**Linkja-crypto**

C library to handle cryptographic functions. Because this is intended to be used by the linkja programs (which are written in Java), this includes the Java Native Interface (JNI) headers with the C code.

## Building on macOS

### Dependencies

Setup and building of the linkja-crypto library requires the following:

1. [CMake](https://cmake.org/) (3.8 or higher)
2. Java (1.8 or higher; [OpenJDK](https://openjdk.java.net/) is allowed)
	1. Requires the JDK to be installed
	2. Will use javac and java binaries
	3. Requires Java Native Interface (JNI), which comes with most JDKs
3. [OpenSSL](https://www.openssl.org/) - the openssl binary needs to be in your path for the build scripts to work. You can test this by executing "openssl version" at the command line.
4. C compiler and make system
	1. For macOS, gcc and make can be used.
	2. For Windows, Visual Studio 2019 Community edition can be used.
5. [cmocka](https://cmocka.org/) for unit tests
	1. For macOS, brew install cmocka
	2. For Windows, [download the latest source code](https://cmocka.org/files/1.1/)
		1. Extract the .tar.xz to a folder
		2. You can use Visual Studio to open the folder and build via CMakeList

### Compiling

There are multiple components that go into building the linkja-crypto library. For macOS, these are wrapped up into the build.sh script (available from the root directory).

1. Create the JNI header file

$JAVA\_HOME/bin/javac -h ./src/include ./src/java/Library.java

1. Ensure cmake targets are built and/or updated

cmake .

1. Build the library

make clean

make

### Testing

To ensure everything is set up correctly, you can compile and run a simple test program using the testing script:

 ./test.sh

## Building on Windows

These instructions were developed on a 64-bit Windows 10 Enterprise version 20H2. To ensure completeness, we started from a completely fresh install of Windows. It's possible that you may already have some of the dependencies installed. Given the number of versions of frameworks and installers, we haven't tested this with more than the specific versions listed. If you have success (or problems) with other verions, please let us know.

### Install Visual Studio 2019

[Download Visual Studio 2019 Community Edition](https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=Community&rel=16)

During the installation and setup, you should select:

* Linux Development with C++
* Desktop Development with C++

NOTE: Although we are installing the Visual Studio 2019 IDE, this is primarily to get the compilers and tooling that it provides. We won't use the IDE to actually compile the code.

**Alternate**: If you already have Visual Studio installed, instructions for setting up these pieces can be found at the following links:

* [Install Linux tools](https://docs.microsoft.com/en-us/cpp/linux/download-install-and-setup-the-linux-development-workload?view=vs-2019)
* [CMake project integration with Visual Studio](https://docs.microsoft.com/en-us/cpp/build/cmake-projects-in-visual-studio?view=vs-2019)

### Download CMocka for MSVC

For this, we used the executable installer of CMocka 1.1.0 MSVC - <https://cmocka.org/files/1.1/cmocka-1.1.0-msvc.exe>

### Java

You will need to inistall Java 1.8 or higher. [OpenJDK](https://openjdk.java.net/) is recommended

1. Linkja requires the JDK (not just the JRE) to be installed
2. Will use javac and java binaries
3. Requires Java Native Interface (JNI), which comes with most JDKs

For these instructions, we will get a Windows installer from [AdoptOpenJdk](https://adoptopenjdk.net/).

For this, we used version OpenJDK 11 (LTS) with the HotSpot JVM.

In the setup, ensure all of the options are set:

* Add to PATH
* Associate .jar
* Set JAVA\_HOME variable
* JavaSoft (Oracle) registry keys

### OpenSSL

You will need OpenSSL tools and libraries. We ran into issues finding a suitable OpenSSL build for Windows that would work for this process. Although you may not wish to blindly and use our version, [we have made a build of OpenSSL 1.1.1f available](https://northwestern.box.com/s/jo0i43676clexastg5p795r0bejzaf10) and will use it in these instructions.

You will need to download the ZIP file, and extract the contents. For these instructions we have placed the files in C:\Program Files\OpenSSL.

