Big Data Security in Apache Projects

Gidon Gershinsky

ApacheCon North America 2022

THIS IS NOT A CONTRIBUTION

Presenter

Gidon Gershinsky

- Committer and PMC member in Apache Parquet
- Works on big data security in Apache Parquet, Spark, Iceberg and other projects
- Designs and builds data security solutions at Apple

Contributors

Apache community!

- Data security work: 2018 - present
- Projects: Parquet, Spark, Arrow, Iceberg, Flink, Trino

Antoine Pitrou, DB Tsai, Eliot Salant, Jack Ye, Jian Tang, Julien LeDem, Gabor Szadovszky, Gyula Fora, Huaxin Gao, Itamar Turner-Trauring, Maya Anderson, Muhammad Islam, Revital Eres, Roee Shlomo, Russell Spitzer, Ryan Blue, Steven Wu, Tim Perelmutov, Tham Ha, Tomer Solomon, Vinitha Gankidi, Xinli Shang, Yufei Gu, Zoltan Ivanfi

Thank you!!

Agenda

Big Data Security

- Our focus and goals

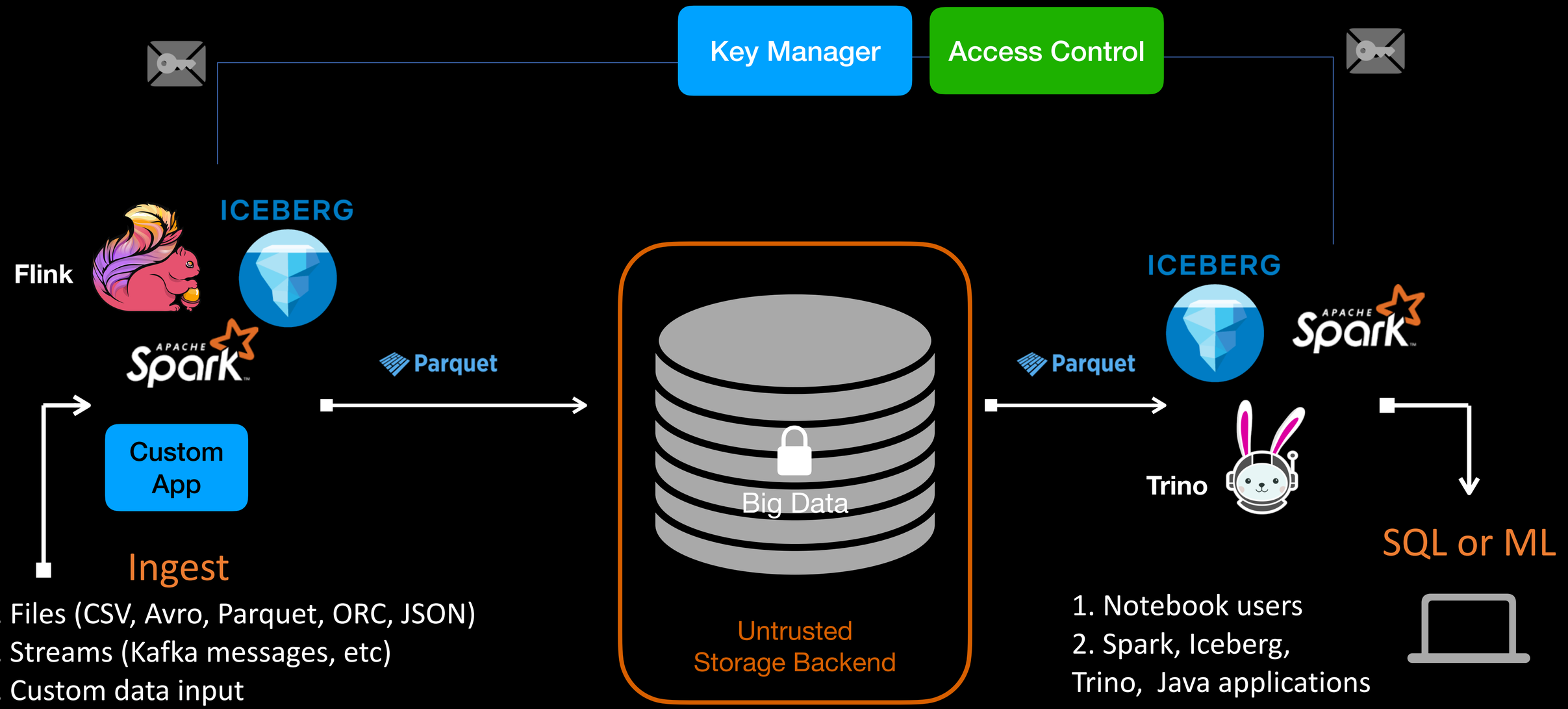
Foundation: Apache Parquet

- Security features; APIs and “HelloWorld” samples
- Performance effect of encryption

Usage in Apache Spark, Iceberg, Arrow, Flink, and Trino

Integration options in other Apache Big Data projects

Data Lake


Key Manager
Access Control
ICEBERG
Flink
APACHE Spark
Parquet
Custom App
