CS 6501 Project Report – Hoos′ Upto No Good

OMKAR BHAT (ODB6PZ), ZOYA YEPREM (ZY8EZ), and VIJAY LINGESH (VB7MZ)

Abstract

There are countless software applications in the kind of world we live in, with new ones being created every day. Also, most of them have their source code hidden which kind of leads to more work when trying to understand the specifics, algorithms, etc. of the said software. This is also where software reverse engineering tools come in. A reverse engineering tool is used to deconstruct a binary file to reveal knowledge such as the programs′ design, architecture or even to find vulnerabilities. There are a handful of tools out there that can be used for Reverse Engineering, however, we do not have a direct comparison between most such publicly available tools. Our goal in this project is to compare these Reverse Engineering tools. Specifically, we aim to analyze Ghidra, Angr and IDA Pro.

Each of these tools has their own implementation of how to go about with reverse engineering and the challenge here is identifying these systematic differences and bringing out a detailed report on which of these tools are better in various aspects. On analysis, we will come up with metrics to evaluate each of them and provide recommendations based on our results.

1 PROJECT DESCRIPTION

In this project, we took a look at reverse engineering applications namely IDA Pro, Ghidra and Angr. Each of these applications is a well-reputed software. However, comparisons are difficult for non-authors because of systematic differences. On the other hand, tool authors often do comparisons, but they are usually biased. Both because they might have been compensated to lean towards a certain tool, or if they are the authors of the tool, they may just promote their own developed application. So in this project, we aim to provide a fair comparison between these three tools. To do so, we considered a list of comparison metrics motivated by [1]. These metrics are as follow GUI - UI/UX, Control Flow Graphs, Supported language, Supported architecture and Fault Tolerance. This will help us better judge the strengths and weakness of each of software. To test this software we considered various programs written in multiple languages. We have a short medium and a large program in each language, this will be discussed further in a dedicated section for each tool.

2 BACKGROUND

Reverse engineering tools are useful in many ways. They are useful for software engineers and maintainers to understand the structure of software by analyzing the source code and represent them in a higher level of abstraction by means of control flow graphs and call graphs. Without the help of such tools, it is extremely difficult to build these for binary files of large complex software. Another major use for a reverse engineering tool is reconstruction or re-engineering where existing legacy software are analyzed from ground zero and potentials vulnerabilities and design flaws are identified and are re-designed. From software security to software maintenance, Reverse engineering is widely used to synthesize high levels of abstraction for complicated programs.
Reverse Engineering takes a series of phases in trying to deconstruct the source code from a binary.
First, the tool loads the binary and parses the compiled machine code the binary holds. This gives the tool basic information such as the architecture of the compiler, available functions, and entry points, isolate sections of written code from run-time initialization code added by the system and possibly even detect the compiler used to create the binary.

Next, the tool disassembles this machine code and builds its own intermediate representation(IR).
After IR is built, the tools try to identify the structuring of the data, variables, and functions. It also tries to figure out which part of the code is exception handlers and error handlers. For most tools the steps till the ones mentioned above are common and what or how the data from the IR is used to do control or flow analysis and in turn how the control flow is created and finally even the code or pseudo code generation, depends on the tool being used.

3 RELATED WORK

The work by Bellay et al. [1] is a comprehensive analysis of 4 RE tools published in the year 1997. They studied and evaluated Refine/C, Imagix4D, SNiFF+, and Rigi. They studied the capabilities of these tools by applying them to real-world embedded software systems. They listed down the benefits and shortcomings of these tools and assessed their applicability, usability, and extensibility. We will be working on similar lines for IDA Pro, Ghidra and Angr.

