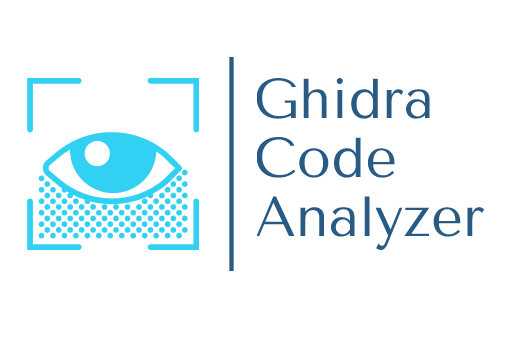
|  |  |
| --- | --- |
| Ghidra Code Analyzer | SAIT  1301 16 Ave NW, Calgary, AB T2M 0L4 |

Formal Report





|  |  |
| --- | --- |
|  | Table of Contents |
|  |

[Executive Summary 2](#_heading=h.gjdgxs)

[Project Scope 4](#_heading=h.30j0zll)

[Project Description 4](#_heading=h.3rdcrjn)

[Lessons Learned](#_heading=h.3znysh7) 5

Challenges6

[Conclusion](#_heading=h.2et92p0) 7

[Acknowledgments](#_heading=h.tyjcwt) 8

[References](#_heading=h.3dy6vkm) 9

[Documentation](#_heading=h.1t3h5sf) 10

[Glossary 1](#_heading=h.4d34og8)3

[Appendices A (Code) 27](#_heading=h.35nkun2)

|  |  |
| --- | --- |
| Section | Executive Summary |
| 1 |

In this document, we will lay out our project’s goals and features, the Ghidra Code Analyzer. The purpose of this project is to export the decompiled code from Ghidra, parse and remove variables from the function, and compare it to C libraries or other bodies of code such as Malware. This makes identifying functionality in code much easier for the analyst.

This report will include the scope of the project (Including out of scope items), the lessons we learned during this project course, any recommendations we might have for the future of the project, and our conclusion.

We will also include the code at the end of this document for reference, as well as our Gantt chart.

|  |  |
| --- | --- |
| Section | Project Scope |
| 2 |

**Project Scope**

* Exports Decompiled Ghidra Function
* Scrubs code of comments and Variable names
* Compares exported code with C libraries or other bodies of code
* Output a percentage match to screen
* Output Percentage match and matched code

**Out of Scope**

* A GUI
* Exporting a specific decompiled functions for comparison

# Project Description

We chose this project to assist in the process of analyzing code by comparing decompiled code to an existing body of code. This is done by exporting a decompiled function form Ghidra, scrubbing them of unneeded data such as comments and unique variable names, and then comparing them against existing bodies of code. Once this is done, the script outputs the percentage of matched items to screen, as well as the matched items to an output file.

|  |  |
| --- | --- |
| Section | Lessons Learned |
| 3 |

* Python Programing (Regex to find and remove specific items from export code)
  + For ease of comparison we needed to find a way to pull unique variable names and replace them with a standard variable name. In order to achieve this we created a Regex pattern to catch the unique variable names Ghidra creates and standardize them.
* Java Programing and structure to integrate our python scripts with the native Java
  + In order to export decompiled functions from ghidra we required to interact with the Ghidra core using Java. While there was potential for us to use Python, learning materials are still quite limited for Python Ghidra development.
* Jython to get our Java and Python to talk to one another
  + Ensure that python logic was used in such a way to allow it to use Java structures and classes through the Jython interpreter. Allowed us to call existing Java code in Ghidra.
* Ghidra scripting environment
  + Ghidra and Jython both require unique syntax for certain functionalities. Originally we developed most of our code outside of Ghidra which led to us having to rewrite some elements to work within Ghidra.
* Application development cycle
  + Utilizing pseudo code to understand how the project will develop. This allowed us to understand our code and effectively plan what we would work on.
* Creating algorithms in Python to compare the code.
  + We wanted the most efficient and effective way to compare the files. Originally, we started off with the idea of comparing the line-by-line which resulted with few matches. We then switched to comparing them word-by-word to get a more effective way to match. This method was successful but it didn’t take into account the size of files since a smaller function file and a larger file to compare would give a higher likelihood of matching and vice versa. To avoid this error, we had to create a function which takes in the percentage and the file sizes to provide more insight into the likelihood of a match.
  + Throughout this process we gained a higher understanding of programming with python and algorithms.

