risecamp
UC Berkeley

ADVANCED DATA SCIENCE

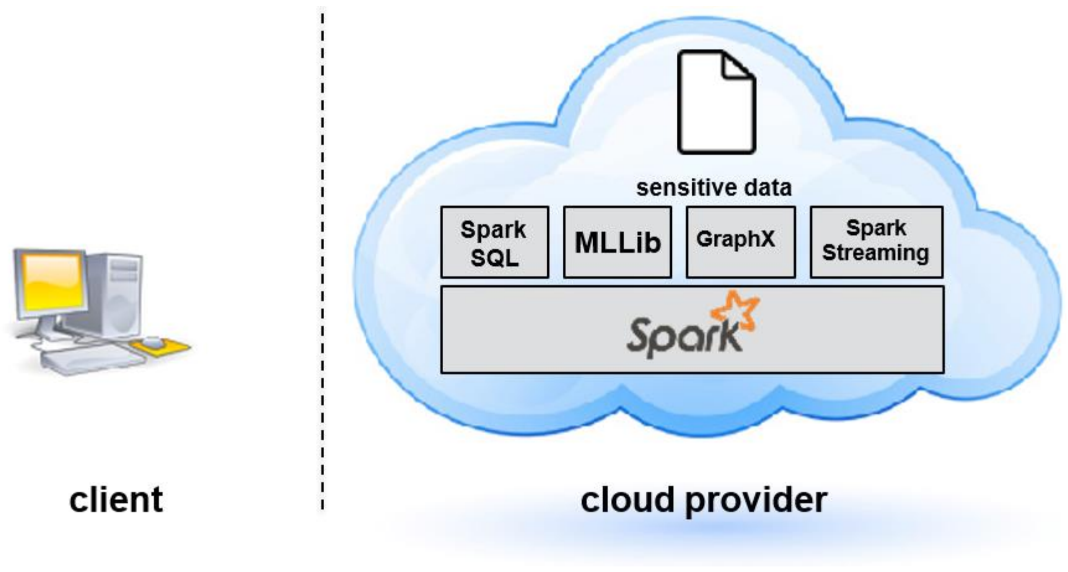
Prof. Sri Krishnamurthy

Team Members:
Sayali Borse
Rishi Rajani
Komal Ambekar

Opaque

Opaque stands for **O**blivious **P**latform for **A**nalytic QUEries

Companies today collect a large amount of sensitive data and recent years cloud-based platforms because they offer a rich set of data analytics that can extract a tremendous amount of value from the data. The cloud consists of the sensitive data of the company. Nowadays, the data is encrypted during the storage as well as in transit from the client to the cloud provider to avoid the attackers steal the data



As enterprises move to cloud-based analytics, the risk of cloud security breaches poses a serious threat because the computation is done on the plaintext. Data breaches have become increasingly common and can cause damages.

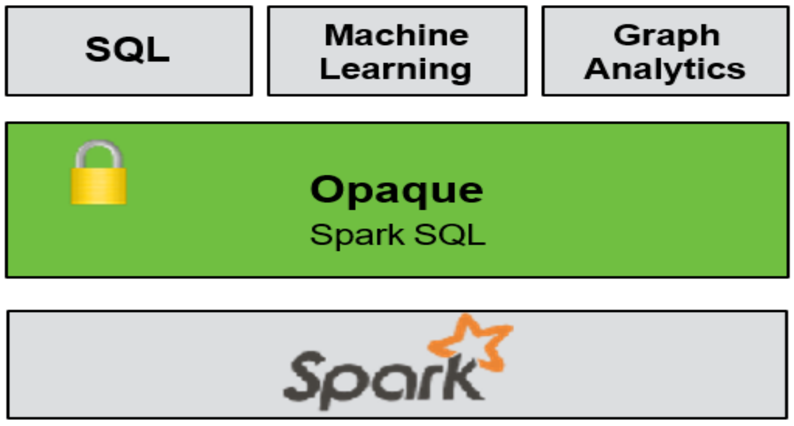
The attackers in these data breaches vary in abilities meaning they are external are of two types:

- External attackers: These attackers break into the network and exfiltrate the data.
- Internal attackers: These attackers are the ones who have the easy access to the data and they also have the administration privileges. This attack is launched from inside who has easy access to the systems.

In order to fully protect the data we must ensure that the entire server should not be compromised by the attackers.

The most important challenge in this type is to protect the data while preserving the functionality to do the complex analytics securely. The performance should not be compromised.

Opaque is a securely distributed algebra platform to perform complex analytical queries. It is a package for Apache Spark SQL that enables very strong security for SQL queries: data encryption, computation verification, and access pattern leakage protection (a.k.a. obliviousness). It extends the Spark SQL with extra sql operators and rules so that the analytical queries can run securely. In this way the attacker only can see the encrypted data on the server side.