### Download linkja-crypto

Clone the repository from <https://github.com/linkja/linkja-crypto.git> using whatever git client you prefer.

**Instructions for Visual Studio 2019**

1. From Visual Studio 2019, click on "Clone a repository".
2. "Repository location" - <https://github.com/linkja/linkja-crypto.git>
3. "Path" - feel free to use the default, or customize. For this guide we are placing the code in C:\Users\Linkja\Source\Repos\linkja-crypto
4. Go ahead and close Visual Studio at this time. As noted above, we will not be using the IDE to compile.

### Open the Developer Command Prompt

In order for the build process to work, you will need to run the x64 Native Tools Command Prompt for VS 2019. This can be navigated to from the Windows Start menu, and can be found under the Visual Studio 2019 folder. There are similarly named options, so please confirm that you have selected the right one.

First, we will set up our command prompt PATH environment variable to include the path to the OpenSSL binaries.

SET PATH=%PATH%;"C:\Program Files\OpenSSL\vc-win64a\bin"

Next we will change to the directory where we downloaded the linkja-crypto code

cd C:\Users\Linkja\Source\Repos\linkja-crypto

The following commands will ensure that any previous build artifacts are removed.

del CMakeCache.txt

del src\linkjacrypto.exp

del src\linkjacrypto.lib

del src\linkjacrypto.dll

del src\linkjacrypto.dll.manifest

The following command is the key one for linkja-crypto. Linkja-crypto uses a generated header file with a random hash, which should only be used once per project and then removed. However, just know that once you delete the header file you won't be able to generate the same crypto library again.

del src\include\linkja\_secret.h

Finally, the following commands will run our CMake scripts and then run the actual compilation process.

cmake -DCMAKE\_BUILD\_TYPE=Release -G "NMake Makefiles" .

nmake clean

nmake

### Finishing Up

At this point you will see in the src subdirectory the following files:

* linkjacrypto.exp
* linkjacrypto.lib
* linkjacrypto.dll
* linkjacrypto.dll.manifest

While all of them can be shared, you only really need linkjacrypto.dll. This should be distributed to those who need to run hashing. You should delete these files as well as the linkja\_secret.h once you are done and the DLL has been distributed.

**Linkja-Salt**

The salt engine generates unique encrypted salt file for each site in the project.

For new projects, the Key Master, will need below inputs:

1. The Project Name
2. Unique Site ID and name for each participating site

There is also an option to add more sites to an ongoing project. For adding sites to an ongoing project, one of the existing participating site should take on the role (acting as Key Master) of generating the encrypted salt file and distributing it to the new site since they already have the encrypted salt file that can be opened with their private key. The acting Key Master will need 2 additional inputs:

1. Encrypted Salt file (file containing salt for project new site is being added to)
2. Private key that was part of the public-private key pair used to encrypt the existing participating site’s shared project Salt file

Additional inputs will be required when using web interface.

The output is an asymmetric encrypted salt file for each site id in the input file. The output file naming convention is ProjectName\_SiteID\_Date.txt

Each encrypted output file is a comma separated value (see Appendix D for Data Dictionary):

1. Site ID – unique ID for each site
2. Site Name – unique Name for each site
3. Private Salt – unique 13 (or more) character string for each site during a single salt generation instance
4. Shared Salt – same 13 (or more) character string for all sites on a project
5. Project ID – same for all sites on a project

The salt engine checks to ensure that the shared salt is different from private salts, and that all private salts are unique during that instance of salt generation. When adding, new sites to an ongoing project, since the acting Key Master will not have other sites’ private salts, there is a risk that newly added site could have private salt collision (i.e., same private salt as other site).

**Instructions for using Java program**

This is a standalone command line program for generated encrypted shareable salt file and does not require any web service connection.

System Requirements:

* Linux, macOS or Windows OS
* Java Runtime Environment (JRE) 1.8 or higher

Download or clone the program from Github (<https://github.com/linkja>). Pre-built JAR is available from the releases page (https://github.com/linkja/salt-engine/releases).

