Below is a detailed breakdown of the key steps from generating **srcML** of the source code to finding vulnerabilities. Here's an overview of the steps involved:

Convert the source code to srcML representation

1. Download **srcML:**
    a. First, you'll need to install **srcML**. You can download it from the official **srcML** website: https://www.srcml.org
    b. Installation instructions can usually be found on the website: https://www.srcml.org/#download.
2. Convert Source Code to **srcML**:
    a. Once you have **srcML** installed, you can use it to convert your source code into an XML representation that preserves the code's syntax information.
    b. Run the following command in your terminal:
    c. `~/srcML/build/bin/srcml "input_code_file_or_archive_url" > "output_file" --position`
    d. Replace `"input_code_file_or_archive_url"` with the name of the file containing your source code or the URL of the code you want to analyze.
    e. Replace `"output_file"` with the name of the file where you want to save the srcML output.
    f. The `--position` option is important because it tells **srcML** to include the position information for identifiers in the code. This position information is crucial for the subsequent steps when **srcSlice** performs its computations.
        i. The output file generated by **srcML** will be in XML format. It will contain structured information about your source code, including the syntax tree, code elements, and their positions within the code. The XML output allows **srcSlice** to understand and analyze the code effectively.

Generate Slice Profiles using srcSlice

1. Download **srcSlice:**
    a. Visit https://www.srcml.org/tools.html and download **srcSlice**, a tool for generating code slice profiles.
    b. Run **srcSlice:** Use the command. ./srcSlice `"name of the srcML archive"` in your terminal. This will utilize the **srcML** representation of your source code to create slicing results.
    c. System Dictionary Creation**: srcSlice** will create a system dictionary containing detailed slice profiles for all identifiers present in your code. These profiles encompass definitions (def), uses (use), dependent variables (dvars), pointers (ptrs), and called functions (cfuncs).
    d. Refer to Tutorials**:** For a comprehensive understanding of the slicing results and how to interpret them, refer to tutorials like the YouTube video at: https://www.youtube.com/watch?v=fk9JiIi032U&t=80s. These resources provide valuable insights into analyzing the generated slicing profiles.

Calculate Slice-based Metrics

In this step, we will calculate the slice-based metrics: Slice Count (SC), Slice Size (SZ), Slice Coverage (SCvg), Slice Identifier (SI), and Slice Spatial (SS) based on the provided slicing results. These metrics provide valuable insights into code characteristics and complexity.

1. **Slice Count (SC):** This metric quantifies the number of slices in relation to the module size. For variables, it indicates the number of slice profiles combined to form the final slice. For functions, it represents the total count of slices encompassing all variables within the function. For files, SC corresponds to the count of slices associated with functions in that file.
2. **Slice Size (SZ):** This metric represents the total number of statements within a complete final slice and plays a role in calculating the SCvg metric.
3. **Slice Coverage (SCvg):** SCvg measures the slice size relative to the module size, quantified in lines of code (LoC).
4. **Slice Identifier (SI):** SI counts the unique identifiers, including variables and method invocations, contained within a slice relative to the module's size. These identifiers are drawn from the Dvars, Ptrs, and Cfuncs fields within the slice.
5. **Slice Spatial (SS):** SS represents the spatial distance in LoC between the initial definition and the ultimate use of the slicing variable. This value is divided by the module size, as expressed by the formula: slice distance = (Sl - Sf)/w, where Sf denotes the first statement in the slice, Sl stands for the last statement in the slice, and w signifies the module size, measured in LoC.

Run Slice-based-metrics.py: this script parses the provided slicing results and calculates the SC, SZ, Scvg, SI, and SS metrics for each slice. You can replace the sample slicing results with your actual data. The metrics are stored in dictionaries where the keys are slice names, and the values are the corresponding metric values.

Generate Slicing Vectors:

To generate slicing vectors from the slice-based metrics, you can use the following Python script. This script will take the slice-based metrics and create slicing vectors with fixed dimensions at variable, function, and file levels. The slicing vectors are generated based on the SC, SZ, Scvg, SI, and SS metrics calculated for each slice.

Run VS-vectors.py: in this script, the `slice_metrics` dictionary contains the calculated metrics for each slice. You can replace the sample data with your actual slice-based metrics. The script then generates slicing vectors at variable, function, and file levels based on the provided metrics. The slicing vectors are stored in the `slicing_vectors` dictionary, where the keys are the vector levels, and the values are dictionaries of slice names and their corresponding vectors.

Apply Locality-Sensitive Hashing (LSH) for Code Clone Detection:

Implement Locality-Sensitive Hashing (LSH) to hash the generated slicing vectors. LSH helps group similar vectors together efficiently, reducing the number of required comparisons during clone detection. To use Locality Sensitive Hashing (LSH) to hash slicing vectors and find code

clones using srcClone, you can follow these steps. Please note that you need to have srcClone and its Python interface properly set up for this script to work.

First, you will need to install the required Python libraries if you haven't already. You can install `numpy` and `pylsh` using pip:

```
pip install numpy
pip install pylsh
```

Run LSH.py : this script uses the `pylsh` library to perform LSH hashing on slicing vectors and then leverages srcClone for identifying code clones. You need to adjust the `slicing_vectors` list with your actual slicing vectors. Additionally, make sure you have the srcClone Python interface set up and configured correctly.

You may need to adjust the LSH parameters (number of bands, number of rows) and similarity threshold to suit your specific use case and data. You can also modify the code to read slicing vectors from a file or database if needed.

Clone detection using srcClone:

Run Clone.py: replace the `vectors` list with your actual vectors. This script uses the `LSHForest` from scikit-learn to find nearest neighbors based on LSH and then applies a similarity threshold to detect code clones. You can adjust the `n_neighbors` and `n_estimators` parameters to fine-tune the search. Please note that you'll need to adapt this script to work with your specific vector data format and similarity measures, but this should give you a starting point for detecting code clones based on your generated vectors.

Clone detection using Nicad (optional):

You can use the open-source clone detection tool like "Nicad" to find source code clones. Here is a Python script that demonstrates how to use Nicad:

```
import os
import subprocess

# Define the path to Nicad executable and the input parameters
nicad_path = "/path/to/nicad/bin/nicad"
input_directory = "/path/to/your/source/code"
language = "java"  # Change this to the appropriate language
(e.g., java, c, c++, etc.)
output_directory = "/path/to/output/folder"

# Create the output directory if it doesn't exist
if not os.path.exists(output_directory):
    os.makedirs(output_directory)

# Run Nicad clone detection
```

```
command = [
    nicad_path,
    "-p", language,
    input_directory,
    "-o", output_directory,
]
subprocess.run(command)

print("Nicad clone detection completed.")
```

Replace the `/path/to/nicad/bin/nicad` with the actual path to the Nicad executable, set `input_directory` to the path of your source code, specify the programming language using the language variable, and set the `output_directory` where you want the results to be stored.

This script will run Nicad on your source code and save the results in the specified output directory. Be sure to have Nicad installed and configured on your system before running the script. You can download Nicad from its GitHub repository:

Please note that the actual usage of Nicad may vary based on your specific codebase and requirements, so you should refer to Nicad's documentation for more details on customizing clone detection parameters.