbox [17] are extremely popular today. Unfortunately, these systems require users to trust the cloud provider with full access to their (generally unencrypted) data. Users wishing to overcome this deficiency can optionally encrypt all of their data on the client before uploading 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 can 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 [3]: 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 [22], but without requiring out-of-band key management. Similar to other file-system-level encryption systems [7, 11, 23], 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 [14, 39] as a stream cipher to transparently stream encrypted data to/from Dropbox's servers on demand (similar in purpose to systems such as [58] and [62]). Since FuseBox leverages Tutamen to store each per-file encryption key, FuseBox natively supports Dropbox's multi-device synchronization use case by allowing a user to access and decrypt their Dropbox files from any device on which they have setup the Tutamen client. FuseBox also makes it simple 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.[9] In this manner, we have 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 [21]. FuseBox, via Tutamen's distributed operation mode, also avoids the sharing pitfalls associated many existing "secure cloud storage" providers [61] by avoiding reliance on a single trusted party to facilitate sharing operations.

5 Evaluation

We've evaluated Tutamen in a variety of scenarios using the applications described in § 4. In this section, we discuss both our existing Tutamen deployment and the performance characteristics of our Tutamen prototype.

5.1 Deployment

We've deployed a group of Tutamen servers, each either hosted locally or by an independent third party to gain geospatial, geopolitical, and administrative diversity. Our current Tutamen deployment includes three access control and storage server pairs, across which we can shard both secrets and access control duties:

France: Scaleway [55] C2S instance[10]
North Virginia: AWS EC2 [2] c4.large instance[11]
Colorado: Self-hosted instance[12]

We've utilized our experimental Tutamen deployment to manually store secrets via the Tutamen CLI, to store encryption keys for our FuseBox app, and to store encryption keys for our LUKS and our QCOW2 full-disk encryption systems. In each case, Tutamen has allowed us to realize use cases not easily attainable before such as the use of full disk encryption on headless servers and the sharding of sensitive files via Dropbox. This deployment demonstrates Tutamen's usefulness as an enabler of previously unattainable functionality in a manner that also minimizes the need for third party trust. While Tutamen is still a prototype, our experience utilizing it thus far leads us to believe it provides a well-designed architecture capable of supporting a wide range of practical secret-storage applications.

[9] While the key sharing process in FuseBox is not yet directly linked to Dropbox's file sharing system, the Tutamen CLI can be used to quickly share the encryption keys between users.

[10] Bare metal, 4-core, 8GB, Intel Atom C2550

[11] VM, 2-core, 3.75GB, Intel Xeon E5-2666

[12] VM, 4-core, 4GB virtual, Intel Xeon E3-1245

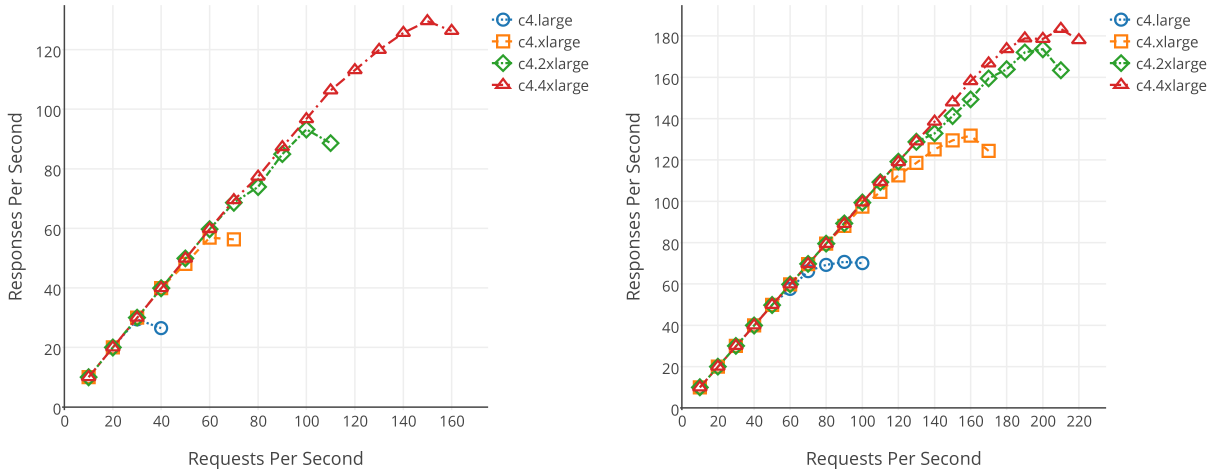
5.2 Performance

All of the scenarios in which we've used Tutamen thus far share the quality that they require a low rate of secret storage/retrieval requests. For example, FuseBox requires only a single secret store per file-create operation and a single secret lookup per file-open operation. The full disk encryption schemes are even less demanding, requiring only a single lookup per system boot. Since our current deployment has easily supported the needs of our existing Tutamen users and applications, we have not yet had a need to optimize our Tutamen deployment for performance. Nonetheless, we have performed a number of performance measurements to better understand the scaling and bottlenecks of the Tutamen system and to target future performance enhancements.