**Input Data**

1. Site definitions – comma separated value (csv) file with site ID, site name, file path to respective site’s public key

**Executing the program**: From command line, run the executable JAR file using the standard Java command: java -jar SaltEngine-1.0-jar-with-dependencies.jar

To generate random project and private salts, provide 2 arguments:

 --projectName = Project Name

 --siteFile = Path to a file containing the site definitions

Example:

Navigate to the directory containing jar file and run below command:

java -jar SaltEngine-1.0-jar-with-dependencies.jar -- projectName Linkja\_project1 --siteFile ./siteKey/project1\_sitedefinitions

To generate random private salt for additional sites after the initial random project salt has been generated, specify the project salt, provide 3 arguments:

 --projectName = Project Name

 --siteFile = Path to a file containing the site definitions

 --projectSalt = Shared project salt to be used

Example:

Navigate to the directory containing jar file and run below command:

java -jar SaltEngine-1.0-jar-with-dependencies.jar -- projectName Linkja\_project1 --siteFile ./siteKey/project1\_sitedefinitions --projectSalt

Basic Troubleshoot:

Cannot find jar file – From command line, navigate to the folder where jar file is located and then run the Java command or specify full path to the JAR file

**Appendix:** Salt file

|  |  |  |
| --- | --- | --- |
| Column | Data Type | Description |
| Site ID | string or number | Unique Site ID\* |
| Site Name | string | Unique Site Name |
| Private Salt\*\* | string | Unique salt for each site  |
| Shared Salt\*\* | string | Salt shared across all sites in the project / study |
| Project Name | string | Project Name |

\*If there are different data sources within a site (e.g. lab system and registration), consider using different site IDs or create a unique record for all patients within site

\*\*Length of Salt is 13 or more characters

**Linkja-Hashing**

Linkja is a java tool to obfuscate PHI. It creates several permutations & combinations from the patient identifiers, hashes the values with salt & crypto algorithm provided by the salt generator, encrypts the hashed records with the public key provided by the aggregator, and packages the output as a .enc file. The tool works in-memory to accomplish above tasks, thus it does not store the real salt used in hashing or pre-encrypted values of data.

To begin, create a project folder to store all project specific files:

* linkjacrypto.dll: Salt-crypto generating agency sends this file. It contains secret algorithm to generate salt for hashing in memory
* saltFile: Salt-crypto generating agency sends this file. It contains seed to be used by the algorithm to generate project salt and site salt in memory
* encryptionKey: Aggregator sends this file. It contains RSA 2048-bit public key to be used in encrypting the hashes
* jar: Latest version should downloaded from the github release page (<https://github.com/linkja/linkja-hashing/releases>)
* project-data: This file contains patient identifiers to be used in hashing (patient id, name, dob, and optional ssn)

**Command:**

On command line (cmd), cd (i.e., change directory) to your project directory. And provide below arguments to begin hashing

Key arguments

-jar <arg> Name of jar file

--encryptionKey <arg> File name to public key sent by aggregator

--saltFile <arg> File name to salt file sent by salt-crypto

--patientFile <arg> File containing patient identifiers (.csv or .txt)

--privateDate <arg> Any random date in format: MM/DD/YYYY

--delimiter <arg> If patient file is .txt, then delimiter used

**Example:**

java -Djava.library.path=. -jar Hashing-1.1.1-jar-with-dependencies.jar --encryptionKey public-agg.key --saltFile project\_x12\_006\_20200509.txt --patientFile project-data.csv --privateDate 01/01/2018

Note: Please update the filenames in above command with your project files and latest jar file.