|  |  |
| --- | --- |
| Section | Challenges |
| 4 |

* Debugging our matching algorithm to get the output that we desired
  + Issues with the math that determines our percentage match and the amount of lines compared where coming out incorrectly. We rewrote the code from a different perspective and were able to get the proper output.
* Understanding regular express patterns to find the desired bits of code to remove
  + We needed to find specific strings within our exported code to remove unique variables to produce a better match in our results. After learning about regular expressions for python, we were able to replace those variables with a generic variable name.
* How to interact with Java from Python using Jython
  + To use the Jython module, we had to ensure that all python was written with version 2.5. We also had issues with them smoothly communicating with each other.
* Trying to manage a project remotely during a pandemic
  + Fortunately, our project was mostly programming which is easy to conduct remotely. The difficulties that we came across was collaboration to overcome obstacles within our project.
* Understanding Java
  + Most of our team was unfamiliar with Java. To overcome this we had to spend some time learning Java from books and online sources.

|  |  |
| --- | --- |
| Section | Improvements to be made |
| 5 |

* Having the option to export one single function from Ghidra rather than all decompiled functions. This would allow an analyst to narrow down what he wants to look at and reduce run time.
* A GUI to improve readability of the program to allow the analyst to better understand the data being presented.
* Have the replacement function replaced with generic data type variables rather than just generic variables. This would maintain more of the original code while still allowing for better results in the output.
* A more formalized report by having higher matched files appear on the top of the outputted report. Also having more in depth analysis on the similarities of the compared files.

|  |  |
| --- | --- |
| Section | Conclusion |
| 6 |

During the duration of this project, we were able to build a script that exports, strips them of variables and comments, and compares Ghidra functions to an already existing body of code. This gives us an output that lets us know the similarities and the percentage match between the code. Along the way we encountered some issues and learned some things from these challenges.

We increased our proficiency in Python, learned about Jython to get our script to interact with Java classes, using regular expressions to remove variable names, braces and comments, and learned about the application development cycle. Each of these lesions aided in the completion of our project, and were able to deliver on most of our scope.

The experience and knowledge that we gain from this project will be essential to our growth in this field in the future, and we look forward to grappling with more challenges like it.

|  |  |
| --- | --- |
| Section | Acknowledgments |
| 6 |

Viktor Lyagutsky

Patricia Castillo

Anthony Leece

Enyi Abajue

Regner Sabillon

Siona Gunn-Graham

Henri StLouis

George Chase

|  |  |
| --- | --- |
| Section | References |
| 7 |

<https://jython.readthedocs.io/en/latest/> - Jython programming guide

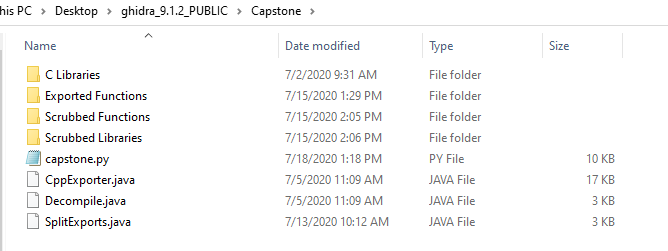
|  |  |
| --- | --- |
| Section | Documentation |
| 8 |

**Quick install:**

1. Download Capstone.rar from the project homepage and extract the contained “Capstone” folder and its contents into your Ghidra root directory. The folder should appear as below:

\*Ghidra\_root\Capstone\

1. Continue to add the folder to your Ghidra script directories.



**Manual Setup:**

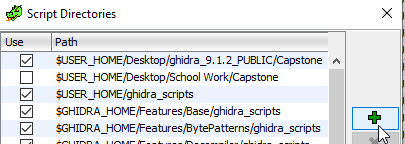
1. Create a folder called “Capstone” within your Ghidra root directory
2. Within the “Capstone” folder create the following folders: (Shown above)
   1. “C Libraries”
   2. “Exported Functions”
   3. “Scrubbed Functions”
   4. “Scrubbed Libraries”
3. Download the “capstone.py” and “SplitExports.java” files from the project homepage.
4. Continue to add the folder to your Ghidra script directory.