Encrypting data at rest and in transit is a major first step. However, data must still be decrypted in memory for processing, exposing it to an attacker who has compromised the operating system or hypervisor. Trusted hardware such as Intel SGX has recently become available in latest-generation processors. Such hardware enables arbitrary computation on encrypted data while shielding it from a malicious OS or hypervisor. However, it still suffers from a significant side channel: ***Access Pattern Leakage.***

Prior Work:

- Computation on Encrypted data:

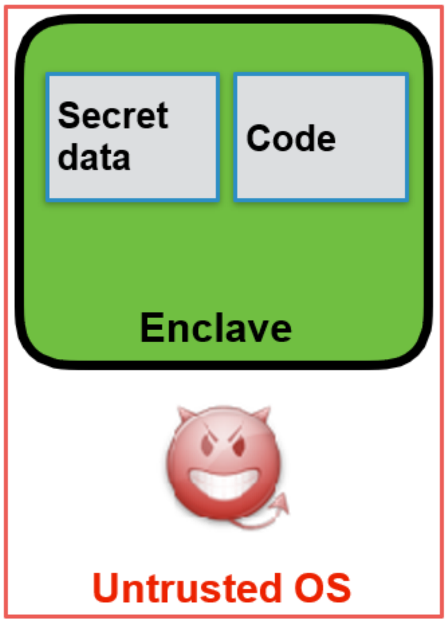
This is purely a software-based approach. It is a cryptographic approach that uses the homomorphic encryption. This approach operates on the cipher text and allows you to multiply to Cipher Text and once you decrypt the product you actually get is the sum of the two versions of the plaintext.
Many systems use this approach

Limitations:

- This approach is very slow
- It has limited functionality

- Hardware based system

These hardware-based systems are the hardware containers which can operate in malicious operating system. They provide a shielded execution for the queries.



The hardware enclave allows you to keep the secret data in the enclave and ensures that the execution is performed securely. Also, it makes sure that the code loaded is correct.

Opaque uses the Hardware-based system. It uses the Intel SGX hardware which is present in the newer computer systems

<u>**Access Pattern Leakage:**</u>

This problem indicates that the pattern in the data is decoded by the attacker. This type of data frequently occurs in the database and can be analyzed by the attackers hence attacking the system. Opaque is used to avoid these data access pattern leakage.

<u>Opaque modes:</u>
Opaque operates in two modes:

- Encryption mode:

This mode is when the External attacker comes into picture.

- Data encryption and authentication
- Computation is integrity protected

- Oblivious mode:

This mode is when the insider/member of the organization attacks the system. Hides the access pattern of the data.

<u>Opaque requirements:</u>

- Runs on Ubuntu 16.04/18.04
- Hardware enclave – Intel SGX
- Spark version – 2.0.2
- Works with Scala only
- Must use DataFrame API

Installation steps:

- Install python to run Scala kernel on jupyter notebook:

```
apt-get update
apt-get upgrade
apt-get -y install python2.7
apt-get -y install python-pip
apt-get -y install python-dev
python --version
pip --version
apt-get -y install ipython ipython3
pip install jupyter
jupyter notebook --allow-root --ip='127.0.0.1'
```

- Install java 8: (Scala works with java version 8 only)

```
sudo apt-get update
java – - version
sudo apt-get install default-jre
```

- Download and install Spark 2.0.2 from the given link
  Spark 2.0.2 download

```
sudo apt-get install scala
```

- Install git to clone/download UCB risecamp repository:

```
sudo apt-get install scala
```

- Clone the UCB risecamp opaque repository from Opaque and follow the installation steps below:
  - *Install dependencies and the Intel SGX SDK with C++11 support:*

For Ubuntu 16.04:

```
sudo apt-get install build-essential ocaml automake autoconf libtool wget python default-jdk cmake libssl-dev
```

For Ubuntu 18.04:

```
sudo apt-get install build-essential ocaml ocamlbuild automake autoconf libtool wget python default-jdk cmake
libssl-dev
```

- git clone https://github.com/intel/linux-sgx.git -b sgx_2.3
- cd linux-sgx
- ./download_prebuilt.sh
- make sdk_install_pkg

  Installer will prompt for install path, which can be user-local
- ./linux/installer/bin/sgx_linux_x64_sdk_*.bin

  On the master, generate a keypair using OpenSSL for remote attestation. The public key will be automatically hardcoded into the enclave code. Note that only the NIST p-256 curve is supported.

- cd ${OPAQUE_HOME}
- openssl ecparam -name prime256v1 -genkey -noout -out private_key.pem

  Set the following environment variables:
- source sgxsdk/environment # from SGX SDK install directory in step 1
- export SPARKSGX_DATA_DIR=${OPAQUE_HOME}/data
- export PRIVATE_KEY_PATH=${OPAQUE_HOME}/private_key.pem

  If running with real SGX hardware, also set export SGX_MODE=HW and export SGX_PRERELEASE=1.

  Run the Opaque tests:

- cd ${OPAQUE_HOME}
- build/sbt test

<u>Following are the commands to use OPAQUE</u>

  run Apache Spark SQL queries with Opaque as follows, assuming Spark 2.3.2 is already installed

  Package Opaque into a JAR:

- cd ${OPAQUE_HOME}
- build/sbt package

Launch the Spark shell with Opaque:

- ${SPARK_HOME}/bin/spark-shell --jars${OPAQUE_HOME}/target/scala-2.11/opaque_2.11-0.1.jar

Steps for docker:

- sudo apt-get update
- sudo apt-get upgrade
- sudo apt install python3.6
- sudo apt install python3.6-pip
- sudo apt install python3.6-dev
- sudo apt install python-pip
- pip install jupyter
- sudo apt-get update
- sudo apt-get install \

```
>    apt-transport-https \
>    ca-certificates \
>    curl \
>    software-properties-common
```

