**Introduction.**

We are going to use homomorphic encryption to implement encryption of data and its operations.

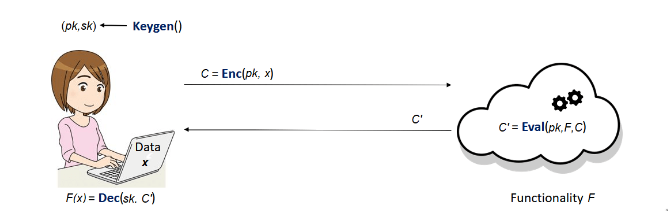
Homomorphic encryption is a form of encryption that allows for computation on ciphertext, generating an encrypted result which, when decrypted, matches the result of the operations as if they had been performed on the plaintext.

ANother simpler definition can, Homomorphic Encryption enables a user to perform meaningful computations on sensitive data while ensuring the privacy of the data. THis is very ideal in our current situation.

**How is Homomorphic encryption different from normal encryption**

Basically if you want to perform useful computations on the encrypted data (e.g. encrypted under classical algorithms like AES), you need to decrypt it first. But once decryption takes place, the privacy of the data is compromised. HE overcomes this seeming contradiction, as it took the cryptographic community more than 30 years to come up with a construction. The first solution was proposed by Craig Gentry in 2009 and was of theoretical interest only.

Besides the traditional encryption (Enc), decryption (Dec) and key generation (Keygen) algorithms, an HE scheme also uses an evaluation algorithm (Eval). This is the distinguishing feature that makes computations on encrypted data possible.



Alice sends her data encrypted, so no one learns anything about **x**.

Computing on the encrypted data C does not involve Alice's secret key **sk**. Only her public key **pk** is used.

To obtain **C'** as the encryption of **F(x)**, the evaluation algorithm uses the description of **F** to do computations on **C** (which encrypts **x**).

By using her secret key, **sk**, Alice manages to recover the information that interests her, namely **F(x)**.

**Implementing a Python based Homomorphic Encryption**

The whole scheme includes the following sections

* Basic functionalities
* Key-generation
* Encryption
* Decryption
* evaluation (add and mul).

**Basic Functionalities**

We import Numpy library and define two helper functions (polyadd and polymul) for adding and multiplying polynomials



In polyadd x, y: two polynoms to be added.

modulus: coefficient modulus.

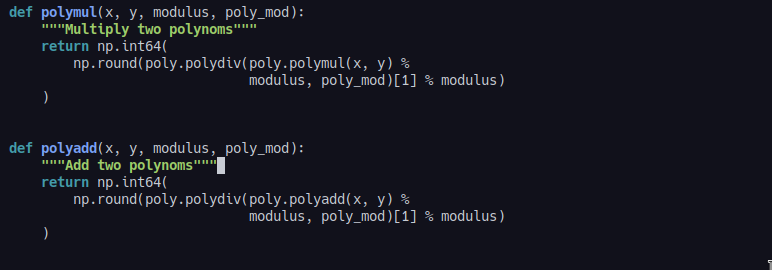
poly\_mod: polynomial modulus.

In polymul x, y: two polynoms to be multiplied.

modulus: coefficient modulus.

poly\_mod: polynomial modulus.

Return a polynomial in Z\_modulus[X]/(poly\_mod).

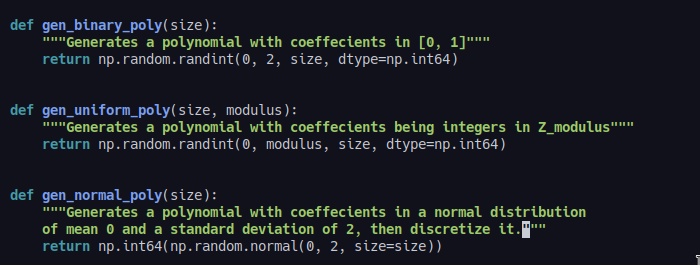


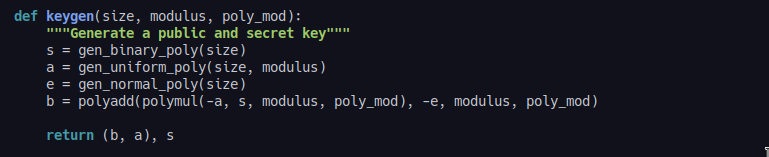
We start by generating a random secret-key sk from a probability distribution, we will use the uniform distribution over R2, which means sk will be a polynomial with coefficients being 0 or 1.

**Key generation**

We start by generating a random secret-key **sk** from a probability distribution, we will use the uniform distribution over **R2**, which means **sk** will be a polynomial with coefficients being **0** or **1**.

For the public-key we first sample a polynomial **a** uniformly over **Rq** and a small error polynomial **e** from a discrete normal distribution over **Rq**. We then set the public-key to be the tuple **pk=([−(a⋅sk+e)]q,a)**. So let's first implement the generation of polynomials from different probability distributions.





The public-key **(b, a)** can then be used for encryption, and the secret-key **sk** for decryption.

**Encryption**

Encrypting an integer as an example by calling encrypt with

pk: public-key.

size: size of polynomials.

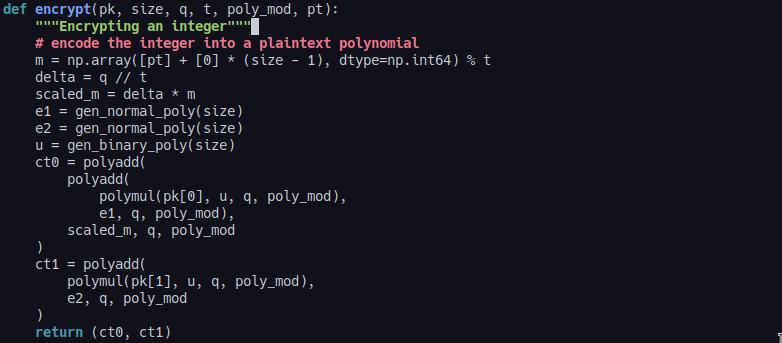
q: ciphertext modulus.

t: plaintext modulus.

poly\_mod: polynomial modulus.

pt: integer to be encrypted.

This function then returns a tuple representing a ciphertext.



