**REEZ: Towards Simplified Static Program Analysis**

**Kirill Kultinov**

**I BACKGROUND**

Reverse Engineering has been an important area of Computer Science for many years. It helps to complete different tasks associated with program analysis, plan recognition, and architecture recovery. More specifically, purposes of reverse engineering are debugging, re-engineering, and understanding malware. Also, the process of reverse engineering is used to recover APIs and interfaces. Many reverse engineering tools and frameworks have been created in the past. They make the process of reverse engineering easier by providing wide variety of APIs and visualizations. Examples of these tools and frameworks are angr, radare2, and IDA. Despite the list of functionalities provided by these tools, there is room for improvement. Moreover, it is very important to automate the process of reverse engineering and tasks related to it. Many reverse engineering tools and disassemblers have some disadvantages and limitations. Fortunately, majority of reverse engineering tools and frameworks allow extensions using scripts and plugins that help to automate or simplify some parts of program analysis.

The purpose of this section is to explain how the most popular tools and frameworks can be extended. Specifically, this section describes what must be done to start development of extensions and what tools are required for it. In addition to that, some details of the preparation process for configuring the development environment are also covered. Finally, this section presents sample extension scripts to familiarize readers with the process of extending these tools. We will be using Linux Ubuntu in cases where it is possible. In case if some tools are only available for Windows, we will use Windows 10 OS.

**Angr**

Angr is a powerful toolkit for analyzing binaries. Essentially, this is a Python framework that contains all required functions to perform debugging, dynamic symbolic executions, and different forms of static analysis on binaries. This framework is available on many platforms including, Windows, Linux, and Mac OS. Primarily, angr is used with python 2.6 and 2.7. It is also compatible with python 3. However, according to the official angr documentation, it may not work properly with python 3 (Docs.angr.io, 2018). This framework provides many opportunities for extensions by writing scripts using angr API. Angr extensions can be used in control-flow graph recovery, symbolic execution, and automatic ROP chain building. In addition to that, angr framework can be used in scripts to solve different crackme problems.

A set of required tools and dependencies must be installed prior working on extending angr. First of all, the following dependencies must be installed: python-dev, libffi-dev, build-essential, virtualenvwrapper. In addition to that, virtual environment virtualenv for python must be installed using python pip (Website & Mobile Application Development, Digital Marketing Agency, 2018). It is important to notice that vritualenv must be exported to make it available from any directory. First, we nee to create a directory for virtual environments to reside by using the following command:

mkdir ~./environment

Then the following lines must be added to “~./bash.rc” file:

export WORKON\_HOME=~/.environments

source /usr/local/bin/virtualenvwrapper.sh

Now everything is ready to start extensions development for angr. It is important to understand basic core concepts of angr to create powerful scripts. First of all, creation process of any extensions requires importing angr module and loading the binary file. Every binary file belongs to a Project object under angr API. The code example will look the following:

import angr

proj = angr.Project('sampleProgram’)

Where Project constructor accepts path to the binary file as a parameter. Using instance of a proj we can access basic information about the binary. “proj.arch” is used to retrieve info about the CPU architecture. “proj.entry” will display starting address of a program in virtual memory.

An interesting example of using angr is to disassemble a binary file. We can extract basic blocks of a binary file by using “proj.factory“ that contains functions that are used frequently. For example, it is possible to print out disassembly of an entry block of a program by calling “proj.factory.block(proj.entry)”. Another useful feature that can be used in angr’s extensions is to represent a program at any given point in time. A state of a program is retrieved using the following code:

state = proj.factory.entry\_state()

simgr = proj.factory.simulation\_manager(state)

We can get a control-flow graph of a binary by writing an extension for angr using Python and angr API. We wrote the following script:

import angr

def main():

proj = angr.Project("../sampleC/hello", auto\_load\_libs=False )

cfg = proj.analyses.CFG()

print(dict(proj.kb.functions))

if \_\_name\_\_ == '\_\_main\_\_':

main()

This python script is then simply executed using command line.

**Radare2**

Radare is a command line utility that allows working with binary files and performing different operations such as disassembling code, debugging programs, and program analysis. It is available on most operating systems. Radare has a wide variety of different commands to perform different operations on binary files. Examples of commands are “pdf” (disassemble binary file), “f flagName” (creating a flag type), “r” (resizing binary file), and “w” (write bytes to the binary file). Executing radare’s commands manually is a time-consuming task to achieve certain goals while doing program analysis. That is why this process must be automated. Fortunately, radare2 is a scriptable tool that can be extended writing scripts in many different programming languages including Python, Swift, JavaScript, and NewLisp. In addition to that, it is possible to write plugins to extend functionality of radare utility.

First, we describe how to extend Radare2 using scripts. Installation of radare2 requires cloning the git repository and running /sys/install.sh bash script. Automation of program analysis is achieved by using r2pipes, which establish a bidirectional communication using pipes. That is why r2pipe package must be also installed. Writing an extension for radare2 requires import of r2pipe package. We wrote a sample Python script that analyzes all symbols and calls and prints out a disassembled binary file:

import r2pipe

r2 = r2pipe.open("../sampleC/hello")

r2.cmd('aa')

print(r2.cmd("pdf@main"))

Generally speaking, each command that needs to be executed is passed to “cmd” function as a parameter.

A more complicated approach of extending radare2 is to write a plugin. According to Radare’s official web page, plugins must be written in C programming language (Radare.gitbooks.io, 2018). Source code of a plugin must be inserted in the following directory “/libr/libName/p/” where “libName” is the name of radare’s component we want to extend. According to the documentaion, the C file must contain the following:

* C struct for initializng the pluging that contains type and data source to the plugin’s information
* C struct containing plugin’s information
* A procedure that will be called when the function is executed

A skeleton for a basic plugin will have the following form:

static int disassemble(RAsm \*a, RAsmOp \*op, const ut8 \*buf, int len) {

//TODO

}

PluginType pluginName = {

.name= “name”

.arch = “architecture”

. bits = 16 | 32 | 64

.desc = “command description”

.disassemble = &disassemble

.assemble = NULL

}

#ifndef CORELIB

RLibStruct radare\_plugin = {

.type = R\_LIB\_TYPE\_ASM,

.data = &pluginName,

.version = R2\_VERSION

};

#endif

When the code is completed, we need to add it in the “/libr/include/headrFile.h” file by inserting the following line “extern PluginType pluginName;” (The Official Radare Blog, 2018). New make file is also needed in “libr/libName/p/” directory. Finally, it is required to modify “plugins.def.cfg” file in the root of radare2 directory to tell compiler if the plugin needs to be compiled as either static or shared. At the end, we need to execute ./configure and make commands.

**Immunity Debugger**

Immunity Debugger is a powerful debugger that makes program analysis way easier and faster. It has many advantages previously described tools because it provides graphical user interface and many features. Most importantly, it can be easily extended using Python scripts. However, Immunity debugger is only available for Windows OS.