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo apt-key fingerprint 0EBFCD88
sudo add-apt-repository \
>   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
>   $(lsb_release -cs) \
>   stable"
```

```
sudo apt-get install docker-ce
sudo apt install git
git clone https://github.com/ucbrise/opaque.git

sudo apt install openjdk-8-jdk
sudo apt install scala
tar xvf spark-2.0.2-bin-hadoop2.7.tgz
```

```
cd spark-2.0.2-bin-hadoop2.7
cd bin


./spark-shell
cd ..
cd ..

sudo docker run -it -m 4g -w /home/opaque/opaque ankurdave/opaque build/sbt test
sudo docker run -it -m 4g -w /home/opaque/opaque ankurdave/opaque build/sbt console and hit enter
```

- Commands:

```
import org.apache.spark.sql.Dataset
import org.apache.spark.sql.DataFrame
import edu.berkeley.cs.rise.opaque.benchmark._
import edu.berkeley.cs.rise.opaque.execution.Opcode._
import edu.berkeley.cs.rise.opaque.execution._
import edu.berkeley.cs.rise.opaque.implicits._

edu.berkeley.cs.rise.opaque.Utils.initSQLContext(spark.sqlContext)

#create data
val data = for (I <- 0 until 10) yield (“foo” , i)
val rdd_data = spark.sparkContext.makeRDD(data, 1)

val words_e = spark.createDataFrame(rdd_data).toDF( “word” , “count”).encrypted

al words_o = spark.createDataFrame(rdd_data).toDF( “word” , “count”).oblivious
```

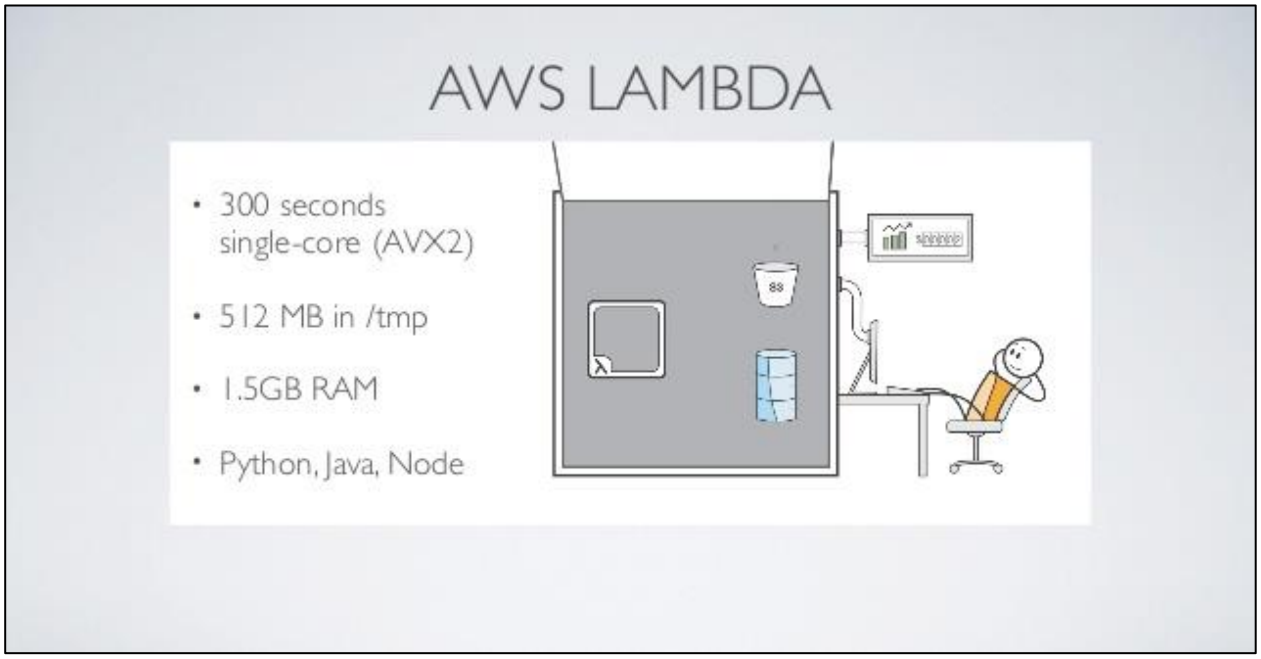
PYWREN

INTRODUCTION:

Pywren serializes a Python function using cloudpickle, capturing all relevant information as well as most modules that are not present in the server runtime. This eliminates the majority of user overhead about deployment, packaging, and code versioning. We submit the serialized function along with each serialized datum by placing them into globally unique keys in S3, and then invoke a common Lambda function. On the server side, we invoke the relevant function on the relevant datum, both extracted from S3. The result of the function invocation is serialized and placed back into S3 at a pre-specified key, and job completion is signaled by the existence of this key. In this way, we are able to reuse one registered Lambda function to execute different user Python functions and mitigate the high latency for function registration, while executing functions that exceed Lambda's code size limit.

AWS LAMBDA:

AWS Lambda is a compute service that lets you run code without provisioning or managing servers. AWS Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second. You pay only for the compute time you consume - there is no charge when your code is not running. With AWS Lambda, you can run code for virtually any type of application or backend service - all with zero administration. AWS Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging. All you need to do is supply your code in one of the languages that AWS Lambda supports (currently Node.js, Java, C#, Go and Python).

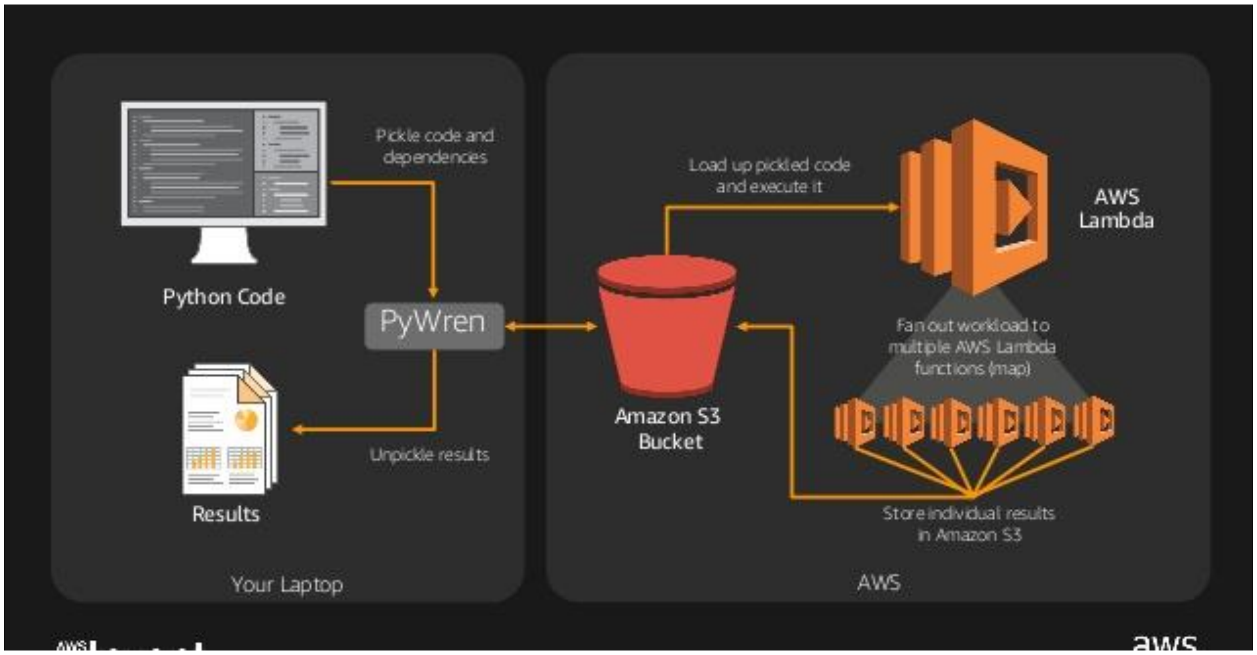


AWS LAMBDA

- 300 seconds single-core (AVX2)
- 512 MB in /tmp
- 1.5GB RAM
- Python, Java, Node

CLOUD PICKLE:

It is used for serializing and de-serializing a Python object structure. Any object in python can be pickled so that it can be saved on disk. What pickle does is that it “serialises” the object first before writing it to file.Pickling is a way to convert a python object (list, dict, etc.) into a character stream.The idea is that this character stream contains all the information necessary to reconstruct the object in another python script.


Pickle code and dependences
Python Code
PyWren
Results
Unpickle results
Your Laptop
Load up pickled code and execute it
AWS Lambda
Amazon S3 Bucket
Fan out workload to multiple AWS Lambda functions (map)
Store individual results in Amazon S3
AWS

HOW TO GET STARTED WITH PYWREN:

Requirements:
Runs on Ubuntu & Fedora

Step 1: Obtain an amazon web services account and then try installing the AWS command-line utilities via

