Overnest Data Platform

Private Computing on the Public Cloud

Paul Lung, Mike Lai, and Ed Yu
Overnest Inc

Dated: March 29, 2017

Overnest is a cloud-hosted, secure data platform that leverages unique searchable encryption. Using Overnest's platform, organizations are able to utilize public cloud infrastructure while retaining the privacy level of a private cloud. Overnest's searchable encryption allows users to leverage end-to-end encryption for unparalleled privacy guarantees while maintaining the ability to perform full-text, boolean and range queries against arbitrarily nested structured data. For the first time, clients can interact with cloud-hosted encrypted data with little to no overhead.

Introduction

Many enterprises are increasing their technology footprint on the public cloud to take advantage of the vast benefits, such as the agility of a pay-as-you-go model, cost savings from economy of scale, tax benefits of operational expenditure, and simply not having to deal with the non-core competent activity of running a data center. This is especially true for small and medium-sized enterprises, where maintaining your own physical infrastructure is simply cost-prohibitive.

As attractive as the public cloud is, there are many reasons why enterprises have not yet made the switch. For organizations that regularly deal with sensitive data and/or have legal or regulatory concerns, the idea of putting data on multi-tenant infrastructure owned and controlled by another entity is a deal breaker. A rogue employee of the service provider may access or even modify the data. Governments may subpoena an organization's data from their provider in secret. An organization's data may also end up as collateral damage in a service provider's data breach. For most companies, security and data privacy remain the single largest reason against public cloud adoption.

Content Encryption

The solution to the privacy problem described above can be solved using encryption. Private data is encrypted on the client to ensure that the service provider never sees the data content in the clear. While this ensures privacy, it also impedes usability, as service providers lose the ability to index and search your data on your behalf. As an enterprise's data footprint invariably grows, this lack of search functionality becomes increasingly restrictive.

Traditional approaches involve a hybrid cloud deployment, where an unencrypted search index is maintained in a private data center, while the encrypted data resides on cloud storage. Alternatively, the index is also encrypted and stored in a public cloud, but must be downloaded and decrypted before use. Both these approaches are suboptimal, requiring an enterprise to either provision large amounts of storage, or consume large amounts of bandwidth.

Overnest introduces the world's first practical searchable encryption. Both data and index and stored in encrypted form in cloud-hosted storage. Search queries themselves are encrypted client-side, and only return relevant encrypted results. The service provider never learns what is being stored or queried for.

Searchable Encryption

The Overnest Data Platform is an encrypted key-object storage platform. Objects consist of structured fields. Fields can be arbitrarily nested, where field values are themselves objects that contain their own set of structured fields. Objects support queries on their fields or across all fields in an object. They can also be queried on a range of combined with boolean operators for more complex queries.

With Overnest, object data is indexed in real-time, when it is written to cloud storage. Indices are specially structured to allow selective access with encrypted queries, retrieving only relevant information from the encrypted index, without the need for mirroring the index locally, or downloading the full copy and decrypting it before becoming searchable.

Key	Object
`Landlord1`	`{ Name: John Smith, Age: 25, Properties: [ { Address: 3571 Nevada Circle, City: San Jose, State: California, Zip: 92347, Years Owned: 5}, { Address: 1245 Poplar Street, City: Las Vegas, State: Nevada, Zip: 92445, Years Owned: 1 } ] }`
`Landlord2`	`{ Name: Mike Jones, Age: 40, Properties: [ {`

```
       Address: 4980 Godfrey Road,
       City: Reno,
       State: Nevada,
       Zip: 92347,
       Years Owned: 20 },
     {
       Address: 3381 Roosevelt Road,
       City: San Jose,
       State: California,
       Zip: 92344,
       Years Owned: 15 }
  ]
}
```

Figure 1: Example keys and data for Overnest key object store

Given the data above, a user can issue simple queries like “Name = John Smith”, and the result would be “Landlord1”. A more complex query utilizing both field and range search like “Properties.City = San Jose AND Age > 30” would produce the result “Landlord2”. Lastly, the user can also perform full-text searches across all fields, such as “Smith AND Circle”, returning “Landlord1” as only that object contains both those words.