Writing plugins for Immunity is a very simple process. First of all, Python 2.6 or 2.7 must be installed. Each Python script has to include the following components:

* Imports for Immunity debugger API’s modules
* Instance of a debugger object (immlib.Debugger())
* Main function (def main(args):)
* Other used defined functions (if any)

Each script then must be placed in the following directory “\Immunity Debugger\PyCommands\” and have “.py” extension. Each script can be executed using a command line using “!scriptName” command. Immunity Debugger API contains many different classes that can be used for different purposes such as disassembling, searching memory, and displaying SEH chains. Another useful feature of Immunity is creating custom windows inside Immunity. For example, we can create a table that will be displayed in a separate window and implement logic for our script by invoking the following method “imm.createTable('table title,['columnName1','columnName2'])”. In general, it is very easy to extend the debugger by writing Python scripts. The only disadvantage is that, there is not much information online and the API documentation can only be read through the debugger itself.

**IDA**

IDA is a powerful debugger that is available on most popular operating systems. According to IDA official website, IDA’s cross-platform debugging includes many characteristics including instant debugging, connection to local and remote processes, and support for 64 bits systems. In addition to that, it is possible to debug binaries for different platforms such as Windows, Linux, iOS, and Dalvik. However, there is room for improvement. IDA allows developing extensions in a form of plugins for the debugger to automate program analysis. Generally speaking, there are two categories of plugins: scriptable and native.

Scriptable plugins are easier to implement than native plugins. Scripts can be written in either IDC (C-like language) or Python. IDC scripts require static “PLUGIN\_ENTRY()” function which return an instance of an object, which implementation is inside of a class declared in the same file (IDA PLUG-IN WRITING IN C/C++, 2018, p.5). The plugin class must have the following:

* Default constructor
* Init() method
* Run(args) method
* Term() method

In addition to that, the file must include “idc.idc” module. The default constructor is used to set the basic properties of the plugin: flags, comment, help, name, and a hotkey to invoke the plugin. Init, run, and term functions are callbacks. More specifically, init function is used to initialize the plugin. Run function is used for plugin’s invocation. Finally, term function is used for termination. A plugin can be started using either a hotkey directly from IDA or programmatically usig “run\_plugin()” function inside “/SDK/loader.hhp” file. Plugin itself must be placed inside “/IDADirectory/Plugins/” folder. Python scripts must have exactly the same structure. However, Using python requires installation of IDAPython.

Native plugins can be written in either C or C++ language and use IDA SDK. The process of writing native scripts is more complex and covered in Steve Micallef’s book called IDA Plug-in Writing in C/C++ (IDA PLUG-IN WRITING IN C/C++, 2018, pp.1-20). Developing native plugins, it is necessary to create Makefile and put plugin.script (.script file from sdk/plugins directory) file into the directory of the plugin. The structure of the plugin is similar to a structure of a scriptable plugins. However, it is required to include the following files: “ida.hpp”, “idp.hpp”, and “loader.hpp”. Finally, the compiled plugin must have “.plw” extension if used on Windows and “.plx” extention on Linux. The copiled file must be placed in “/Plugins/” directory.

**BinNavi**

BinNavi is a binary analysis IDE that allows performing different actions on binary files including inspection, navigation, and annotation of control-flow graphs (Zynamics.com, 2018). BinNavi is developed by Google using Java programming language. The primary advantage of this IDE over other tools is that it can be easily extended using scripts and plugins. Moreover, there are many examples and a well-documented API.

According to BinNavi’s documentation, writing scripts can be done in two different modes (Zynamics.com, 2018). First mode involves usage of a built-in scripting dialog that is accessible directly from BinNavi. It is important to know that writing scripts to analyze binaries involves using the most important object called navi. navi object is used to access BinNavi and all its functionality. This object is already pre-defined in the scripting dialog. In addition to that, the following objects are also pre-defined for usage simplicity:

* dbs (an object used to access database)
* cdb (an object that has a reference to a database where a control-flow graph is stored)
* cg (an object used to access elements of the control-flow graph)
* cf (a reference to an object that deals with window frame representation)

Standalone scripts should be written in Jyphon language. However, it is also possible to use Python. When we start writing scripts in Jyphon, we have to give the script a certain structure. First, we need to import “sys” and include a path to the BinNavi API. It can be done in the following way: sys.path.append(‘/path/binnavi.jar’). Importing of the StandAlone utility is also required: “from com.google.security.zynamics.binnavi.API.plugins import StandAlone” (Zynamics.com, 2018). Finally, we can defile a skeleton for our standalone script following the steps described on BinNavi’s website:

* define plugin interface object (pi) using “StandAlone.getPluginInterface()”
* define database manager object (dbm) using “pi.databaseManager”
* initialize a new database using “addDatabase” contructor
* retrieve our database (db) using “dbm.databases[0]”
* connect and load the database using “connect()” and “load()” functions.

The process of writing scripts in Python is very similar. However, we must redirect output of the Python script to the scripting dialog by using “sys.stdout = SCRIPT\_CONSOLE” command at the begging of the script. Any Script must end with “sys.exit(0)” command. Then the script must be place in any other directory but Scripts.

Writing plugins is not a very hard. However, it is a tricky process to make plugins work. There are two general approaches to extending BinNavi using plugins. The most generic approach is to write plugins in Java. Source code of a plugin must be either class or jar files. Any plugin must be placed inside the “/binnavi/plugins” directory. Class plugins must implement at least one Plugin interface of the API. Jar files have to define the main class in a manifest file. In addition to that, the main file has to contain an implementation of IPLuginIterface. Custom plugins are loaded automatically when BinNavi starts up.

BinNavi’s documentation states that plugins written in scripting languages must have the following:

* same extension as regular scripts
* implement BinNavi Plugin interfaces
* access “navi” object of type “PluginInterface”

In addition to that plugins must have two classes “MessageAction(AbstartAction)” and “PluginName(IMainWindowMenuPlugin)”. “MessageAction” class has two implement two methods. The first method is a constructor that has the following form “\_\_init\_\_(self, pi)”. The second method is called “actionperformed” that accepts two parameters “self” ad “e”. “PluginName” class has to have the following methods:

* getName(self)
* getGuid(self)
* getDescription(self)
* init(self, pi)
* unload(self)
* extendPluginMenu(self)

Finally, the plugin file must add itself to BinNavi’s plugin registry using “navi.getPluginRegistry().addPlugin(customPluginInstance)” line.