```
pip install awscli
```

Step 2: Installation of pywren

```
$ pip install pywren
```

This installs the pywren library as well as the pywren command-line tool.

Step 3: Interactive pywren setup

```
$ pywren-setup


This is the PyWren interactive setup script

Your AWS configuration appears to be set up, and your account ID is 71825125821

This interactive script will set up your initial PyWren configuration.

If this is your first time using PyWren then accepting the defaults should be fine.

What is your default aws region? [us-west-2]:

Location for config file:  [~/.pywren_config]:

PyWren requires an s3 bucket to store intermediate data. What s3 bucket would you like to use? [
jonas-pywren-604]:

Bucket does not currently exist, would you like to create it? [Y/n]: Y

PyWren prefixes every object it puts in S3 with a particular prefix.

PyWren s3 prefix:  [pywren.jobs]:

Would you like to configure advanced PyWren properties? [y/N]:

PyWren standalone mode uses dedicated AWS instances to run PyWren tasks. This is more flexible,
but more expensive with fewer simultaneous workers.

Would you like to enable PyWren standalone mode? [y/N]:
```

```
Creating config /Users/jonas/.pywren_config
new default file created in ~/.pywren_config
lambda role is pywren_exec_role_1
Creating bucket jonas-pywren-604.
Creating role.
Deploying lambda.
Pausing for 5 seconds for changes to propagate.
Pausing for 5 seconds for changes to propagate.
Successfully created function.
Pausing for 10 sec for changes to propoagate.
function returned: Hello world
```

MANUAL SETUP:

Before you get started, make sure you have your AWS credentials set up properly for use via Boto, the python AWS library.