**Outputs:** There are 3 outputs generated by Linkja

1. encrypted file: This file has an extension of “.enc”. It contains hashed identifiers and should be shared with the aggregator for matching

2. crosswalk: This file contains crosswalk between site’s local patient ids and hashed patient id. This crosswalk can be used to link back to site’s data

3. invalid data file: This file contains records that linkja could not process due to errors in data (e.g., missing first name). To process these records, fix the error and re-run linkja

For testing, please download:

1. <https://github.com/linkja/linkja-test-data/tree/master/hashing>
2. <https://github.com/linkja/linkja-test-data/tree/master/crypto/secret>

**Appendix**: Patient data

The patient data should be csv or text delimited file (headers required)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description |
| 1 | Patient ID | string or number | Unique patient identifier\* |
| 2 | First Name | string | Patient first name |
| 3 | Last Name | string | Patient last name |
| 4 | DOB | string | Date of birth\*\*\*\* |
| 5\*\* | SSN | string | Social Security Number\*\*\* |

\*Each patient record should have a unique ID

\*\* SSN is optional, if the SSN is missing, hashes that require SSN (e.g. hash1 - fnamelnamedobSSN) will remain blank.

\*\*\* The application can handle last 4 digits of SSN as well as full Social Security Number (only last 4 numbers get used in hashing)

\*\*\*\*The application can handle several date formats YYYY-MM-DD (e.g. 1960-12-31), YYYYMMDD (e.g. 19601231), MM/DD/YYYY (e.g. 31/12/1960). Select 1 type of format for all records.

**Appendix**: Output Hashed Encrypted file (.enc extension)

This is the only file that should be shared with the aggregator. Linkja in memory, hashes the record, encrypts the hashed records, and packages the output as a .enc file. Only the aggregator with private key can open this file. The file contains below values (the values will not be visible to the site as only aggregator can open this file):

|  |  |  |  |
| --- | --- | --- | --- |
|  | Field Name | Data Type | Description  |
| 1 | Site ID | String | Site ID |
| 2 | Project ID | String | Project ID |
| 3 | PIDHASH | String | Patient ID + Site ID + Date Offset (Private Date and DOB)  |
| 4 | hash1 | String | First Name + Last Name + DOB + L4 SSN |
| 5 | hash2 | String | Last Name + First Name + DOB + L4 SSN |
| 6 | hash3 | String | First Name + Last Name + DOB |
| 7 | hash4 | String | Last Name + First Name + DOB |
| 8 | hash5 | String | First Name + Last Name + Transposed DOB + L4 SSN |
| 9 | hash6 | String | First Name + Last Name + Transposed DOB  |
| 10 | hash7 | String | 3 Initials First Name + Last Name + L4 SSN |
| 11 | hash8 | String | 3 Initials First Name + Last Name + DOB  |
| 12 | hash9 | String | First Name + Last Name + DOB +1D + L4 SSN |
| 13 | hash10 | String | First Name + Last Name + DOB +1Y + L4 SSN |

DOB=date of birth (YYYY-MM-DD)

Transposed DOB = Month and Date Transposed in date of birth (YYY-DD-MM)

1D = 1 day offset in date of birth

1Y = 1 year offset in date of birth

L4 SSN = Last 4 Social Security Numbers

Fields 3 – 13 are SHA256[[1]](#footnote-1) hashes (64 hexadecimal characters)

**Linkja-Matching**

To begin, on cmd, cd to directory containing linkja matching jar file and linkjacrypto.dll (for matching)

**Modes**

There are 4 modes available with linkja matching: decrypt, load, match & report data

Key arguments

-jar <arg> Path to jar file

--directory <arg> Path to project root directory

Below are mode specific arguments

**DECRYPT:** It looks inside specified root directory’s > data > input to load files with matching suffix (.txt or .csv) & prefix on to sqlite database

java -Djava.library.path=. -jar Matching-1.3.0-jar-with-dependencies.jar --decrypt --directory "P:\linkja-matching-master\project2" --prefix hashes --suffix .enc --decryptionKey private-agg.key

--decrypt activate decryption mode

--suffix <arg> suffix of files that should be decrypted

--prefix <arg> file extension of files that should be decrypted (.enc)