Ingest
1. Files (CSV, Avro, Parquet, ORC, JSON)
2. Streams (Kafka messages, etc)
3. Custom data input
Big Data
Untrusted Storage Backend
Parquet
ICEBERG
APACHE Spark
Trino
SQL or ML
1. Notebook users
2. Spark, Iceberg, Trino, Java applications

Goal: Protect Sensitive Data-at-Rest

Keep the data Confidential

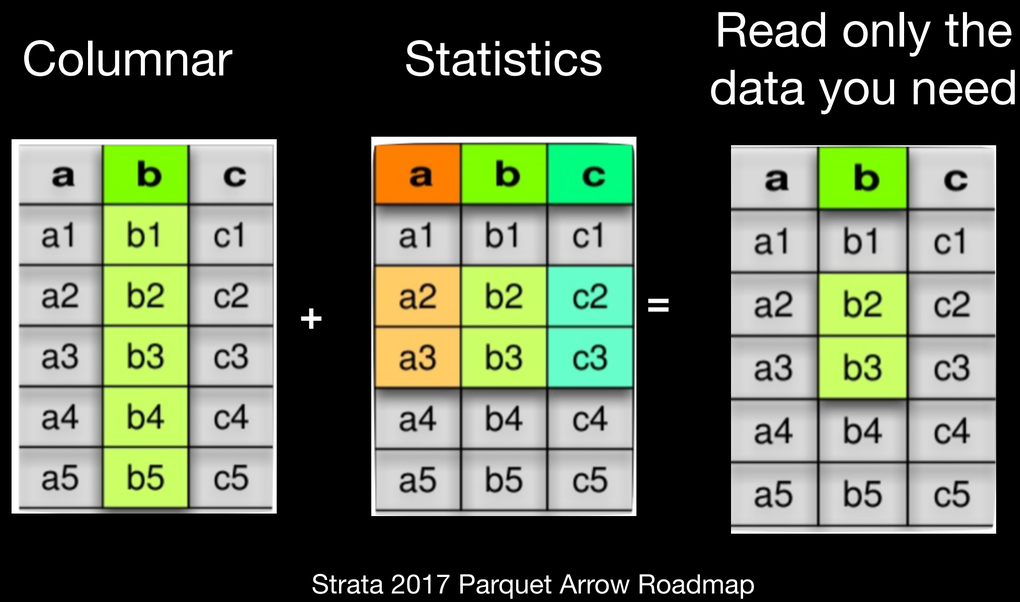
- -hiding sensitive information in storage
- -via encryption

Keep the data Tamper-Proof

- -protecting integrity of sensitive information in storage
- -via crypto-signatures and module IDs

Apache Parquet

- **Popular columnar storage format**
- **Encoding, compression**
- **Advanced data filtering**
  - columnar projection: skip columns
  - predicate push down: skip files, or row groups, or data pages
- **Built-in encryption since 2021**



Strata 2017 Parquet Arrow Roadmap

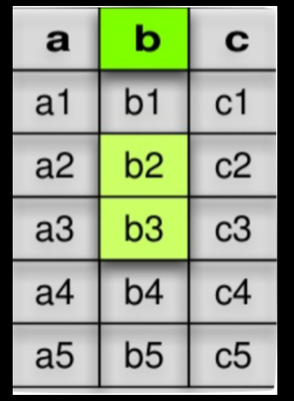
Parquet

Parquet Modular Encryption

Goals

- **Encrypt and sign all modules in Parquet files (data and metadata modules)**
- **Preserve full Parquet capabilities** (columnar projection, predicate pushdown, compression, etc) **in encrypted files**
- **Preserve performance of analytic engines with encrypted files**