Figure 7 shows the response vs request rates of single access control and storage server deployment across a range of increasingly powerful flavors of Amazon EC2 compute-optimized instances. The curve for each instance tops out once the server reaches its maximum processing capability. Increasing the request rate beyond that point only serves to increase the response latency, and eventually leads to diminished performance due to server thrashing. Therefore, we only graph through the first data point that shows a decrease in response rate relative to previous data point, representing the asymptotic performance limit of each EC2 flavor.

Figure 7a shows performance of a stream of "Get Authorization Token" requests to a single Tutamen access control server for a verifier that requires only account membership (e.g., no authenticator modules). The maximum request rate scales fairly linearly with the number of CPU cores, ranging from around 30 RPS on a c4.large to around 130 RPS on a c4.4xlarge. Processing authorization token requests tends to be the most computationally difficult Tutamen operation: the process requires verifying both cryptographic assertions (e.g., a client certificate via the account ID) and each authenticator module (where required).[13] Figure 7b shows a similar set of curves for a stream of request to fetch a secret from a Tutamen storage server. As in the access control case, these operations scale linearly with the number of CPU cores, up until the point where they top out via the diminishing return of a c4.4xlarge instance relative to a c2.2xlarge instance. While we have not fully determined the exact cause of this limit, we believe it is a combination of our Python test client itself being only capable of generating 180 TLS requests per second and the fact that the Redis database starts to reach the limit of the instances I/O performance

[13] Processing authorizations can incur human-scale delays far in excess of the computational limits, as in cases where an SMS authenticator requires a human to receive and then respond to a text message. Tutamen clients must thus account for the possibility of significant delays when requesting tokens, generally via timeouts or asynchronous callbacks.



(a) AC Server – Get Authorization Token

(b) Storage Server – Fetch Secret

Figure 7: Scale Up Performance of Tutamen Servers on Amazon EC2 Gen4 Compute-Optimized Instances

around this point, leading us to become I/O instead of CPU bound. It should be noted that the request secret operation requires significantly fewer computational resources than the "get token" operation, leading to a performance range of 70 RPS to 180 RPS.

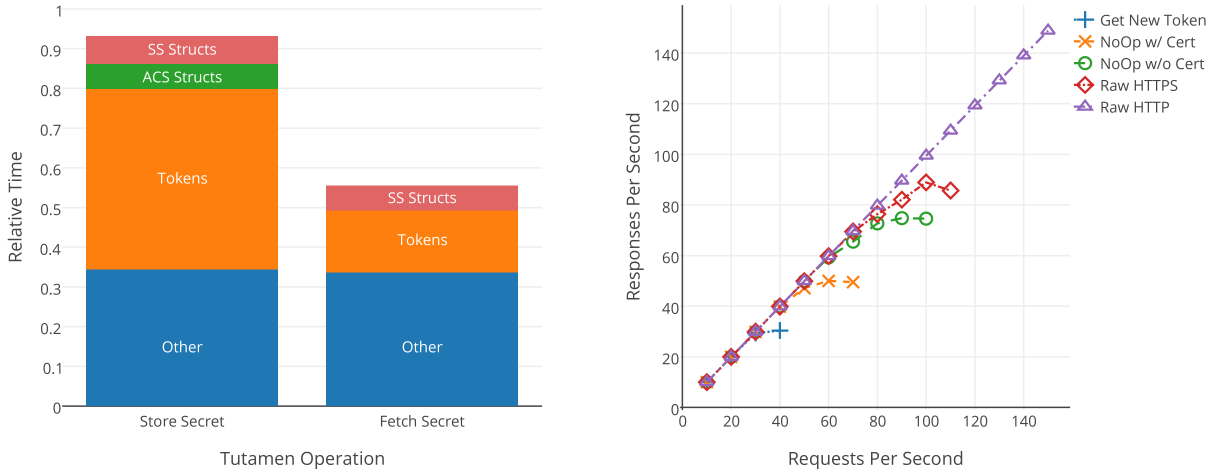
Figure 8a shows the breakdown of the relative time required to complete two of the most common Tutamen operations: storing a new secret and retrieving a previously stored secret. We profiled the amount of time the Tutamen CLI application spent performing various parts of each of these two Tutamen operations. In both operations, the bulk of the server-related runtime is spent requesting and retrieving the authorization tokens required to complete the associated operations. In the secret creation case, five tokens are required.[14] In the secret read case, only a single token is required.[15] The remainder of the server-related time is spent either creating AC and storage data structures (e.g., verifiers, collections, etc), or reading existing data structures. The "other" time is spent reading the Tutamen config files, loading the necessary client certificates, and dealing with the overhead required to setup the TLS connections and interpret the Python-based CLI.