**Decryption**

The function decrypts a ciphertext. We pass it the following values

sk: secret-key.

size: size of polynomials.

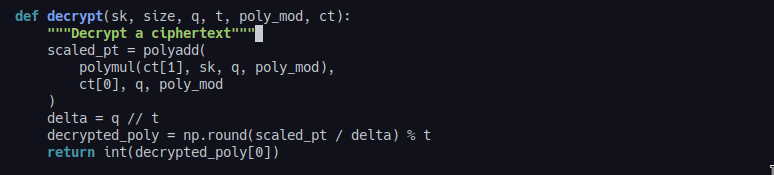
q: ciphertext modulus.

t: plaintext modulus.

poly\_mod: polynomial modulus.

ct: ciphertext.

After all the operations, it returns an integer representing the plaintext.

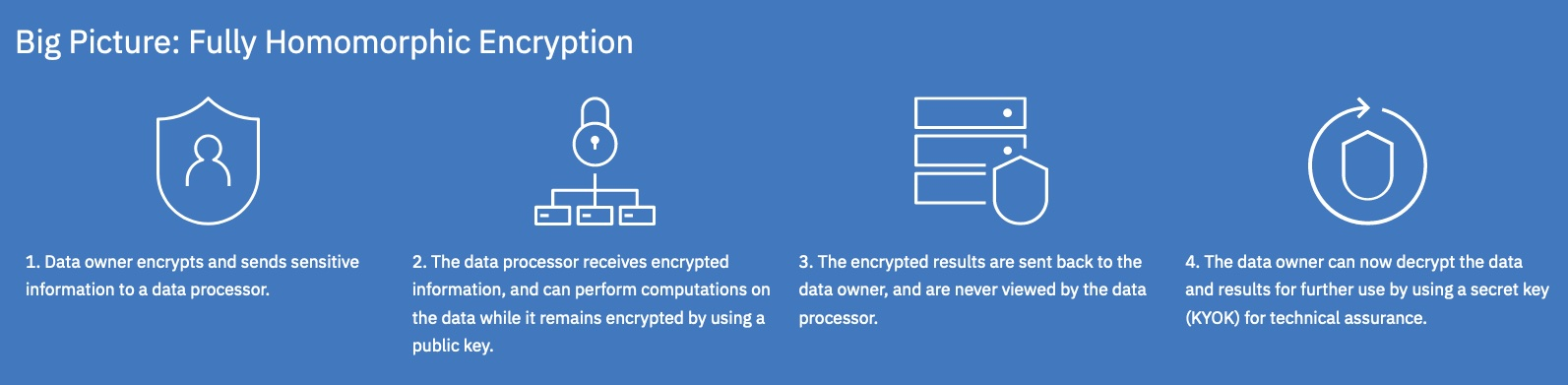


Key generation algorithm outputs two keys, a public-key that can encrypt messages, and a secret-key that can decrypt messages.

**Demonstration**

IBM Fully Homomorphic Encryption (HELayers) SDK for Linux

FHE Toolkit called IBM HElayers, a software development kit (SDK) for the practical and efficient execution of encrypted workloads using fully homomorphic encrypted data.



**Getting started**

HElayers is a Linux based Docker container and can be implemented entirely in software; there is no hardware dependency. The only software that is required is Docker version 19 or higher with the necessary user privileges to run docker commands. It is also assumed you have a working internet connection.

**Step 1: Cloning the IBM FHE Toolkit Repository**

First, from a terminal window, issue the following command to clone this git repo:

git clone <https://github.com/ibm/fhe-toolkit-linux>

**Step 2: Fetching the Toolkit Docker Images**

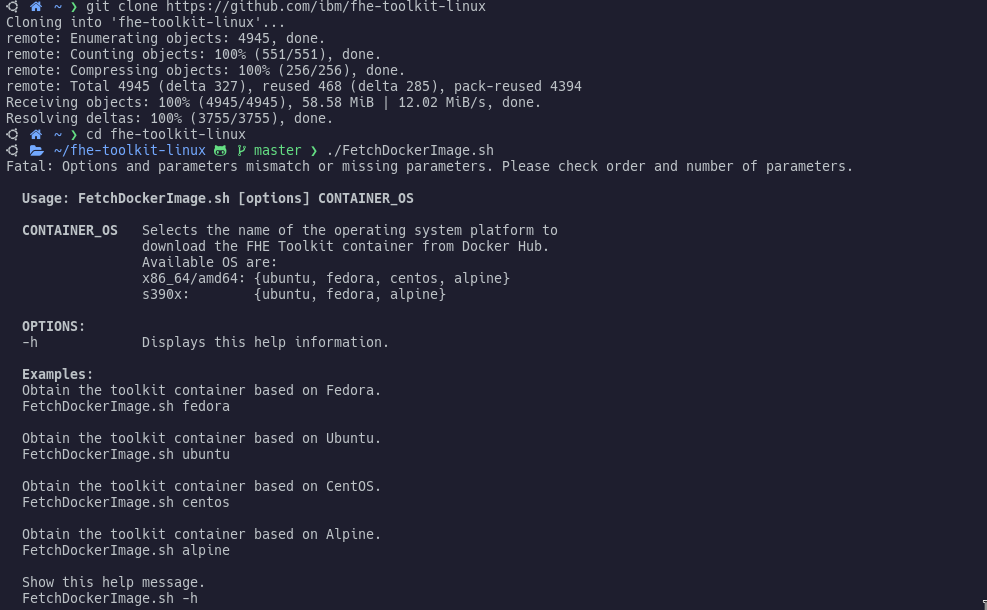
To fetch a Toolkit Docker image, first go to the folder where you cloned the toolkit and cd into the toolkit project folder.

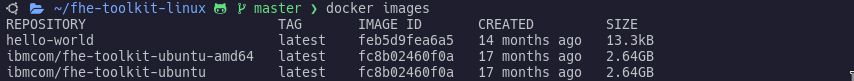
cd fhe-toolkit-linux

Fetch the Toolkit Docker images by invoking the **FetchDockerImage.sh** script followed by **<platform>** to select one of the supported platforms (centos, ubuntu, fedora or alpine). The remaining of these instructions will use CentOS as the example platform.