**II INTRODUCTION**

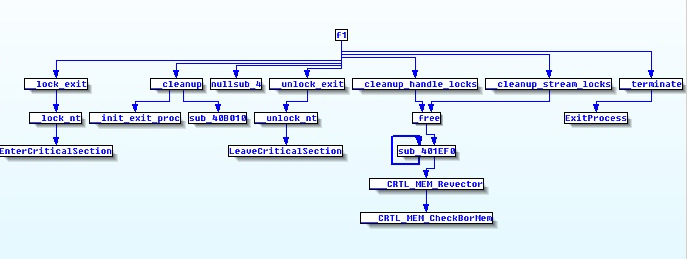
Reverse engineering and static program analysis are very tedious tasks because it involves working with disassembly of a program directly. The disassembly can be represented in either hex format or assembly language. Looking at assembly instructions and trying to build a big picture about the program and is a time consuming task. Moreover, many reverse engineers spend most of the time trying to recognize algorithms used in a program because assembly is a low-level programming language that is not easily readable by humans. However, many tools and disassemblers have been created in order to simplify the process of reverse engineering. Most of these tools are able to provide various types of information including disassembly of a file, hex view of all bytes, and different types of visualizations such as call and control-flow graphs. Examples of some popular tools are IDA pro, Ollydbg, Hopper, Radare2, and angr. First three tools are stand alone software suitable for both reverse engineering and debugging, while last two provide their own APIs and usually used in scripts for program analysis tasks and scientific researches.

In general, all these tools provide a way to visualize information about a binary by displaying different types of graphs because it is one of the most common ways to simplify the process of program analysis. According to a research survey done by Rainer Koschke, graphs are identified as the most often used kind of visualization (Rainer Koschke, 2000, pp.8-16). It is true because graphs are simple and powerful enough to express information about binaries. In other words, they help to generalize information from a disassembly (lower lever type of information) and bring it to a higher level of representation that is easier to understand.

**Previous work**

Existing reverse-engineering tools have limitations and to do not provide a way to generalize information about a program’s structure in an easy fashion. Our idea is to create a reverse engineering tool that is an improvement of existing ways to visualize information about binaries. The tool will be used for static program analysis. Its goal is to generalize information about a given binary and displaying that information in a form of a graph, so reverse engineers can get a sense how program is structured and how it operates. The goal of our tool is to make the process of static program analysis faster. This section is structured in the following way. First, we describe existing tools along with their functionalities, identifying their disadvantages and limitations. Then, we describe our idea that solves the described problem. Finally, we describe challenges and opportunities for our project.

First we take a look at IDA pro tool. IDA provides a wide variety of information about binaries including list of procedures, machine instructions, hex view of a program, list of used strings, etc. In reverse engineering it is important to work with both machine instructions and control-flow graphs. IDA Pro provides a good visualization of control flow graph by displaying control flow dependency between blocks using edges of different colors along with machine instructions for each basic block. However, there is no interaction between the user and control flow graph. In addition to that, switching between control-flow graphs for each procedure is not very intuitive. In addition to that, if we want to look at machine instructions of one function and a CFG of another procedure, we have to switch between tabs back and forth. It is not convenient and requires memorizing certain type of information at any given point in time. This is may not be suitable for everyone. Also, there is no built-in way to view call graphs. However, we were able to find a plugin able to generate call graphs. The picture below shows a generated call graph by the plugin:



As we can see the call graph allows figuring out the structure of a program by looking at the relationship among functions. However, there is no any interaction allowed with the call graph, which is a primary disadvantage. Ollydbg tool provides similar functionalities and preserves same disadvantages.

Another interesting reverse engineering tool is called Hopper. In general, Hopper does a better job allowing interaction with control flow graphs and switching between different kinds of views. However, Hopper provides only one window for viewing information. It means that it is also may be a disadvantage in some cases. More precisely, it is impossible to view disassembly of a file and a CFG for a given function at the same time. In addition to that, Hopper does not provide a way to view call graphs, which are useful for understanding overall structure of a program.

Next tool we look at is Radare2. It is a command line utility that can be used for different reverse engineering purposes. Moreover, it is possible to write powerful scripts and programs using Radare API and r2pipes. Radare provides a way to display a control-flow graph. However, since it is a command-line utility, it will require issuing different commands to view small portions of information about a binary, which uncomfortable and will require memorizing certain details that we are working with during an analysis. Also, it is possible to recover a call graph by sing “agC” command that gives the following output:

“digraph code {

graph [bgcolor=white fontname="Courier" splines="ortho"];

node [fillcolor=gray style=filled shape=box];

edge [arrowhead="normal"]; …

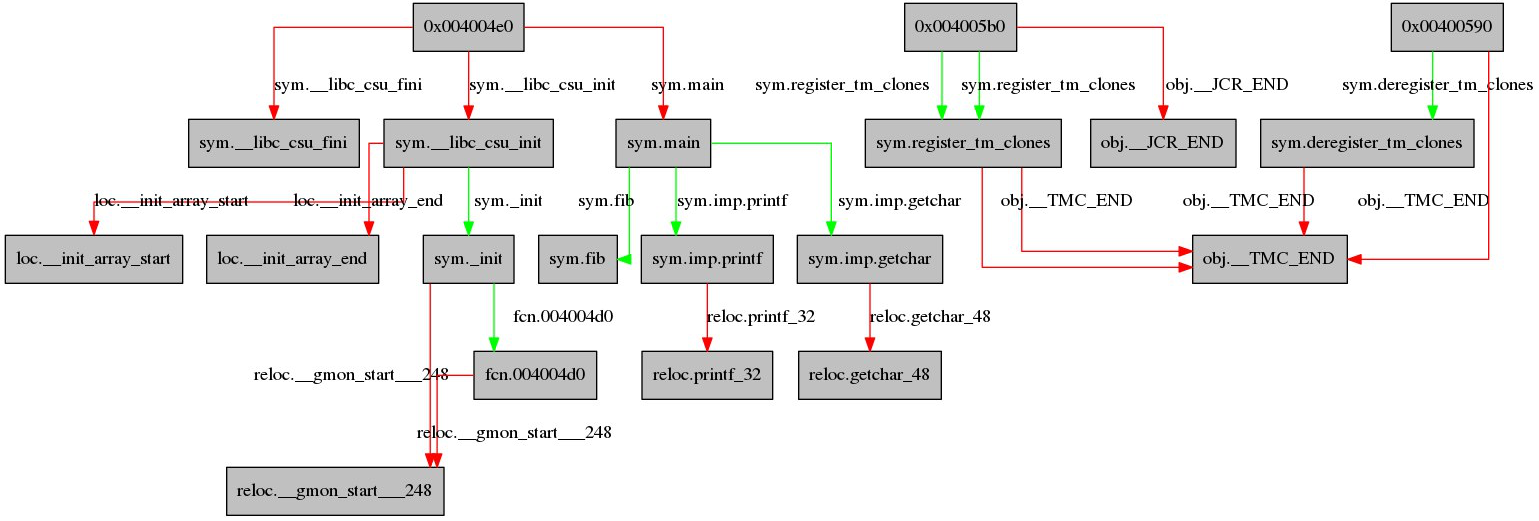
"0x004006df" [label="sym.main" URL="sym.main/0x004006df"]; …

"0x00601048" [label="obj.\_\_TMC\_END" URL="obj.\_\_TMC\_END/0x00601048"];

"0x00400550" -> "0x00601048" [label="obj.\_\_TMC\_END" color="red" URL="obj.\_\_TMC\_END/0x00601048"];

"0x00601048" [label="obj.\_\_TMC\_END" URL="obj.\_\_TMC\_END/0x00601048"]; … }”.