**Add the “Capstone” folder to your Ghidra Scripts directories:**

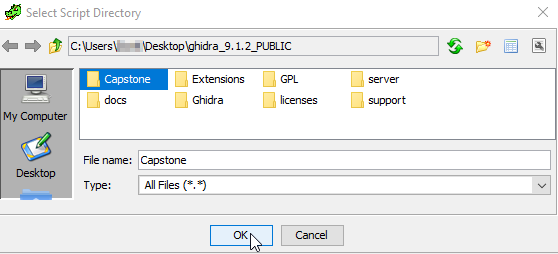
1. In Ghidra, select the “Window” menu option from the top bar and navigate to the “Script Manager”
   1. Alternatively, select the  symbol in the toolbar.
2. Select the “Script Directories” icon



1. Click the green cross to add a new directory



1. Navigate to your Ghidra installation and select the contained “Capstone” folder and click OK

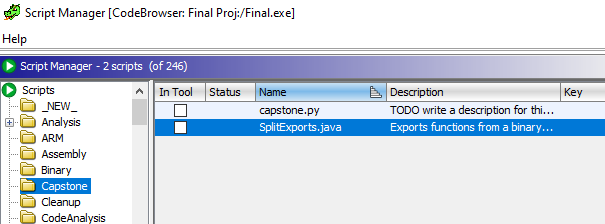


1. You should now see the “capstone.py” and “SplitExports.java” files in your Script Manager.

**Usage:**

\*\*Note\*\* Within the Capstone folder there are 4 contained folders.

* C Libraries - Contains Standard C libraries for testing and comparison
* Exported Functions - Destination of functions exported from Ghidra
* Scrubbed Functions - Destination for exported functions once variables have been standardized (This is the folder the script will scan for functions for comparison.)
* Scrubbed Libraries - Destination for libraries and functions once the comments have been stripped. (This is the folder containing functions to be compared against.)

1. Within Ghidra, open an executable and decompile it for analysis.
2. Once the executables functions have been decompiled select the “Window” menu option from the top bar and navigate to the “Script Manager”
   1. Alternatively, select the  symbol in the toolbar.
3. The scripts can be found under the “Capstone” folder on the left hand side bar. 
4. Execute the SplitExports.java script to extract the functions from the decompiled executable. These will be output to the “Exported Functions” folder.
5. Run capstone.py and it will ask you how you wish to proceed.
   1. (S)crub Libraries: This will go through the files in “C Libraries” directory and clear all C style comments to prevent them from being analyzed. Outputs into the “Scrubbed Libraries” folder
   2. (R)eplace Variables: This function will go through the files in “Exported Functions” and alter all variable names to a standard variable name for analysis outputting them to the “Scrubbed Functions” directory.
   3. (C)ompare Functions: Compares the files in “Scrubbed Functions” against those in “Scrubbed Libraries” produces an output file in the Ghidra root directory. This file will contain the results of analysis and comparative percentages of the functions analyzed.

|  |  |
| --- | --- |
| Section | Glossary |
| 9 |

**Terminology:**

Ghidra - NSA’s executable reverse engineering and static analysis toolset.

Regex - Regular expressions used to search for specific words within code.

Jython - A module used to recognize python code within java.