```
$ pywren get_aws_account_id
Your AWS account ID is 942315755674
```

Run the following from the prompt:

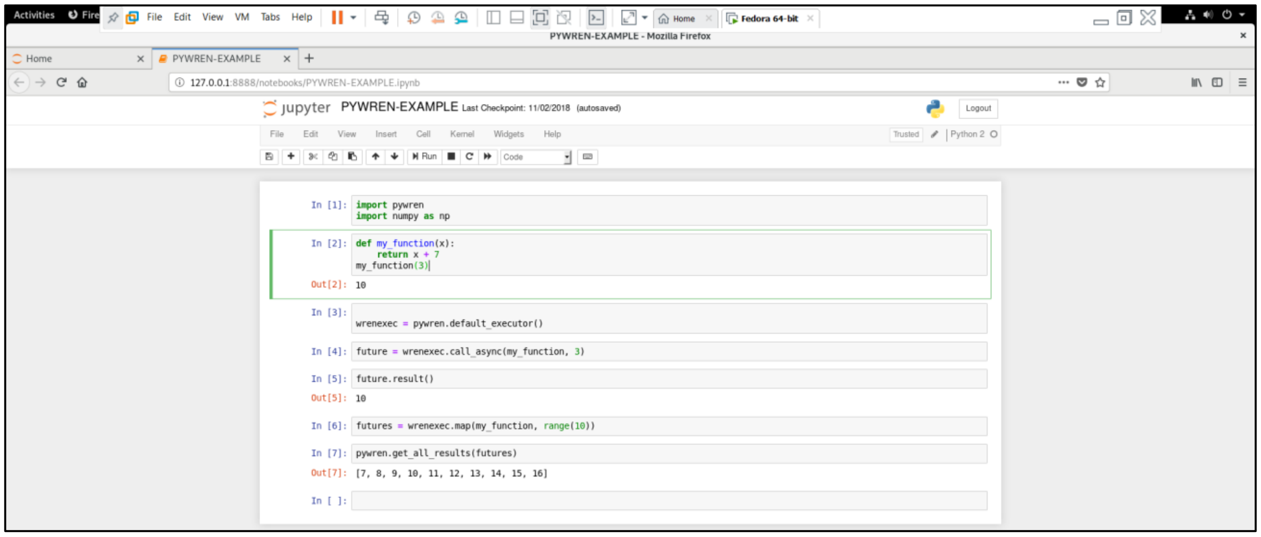
```
$ pywren create_config
$ pywren create_role
$ pywren create_bucket
$ pywren deploy_lambda
```

- This will create a default configuration file and place it in ~/.pywren_config.
- Create the default IAM role to run the lambda process as pywren_exec_role.
- Deploy the lambda function to AWS using your account as pywren1.
- Create a bucket in your default AZ named pywren-bucket.
- Place all intermediate data in pywren-bucket/pywren.jobs.

After Installing pywren through command line the jupyetr notebook should be opened via following command:

```
jupyter notebook --allow-root --ip='127.0.0.1'
```

EXAMPLE:


PYWREN-EXAMPLE - Mozilla Firefox
127.0.0.1:8888/notebooks/PYWREN-EXAMPLE.ipynb
jupyter PYWREN-EXAMPLE Last Checkpoint: 11/02/2018 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 2
Run
Code

```
In [1]: import pywren
        import numpy as np

In [2]: def my_function(x):
            return x + 7
        my_function(3)

Out[2]: 10

In [3]:
        wrenexec = pywren.default_executor()

In [4]: future = wrenexec.call_async(my_function, 3)

In [5]: future.result()

Out[5]: 10

In [6]: futures = wrenexec.map(my_function, range(10))

In [7]: pywren.get_all_results(futures)

Out[7]: [7, 8, 9, 10, 11, 12, 13, 14, 15, 16]