However, it is hard to extract useful information from this output. We can redirect the output to a .dot file and then use a “!!dot” utility to produce the following .png image:



This format can give us an idea about a program’s structure and the relationship between procedures. However the amount of actions needed to visualize this information is not appropriate. We also face the same problem of inconvenience when navigating between .png images and the command line.

Angr gives us an ability to recover a control-flow graph in the following format:

“{4195552L: <Function \_init (0x4004e0)>,

4195600L: <Function plt.puts (0x400510)>,

4195616L: <Function plt.printf (0x400520)>,

4195632L: <Function plt.read (0x400530)>,

4195648L: <Function plt.\_\_libc\_start\_main (0x400540)>, … }”.

This representation is easier to understand. However, we face same issues if we were using Radare2.

All of the mentioned tools carry one common problem. They do not provide an easy way to get information about general structure of a program and relationship among functions along with viewing disassembly of a program at the same time. It is important to display just enough information about a binary on the same screen so reverse engineers can focus on certain things without a need to memorize details about a program when they switch between windows to extract more information about binaries. In addition to that, it would be useful to have an ability to interact with graphs.

**Our approach**

Our idea is to create a simple but useful reverse engineering tool that will eliminate described problems and simplify process of static program analysis. Our project focuses on call graph visualizations. “A call graph is a directed graph depicting the calling relationships between the procedures of the program, i.e. which procedures contain calls to which other procedures” (Malware and Machine Learning, 2015). The visualized call graph looks similar to a presented example of a .png image. It allows viewing high-level structure of a program and better understanding relationships between functions. This solves a problem of readability because there is no need to spend time to go through a disassembled code in order to extract general information about the binary. In addition to that, the tool does not multiple tabs, which may affect performance of reverse engineers when the switch between different representations. Call graph visualizations in reverse engineering tools are not a novel approach. However, all previous work does not allow direct interaction with graphs. In our project, we implement listeners for each node of the call graph. A double click triggers an algorithm that will display disassembly of a chosen procedure in a view holder next to the call graph. This will allow working with two things at the same time and have an ability to scroll through the disassembly and the call graph. The displayed graph will have useful features implemented in other tools. More precisely, each edge in the graph is colored depending on its properties (conditional paths, paths always taken, etc.). Call graphs can get extremely large. It may be a challenge to figure out the relationship between functions on a large graph. We solve this problem by coloring edges of the graph. Edges of green color represent a function call, which is made to a function defined in the binary. Red colored edges indicate a call to a function outside the binary (library and system calls).

We identified two primary challenges during the research. First of all, it may be a problem to recover an accurate call graph from a binary. We are using Radare2 for this purpose, which is able to provide an accurate json-like representation for a graph including all necessary information for us. Another challenge is properly visualizing graph. That is why we chose to use Python programming language. It has two powerful libraries for drawing graphs “graphviz” and “xdot”. The most time consuming part of the project is parsing the dot output from Radare2 and drawing the graph on the screen.

We realize that our project is a simple reverse engineering tool and may not be used as the only tool during reverse engineering process. However, it is useful for cases when we want to see a big picture about a program’s structure along with extracting some details from the binary. The simplicity of the tool’s UI will provide an overview of a program along with some functionality as a starting point during the process of reverse engineering without a need to focus on many different things at the same time.

**III DESIGN**

**Use cases**

We name our reverse engineering tool REEZ (Reverse Engineering is Easy). REEZ is primarily designed to simplify the process of reverse engineering and make the process of program analysis easier by providing high-level structure of the program in a form of call graph. More precisely, the tool helps reverse engineering to learn the structure of binaries without the need to manually go over the disassembly and trying to figure out the flow of the program. This is especially helpful during analysis of unknown binaries. Manual analysis is a tedious task because it requires reading large amounts of assembly code and trying to properly reconstruct the relationship among functions, which can be very complex. In addition to that, our software can be helpful during static malware analysis. The tool allows viewing all library calls that are made by a given program. Analyzing the flow of the program and knowing potentially malicious system calls (opening files, manipulating buffers, and network usage) can give an idea the vulnerable parts of the code and navigate directly to the part of a program using the call graph for further analysis. It is possible because our tool allows viewing disassembly of functions included in the call graph. Summarizing the information, the tool can be used as a starting point during program analysis to understand the high-level structure of the program. In addition to that, it is possible to analyze specific parts of the binary by selecting the most important parts of the binary and reading the disassembly of selected functions. Combining both of these scenarios, the tool can be very useful when reverse engineers are trying to understand algorithms used in the program. It will not be a hard task assuming that a program is following proper styling conventions. Reverse engineers can get an idea what algorithms are used by tracking the flow of function calls and associating them with names of procedures and analyzing their disassembly if needed.

Our software is intended to have a simple user-friendly design displaying only important information without making the user to look at many different types of information at the same time. Presented use cases show that both professional reverse engineers and beginners can use the tool. Primarily, the tool is very useful for people who do not have much experience in analyzing binaries but wish to learn more about structures of various programs.

**High-level design**

This section provides a high-level design of the program by introducing libraries and techniques used during the implementation process. In addition to that, we provide a user-software interaction diagram to show the functionality of the tool and how users interact with it.

The software is implemented using Python programming language. This language is chosen because it is very flexible and provides the functionality we need for our purposes. In addition to that, the development process of the software will take less time considering time constrains for this project. Finally, there are all available libraries that satisfy our needs. First of, Radare2 provides r2pipe library for Python that allows us to get a call graph representation of binaries in the dot format. qdot tool is very helpful for us to parse the .dot file and retrieve all required information for visualizing the call graph (Qdot, 2013). Finally, we can use PyQT that is a Python binding for a powerful QT library that allows us creating GUI for our application.

Figure 1 shown below presents all possible cases of user interaction with the software. In the diagram, each rectangle represents an action taken by a user, while ellipses represent set of actions taken by the program.