./FetchDockerImage.sh ubuntu

docker images

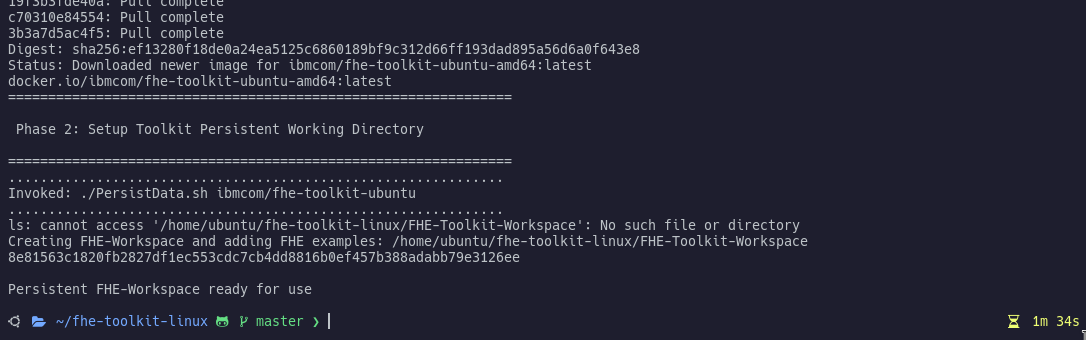




**Step 3: Running the Toolkit**

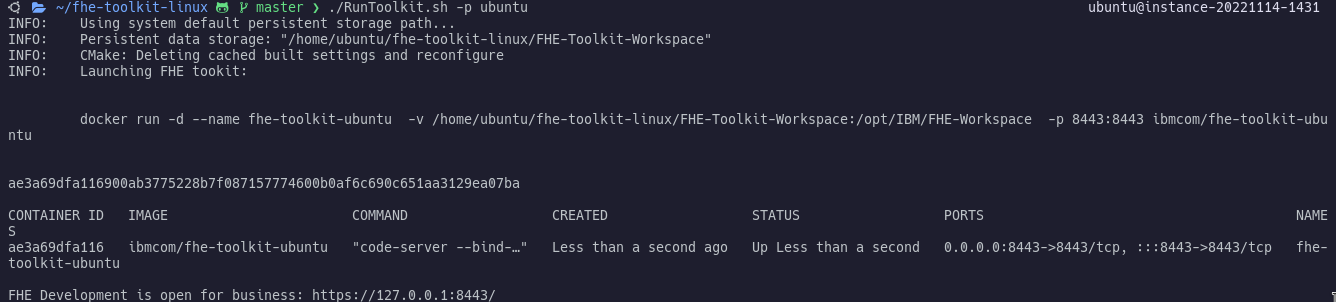
Once the fetch script has completed, invoke the **RunToolkit.sh** script with the **-p <platform>** flag. This will start the FHE Toolkit container with the IDE running as a daemon, ready to be accessed with a web browser.

./RunToolkit.sh -p ubuntu



**Step 4: Accessing the Toolkit**

Open a web browser on your host machine (not the docker container instance) and browse to **https://127.0.0.1:8443/**. This will connect you to the IDE running in the FHE toolkit Docker container.