|  |  |
| --- | --- |
| Section | Appendices A (Code) |
| 10 |

**SplitExports.java:**

|  |
| --- |
| // Exports functions from a binary to a set of C files. Each function that  // ends with \_\_\_FILENAME will be exported to FILENAME.c  //@author GW\_Ponder  //@category Capstone  //@keybinding  //@menupath  //@toolbar  //  import java.io.BufferedWriter;  import java.io.FileWriter;  import ghidra.app.decompiler.flatapi.FlatDecompilerAPI;  import ghidra.app.script.GhidraScript;  import ghidra.program.model.listing.Function;  import ghidra.program.model.listing.FunctionIterator;  import ghidra.program.model.listing.Listing;  public class SplitExports extends GhidraScript {    @Override  public void run() throws Exception {    // Identify directory for files  // This example puts them in user's home directory in Windows.  String path\_name = System.getProperty("user.dir") + "\\Capstone\\Exported Functions\\";    // create a FlatDecompilerAPI to handle the decompilation  FlatDecompilerAPI fda = new FlatDecompilerAPI(this);    monitor.setMessage("Selecting functions to export...");  Listing listing = state.getCurrentProgram().getListing();  FunctionIterator iter = listing.getFunctions(true);    // While there are more functions and the user hasn't cancelled  while (iter.hasNext() && !monitor.isCancelled()) {  // get the next function  Function func = iter.next();    // get the name of the function  String func\_name = func.getName();    // only export functions that have names that end in \_\_\_FILENAME  // where filename is the name of the file to append the function to  if (func\_name.contains("\_\_\_") ) {  println("Found function " + func\_name);    String file\_name = path\_name + func\_name.split("\_\_\_")[1] + ".c";  println(" Writing to " + file\_name);    // Open FILENAME.c for append  BufferedWriter c\_file = new BufferedWriter(  new FileWriter(file\_name, false));    // Write the decompiled version of the function  c\_file.write(fda.decompile(func));    // close the function  c\_file.newLine();  c\_file.close();  }  }  }  } |