In [ ]:
```

Code:

This is a simple Hello World example, showing how to take a function and run it with pywren. First we import the necessary libraries.

In [1]:
```
import pywren
import numpy as np
```

Pywren is designed to run any existing python functions you have, in parallel, at scale, on the cloud. So first, we create an example python function. The funtion must take a single argument:

In [2]:
```
def my_function(x):
    return x + 7
my_function(3)
```

Out[2]:
10

To start using pywren, we first create an executor.

In [3]:
```
wrenexec = pywren.default_executor()
We can call my_function(3) remotely via call_async:
```

In [4]:
```
future = wrenexec.call_async(my_function, 3)
```
Future is a placeholder for the returned value from applying my_function to the number 3. We can call result on it and get the result. Note that this will block until the remote job has completed

In [5]:
```
future.result()
```
Out[5]:
10

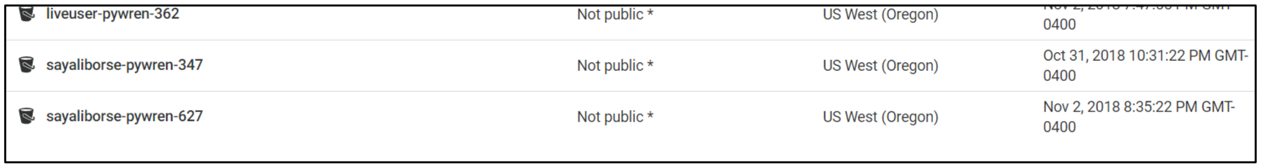
You can apply my_function to a list of arguments, and each will be executed remotely at the same time.

In [6]:
```
futures = wrenexec.map(my_function, range(10))
```
The pywren get_all_results function will wait until all of the futures are done and return their results

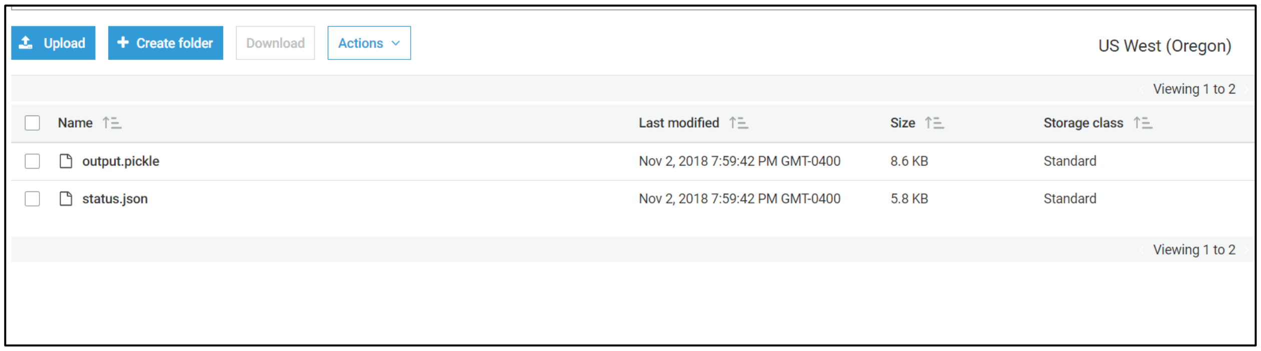
In [8]:
```
pywren.get_all_results(futures)
```

Out[8]:
[7, 8, 9, 10, 11, 12, 13, 14, 15, 16]

AWS LAMDA OUTPUT ON S3:



liveuser-pywren-362	Not public *	US West (Oregon)	0400
sayaliborse-pywren-347	Not public *	US West (Oregon)	Oct 31, 2018 10:31:22 PM GMT-0400
sayaliborse-pywren-627	Not public *	US West (Oregon)	Nov 2, 2018 8:35:22 PM GMT-0400



Name	Last modified	Size	Storage class
output.pickle	Nov 2, 2018 7:59:42 PM GMT-0400	8.6 KB	Standard
status.json	Nov 2, 2018 7:59:42 PM GMT-0400	5.8 KB	Standard

WAVE

Wide Area Verified Exchange

Wave in a nutshell is an authorization system that permits delegation of permissions without a central authority. Which means if one component had permission to access something or control something, they are able to delegate a subset of their permissions to another device or service and all of this happens without trusting a centralized server. If at all we have an active directory or a live database connection this gets very easy, but we want to eliminate the dependency on a central server and hence this mechanism is enforced using cryptography. In other words, WAVE is an authorization system spanning across different administrative domains without relying on a central authority. Each party can express fine-grained trust into another party without implicitly having to trust a central third.

Today with the trend of smart homes and other smart devices, IoT has become a significant part of everyone's' routine. In a global Internet of Things, who can be trusted to authorize the world? Also, existing systems have a central authority which cannot be the optimal method hence forth. Centralization is a point of weakness and over the last two years companies have lost millions of dollars due to breaches in the central server. Avoiding the central point of weakness drastically reduces the risk associated with your service.