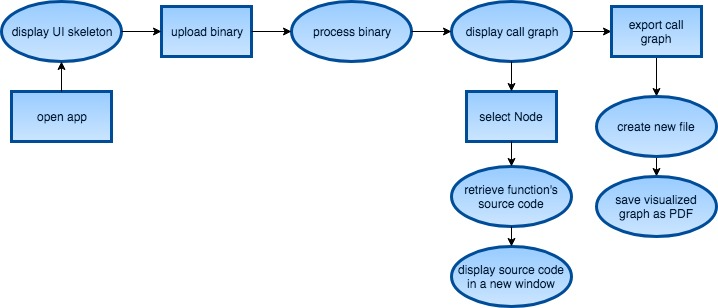


Figure 1: User-software interaction

The user interacts with the software as following. Whenever a user opens the app, the UI skeleton of REEZ is displayed. It includes general UI components such as menu items and a toolbar. Then a user can load a binary. The binary is analyzed and processed to display a call graph. Also, view holders for the call graph and disassembly are initialized. The view holders are separated by a splitter, which allows resizing the areas used for visualizations. Our tool gives an option to navigate through the call graph by zooming in/out and scrolling to different regions of the graph. Finally, users are able to interact with the call graph by double clicking on any node of the call graph to view the disassembly of the selected function, which is displayed in a disassembly’s view holder that takes approximately 1/2 of the screen’s width. It allows the user to navigate through the call graph and have an ability to view assembly code at the same time.

**Data flow**

This section presents a data flow diagram of our software to describe the data input and outputs between the most important components of the application and describe details of the design approach. The diagram is divided into two parts, which is shown on figure 2. These parts present data flow inside the main components of the application. The first part of the diagram described inputs and outputs of the main functionality – call graph visualization (Call Graph Visualization DFD on figure 2). It contains 4 levels, where each level is broken down to nested levels to describe the data flow between parts of the component in more details.

Level 1 shows the general overview of the data flow between the user and our tool. A loaded binary is an input for our program. REEZ visualizes a call graph using the input. In addition to that, the user can export the visualized graph to a pdf file.

Level 2 shows two main components of the tool and associated data flow. User’s input is passed to Radare2 using r2pipe in order to retrieve the call graph representation in the dot format. Then it passes the output to a processing unit called qdot, which renders the dot representation for further visualizations.

We explain why we chose qdot to be a part of our project. One of the most challenging tasks of developing this reverse engineering tool is to properly parse call graph, which is represented in dot format. In addition to that, we need to traverse through the parsed file and draw the call graph in order to allow users to interact with it. We embed an existing open source dot viewer called qdot in our application. However, we have to modify its functionality to meet our goals. We chose qdot because it is the only solution that can be integrated with PyQt and displays call graphs in an appropriate for us fashion. We researched other libraries that can be used for visualization of graphs. These libraries are NetworkX, Pydot, and pygraphviz. NetworkX is a powerful able to parse and display graphs from a dot file. However, it is extremely hard to take over some functionality of the library. In other words, we can parse the dot file but the traversing through a built graph data structure is very complex. Even finding a root node of the graph is a challenging task. NetworkX graphs can be visualized using matplotlib library. However, the result image is displayed as a network graph in an unorganized fashion. Most importantly, matplotlib has very limited abilities to make the graph interactive using Python and PyQt. Both Pydot and pygraphviz do a good job in parsing the dot files. However, the only way to visualize the graph is to write it to either png or pdf file. We will be using graphviz library in order to allow exporting call graphs to a file inside our application. According to Anthony Liu, Qdot is based on xdot and implements its own parser, lexer, and tokenizer (Qdot, 2013). Finally, Qdot is essentially an implementation of graphviz that uses primitive PyQt features to visualize information and draw objects.

Qdot consists of two major parts: parsing engine and visualization unit. The dot file received from Radare2 is parsed using XDot parser, which is a part of qdot. The parser produces a data structure that contains all information about nodes and their relationship with each other. Finally, the data structure is passed to the visualization unit, which is responsible for displaying the interactive graph to the user. Visualization unit is broken down to two units in level 4 of the diagram. The call graph is always visualized and displayed in a form of interactive graph with a help of PyQT and qdot. Qdot draws every element of the graph’s data structure. If a user wants to export the call graph, then the graph data structure passed to a block of code that uses graphviz library to handle this type of task.

Another main component of our software deals with displaying a disassembly of a selected function.

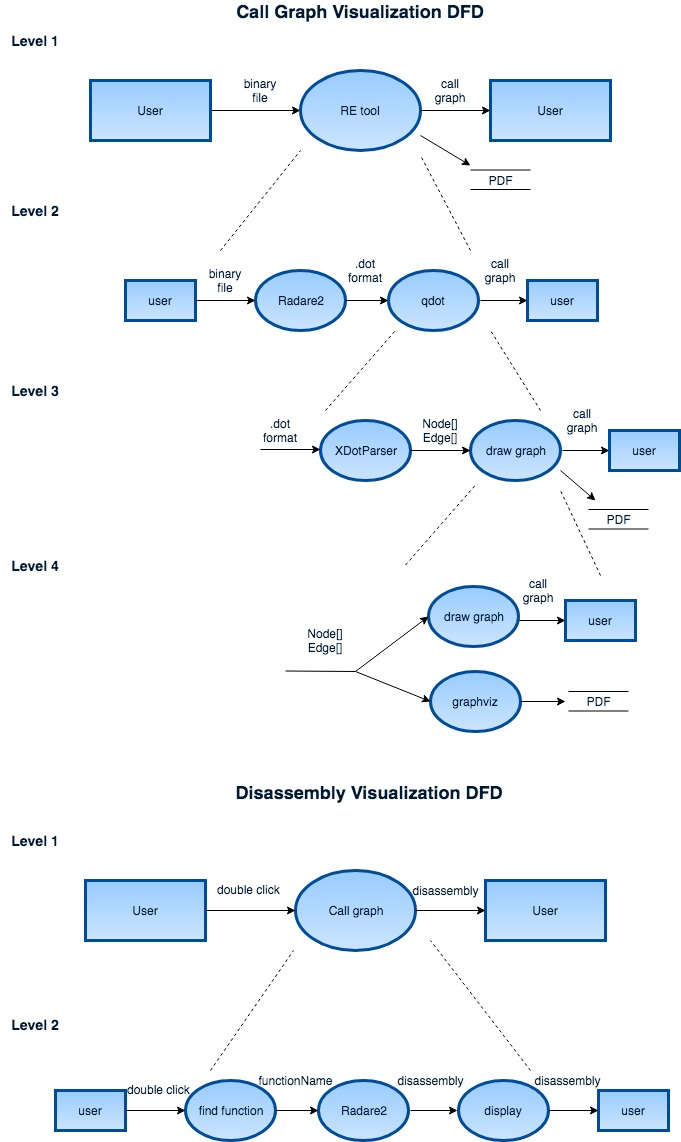


Figure 2: Data flow diagram

When a user double clicks on a certain node of the graph, the coordinates are passed to a method that is responsible for associating coordinates of the click with coordinates of the selected node. If the selected function is found, then the information is passed to Radare2 to get the disassembly of the function, which is displayed in the second view holder the help of PyQT library.

**IV IMPLEMENTATION**

In this section, we describe the structure of REEZ in details. In addition to that, we present a class UML diagram to show interaction between main components and how the app is implemented. Finally, we describe the structure of the qdot open-source solution that is used in our project because we need to modify its functionality to meet our goals.