Read only the data you need



a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5

2017 Parquet Arrow Roadmap

Parquet

Parquet Modular Encryption

Open standard for safe storage of analytic data

- ***Parquet Format*** **spec for encryption approved in 2019**
- **Works the same in any storage**
  - cloud or private, file systems, object stores, archives
- **Integrated in a number of Apache Big Data frameworks**
  - write (encrypt) with one framework, read (decrypt) with another
- **Supports any KMS (key management service)**
- **Per-file and per-column encryption keys**

Parquet

Parquet Modular Encryption

Use-cases

- **Encryption and tamper-proofing of sensitive big data in storage**

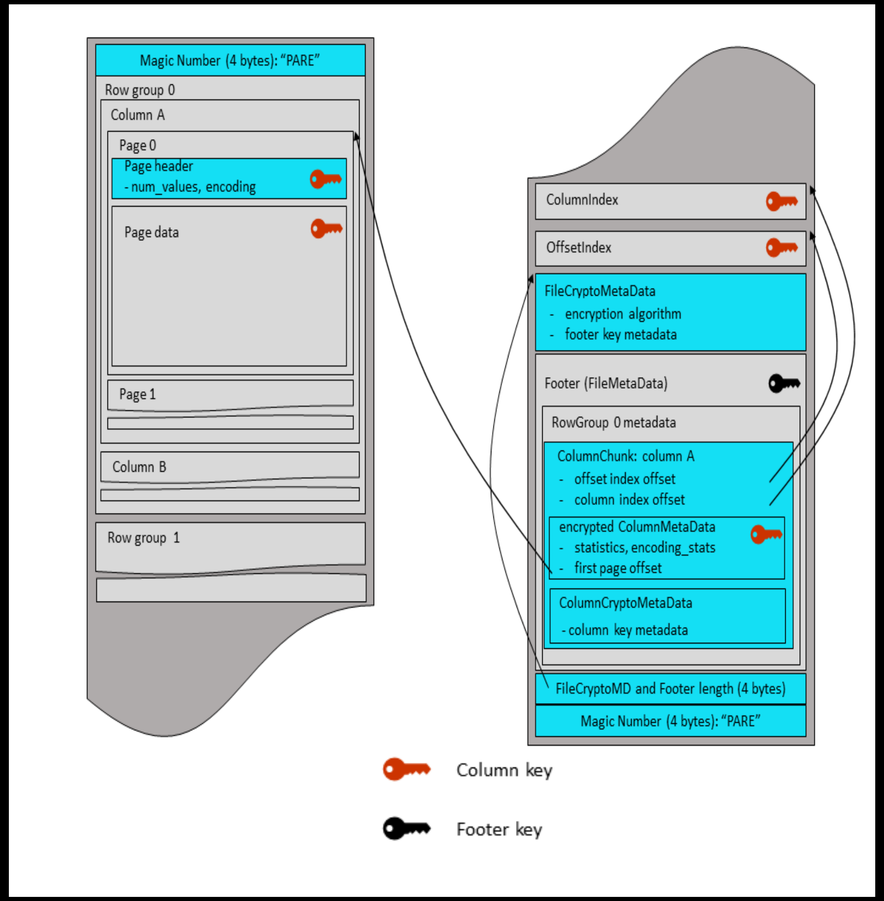
- **Efficient migration from one storage to another**
  - no need to import / decrypt / encrypt / export
  - no need to change key management system (KMS)
  - simply move the files: eg cloud -> cloud; or on-prem -> cloud -> archives

- **Sharing data subset / table column(s)**
  - no need to extract / encrypt a copy for each user
  - simply provide key access to eligible users (table keys or column keys)

Parquet

Keep Data Confidential

- **Full encryption mode**
  - all modules are hidden
- **Plaintext footer mode**
  - footer is exposed for legacy readers
  - sensitive metadata is hidden
- **Separate keys for sensitive columns**
  - column access control
- **New: uniform encryption key (all file columns)**
- **“Client-side” encryption**
  - storage backend / admin never see data or keys