When the query is performed, the query is encrypted before leaving the client, and the query result is decrypted only after it reaches the client. The service provider learns neither the query, nor the result.

Encrypted Term Query

A term query finds which keys contain a certain word, or a series of words. For example, a simple term query could be “Smith” or “John Smith”, both of which would result in “Landlord1”. This type of search is achieved by building an inverted index of all the terms, encrypting it, and uploading it to the cloud server.

Index Creation	
Plain Text Index	Encrypted Index
`John -> [“Landlord1”]`	`hmac(John) -> aes256([“Landlord1”])`
`Smith -> [“Landlord1”]`	`hmac(Smith) -> aes256([“Landlord1”])`
`Name -> [“Landlord1”, “Landlord2”]`	`hmac(Name) -> aes256([“Landlord1”, “Landlord2”])`

Figure 2: Plaintext terms like “John” and “Name” are HMAC-hashed, and the index content is encrypted with AES256.

Once an encrypted index is generated and storage on the server, a client can perform queries by generating a HMAC hash of the term, transmitting that to the cloud server. The server responds with a matching encrypted index, while the client decrypts the index to find matching keys for the query term.

Proximity searches can also be supported by including positional information in the index. For example, searching for “John Smith” is achieved by retrieving the index for both “John” and “Smith”, and returning keys where both terms are adjacent.

Encrypted Range Query

A range query requires a different method from a term query, and the performance tends to be quite a bit slower. Range queries are achieved by manipulating an encrypted B+ Tree remotely[4].

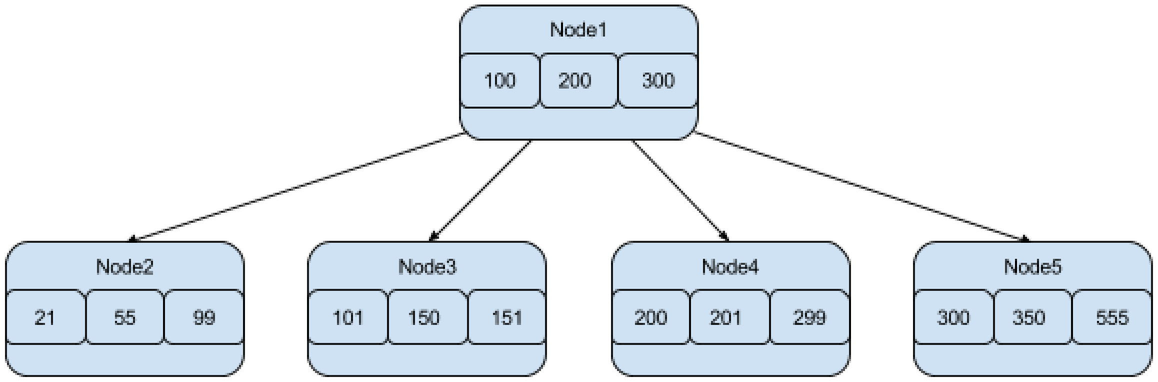

Figure 3: Encrypted B+ Tree.

The client can perform a range query by asking the server for the root encrypted node, decrypting it, and then determining which child nodes to obtain next. For example, if the client wishes to perform the query “26 < n < 160”, it would first get Node1, the root node. Decrypting Node1 would reveal the need to fetch Node2 and Node3. Finally, decrypting both Node2 and Node3 will yield the final answer of “55, 99, 101, 150, 151”.

Unlike a term query, a range query is inherently serialized. While client-side caching can reduce the number of server requests, the client still must query and decrypt each level before knowing the next level of nodes to get, since even the “child” pointers in each level are encrypted. It is also possible to make the child pointers unencrypted, which would allow the server to return an arbitrary number of nodes in bulk. However, while this scheme increases performance, it leaks relational information to the service provider, since the parent-child relationship is unencrypted.

Encrypted Field Query

Overnest supports querying on specific fields within a stored object. Using the landlord example in Figure 1, querying for “John Smith” would return the result key “Landlord1”. However, querying “John Smith” in the field “Properties.Address” would result in zero matches, since no objects contain the terms “John Smith” in the “Properties.Address” field.