As it was said before, we are using Python programming language to implement the reverse engineering tool. The GUI of the application is built using PyQT4 library since it allows building applications suitable for our needs. This allows us to create cross-platform software that can be installed on UNIX operating systems. The only thing is needed from the end-user is to install all required packages and libraries to be able to compile our application. All operations with binary files are handled using r2pipes to execute appropriate Radare2 commands to analyze the binary file and retrieve information about a given program.

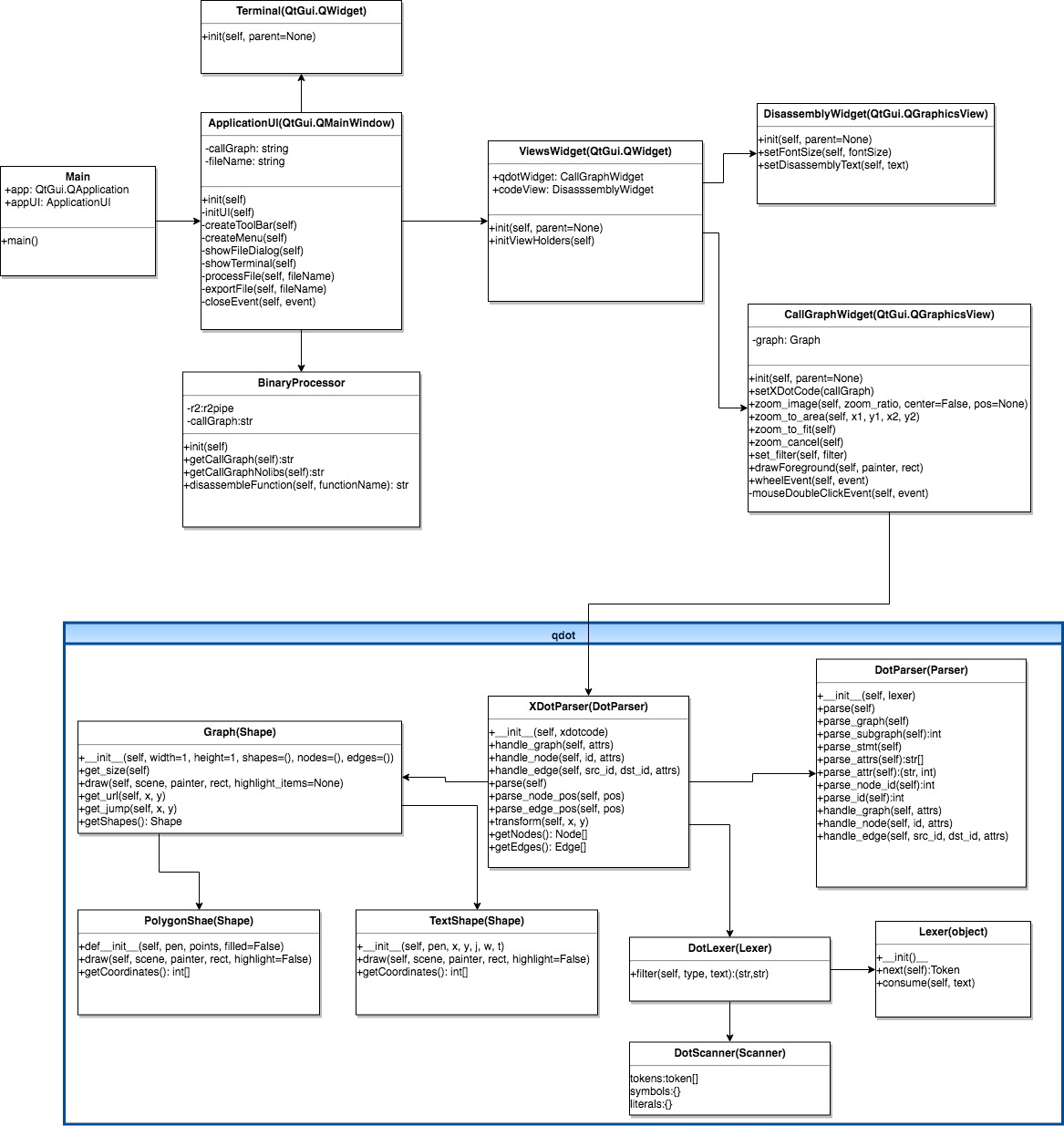


Figure 3. UML Diagram

Figure 3 presents UML class diagram of our application and describes relationship between objects. Main class is used to initialize the GUI of the application by creating instance of QApplication and a custom class ApplicationUI, which extends QMainWindow class from PyQt4 library and contains the implementation of GUI for the main window. Different actions such as importing binaries and exporting visualized graphs are handled using corresponding functions inside ApplciationUI class. When user loads a binary, we initialize main view holder using ViewsWidget class that extends PyQt’s QWidget class. ViewsWidget is a controller of the main view that contains two view areas of the visualized information. Initialization of these two areas is also performed inside ViewsWidget class. First view area is used to hold visualization of a call graph. Its implementation is located inside CallGraphWidget class. CallGraphWidget class is used to set up elements of user interface to handle interaction with the graph. We create several buttons to allow scrolling through and zooming parts of the graph. Corresponding functions serve as a connection with the main holder of the call graph that is an instance of QGraphicsScene class that belongs to CallGraphWidget widget. In addition to that, we do an initial process of binary file here by passing the path to the binary file to a BinaryProcessor object that analyzes the binary and returns a call graph in dot format as a string. This string is then passed to the modified version of qdot for further parsing and drawing the graph inside the QGraphicsScene. Most importantly, CallGraphWidget implements DoubleClickListener method that provides interactivity feature with the call graph. When a user double clicks on a node, the method finds the selected function by comparing coordinates of the click with coordinates of the selected function. The challenge of this task is that we need to map coordinates of a mouse pointer to the coordinate system of the QGaphicsScene. This is not a very effective approach to implement click listeners. We tried to re-implement qdot’s functionality and take advantage of visualizing nodes of the graph using QRectangle class. This would allow us to associate click listeners directly with the object on QGraphicsScene. However, qdot’s implementation is very complicated and we failed to make this change within given time period for the project. We decided to use less efficient approach that works. In section V, we show that this approach is still pretty efficient and does not introduce performance overhead. The name of the selected function is retrieved using qdot and passed to BinaryProcessor to obtain disassembly of the function using disassembleFunction procedure. CallGraphWidget calls DisassemblyWidget’s method responsible for setting up a text view of the disassembly. The retrieved disassembly is passed to setDisassemblyText procedure, which visualizes the assembly code of a selected function.