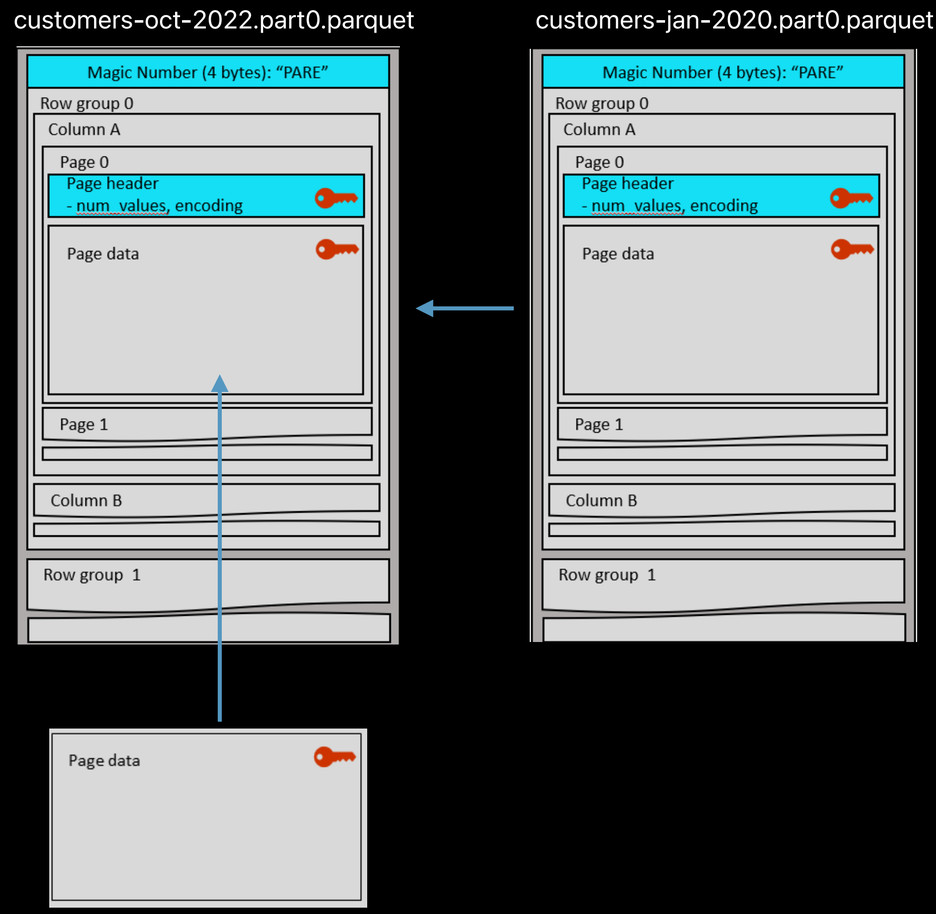


Parquet

Keep Data Tamper-proof

- **File contents not tampered with**
- **File not replaced with wrong file**

- Signs data and metadata modules
  - with unique module IDs and file IDs
- AES GCM: “authenticated encryption”



Parquet

Performance of Encryption

AES ciphers implemented in CPU hardware (AES-NI)

- Gigabyte(s) per second in each thread
- Order(s) of magnitude faster than “software stack” (App/Framework/Parquet/ compression)
- C++: OpenSSL EVP library

Java AES-NI

- AES-NI support in HotSpot since Java 9
- Java 11.0.4 – enhanced AES GCM decryption
- Thanks Java community!