Figure 8a shows the response vs request rate for a standard token request operation, two "No-Op" access control API operations (one that sends and verifies the client TLS certificate and one that does not), and raw Apache HTTPS and HTTP pageloads as served by an Amazon EC2 c4.large instance. As these curves show, token verification on such an instance tops out around 30 RPS. The null AC API operation with client certificates tops out around 50 RPS, and the null operation without client certificates tops out at about 75 RPS. Raw HTTPS tops out around 90 RPS. HTTP topped out around 550 rps (curve truncated for viewability). Our Tutamen prototype is thus primarily limited by the TLS overhead required to serve the application and verify client certificates. Token verification itself also incurs additional computational requirements, including cryptographic signing operations. Finally, the data retrieval itself incurs some overhead.

While the current Tutamen performance has been sufficient for our in-house needs, we have plans to optimize and increase the performance of our Tutamen deployment. On the server-side, Tutamen is designed to allow both horizontal and vertical scaling, and we foresee large deployments overcoming both the computational and I/O barriers of our current deployment through a mix of balancing requests across multiple backend instances (i.e., to overcome database and I/O limits of single instance) and the use of more powerful processing capabilities (e.g., cryptographic accelerators). We also have plans to streamline the cryptographic code and the database abstraction layer to decrease Tutamen's computational and I/O requirements. On the client, we also have plans to take advantage of Tutamen's support for long-lived (e.g., minutes to hours vs seconds) tokens to decrease the frequency with which clients must make requests to the Tutamen AC server. Finally, we also have plans to amend the Tutamen protocol to allow for batching multiple token and storage sever requests together to decrease the ratio of authentication overhead to useful work possible in a single round trip request. We believe a combination of these tech-

[14] I.e., one token to create the permissions for a new verifier, one token to create a new verifier itself, one token to create permissions for a new collection, one token to create the collection itself, and one token to store a new secret within the collection.

[15] I.e., to read a secret within the collection.



(a) Relative Time Spent Processing Store and Fetch Operations

(b) Relative Performance of Tutamen AC Server Operations

Figure 8: Timing and Performance Comparison of Tutamen Server Operation Sub-components

niques would allow us to significantly increase the performance of our Tutamen deployment with only moderate additional 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 three key secret-storage capabilities: the ability to support operation outside of a single administrative domain, the ability to 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) 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 to explore interfacing Tutamen audit logs with intrusion detection systems in order to expose more actionable intelligence to Tutamen authenticator modules. We are also considering tying Tutamen's logging infrastructure to a public audit system similar to [32]. 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 in a manner that also minimizes third party trust.

References

[1] AgileBits. 1Password. `http://agilebits.com/onepassword`.

[2] Amazon.com, Inc. Elastic Cloud Compute. `http://aws.amazon.com/ec2`.

[3] T. Andrews. FuseBox source code. `https://github.com/taylorjandrews/FuseBox`.

[4] Apache Software Foundation. Apache HTTP server project. https://httpd.apache.org/.

[5] F. Bellard and Others. QEMU: Open source process emulator. http://wiki.qemu.org/Main_Page.

[6] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the Sixth Conference on Computer Systems*, pages 31–46, 2011.

[7] M. Blaze. A cryptographic file system for UNIX. *Proceedings of the First ACM conference on Computer and Communications Security*, pages 9–16, 1993.

[8] M. Blaze. Oblivious key escrow. *Information Hiding*, 1174, 1996.

[9] M. Broz and Others. dm-crypt. http://code.google.com/p/cryptsetup/wiki/DMCrypt.

[10] J. M. A. Calero, N. Edwards, J. Kirschnick, L. Wilcock, and M. Wray. Toward a multi-tenancy authorization system for cloud services. *IEEE Security & Privacy Magazine*, 8(6):48–55, November 2010.

[11] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for UNIX. In *USENIX Annual Technical Conference*, pages 199–212, 2001.

[12] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280: Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Technical report, Internet Engineering Task Force.

[13] D. Crockford. Introducing JSON. http://www.json.org.

[14] J. Daemen and V. Rijmen. AES proposal: Rijndael. Technical report, 1999.

[15] D. E. Denning and D. K. Branstad. A taxonomy for key escrow encryption systems. *Communications of the ACM*, 39(3):34–40, March 1996.

[16] T. Dierks and E. Rescorla. RFC 5246: The transport layer security (TLS) protocol - version 1.2. Technical report, Internet Engineering Task Force, 2008.

[17] Dropbox, Inc. Dropbox. http://www.dropbox.com.

[18] P. Eby. PEP 3333 – Python web server gateway interface v1.0.1. Technical report, Python Software Foundation, 2010.