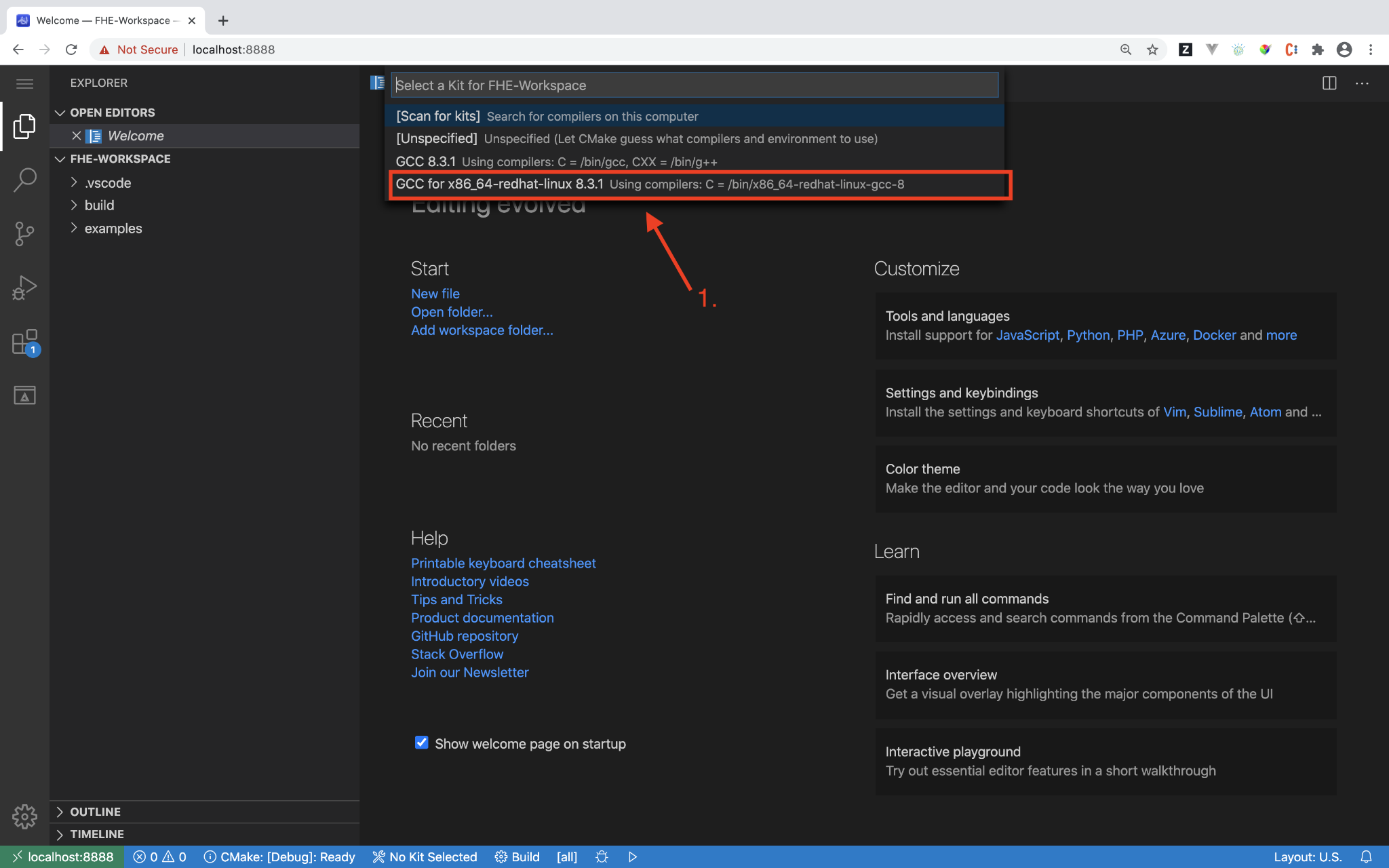


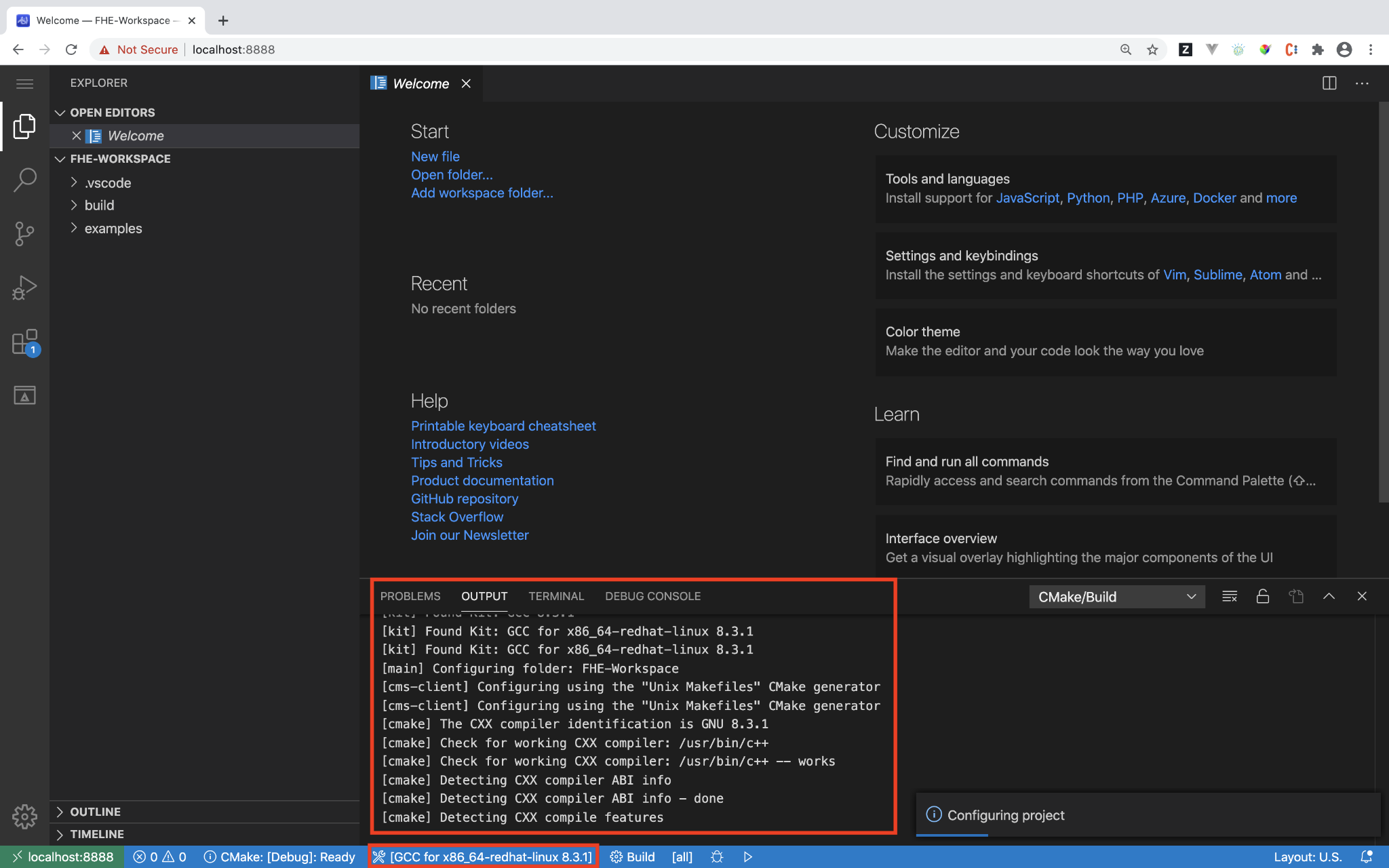
Note that this connection is secured over https using a self-signed certificate. You'll therefore need to tell your browser to trust it each time you connect to a new instance of the toolkit. In Chrome, you can do this by clicking anywhere on the warning text and typing “thisisunsafe”.

**Step 5: Configuring the Toolkit**

Once in the toolkit, you should automatically be prompted to select a kit for the FHE-Workspace to use. Select one of the kits from the dropdown. Configuration of the workspace will begin which you will be able to see in the Output Window. You'll also notice the kit you selected is now shown in the CMake Tools status bar at the bottom of the window.