Bottom line: Encryption won’t be your bottleneck

app workload, **data I/O**, compression, etc

Envelope Encryption

- **Parquet file modules are encrypted with “Data Encryption Keys” (DEKs)**
  - “Big Data” - lots of DEKs (one per file or even file/column - per NIST requirements for AES GCM)
- **DEKs are encrypted with “Master Encryption Keys” (MEKs)**
  - Result is called “key material” and stored either in Parquet file footers, or in separate files “close to data”
- **MEKs are stored and managed in “Key Management Service” (KMS)**
  - Access control verification
  - Number of MEKs << number of DEKs (1 MEK can encrypt ~ 2 billion DEKs)

Apache Parquet-MR Encryption API

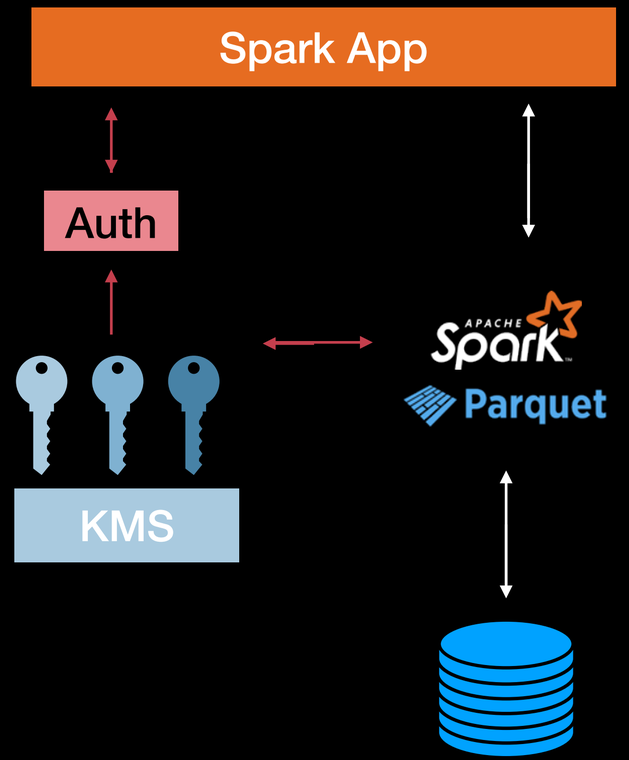
- **High-level API**
  - Encryption parameters passed via Hadoop configuration
  - Envelope encryption and DEK storage handled by Parquet
  - Out-of-box integration (just update parquet version to 1.12+, and plug your KMS)

- **Low-level API**
  - Direct calls on low-level Parquet API (before encryption)
  - Direct DEK handling

Parquet

Spark with Parquet Encryption

- **Configure encryption via Hadoop parameters**
  - Pass list of columns to encrypt
  - Specify IDs of master keys for these columns
  - Specify ID of master key for Parquet footers
  - Pass class name for client of your KMS
  - Activate encryption



- **Since Spark 3.2.0**
  - doc: spark.apache.org/docs/latest/sql-data-sources-parquet.html#columnar-encryption

- **Coming up: uniform file encryption**
  - Released in Parquet, and merged in Spark master branch

HelloWorld: Writing Encrypted Files

- Run spark-shell
- “Activate” encryption

```
sc.hadoopConfiguration.set(“parquet.crypto.factory.class" ,
                “org.apache.parquet.crypto.keytools.PropertiesDrivenCryptoFactory”)
```

- Pass master encryption keys (demo only!)

```
sc.hadoopConfiguration.set("parquet.encryption.kms.client.class" ,
                      "org.apache.parquet.crypto.keytools.mocks.InMemoryKMS")

sc.hadoopConfiguration.set(“parquet.encryption.key.list” ,
                      “k1:AAECAwQFBgcICQoLDA0ODw== ,  k2:AAECAAECAAECAAECAAECAA==")
```

HelloWorld: Writing Encrypted Files

- Write dataframe: “columnA” will be encrypted

```
sampleDF.write.
    option("parquet.encryption.footer.key" , "k1").
    option("parquet.encryption.column.keys" , "k2:columnA").
  parquet("/path/to/table.parquet.encrypted")
```

- Column key format

```
"<masterKeyID>:<colName>,<colName>;<masterKeyID>:<colName>, ..
```

HelloWorld: Reading Encrypted Files

- Run spark-shell
- “Activate” decryption

```
sc.hadoopConfiguration.set(“parquet.crypto.factory.class" ,
                “org.apache.parquet.crypto.keytools.PropertiesDrivenCryptoFactory”)
```

- Pass master encryption keys (demo only!)

```
sc.hadoopConfiguration.set("parquet.encryption.kms.client.class" ,
                      "org.apache.parquet.crypto.keytools.mocks.InMemoryKMS")

sc.hadoopConfiguration.set(“parquet.encryption.key.list” ,
                      “k1:AAECAwQFBgcICQoLDA0ODw== ,  k2:AAECAAECAAECAAECAAECAA==")
```