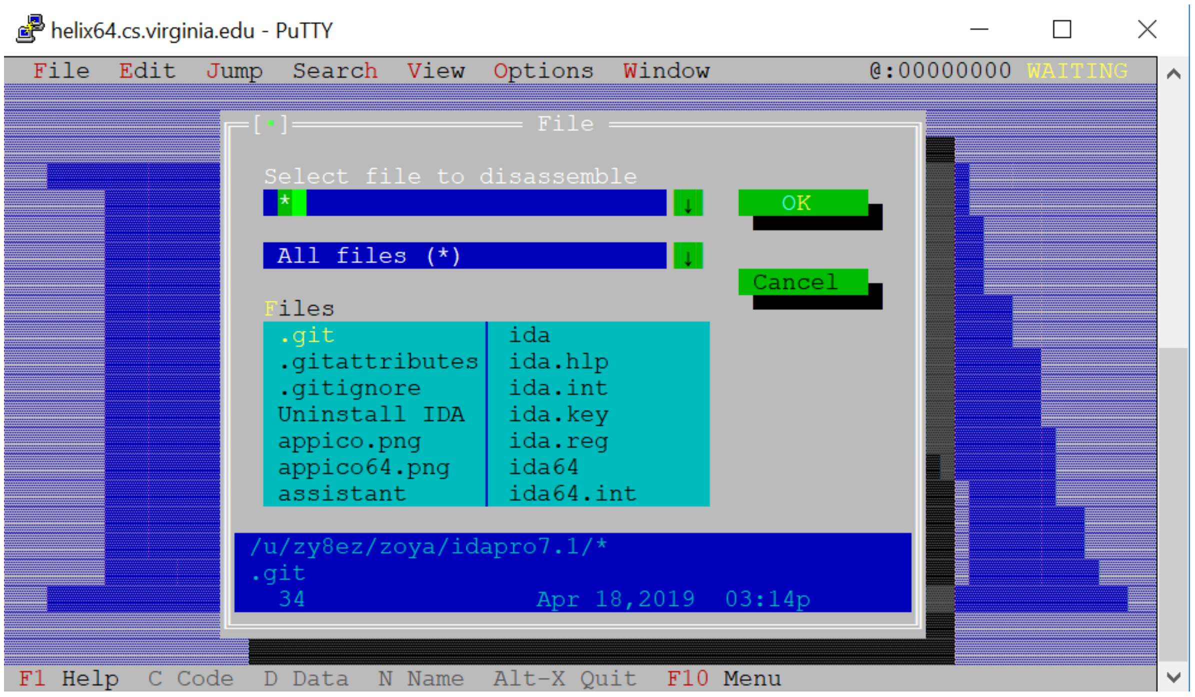
4 EVALUATION OF CAPABILITIES OF THE THREE TOOLS

4.1 IDA Pro

IDA is a Windows, Linux or Mac OS X hosted multi-processor disassembler and debugger - meaning it supports multiple debugging targets and can handle remote applications, via a "remote debugging server". However, in this project, we only focus on IDA disassembler. IDA is licensed and is sold for a very high licensing price, but it delivers so many functionalists that security and binary analysis researchers do consider paying for its Pro edition.

4.1.1 Supported Architectures. IDA Pro supports over 50 different architectures which is very impressive between these three tools we analyzed. Following is a short list of supported architectures, however full list can be found here.

- x64 architecture (Intel x64 and AMD64)
- ARM64 Architecture (aka AArch64)



```
helix64.cs.virginia.edu - PuTTY
 File  Edit  Jump  Search  View  Options  Window          @:00000000 WAITING
                      [*]────────────── File ──────────────
                       Select file to disassemble
                       *                                 ↓    OK

                       All files (*)                     ↓
                                                              Cancel
                       Files
                        .git              ida
                        .gitattributes    ida.hlp
                        .gitignore        ida.int
                        Uninstall IDA     ida.key
                        appico.png        ida.reg
                        appico64.png      ida64
                        assistant         ida64.int

                      /u/zy8ez/zoya/idapro7.1/*
                      .git
                        34                 Apr 18,2019   03:14p
 F1 Help  C Code  D Data  N Name  Alt-X Quit  F10 Menu
```

Fig. 1. IDA Pro User Interface

- Analog Devices AD218x series (ADSP-2181, ADSP-2183, ADSP-2184(L/N), ADSP-2185(L/M/N), ADSP-2186(L/M/N), ADSP-2187(L/N), ADSP-2188M/N, ADSP-2189M/N)
- Dalvik (Android bytecode, DEX)
- DEC Alpha
- DSP563xx, DSP566xx, DSP561XX (comes with source code)
- TI TMS320C2X, TMS320C5X, TMS320C6X, TMS320C64X, TMS 320C54xx, TMS320C55xx, TMS320C3 (comes with source code)
- TI TMS320C27x/TMS320C28x
- Hewlett-Packard HP-PA (comes with source code)
- Hitachi/Renesas SuperH series: SH1, SH2, SH3, Hitachi SH4 (Dreamcast), SH-4A, SH-2A, SH2A-FPU
- IBM/Motorola PowerPC/POWER architecture

4.1.2 Supported File Types. IDA Pro is capable of disassembling almost any popular file format. A short list of these formats can be found below. However full list can be found here.