**capstone.py:**

|  |
| --- |
| #TODO write a description for this script  #@author Scott Matheson, Shayne Gradwell, Tyson Jamison  #@category Capstone  #@keybinding  #@menupath  #@toolbar  import os  import re  import sys  from sets import Set  #https://gist.github.com/ChunMinChang/88bfa5842396c1fbbc5b  def comment\_remover(text):  def replacer(match):  s = match.group(0)  if s.startswith('/'):  return "" # note: a space and not an empty string  else:  return s  pattern = re.compile(  r'//.\*?$|/\\*.\*?\\*/|\'(?:\\.|[^\\\'])\*\'|"(?:\\.|[^\\"])\*"',  re.DOTALL | re.MULTILINE  )  return re.sub(pattern, replacer, text)    #Step through all files in directory  def library\_scrubber():  for filename in os.listdir("Capstone\C Libraries"):  if filename.endswith(".h"):  print(os.path.join("Capstone\C Libraries", filename))  f=open(os.path.join("Capstone\C Libraries", filename), "r")  outfile=open(os.path.join("Capstone\Scrubbed Libraries", filename), "w")  fileData=(f.read())  cleanData=(comment\_remover(fileData))  outfile.write(cleanData)  outfile.close()  f.close()  def replace():  print "Scrubbing variables in Exported Functions"  repFiles=os.listdir("Capstone\Exported Functions")  for file in repFiles:  outFile=open(os.path.join("Capstone\Scrubbed Functions", file), "w")  infile=open(os.path.join("Capstone\Exported Functions", file), "r")  funcPatt = "FUN\_"+'[ab-z]?\d+[ab-z]?\d+'  for line in infile:  variables=["local\_","local\_res","uVar","iVar","puVar","ppHVar","pcVar","lVar","pFVar","pHVar"]  strip = line.strip()  for index in variables:  #print index  pattern = index+'[ab-z]?\d+'  if re.search(pattern,strip):  strip = re.sub(pattern,"repVar",strip)  if re.search(funcPatt,strip):  strip = re.sub(funcPatt,"repFunc",strip)  outFile.write(strip+"\n")  outFile.close()  infile.close()  print "Done!"    #Compares the matched percentage against the size of the files.  #A function that is much smaller than the file its being compared against is more likely to have a match and vice versa.  def enhanced\_comparison(sizeComp, matchPercentage):  if sizeComp >= 200.00:  if matchPercentage >= 75.00:  resLik = "high"    elif matchPercentage >= 50.00:  resLik = "high "    elif matchPercentage >= 25.00:  resLik = "medium"    elif matchPercentage >= 10.00:  resLik = "low"    else:  resLik = "improbable"    elif sizeComp >= 100.00:  if matchPercentage >= 75.00:  resLik = "high"    elif matchPercentage >= 50.00:  resLik = "medium"    elif matchPercentage >= 25.00:  resLik = "medium"    elif matchPercentage >= 10.00:  resLik = "low"    else:  resLik = "improbable"    elif sizeComp >= 80.00:  if matchPercentage >= 75.00:  resLik = "high"    elif matchPercentage >= 50.00:  resLik = "medium"    elif matchPercentage >= 25.00:  resLik = "low"    elif matchPercentage >= 10.00:  resLik = "low"    else:  resLik = "improbable"    elif sizeComp >= 50.00:  if matchPercentage >= 75.00:  resLik = "medium"    elif matchPercentage >= 50.00:  resLik = "medium"    elif matchPercentage >= 25.00:  resLik = "low"    elif matchPercentage >= 10.00:  resLik = "improbable"    else:  resLik = "improbable"    elif sizeComp >= 25.00:  if matchPercentage >= 75.00:  resLik = "medium"    elif matchPercentage >= 50.00:  resLik = "low"    elif matchPercentage >= 25.00:  resLik = "low"    elif matchPercentage >= 10.00:  resLik = "improbable"    else:  resLik = "improbable"    else:  if matchPercentage >= 75.00:  resLik = "low"    elif matchPercentage >= 50.00:  resLik = "low"    elif matchPercentage >= 25.00:  resLik = "low"    elif matchPercentage >= 10.00:  resLik = "improbable"    else:  resLik = "improbable"    return resLik    def compare():  cLibs = os.listdir("Capstone\Scrubbed Libraries") #Gets the files and subdirectories from the specified directory.  compFuncs = os.listdir("Capstone\Scrubbed Functions")  for scanning in cLibs: #Will loop for every opened library file.  if scanning.startswith("rep"):  scanLibs = os.path.join("Capstone\Scrubbed Libraries", scanning)  for func in compFuncs:  scanFunc = os.path.join("Capstone\Scrubbed Functions", func)    funcFile = open(scanFunc, 'r')  libFiles = open(scanLibs, 'r')  outputTxt = open('CodeAnalyzerResults.txt', 'a+')  print "Comparing files %s & %s" % (scanLibs, scanFunc)    remBrac = 0  funcWordCount = 0  libWordCount = 0  funcMatch = 0  matchPercentage = 0.0  sizeComp = 0.0      funcWords = funcFile.read().split() #Splitting the files into individual words.  libWords = libFiles.read().split()  matchedWords = set(funcWords) & set(libWords) #Reading for what words are matched.  funcWordCount = len(funcWords) #Counting the amount of words.  libWordCount = len(libWords)  sizeComp = (float(funcWordCount)/float(libWordCount))\*100    for word in matchedWords:  if word == '{' or word == '}':  if funcWords.count(word) > libWords.count(word): #A loop to exclude brackets.  remBrac += libWords.count(word)    else:  remBrac += funcWords.count(word)    else:  outputTxt.write('MATCHED: {} Occurs {} times within the function and {} times within {}.\n' .format(word, funcWords.count(word), libWords.count(word), scanning))      if funcWords.count(word) > libWords.count(word): #A loop to exclude multiple matches.  funcMatch += libWords.count(word)    else:  funcMatch += funcWords.count(word)    outputTxt.write("\nAbove results are from: {} compared against {}\n" .format(func, scanning))  outputTxt.write("Total words scanned in for comparison = {}\n" .format(libWordCount))  outputTxt.write("Total matched words in the function = {}\n" .format(funcMatch))  outputTxt.write("Brackets filtered out = {}\n" .format(remBrac))    funcWordCount = funcWordCount - remBrac  matchPercentage = (float(funcMatch)/float(funcWordCount))\*100  outputTxt.write("Exported function word count: {} and compared file word count: {}\n" .format(funcWordCount, libWordCount))  outputTxt.write("Matched: {} / {}. Percentage of match is = {}%\n" .format(funcMatch, funcWordCount, round(matchPercentage,2)))  outputTxt.write("Likelihood of a match is: {}\n\n\n" .format(enhanced\_comparison(sizeComp, matchPercentage)))  funcFile.close()  libFiles.close()  outputTxt.close()  def main():  userIn = 'a'  while userIn.lower() != 'q':  userIn = askString("Action Selection", "Would you like to (S)crub libraries, (R)eplace variables, (C)ompare functions or (Q)uit: ")  if userIn.lower() == 's':  library\_scrubber()  elif userIn.lower() == 'r':  replace()  elif userIn.lower() == 'c':  compare()  elif userIn.lower() == 'q':  print "Exiting..."  else:  popup("Error: Unrecognized input.")  print(os.getcwd())  #SplitExports.run()  main() |