Field search is achieved by maintaining the following structure in the encrypted index:

Term	Index

San	{Properties.City: [
	    {Landlord1: [1]},
	    {Landlord2: [2]}
	]}

Jose	{Properties.City: [
	    {Landlord1: [1]},
	    {Landlord2: [2]}
	]}

Nevada	{Properties.Address: [
	    {Landlord1: [1]}
	],
	 Properties.State: [
	    {Landlord1: [2]},
	    {Landlord2: [1]}
	]}

. . .	. . .

Figure 4: Index structure for encrypted field query

This structure describes the fields and the corresponding keys that a term appears in, and is the means by which results can be filtered by the field they appear under. For example, the term “Nevada” appears both in the “Properties.Address” field of “Landlord1”, and “Properties.State” field of both “Landlord1” and “Landlord2”. If the field is an array, we also record the position(s) in the array that the term appears in. This is required to return the proper result for a query like “Properties.City = San Jose AND Properties.State = Nevada”. Since no single entry within the “Properties” list contains both “San Jose” as the city and “Nevada” as the state, the query result should be empty.

Other Encryption Schemes Considered

Other encryption schemes were considered but ultimately disregarded. These include Fully Homomorphic Encryption (FHE)[1], Partially Homomorphic Encryption (PHE)[2], and Multiparty Computation (MPC)[3]. FHE and PHE were dismissed for their impractical performance. For example, it takes somewhere in the order of 5-10 minutes just to bootstrap a relatively small amount of plaintext for encryption. The same problem applies to MPC algorithms. AES encryption circuits can be built out of MPC, but such circuits take somewhere in the order of 20 minutes to compute even a small amount of ciphertext.

Key Distribution and Object Sharing

Overnest additionally enables users to share access to encrypted data. Objects are separated by user-defined “partitions”, where each partition’s data shares a single encryption key. Since Overnest relies on AES for encryption, sharing is achieved via key distribution mechanisms to authorized users in additional to traditional access control lists (ACLs). When a user’s access is revoked, the platform immediately disables access via ACLs, and optionally re-encrypts previously shared objects in the background with a new, unique AES key. This is only required if forward secrecy is desired, further cryptographically enforcing access control.

Granting Access To Shared User

Granting permission to a user utilizes asymmetric encryption such as RSA, coupled with proxy re-encryption. Since asymmetric encryption is typically much slower than schemes like AES, and work on a small upper-bound of data, this is only used to transmit symmetric encryption keys securely. Every user (or client) in the Overnest system has a public/private key pair. Public keys are stored on the Overnest server along with user account information.

When User A shares access to a data partition with User B, the encryption key is encrypted with User B’s public key, and the result is stored with Overnest. User B is later able to retrieve the encrypted payload and decrypt on the client with their private key. However, when sharing a large number of partitions, this scheme places undue burden on User A to encrypt multiple keys. This problem is solved using proxy re-encryption.