- MS DOS
- EXE File
- MS DOS COM File
- MS DOS Driver
- Binary File
- ZIP archive
- JAR archive

4.1.3 supported Languages. To determine if each of our tools is capable of disassembling and analyzing different files, we collected several different programs written in different programming languages, then ran them against target tools. Programming languages we selected for this purpose are as follow C, C++, Perl, Python, Fortran, Ruby, Shell, and .exe files. IDA Pro successfully read, disassembled and analyzed programs written in all of these languages.

4.1.4 GUI - UI/UX. IDA Pro is one of the tools which is said to have comparatively very user-friendly UI. However, we had access to IDA Pro through Helix64 machine that was provided to us by the UVA CS department. To run IDA Pro on this machine we first had to connect to it via SSH then run IDA from the terminal. This caused the GUI to look very different than what we expected to see. This UI is very similar to MS-DOS applications UI (Figure 1) so it is not very user-friendly. Also, by researching about IDA Pro, we found that originally it comes with so many different toolbars and features that were not present in the GUI we worked with. More specifically, features regarding drawing control flow graph were not available in our GUI which will be discussed further in section 4.1.5. For this reason, based on our tests and analysis, IDA Pro's GUI is less desirable compared to Ghidra.

4.1.5 Flow Graph. IDA Pro advertises for flow graphing feature which was the reason we chose to investigate and compare this feature among all three tools we are comparing. IDA Pro supports graphing through a VCG Port. from there it is able to produce standard GDL graphs which are then passed to Wingraph32 for drawing. Wingraph32, a partial port of the VCG graphing library which is available since IDA Pro 4.17. Graphing is as easy as selecting the chunk of code you'd like to draw the graph for, then use the graphing commands that are available in the "graph toolbar" to generate your desired graph with your defined settings. However, as mentioned in the previous section, the GUI we worked with during this project did not look anything like the original IDA Pro GUI. Using this GUI we were able to generate GDL file but it was missing the options to draw the graph from it.

4.1.6 Fault Tolerance. To measure how fault tolerant each of tools is, we modified a few bytes in a binary file using ‘dd‘ command and checked if the tool could detect and analyze the file. IDA Pro successfully read the corrupted file without any error messages and was able to extract some information from it. However, we note that the extracted information from the corrupted file was not fully correct but it was accurate for the most part.