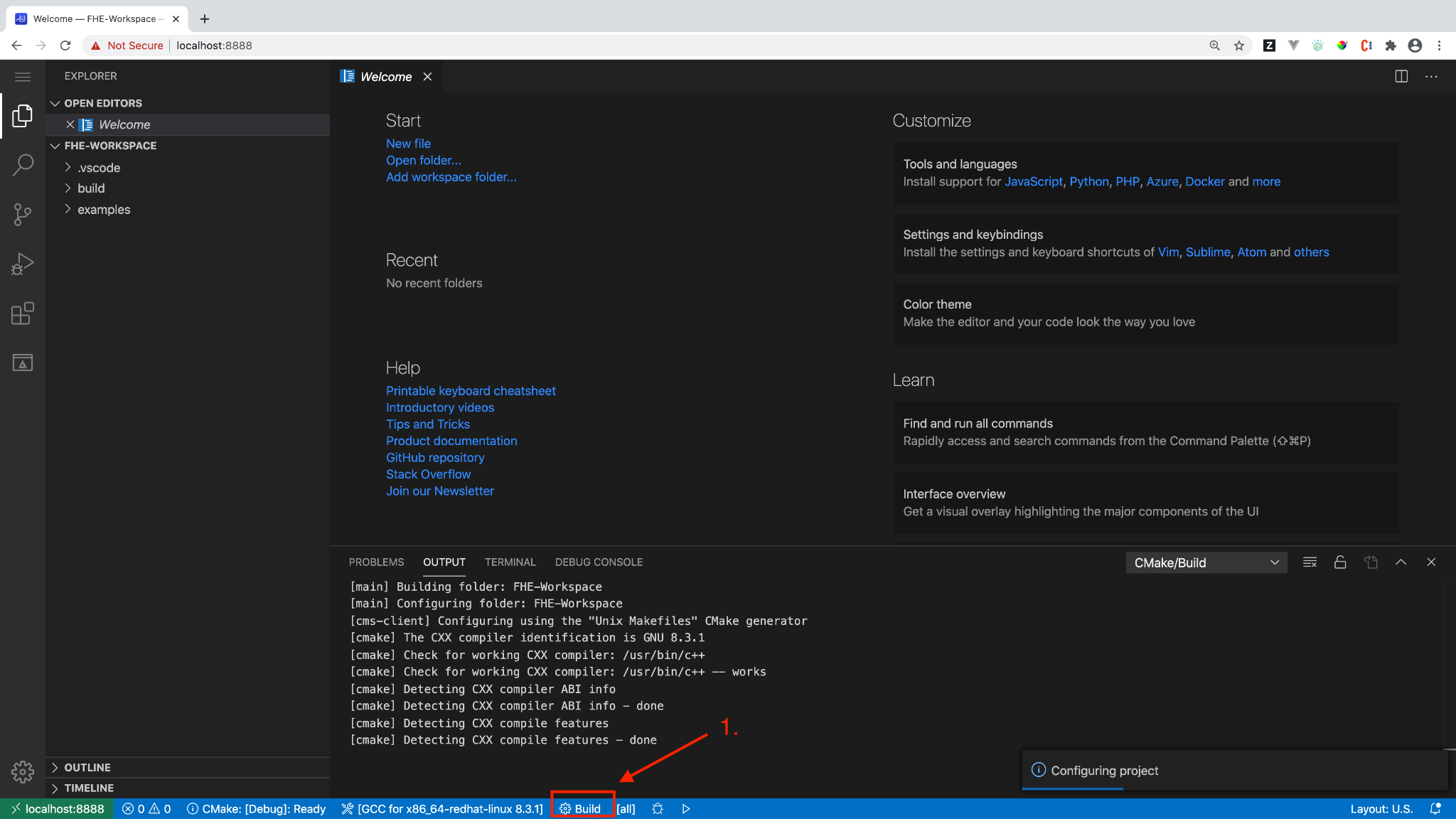
If you are not automatically prompted to select a kit after a few seconds of loading the toolkit, try refreshing your browser. If you're still not prompted, check the CMake Tools status bar at the bottom of the window to see if a kit has already been selected. You'll need to select a kit each time you start a new instance of the Toolkit.





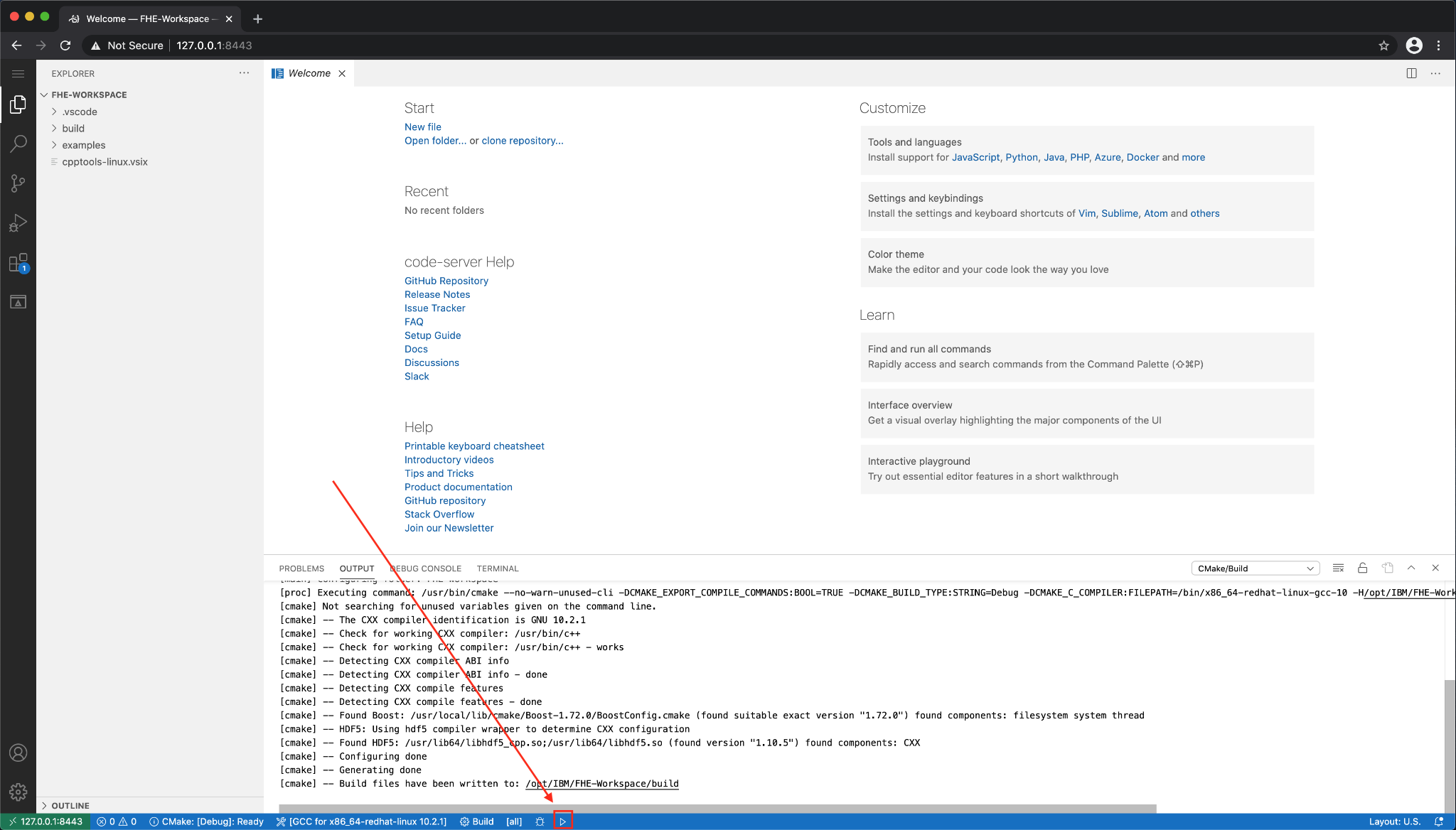
**Step 6: Building Your First HElib Example**

Click "Build" in the CMake Tools status bar to build the selected target.