WAVE can be used to secure communications and encrypt any messages that are being passed between any components and services. We can attach proofs that the permission / action is authorized which eliminates the existence of a central authority. As it can be used to authorize and authenticate any actions it can replace your active directory, any bearer token authorization frameworks that exist today. Wave connects Sensors, actuators, controllers, drivers, apps and people.

Working of WAVE:

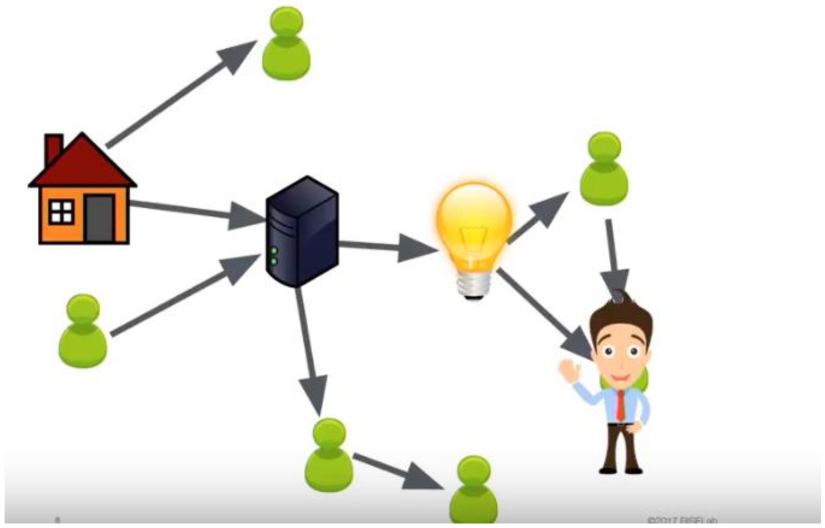
WAVE allows participants to form a proof of permissions they have and attach this proof to all interactions that they have with other parties or services. This proof can now be verified by the receiving party non-interactively. This is quite a big change from the current authorization systems. The receiving party does not need to interact with sending party which allows it for any asynchronous actions or any publish / subscribe communications. Permissions can be granted to any participant, which means if a component belongs to company A, he can communicate with a component of company B which allows WAVE to be used for federation across organizations.

Fundamentals of WAVE:

Entity: Any user within WAVE let that be a person or a device can be called an ENTITY. We can relate to this as a username / email address. It is the identifies that is associated with permissions.	This could a participant, service or an IOT device. Each entity has a public part and a private part, and each entity contains multiple public and private keys. The public part of the entity is referred to by its hash. However, we can use a globally immutable alias like Rishir29 or Komal30 that helps in in remembering the namespaces.

Attestations: All entities that interact amongst each other with permissions and these permissions are called Attestations. These attestations are encrypted but are publicly discoverable.

These attestations form a Global Graph of Authorization which crosses multiple administrative domains.

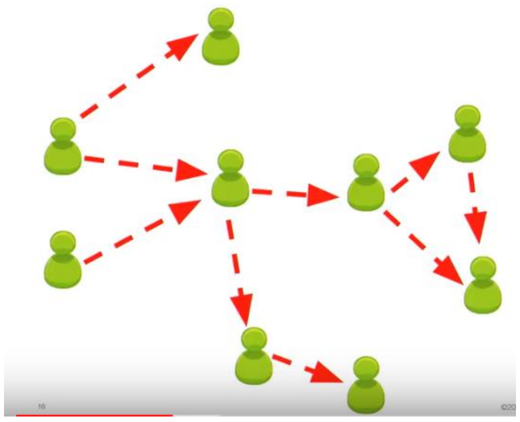


We can relate to this diagram by building an analogy where we have an entity that is a house which is connected to a server which is connected to a light and which is eventually connected to a person, along with multiple other entities that are present within the graph and can interact with each other.

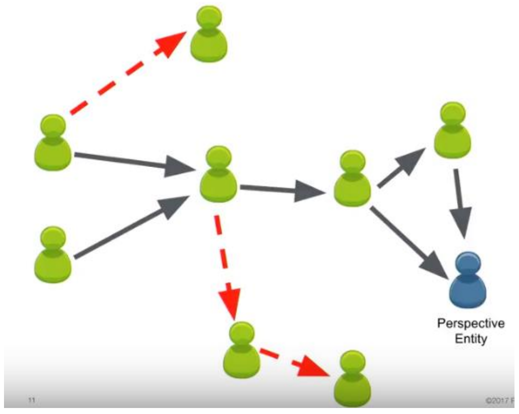
The global graph contains every attestation that is created within the system which exist between people, services and devices. It spans across all authorization domains. For wave everything just looks like an entity. It doesn't matter if it's a person or a device, WAVE will treat every component only as an entity.

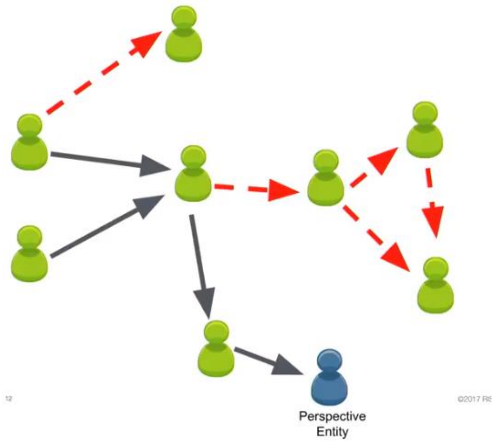
Using the graph every entity can discover attestations which would enable it to use them (permissions for other entities). The entity will traverse through the Global Graph and decrypt a certain part of it which is can use to create proofs for itself. This decrypted / subset graph is called a Perspective Graph. As a user we do not have to manually decrypt any of these graphs. WAVE would do it for us.

Let's consider the below Global Graph:



For every entity the Perspective Graph is different. The below two images indicate a perspective graph for two different entities that belong to the same system.



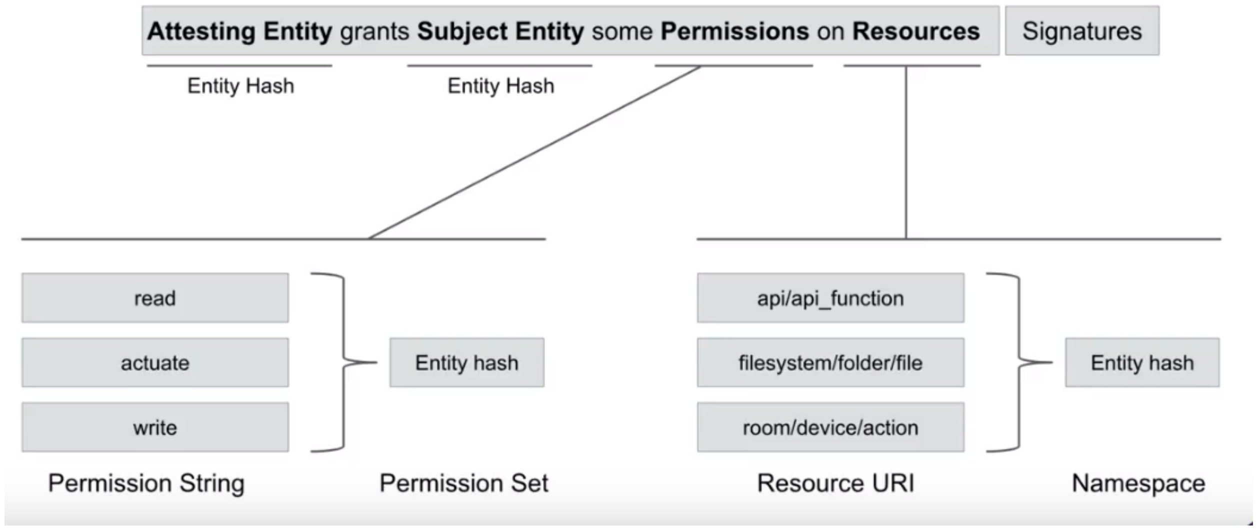


We can safely say that incase an entity does not have a certain attestation within its graph it doesn't have the capability to access it any way.

Format of an attestation:
An attestation is an object that contains the attesting entity – the one that has permissions and is granting it, the subject entity – the one that is receiving that permissions, the description of the permissions and which resources that permission concerns. The Entities are referred to by their hash and we also attach some cryptographic signatures that cannot be forged which means that only the attesting entity can create the objects because it was signed by that entity.

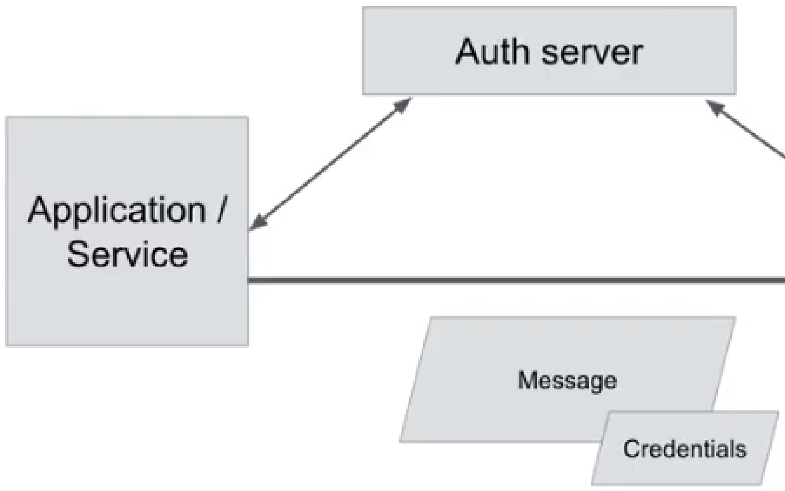
The permissions are a combination of permission strings which allow functions like read write etc. and a permission set which determines the entity for which the permissions are granted. It acts as a name space for the permissions, so it doesn't get confused with any other entities.

Our resources are a combination of our URI and each of these URI are within a namespace and every URI starts with a namespace which is the entity it belongs to and has the capability to grant permissions to the other entities with this URI.



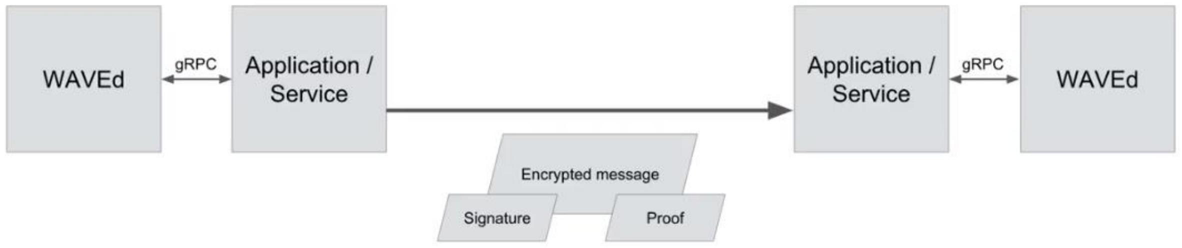
Wave Architecture:

Typical Authorization System Architecture



When we look at a regular authorization mechanism we notice that there are two entities (could be applications or services) where one of them requests permissions on the other. The requesting entity sends out a message with its credentials and the receiving entity then liaisons with the central server and grants the required permission.

In wave every entity/ resource has its own WAVE Daemon running which is local to it. Let's assume we are running our applications on a server, the WAVE Daemon would be installed on the server and then communicate with all applications. The other entity you would like to communicate with will also have its WAVE daemon. So, the requesting entity sends an encrypted message to the desired entity. In addition to the encryption this message is sent with a signature and a proof which the receiving party decrypts and assures the authenticity by first verifying the signature decrypting message. Steps are as follows, creating entity > Building a proof > Encrypt a message > Sign the message (message has now reached receiving entity. The receiving entity first verifies the signature, decrypts the message and then it verifies the proof.



References:

Opaque:

- [1] Wenting Zheng, Ankur Dave, Jethro Beekman, Raluca Ada Popa, Joseph Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. NSDI 2017, March 2017.
- https://www.youtube.com/watch?v=2q3sOYJot5U
- https://github.com/ucbrise/opaque

PYWREN:

- https://www.youtube.com/watch?v=OskQytBBdJU&t=890s
- https://github.com/ucbrise/risecamp/tree/risecamp2018/pywren
- https://arxiv.org/pdf/1702.04024.pdf

WAVE:

- https://www.youtube.com/watch?v=l3gM2Sl2Cug&t=267s
- http://bets.cs.berkeley.edu/publications/2017techreport_wave.pdf
- https://github.com/ucbrise/risecamp/tree/risecamp2018/wave