DisassemblyWidget class is initialized inside ViewsWidget class. DisassemblyWidget is an extension of PyQt’s QGraphicsView class. This class implements functionality for visualizing assembly code of selec ted functions. The init method is used to set up main UI components of the disassembly view. Assembly code is displayed inside a text view. We place the text view inside a QScrollArea that is located inside QGroupBox in order to allow customizations of the view. QGroupBox uses QFormLayout to allow flexible resizing of the view and changing properties of the scroll area whenever a user modifies its properties inside an instance of ViewsWidget object.

One of the most important parts of our reverse engineering tool is a dot parser. As mentioned above, we will modify qdot to achieve our goals. Source code of qdot is about 2000 lines and contains 15 classes. We carefully analyzed the code and determined how this code can be modified. The structure of most important classes is depicted inside the UMl diagram shown on Figure 3. Further, we describe the organization of qdot’s main components and how we modify them.

Qdot contains an implementation of XDotParser class used to handle parsing and drawing of all components of the dot graph. XdotParser uses DotParser and DotLexer classes to parse the file and retrieve nodes and edges of the file along with corresponding attributes. This information is collected using Graph object that contains information about nodes and their relationships (edges). However, Graph object is not used in reality. Instead, we have to modify functionality of DotParser and XDotParser in order to get access to parsed nodes and edges of the graph. Qdot is intended to be used in a different context and accept .dot files as input. We get rid of this functionality and change it in a way that is suitable for our purposes. More precisely, we change qdot’s base class so it can be embedded inside QGraphicsScene. Finally, we change XDotParser structure in order to parse dot strings directly. We also modify PolygonShape and TextShape classes to allow us retrieve information about their location on the graph along with the corresponding text labels. Finally, we modify these classes in order to achieve a better appearance of the graph.

**V EVALUATION**

In this section we present functionality tests and performance evaluation of our reverse engineering tool. We evaluate REEZ using the following criteria: correctness, performance, and time and space complexities. We tested our application on Linux Ubuntu 16.04 VM with 2GB of RAM running on a host, which has Intel i5 dual-core processor and 16GB of RAM. First we describe the evaluation of correctness of REEZ.

In our case, correctness describes functional correctness of input-output behavior of algorithms used inside our tool. In other words, we need to make sure that each component of the software works as intended and produces correct output. We divide this evaluation criterion into two parts. First part of testing is associated with the UI skeleton. More precisely, we test general functionality of the app’s UI, which is not related to graph visualization itself. The second part involves testing call graph visualization since it is the most complex and important part of the reverse engineering tool. In addition to that, we also test interactive features of the graph.

General components of the UI are toolbar, menu, and resizable view holders for a call graph and disassembly of functions. We test each component 15 times and count the number of correctly produced outputs. Table 1 summarizes test results of general UI components. In addition to that, it is important to mention that toolbar contains 5 UI elements: 4 buttons and 1 dropdown list. Buttons are associated with corresponding actions: exit application, load binary, export call graph, and open terminal. “Exit application” button triggers showing a pop up dialog that asks user if the application needs to be closed. We also include the output of buttons inside the pop up dialog in the evaluation of the “exit application” button because these actions are associated with semantics of the toolbar element. “Load binary” button opens a dialog that allows users to select a binary for further analysis. Testing “export call graph” button involves checking if the visualized graph is successfully saved to a pdf file in a right directory that is provided by a user using file dialog. “Open terminal” is responsible for opening a terminal inside our application. So, we test if this action is correctly performed when the button is clicked. Finally, the dropdown list allows a user to select a font size of a text view, which holds disassembly of a selected function. In our evaluation we say that toolbar works correctly if all elements of a toolbar produce a right output. We follow the same logic during testing the menu items of our tool. As for view holder, we test if views are displayed as intended. In other words, we check if view holders for a call graph and disassembly are initialized and do not trigger any errors when resized.

|  |  |  |  |
| --- | --- | --- | --- |
| **UI component** | **Number of tests** | **Number of correct outputs** | **Percentage of correct outputs** |
| Toolbar | 15 | 15 | 100% |
| Menu | 15 | 15 | 100% |
| View Holder | 15 | 15 | 100% |

Table 1. Evaluation of general UI components

Table 1 shows that all UI components have correct input-output behavior in all test cases. Since toolbar has 5 elements, we perform 75 tests total but combine 5 tests (one for each toolbar element) into one. The test results satisfy our requirements. We want general UI components to work without any errors because they represent a skeleton of our application and are crucial for providing the main functionality. Main components of our application are call graph visualization and disassembly view of a selected function. It means we need to test the correctness of a visualized graph obtained from a binary. In other words we check if all nodes and edges of the graph are displayed correctly with appropriate labels. We visualize call graphs of 10 different binaries and compare the results with graphviz’s dot utility because qdot used in our application is an implementation of graphviz in the context of PyQt. In addition to that, we test interactive feature of the call graph. We check if a displayed disassembly of selected functions on the graph is correct by comparing the output of our application with the output produced by Radare2. We select 10 different functions (nodes of the call graph) from 10 different visualized graphs and compare the displayed disassembly with an output of Radare2’s command used to disassemble procedures by manually typing these commands with a corresponding argument that is a function’s name. We say the output of our application is correct if displayed disassembly of a selected function is identical to an output of Radare2 for the same function. Finally, we perform testing of exported graph visualizations in the same way we test call graphs displayed inside REEZ. In other words, we compare the visualized graph exported to a PDF file with a visualization produced by graphviz’s dot utility. Test results of these three kinds of visualizations are presented in table 2. The table shows that call graph is visualized correctly in very test scenario. It means that all nodes and edges are correctly displayed with appropriate labels. The same is true for exported graphs. Finally, a selection of each function results in correct visualization of disassembly for that particular function without any errors and alternations. It means that interactive features of call graphs work as intended.

|  |  |  |  |
| --- | --- | --- | --- |
| **Visualization** | **Number of tests** | **Number of correct outputs** | **Percentage of correct outputs** |
| Interactive call graph | 10 | 10 | 100% |
| Exported call graph | 10 | 10 | 100% |
| Disassembly | 100 | 100 | 100% |

Table 2. Correctness of visualizations

Performance evaluation is an important part of our research. Most importantly, we measure how much time it takes for our program to provide main functionalities such as call graph visualization and displaying disassembly. We use Python’s “time” package for this purpose. The package measures how much time it takes for certain parts of code to run including all interruptions performed by the kernel. It means we are measuring the real time taken by a certain task. We chose this way of evaluating the performance because the results describe what users will experience in reality using our tool. First, we evaluate performance of our algorithm used to visualize a call graph. We start the timer right after user selects binary, which need to be processed. We stop the timer right after the process of call graph visualization is completed. We take measures using binaries of different sizes. More precisely, there are three scenarios in evaluating this part of our program. First scenario deals with measuring time taken to visualize graphs of a small size (10-25 nodes). We use medium size graphs for the second scenario where number of nodes in a graph is between 26 and 60 nodes. Finally, the last scenario deals with measuring time taken to visualize larger graphs that have 61-100 nodes. Test results of these scenarios are shown in table 3.

|  |  |  |  |
| --- | --- | --- | --- |
| **Call graph size (nodes)** | **Average time (seconds)** | **Best time (seconds)** | **Worst time (seconds)** |
| 10-25 | 1.229 | 0.301 | 3.245 |
| 26-60 | 1.509 | 0.464 | 3.533 |
| 61-100 | 1.647 | 0.572 | 3.746 |

Table 3. Time taken to visualize call graphs.