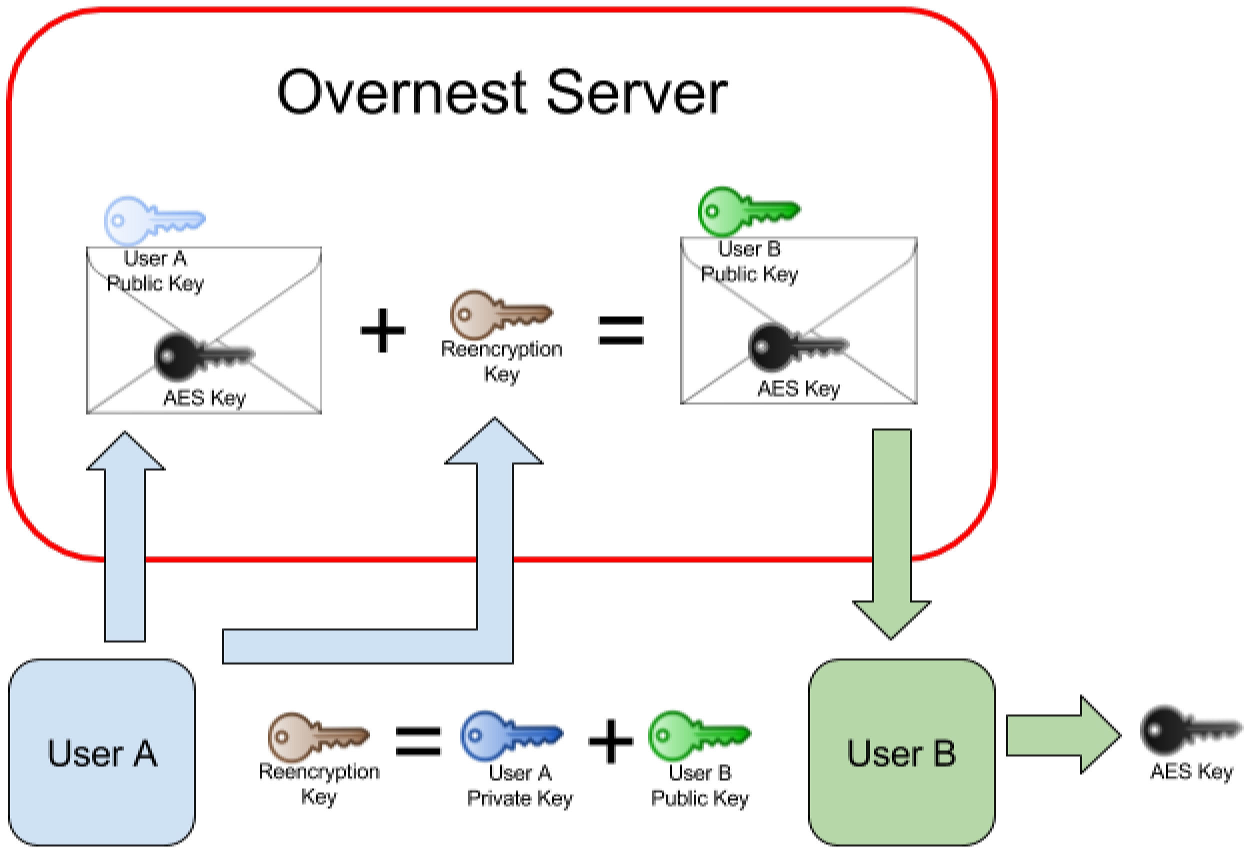

Figure 5: Proxy Re-encryption For Key Distribution

Using proxy re-encryption, when User A shares an AES key with User B, User A does the following:

1. Encrypts the AES key with User A’s public key and uploads it to the server. This step is not actually necessary when granting access, because the encrypted AES key is typically uploaded to the server and stored with the object when the object itself is written for the first time.
2. Generates a re-encryption key (Proxy Key) by combining User A’s private key and User B’s public key. Upload the Proxy Key to the server.

When User B retrieves a shared encrypted key, the Overnest server can use the Proxy Key to convert the AES key encrypted with User A’s public key into an AES key encrypted with User B’s public key. Without access to either User A or User B’s private keys, Overnest is still unable to produce a valid decryption key from these components. This allows Overnests to assume the primary burden of key sharing.

Revoking Access To A Shared User

Access revocation is achieved using standard access control methods. However, since the revoked user may already have an object’s AES key, she can, in theory, regain access to the encrypted object stored on the server if access control is somehow bypassed. To guarantee forward secrecy, a new AES key must be generated and the object re-encrypted, rendering the old AES key from the revoked user useless.

Unfortunately, proxy re-encryption does not exist for symmetric ciphers like AES. It is not currently possible for the server to re-encrypt data on behalf of users without access to the data's encryption key. To mitigate the requirement for client-side re-encryption, this can be performed only on object writes. Since write operations require the retrieval, decryption and re-encryption of objects, re-encrypting with a new key at this time introduces no new overhead. Prior to an object write, revoked users would have access to no additional information from the time their access was revoked.

---

[1] Frederik Armknecht , Colin Boyd , Christopher Carr , Kristian Gjøsteen , Angela J¨aschke , Christian A. Reuter, and Martin Strand. “A Guide to Fully Homomorphic Encryption”.
[2] Liam Morris. “Analysis of Partially and Fully Homomorphic Encryption”
[3] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. “Secure Two-Party Computation is Practical”
[4] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, Yirong Xu. “Order Preserving Encryption for Numeric Data”