- Read encrypted dataframe

```
val df = spark.read.parquet(“/path/to/table.parquet.encrypted")
```

Coming up: Uniform encryption

- Set same Hadoop configuration as before

- Write dataframe with uniform encryption

```
sampleDF.write.
   option("parquet.encryption.uniform.key" , "k1").
 parquet("/path/to/table.parquet.encrypted")
```

- Read encrypted dataframe

```
val df = spark.read.parquet(“/path/to/table.parquet.encrypted")
```

HelloWorld: Parquet API

Create encryption properties

```
Configuration hadoopConfiguration = new Configuration();

hadoopConfiguration.set("parquet.crypto.factory.class" ,
                "org.apache.parquet.crypto.keytools.PropertiesDrivenCryptoFactory");

hadoopConfiguration.set("parquet.encryption.kms.client.class" ,
                "org.apache.parquet.crypto.keytools.mocks.InMemoryKMS");

hadoopConfiguration.set(“parquet.encryption.key.list" ,
                “k1:AAECAwQFBgcICQoLDA0ODw==, k2:AAECAAECAAECAAECAAECAA==");

hadoopConfiguration.set("parquet.encryption.footer.key" ,  "k1");

hadoopConfiguration.set("parquet.encryption.column.keys" ,  “k2:columnA”);
```

Parquet

HelloWorld: Parquet API

Write data

```
EncryptionPropertiesFactory cryptoFactory =
    EncryptionPropertiesFactory.loadFactory(hadoopConfiguration);


FileEncryptionProperties fileEncryptionProperties =
    cryptoFactory.getFileEncryptionProperties(hadoopConfiguration,
                                              </path/to/file>, null);

ParquetWriter writer = ParquetWriter.builder(<path/to/file>)
    .withConf(hadoopConfiguration)
    …
    .withEncryption(fileEncryptionProperties)
    .build();


// write as usual
```

Parquet

HelloWorld: Parquet API

Read data

```
DecryptionPropertiesFactory cryptoFactory =
    DecryptionPropertiesFactory.loadFactory(hadoopConfiguration);


FileDecryptionProperties fileDecryptionProperties =
    cryptoFactory.getFileDecryptionProperties(hadoopConfiguration,
                                              </path/to/file>);

ParquetReader reader = ParquetReader.read(InputFile)
    …
    .withDecryption(fileDecryptionProperties)
    .build();

// read as usual
* No need to pass footer and column key properties
```

Parquet

Real World

- **Master keys are kept in your KMS (Key Management Service) in your platform**
  - controls key access for users in your organization
  - if your org doesn't have KMS: use cloud KMS, or install an open source KMS
- **Develop client for your KMS server**
  - Implement KMS client interface

```
public interface KmsClient {


  // encrypt e.g. data key with master key (envelope   encryption)
  String wrapKey(byte[] keyBytes, String masterKeyIdentifier)

  // decrypt key
  byte[] unwrapKey(String wrappedKey, String masterKeyIdentifier)
}
```

Parquet

Example: Hashicorp Vault

- Open source KMS
- Search for VaultClient in github.com/apache/parquet-mr
- Set up encryption

```
sc.hadoopConfiguration.set("parquet.crypto.factory.class" ,
                           “org.apache.parquet.crypto.keytools.PropertiesDrivenCryptoFactory”)

sc.hadoopConfiguration.set("parquet.encryption.kms.client.class"  ,
                           “org.apache.parquet.crypto.keytools.samples.VaultClient”)

sc.hadoopConfiguration.set("parquet.encryption.key.access.token"  , "<vault token>")

sc.hadoopConfiguration.set("parquet.encryption.kms.instance.url"  , "<vault server url>")
```

Example: Hashicorp Vault

- Write dataframe with encrypted columns

```
sampleDF.write.
  option("parquet.encryption.footer.key" , "k1").
  option("parquet.encryption.column.keys" , "k2:columnA").
parquet("/path/to/table.parquet.encrypted")
```

- Read dataframe with encrypted columns

```
val df = spark.read.parquet("/path/to/table.parquet.encrypted")
```