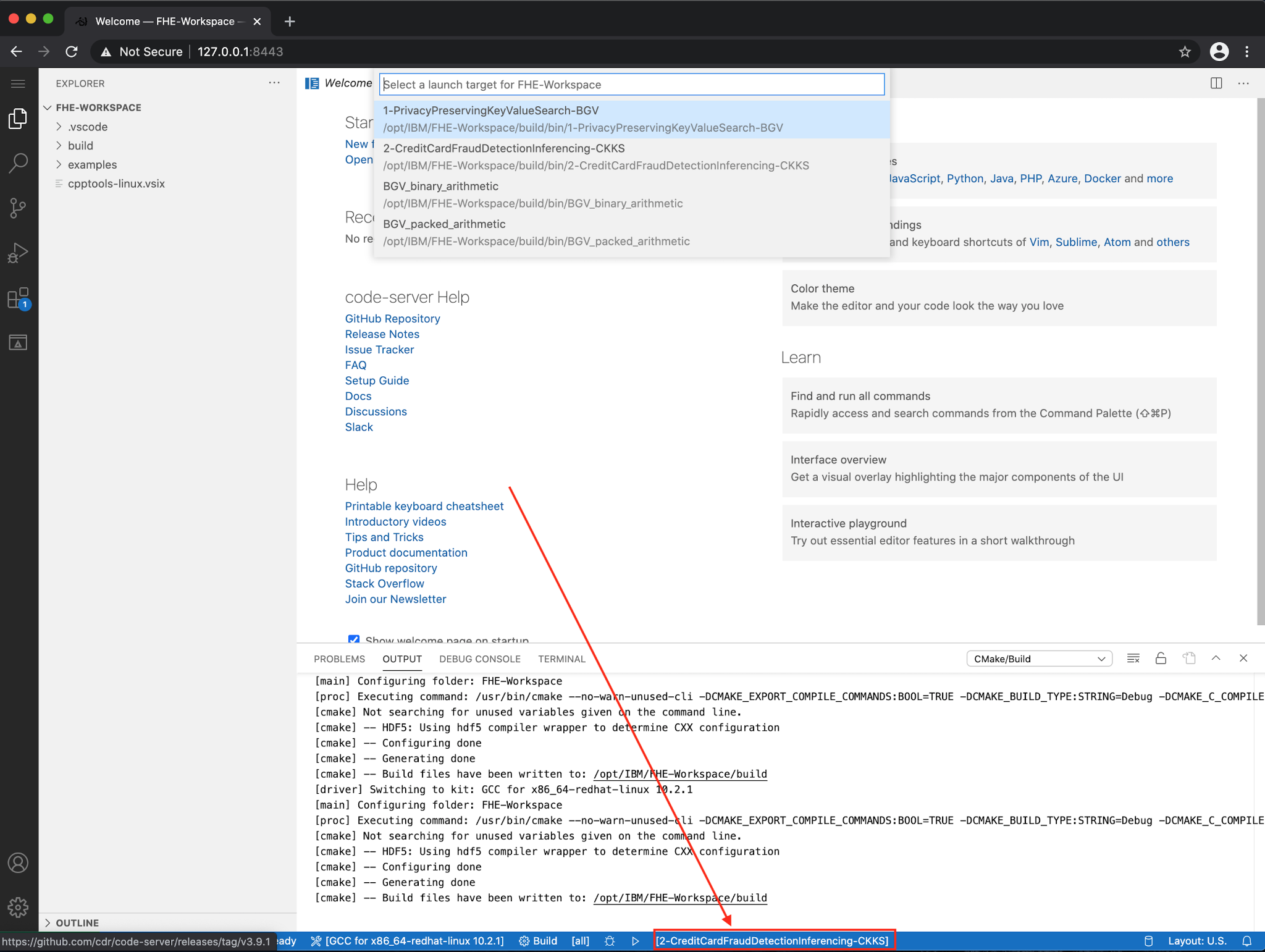


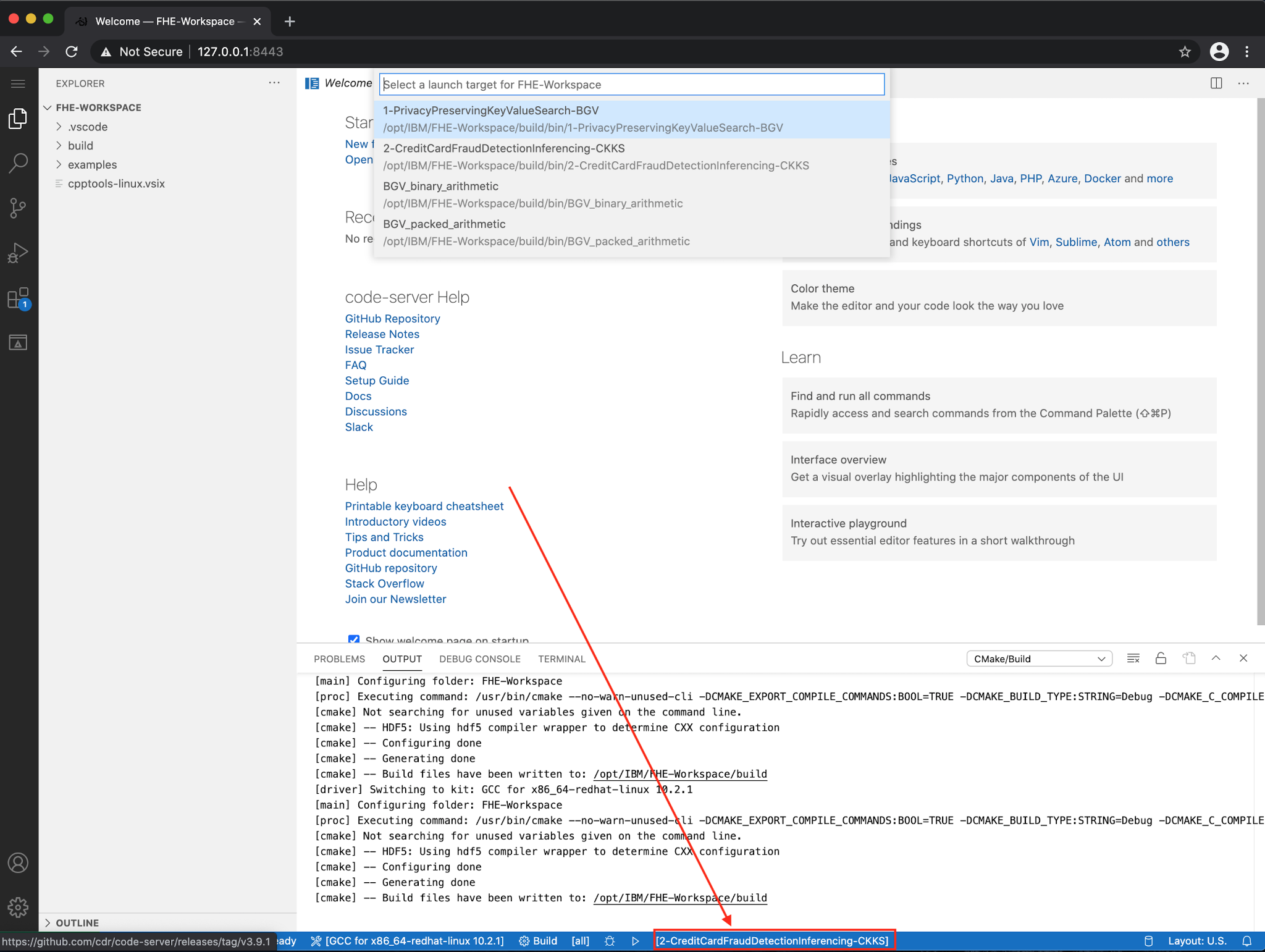
**Step 7: Running the HElib Example (Point and Click)**

When the build has finished, if there is nothing already selected (if this is the first time running the toolkit, there won't be), click "Launch". It will cause a drop down to animate on screen with the list