--decryptionKey <arg> path and filename of the RSA 2048 bit private key

**LOAD DATA**: It looks inside specified root directory’s > data > input to load files with matching suffix (.txt or .csv) & prefix on to sqlite database

java -jar Matching-1.3.0-jar-with-dependencies.jar --directory "P:\linkja-matching-master\project2" --load --prefix hashes --suffix .csv

--load activate mode to load data files to database

--suffix <arg> suffix of files that should be loaded to database

--prefix <arg> file extension of files that should be loaded

Every time load data script runs, it appends the data to database. To remove previous data, please use delete statements below and then click ‘write changes’

Delete from GlobalMatch;

Delete from InclusionPatients;

Delete from ExclusionPatients;

**MATCH DATA**: It matches data in the database inside root directory’s > data using the specified rule number & starting ID/seed

java -jar Matching-1.3.0-jar-with-dependencies.jar --directory " P:\linkja-matching-master\project2" --match 0 --seed 1

--match <arg> provide matching rule to match data in database

--seed <arg> number between 1 and 1000, global id assignment will begin from seed+1

**REPORT DATA**: It extracts data (site id, project name, patient id & global match id) for the site and saves the resulting file in data>output folder

java -jar Matching-1.3.0-jar-with-dependencies.jar --directory " P:\linkja-matching-master\project2" --report 6

--report <arg> provide site id to extract site id & global id

**GUI:** For smaller datasets, GUI can be used to load, match & report data

java -jar Matching-1.3.0-jar-with-dependencies.jar --directory " P:\linkja-matching-master\project2" --GUI

**Linkja matching folder structure**

1. Project root directory:



* Replace the private key file with project’s RSA 2048-bit private key
* Download latest jar file from <https://github.com/linkja/linkja-matching/releases>
* Download latest matching linkjacrypto.dll from <https://github.com/linkja/linkja-matching/tree/master/lib>

2. config:



No changes

3. data:



* ***input*** Contains all data files (encrypted files and decrypted files)
* ***output*** *Contains* distributable file with site id, project id, patient id, global id to share with the data contributors
* ***processed*** Contains log report

**Additional Utilities**

**1. OpenSSL:** Aggregator can generate RSA 2048-bit public-private key pair using openssl. Windows build is available on box <https://northwestern.box.com/s/jo0i43676clexastg5p795r0bejzaf10>

On cmd, cd to openssl.exe (.\OpenSSL\vc-win64a\bin) and to generate keys,

private key: openssl genrsa -out private-agg.key 2048

public key from the private key: openssl rsa -in private-agg.key -outform PEM -pubout -out public-agg.key

Name the files (private-agg and public-agg) as desired but keep extension .key

**2. SQLite database:** Download 64-bit DLL (x64) for SQLite version 3.31.1 from <https://www.sqlite.org/download.html>

**3. SQLite browser:** Download standard installer for 64-bit Windows from <https://sqlitebrowser.org/dl/>

- To create new database or open database, click on file, navigate to your project>data folder e.g., P:\linkja-matching-master\project2\data, and create/save GlobalMatchData database





When edit table dialog opens, hit cancel and follow DDL instructions below to create tables



**4. DDL** to create tables in SQLite browser is in Appendix. Copy & paste the ddl in Execute SQL tab, and hit play/run button. Make sure to click ‘write changes’ after executing DDL statements to commit changes to database



**5. To delete data from SQLite browser**

Delete from GlobalMatch;

Delete from InclusionPatients;

Delete from ExclusionPatients;

Make sure to click ‘write changes’ after executing delete statements to commit changes to database

**6. Optional JVM memory parameters** to set starting and maximum JVM memory size:

java -Xms3G -Xmx6G

sample usage:

java -Xms3G -Xmx6G -jar Matching-1.3.0-jar-with-dependencies.jar --directory "P:\linkja-matching-master\project2" --load --prefix hashes --suffix .csv