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

[20] C. Fruhwirth. LUKS. https://code.google.com/p/cryptsetup.

[21] 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.

[22] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. *NDSS*, (0121481), 2003.

[23] M. A. Halcrow. eCryptfs : An enterprise-class cryptographic filesystem for Linux. In *Ottawa Linux Symposium*, pages 201–218, 2005.

[24] HashiCorp. Vault: A tool for managing secrets. https://www.vaultproject.io/.

[25] R. J. Hulsebosch, A. H. Salden, M. S. Bargh, P. W. G. Ebben, and J. Reitsma. Context sensitive access control. In *Proceedings of the Tenth ACM symposium on Access control Models and Technologies*, page 111, New York, New York, USA, 2005. ACM Press.

[26] M. Jones, J. Bradles, and N. Sakimura. RFC 7515: JSON web signature (JWS). Technical report, Internet Engineering Task Force, May 2015.

[27] M. Jones, J. Bradles, and N. Sakimura. RFC 7519: JSON web token (JWT). Technical report, Internet Engineering Task Force, May 2015.

[28] 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.

[29] H. Krawczyk. Secret sharing made short. In D. R. Stinson, editor, *Advances in Cryptology-CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 136–146. Springer Berlin Heidelberg, Berlin, Heidelberg, July 1994.

[30] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao.

Oceanstore: An architecture for global-scale persistent storage. *ACM SIGPLAN Notices*, 35(11):190–201, 2000.

[31] LastPass. LastPass password manager. https://lastpass.com.

[32] B. Laurie, A. Langley, and E. Kasper. RFC 6962: Certificate transparency. Technical report, Internet Engineering Task Force, 2013.

[33] P. Leach, M. Mealling, and R. Salz. RFC 4122: A universally unique identifier (UUID) URN namespace. Technical report, Internet Engineering Task Force, 2005.

[34] M. A. P. Leandro, T. J. Nascimento, D. R. dos Santos, and C. B. C. M. Westphall. Multi-tenancy authorization system with federated identity for cloud-based environments using Shibboleth. *ICN 2012, The Eleventh International Conference on Networks*, pages 88–93, 2012.

[35] Lyft. Confidant: Your secret keeper. https://lyft.github.io/confidant/.

[36] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems*, 29(4):1–38, December 2011.

[37] M. Monaco. Tutamen ask-password source code. https://git.monaco.cx/matt/tutamen-ask-password.

[38] M. Monaco. Tutamen golang library source code. https://git.monaco.cx/matt/go-tutamen.

[39] National Institute of Standards & Technology. Announcing the advanced encryption standard (AES). Technical Report 197, U.S. Dept. of Commerce, 2001. Federal Information Processing Standards Publication.

[40] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, 1994.

[41] Opscode. Chef. https://www.opscode.com/chef.

[42] J. Padilla and J. Lindsay. pyJWT: A Python implementation of RFC 7519. https://github.com/jpadilla/pyjwt.

[43] Puppet Labs. Puppet. http://puppetlabs.com.

[44] Redis Team. Redis: Remote dictionary server. http://redis.io/.

[45] A. Ronacher. Flask: Web development one drop at a time. http://flask.pocoo.org/.

[46] 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.

[47] A. Sayler. pytutamen client source code. https://github.com/asayler/tutamen-pytutamen.

[48] A. Sayler. pytutamen server source code. https://github.com/asayler/tutamen-pytutamen_server.

[49] A. Sayler. Tutamen access control API source code. https://github.com/asayler/tutamen-api_accesscontrol.

[50] A. Sayler. Tutamen command line interface. https://github.com/asayler/tutamen-tutamencli.

[51] A. Sayler. Tutamen storage API source code. https://github.com/asayler/tutamen-api_storage.

[52] A. Sayler. *Securing Secrets and Managing Trust in Modern Computing Applications*. PhD thesis, University of Colorado Boulder, April 2016.

[53] A. Sayler and D. Grunwald. Custos: Increasing security with secret storage as a service. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, Broomfield, CO, October 2014. USENIX Association.

[54] A. Sayler and Others. Tutamen-aware QEMU port. https://github.com/asayler/qemu.

[55] Scaleway. Deploy baremetal ssd cloud servers in seconds. https://www.scaleway.com/.

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

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

[58] 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.

[59] Twilio. Twilio: A messaging, voice, video and authentication API for every application. `https://www.twilio.com/`.

[60] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, pages 21–26, New York, New York, USA, 2008. ACM Press.

[61] D. Wilson and G. Ateniese. To share or not to share in client-side encrypted clouds. *arXiv:1404.2697*, November 2014.

[62] Y. Xia, Y. Liu, C. Tan, M. Ma, H. Guan, B. Zang, and H. Chen. TinMan: Eliminating confidential mobile data exposure with security oriented offloading. *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*, pages 1–16, 2015.