Where to Store “Key Material”

- **“key material”: wrapped DEK plus MEK ID. No secret info!**
- **Storage options**

  1: Inside parquet file (footer)

  - straightforward
  - default mode
  - shortcomings: can’t rewrap DEKs (change MEK), or change KMS

Parquet

Where to Store “Key Material”

- **Storage options (continued)**

  2: Small separate “key_material” files (one per parquet file, same folder)

```
sc.hadoopConfiguration.set(“parquet.encryption.key.material.store.internally" ,
                           “false”)
```

  - allows to re-wrap / re-encrypt DEKs
    - with new MEK or MEK version: if compromised, or change in user access rights
    - migrate to new KMS
  - shortcomings: addition cost of reading the small key_material file, for each Parquet file

Where to Store “Key Material”

- **Storage options (continued)**

3: Apache Iceberg: native metadata files!
Wrapped DEKs e.g. in manifest files

- no shortcomings! (re-wrapping is possible, no extra files)

Apache Spark

Standalone Spark

- uses high-level Parquet-MR API
- parquet encryption works out of box
- parquet-mr updated to 1.12 in Spark version 3.2.0
- encryption activation and configuration via Hadoop parameters
- DEK storage in either parquet file footers or external key material mode

Spark with Iceberg

- WIP on Table encryption in Apache Iceberg community
- DEK storage in Iceberg metadata files

Apache Flink

Standalone Flink

- Uses high-level Parquet-MR API
- Parquet encryption works out of box, like in Apache Spark
- Parquet-mr updated to 1.12.2 in Flink version 1.15.0
- Encryption activation and configuration via Hadoop parameters
- DEK storage only in parquet file footers (no external key material mode)
  - if need external key material, use Spark to collect/ compact Flink output
  - or update Apache Flink code to enable external mode

Flink with Iceberg

- WIP on Table encryption in Apache Iceberg community
- DEK storage in Iceberg metadata files

Apache Trino

Standalone Trino

- Uses its own parquet implementation (with parts of parquet-MR)
- Xinli Shang: PR for read part https://github.com/trinodb/trino/pull/9871
  - Apache Trino community: please review and merge!
- Decryption works via Hadoop parameters
  - DEKs stored either in parquet file footers or in external key material files
- Interoperable with Spark and Flink

Trino with Iceberg

- WIP on Table decryption in Apache Trino and Iceberg communities
- DEK storage in Iceberg metadata files

Apache Arrow

- **C++ parquet implementation (including low-level encryption API)**

- **Python implementation of high-level parquet encryption API**
  - encryption activation and configuration via PyArrow parameters (similar to Hadoop examples)
  - supports envelope encryption
  - currently, DEK storage only in parquet file footers
    - external key material mode WIP (PR)
  - interoperable with Spark and Flink

Apache Iceberg

- **Has metadata files!!**
  - allows for optimal storage and management of encryption keys (DEKs)
- **Based on parquet-MR, with some low-level calls and changes**
- **Encryption project (github.com/apache/iceberg/projects/5)**
  - use parquet encryption standard format
  - build key management: based on envelope encryption, and DEK storage in metadata files
- **MVP version**
  - Uniform data encryption : Table key
    - all table columns encrypted with same MEK

Apache Iceberg

- **Data security roadmap**
  - MVP version (some PRs are merged, some WIP)
    - uniform data encryption : Table key
  - Manifest encryption
  - Column encryption
  - Manifest list encryption
  - Beyond parquet: ORC and Avro data files
  - KMS optimization
    - double-wrapping (no KMS calls per file / column)
    - two-tier key management (only driver node talks to KMS, not worker nodes)
  - Table integrity / tamper-proofing
    - encrypting and signing each data and metadata file
    - end to end protection of big data tables! (confidential AND tamper-proof)

Other Apache Big Data projects

- **Adding encryption can range from 0-effort (just turn it on) to a sizable effort**

- **Encryption checklist for each project**
  - supports Parquet format?
    - which implementation? (own? parquet-mr? pyarrow? ..)
  - uses high-level or low-level parquet API?
  - has its own metadata files?

- **Feel free to contact me!** **gershinsky@apache.org**

Questions!

gershinsky@apache.org