of available demos. Select one, and it will start running that program.



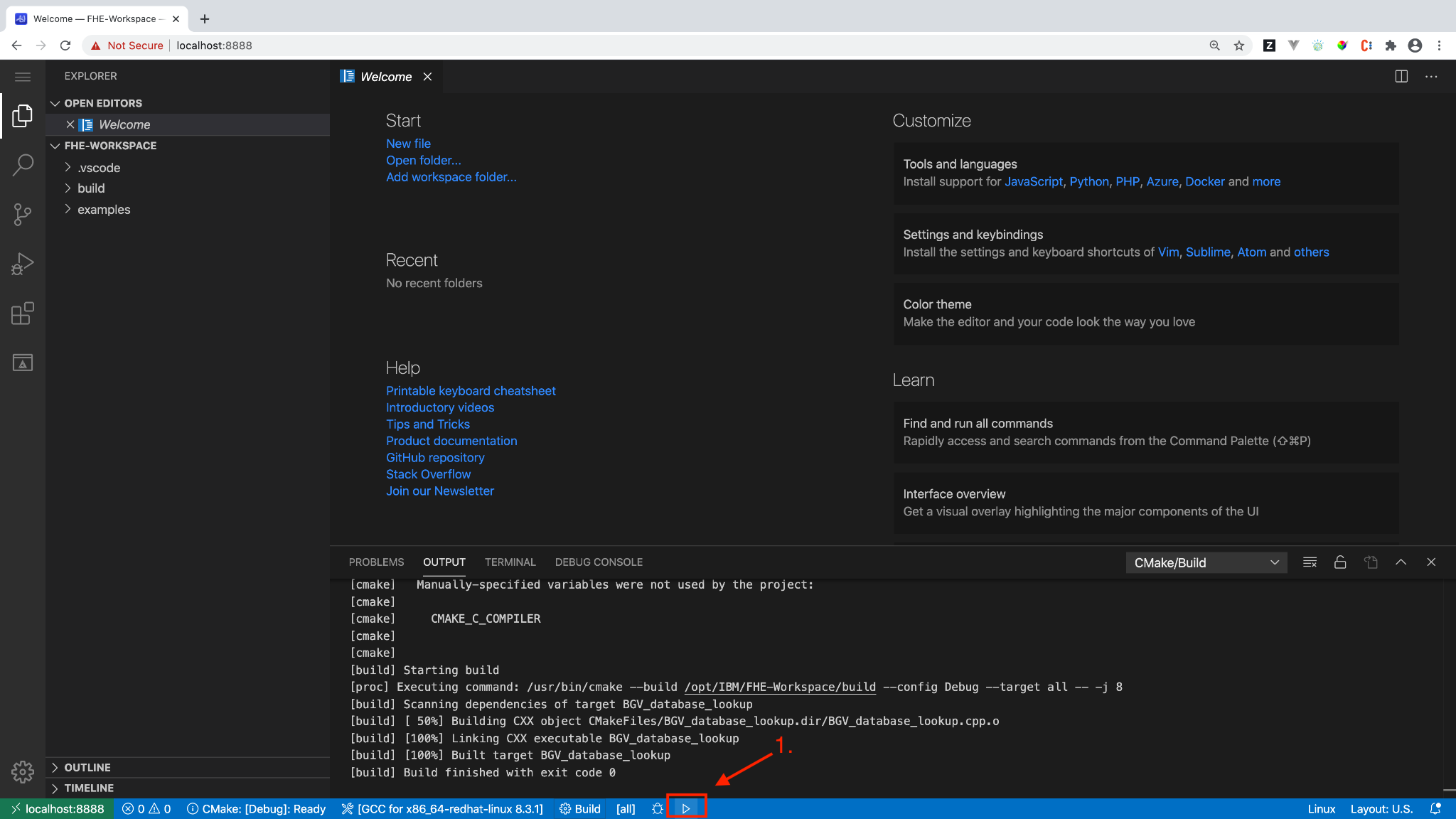
If something has been previously selected it's title will be displayed to the right of the "Launch" button. To change the selected demo click on the title and the dropdown will animate on screen with the list of available demos.