**Appendix: DDL Statements**

CREATE TABLE IF NOT EXISTS InclusionPatients (

 id integer PRIMARY KEY,

 globalId integer,

 siteId text,

 projectId text,

 pidhash text NOT NULL,

 hash1 text,

 hash2 text,

 hash3 text,

 hash4 text,

 hash5 text,

 hash6 text,

 hash7 text,

 hash8 text,

 hash9 text,

 hash10 text

);

CREATE TABLE IF NOT EXISTS ExclusionPatients (

 id integer PRIMARY KEY,

 globalId integer,

 siteId text,

 projectId text,

 pidhash text NOT NULL,

 hash1 text,

 hash2 text,

 hash3 text,

 hash4 text,

 hash5 text,

 hash6 text,

 hash7 text,

 hash8 text,

 hash9 text,

 hash10 text

);

CREATE TABLE IF NOT EXISTS GlobalMatch (

 id integer PRIMARY KEY,

 globalId integer,

 siteId text,

 projectId text,

 pidhash text NOT NULL,

 hash1 text,

 hash2 text,

 hash3 text,

 hash4 text,

 hash5 text,

 hash6 text,

 hash7 text,

 hash8 text,

 hash9 text,

 hash10 text

);

CREATE INDEX match0 ON GlobalMatch (hash1,hash2,hash3,hash4,hash5,hash6,hash7,hash8,hash9,hash10,id);

CREATE INDEX match1 ON GlobalMatch (hash1,hash2,hash5,hash9,hash10,id);

CREATE INDEX match2 ON GlobalMatch (hash3,hash4,hash6,id);

CREATE INDEX match3 ON GlobalMatch (hash1,id);

CREATE INDEX match4 ON GlobalMatch (hash1,hash2,id);

CREATE INDEX match5 ON GlobalMatch (hash1,hash5,id);

CREATE INDEX match6 ON GlobalMatch (hash1,hash9,id);

CREATE INDEX match7 ON GlobalMatch (hash1,hash10,id);

CREATE INDEX match8 ON GlobalMatch (hash3,id);

CREATE INDEX match9 ON GlobalMatch (hash3,hash4,id);

CREATE INDEX match10 ON GlobalMatch (hash3,hash6,id);

CREATE INDEX match11 ON GlobalMatch (hash7,id);

CREATE INDEX match12 ON GlobalMatch (hash8,id);

CREATE INDEX pidindex ON GlobalMatch (pidhash,siteId,projectId,id);

COMMIT;

CREATE VIEW report1 AS

SELECT siteId, projectId, pidhash, globalId FROM GlobalMatch;

**Appendix: Matching Rules**

Deterministic algorithms determine whether record pairs agree or disagree on a set of identifiers, where agreement on a given identifier is assessed as a discrete—“all-or-nothing”—outcome[[2]](#footnote-2). The match occurs on a set of identifiers that have been standardized, combined and hashed similarly across all participating sites (handled in Module 2: Hashing). When using composite identifiers (e.g., First Name + Last Name + Date of Birth + Last 4 SSN), equal weight is assigned to each data element and only when the entire composite ID matches, it is flagged as a match. In our current build, 10 composite identifiers are processed that allow syllogistic matches including, iterative match, hierarchical match (from more specific to more sensitive), and fuzzy match (partial name match). Also, see Appendix C, Table 1 for full list of rules. Deterministic algorithm is simple to understand, easy to implement and quality of matches can be improved greatly by improving data normalization techniques and increasing identifiers. Below are key match rules:

Rule 0: any hash <--> any hash

Rule 1: any hash with Full Name DOB SSN <--> any hash with Full Name DOB SSN

Rule 2: any hash with Full Name DOB <--> any hash with Full Name DOB

Rule 3: Hash 1 Full Name DOB SSN <--> Hash 1 Full Name DOB SSN

Rule 4: Hash 1 Full Name DOB SSN <--> Hash 2 Transposed Full Name DOB SSN

Rule 5: Hash 1 Full Name DOB SSN <--> Hash 5 Full Name Transposed DOB SSN

Rule 6: Hash 1 Full Name DOB SSN <--> Hash 9 Full Name DOB 1 Day SSN

Rule 7: Hash 1 Full Name DOB SSN <--> Hash 10 Full Name DOB 1 Year SSN

Rule 8: Hash 3 Full Name DOB <--> Hash 3 Full Name DOB

Rule 9: Hash 3 Full Name DOB <--> Hash 4 Transposed Full Name DOB

Rule 10: Hash 3 Full Name DOB <--> Hash 6 Full Name DOB 1 Year SSN

Rule 11: Hash 7 Partial Name DOB SSN <--> Hash 7 Partial Name DOB SSN

Rule 12: Hash 8 Partial Name DOB <--> Hash 8 Partial Name DOB

Notes:

* Full Name = First Name + Last Name
* Transposed Full Name = Last Name + First Name
* DOB = Date of Birth (MM/DD/YYYY)
* Transposed DOB = Transposed Date and Month in Date of Birth (DD/MM/YYYY)
* SSN = Last 4 Social Security Numbers
* Partial Name = Initial 3 characters in First Name + Last Name
* Rules 0, 1, and 2 are legacy rules. These are available for backward compatibility and will be phased out
* Multiple rules can be combined to increase sensitivity e.g., --match 3,4,5,6

Table1: Complete syllogistic matching set rules available in Java:

|  |  |  |
| --- | --- | --- |
| **Rule No. (Match set)** | **Composite identifiers (Hashed)** | **Match Rule Description** |
| 0\* | All hashes with each other | All hashes with each other |
| 1\* | hash1,hash2,hash5,hash9,hash10 with hash1,hash2,hash5,hash9,hash10 | All Full name, dob & SSN |
| 2\* | hash3,hash4,hash6 with hash3,hash6,hash6 | All Full name & dob  |
| 3 | hash1 with hash1 | Full name, dob & SSN  |
| 4 | hash1 with hash2  | Full transposed name, dob & SSN |
| 5 | hash1 with hash5 | Full name, transposed dob & SSN |
| 6 | hash1 with hash9 | Full name, day diff dob & SSN |
| 7 | hash1 with hash10 | Full name, year diff dob & SSN |
| 8 | hash3 with hash3 | Full name, dob |
| 9 | hash3 with hash4 | Full transposed name, dob |
| 10 | hash3 with hash6 | Full name, transposed dob |
| 11\*\* | hash7 with hash7 | Partial name, dob & SSN\* |
| 12\*\* | hash8 with hash8 | Partial name, dob\* |

\*Rules 0,1,2 are legacy match rules carried forward for backward compatibility

\*\*Rules 11 and 12 are fuzzy matches allowing first 3 initials of first name combined with rest of the elements as specified in description

The rules can be combined to form more complex algorithms. Recommended cross match set rule combinations:

3 – most specific

3,4,5,6,7

3,4,5,6,7,11

8,9,10

12

12,9,10,6,7 – most sensitive

**Appendix: Config files**

**1. Create and save global-match.properties with below code**

############ Define Global Patient Match properties ###########

# database information

Database=SQLite

DbDirectory=%ProjectRoot%/data/

DbName=GlobalMatchData.db

# directory for configuration files

ConfigFilesDirectory=%ProjectRoot%/config

# directory for input files

InputFilesDirectory=%ProjectRoot%/data/input

# directory for CDA output files

OutputFilesDirectory=%ProjectRoot%/data/output

# directory for processed files

ProcessedFilesDirectory=%ProjectRoot%/data/processed

#Input file name prefix - suffix for hash files

InputFileNamePrefix=hash\_

InputFileNameSuffix=.dx

#Sequence of Patient Matching Rules. comma separated

MatchingRules=3

**2. Create and save global-match-globalId.txt with below code**

1 | 2020-05-10

1. https://en.wikipedia.org/wiki/SHA-2 [↑](#footnote-ref-1)
2. https://www.ncbi.nlm.nih.gov/books/NBK253312/ [↑](#footnote-ref-2)