From table 3 we can see that on average it takes between 1.229 and 1.647 seconds to visualize call graphs of different sizes. We think it is a pretty good result since it takes less than two seconds to display a call graph on the screen considering the amount of computations performed to make the visualization appear on the screen. More precisely, Radare2 must process the loaded binary in order to obtain the graph representation in dot format, which is a time consuming process. Then, the dot representation of a call graph must be parsed. Parsing has time complexity O(n3) and space complexity O(n2). Qdot draws all elements on the QScene view holder using primitive shapes and QPen objects, which is not a very efficient approach. Drawing has time and space complexity O(n2). Finally, the view holders must be created along with an additional toolbar used to change appearance of the call graph. Considering all these subtasks average time taken to visualize call graphs is sufficient. Another consideration is that we observe a big difference between best and worst time taken to display call graphs. During our research we observed that roughly 4 out of 10 visualizations take 10 times longer than normal even if the same binary is used to obtain a corresponding call graph. We suppose it happens because of kernel interruptions. Also, it is important to understand that performance evaluation was conducted on a VM that has limited resources.

Finally, we evaluate performance of a component responsible for displaying disassembly of selected functions. Again, we use Python’s “time” package to measure time taken by the algorithm used to provide visualizations of a disassembled functions. We start our timer when double click on the call graph is observed. We stop the timer when the disassembly of a selected function is displayed. The results of this evaluation are shown in table 4. We break down the performance evaluation into two parts. First, we measure how long it takes to visualize disassembly that have 1-50 lines of assembly code. The second scenario deals with larger portions of assembly code (51-150 lines).

|  |  |  |  |
| --- | --- | --- | --- |
| **Disassembly size (lines of code)** | **Average time (seconds)** | **Best time (seconds)** | **Worst time (seconds)** |
| 1-50 | 0.0036 | 0.0004 | 0.0050 |
| 51-150 | 0.0146 | 0.0106 | 0.0381 |

Table 4. Time taken to visualize assembly code of a selected function.

The algorithm used for displaying disassembly of a selected function can be broken down into several subtasks. However, we are most interested in time and space complexities of determining what function a user has selected. Time and space complexities for this part of algorithm are O(n2) and O(n) respectively. The algorithm was tested on a graph of a large size defined in table 3. Table 4 shows that the worst-case scenario takes 0.0381 seconds to display assembly code of a function. We think it is sufficient since a user will not notice any delays after selecting a function on a call graph.

**VI FUTURE WORK**

Current state of the implemented reverse engineering tool is far away from being perfect. There is a lot of room for improvement. Future changes are associated with the main functionality of the application – call graph visualization. Arun Lakhotia states that a call graph is generally used to show which functions can call other functions, not to only show that a function did call another function in a single execution path (Malware and Machine Learning, 2015, p.24). Using this property of call graphs, we would like to add labels to graph’s edges, which represent an order in which functions are called in the static context. We also plan to add a dynamic analysis feature to our application. In this case, edges will have corresponding labels that indicate the order in which functions were actually called. Another useful feature we plan to add is an ability to choose different representations of disassembly of selected functions. More precisely, we plan to add the following representations: hex view, CFG, and binary view. Finally, we plan to add an ability to hide library and system calls on the visualized call graph. This will be a very useful feature when users perform static program analysis on large binaries.

**VII CONCLUSION**

We presented a new reverse engineering tool for static program analysis. The tool focuses on call graph visualizations because this type of visualization makes the process of program analysis easier and faster by provided a generalized overview of a program’s structure. In addition to that, we presented a new feature that is not implemented in previous approaches. More precisely, we implement an interactive call graph, which allows retrieving assembly code of functions displayed on the graph by selecting corresponding node on the graph. Implemented user interface is simple but powerful. It allows reverse engineering to work with two type of information at the same time in the same window without need to switch between representations. Moreover, the evaluation of REEZ show it works correct and fast. The current state of the project may not be sufficient enough to use REEZ as a stand-alone disassembler but it can be very useful in conjunction with other reverse engineering tools and frameworks.

**References**

Docs.angr.io. (2018). Introduction · angr Documentation. [online] Available at: https://docs.angr.io/ [Accessed 12 Feb. 2018].

IDA PLUG-IN WRITING IN C/C++. (2018). [ebook] Steve Micallef, pp.1-20. Available at: http://www.openrce.org/reference\_library/files/ida/idapw.pdf [Accessed 12 Feb. 2018].

Koschke, R. (2018). Software visualization in software maintenance and reverse engineering. [ebook] pp.8-16. Available at: http://www.sdml.cs.kent.edu/library/Koschke02a.pdf [Accessed 26 Feb. 2018].

Malware and Machine Learning. (2018). Springer International Publishing Switzerland: Arun Lakhotia, p.24.

Radare.gitbooks.io. (2018). Scripting · Radare2 Book. [online] Available at: https://radare.gitbooks.io/radare2book/content/scripting/intro.html [Accessed 12 Feb. 2018].

team, T. (2018). Extending r2 with new plugins · The Official Radare Blog. [online] Radare.today. Available at: http://radare.today/posts/extending-r2-with-new-plugins/ [Accessed 12 Feb. 2018].

The Beginner's Guide to IDAPython. (2018). [ebook] Alexander Hanar, pp.5-15. Available at: https://leanpub.com/IDAPython-Book [Accessed 12 Feb. 2018].

Qdot, GitHub. (2013). pakls/qdot. [online] Available at: https://github.com/pakls/qdot [Accessed 25 Mar. 2018].

Website & Mobile Application Development, Digital Marketing Agency, Delhi India. (2018). mkvirtualenv command not found - virtualenvwrapper. [online] Available at: https://techstricks.com/mkvirtualenv-command-not-found-virtualenvwrapper/ [Accessed 12 Feb. 2018].

Xdot, GitHub. (2018). jrfonseca/xdot.py. [online] Available at: https://github.com/jrfonseca/xdot.py [Accessed 25 Mar. 2018].

Zynamics.com. (2018). zynamics BinNavi 5.0 Manual - Using Scripts in BinNavi. [online] Available at: https://www.zynamics.com/binnavi/manual/html/scripting\_usage.htm [Accessed 12 Feb. 2018].

Zynamics.com. (2018). zynamics BinNavi 5.0 Manual - Writing BinNavi Plugins. [online] Available at: https://www.zynamics.com/binnavi/manual/html/plugins.htm [Accessed 12 Feb. 2018].