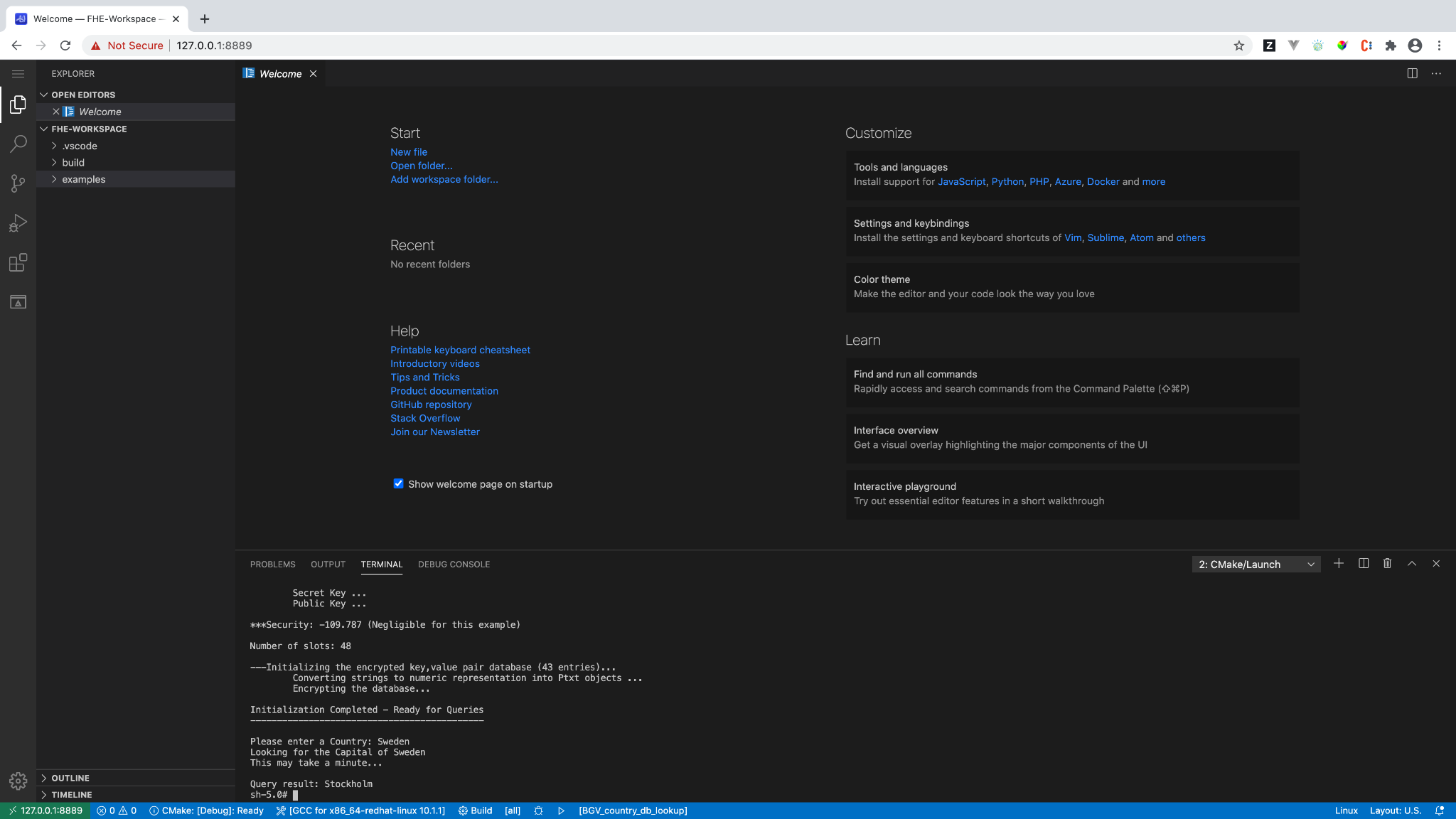


Select one, then click "Launch", and it will run that program.

We are going to use the BGV World Country Database Lookup Example Documentation which contains the information to run a complete example of a privacy preserving search against an encrypted database. The database is a key value store prepopulated with the english names of countries and their capital cities from around the world. Selecting the country will perform a search of the matching capital.



Follow the text in the demo specific documentation you have chosen. For example, if using the privacy preserving country database search, enter a country as input to find out its capital city. In the image below, the example used is **Sweden**



**Step 8: Stopping the Toolkit**

Once you've finished using the toolkit instance, run the helper script **./StopToolkit.sh** from the terminal in your host system to stop and remove all toolkit instances.

./StopToolkit.sh

**References:**

Ronald Rivest, Leonard Adleman and Mike Dertouzos

On Data Banks and Privacy Homomorphisms

<http://people.csail.mit.edu/rivest/RivestAdlemanDertouzos-OnDataBanksAndPrivacyHomomorphisms.pdf>

Armknecht, Frederik; Boyd, Colin; Gjøsteen, Kristian; Jäschke, Angela; Reuter, Christian; Strand, Martin (2015). "A Guide to Fully Homomorphic Encryption". Cryptology ePrint Archive.

D. Boneh, E. Goh, and K. Nissim. Evaluating 2-DNF Formulas on Ciphertexts. In Theory of Cryptography Conference, 2005.

Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. In Theory of Cryptography Conference, 2007.

Shai Halevi and Victor Shoup

HELib: An Implementation of Homomorphic Encryption

https://github.com/shaih/HElib

Hao Chen, Kim Laine and Rachel Player (Microsoft Research)

SEAL: Simple Encrypted Arithmetic Library

https://www.microsoft.com/en-us/research/project/homomorphic-encryption/