4.2 Angr

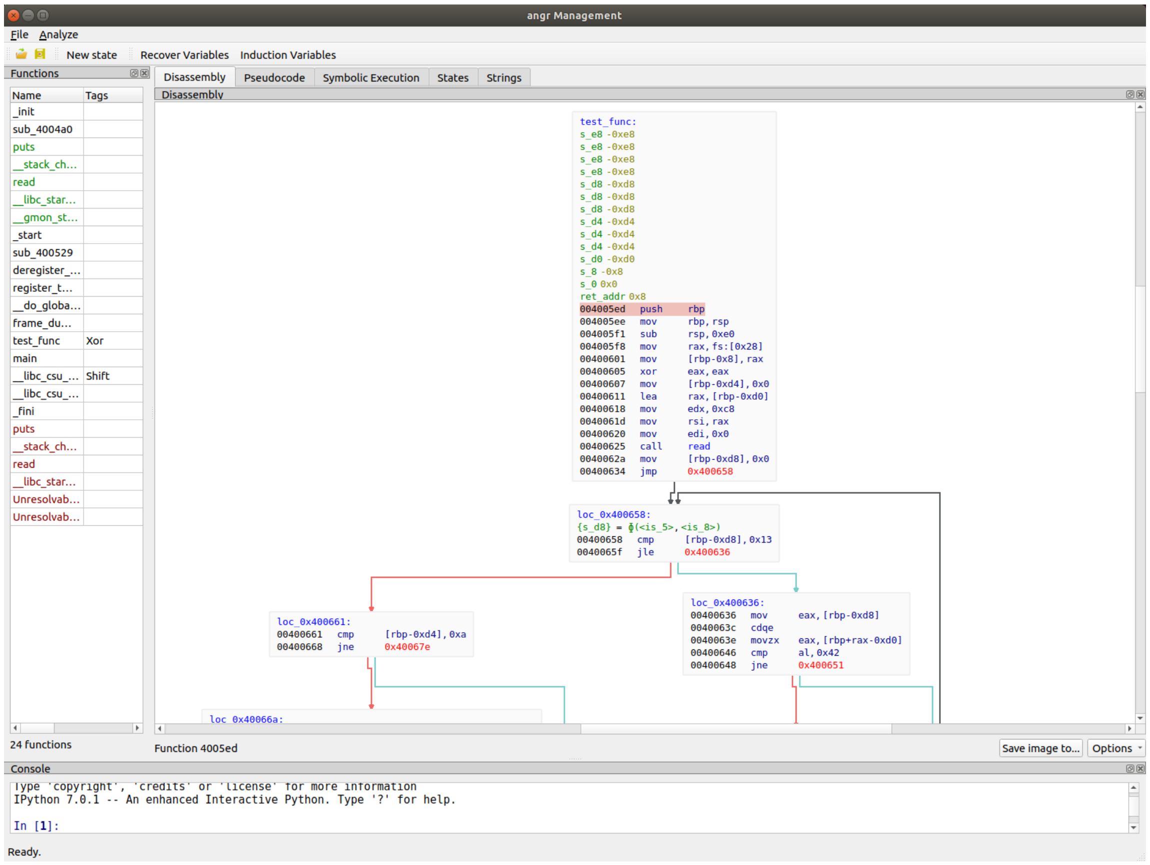
Angr[4] is a binary analysis framework built on python. It was built by the computer security research team at UC Santa Barbara and the Security Engineering of Future Computing team at Arizona State University. Angr is built around the Intermediate Representation that is created by means of the python version of VEX, PyVEX. Angr is a wordplay on getting Angry when someone is VEXed. VEX is a dynamic instrumentation infrastructure by Valgrind[3] On creating the IR using PyVEX, angr stores this state and spins up a simulation manager. Using this sim, a symbolic representation[5] is created and using this, a combination of static and dynamic analysis is possible. Angr is commonly used to solve CTF challenges.

4.2.1 Supported Architectures. Angr supports binaries built on all architectures that are supported by VEX. This includes most CISC and RISC based architectures. VEX is an architecture-agnostic, side-effects-free representation of a number of target machine languages. It abstracts machine code into a representation designed to make program analysis easier. The most popular ones it supports are:

- x86
- ARM

- MIPS
- PowerPC

4.2.2 GUI - UI/UX. Angr does not have it's own native GUI. It is a python module installed and uses CLI to execute over a python shell. Users can use IPython to interactively step in and step out of program counter just like a debugger would. There are multiple third-party applications that help visualize and build control flow graphs using angr. A few notable ones are Fig 2 - Angr-Management and Fig 3 - CFG-Utils. Angr management is a much more capable interface with the ability to select each of the basic blocks and analyze them or export them as images. But, this does not let us edit the values in the blocks. Auto



Name	Tags
_init	
sub_4004a0	
puts	
__stack_ch...	
read	
__libc_star...	
__gmon_st...	
_start	
sub_400529	
deregister_...	
register_t...	
__do_globa...	
frame_du...	
test_func	Xor
main	
__libc_csu_...	Shift
__libc_csu_...	
_fini	
puts	
__stack_ch...	
read	
__libc_star...	
Unresolvab...	
Unresolvab...	

```
test_func:
s_e8 -0xe8
s_e8 -0xe8
s_e8 -0xe8
s_e8 -0xe8
s_d8 -0xd8
s_d8 -0xd8
s_d8 -0xd8
s_d4 -0xd4
s_d4 -0xd4
s_d4 -0xd4
s_d0 -0xd0
s_8 -0x8
s_0 0x0
ret_addr 0x8
004005ed   push    rbp
004005ee   mov     rbp, rsp
004005f1   sub     rsp, 0xe0
004005f8   mov     rax, fs:[0x28]
00400601   mov     [rbp-0x8], rax
00400605   xor     eax, eax
00400607   mov     [rbp-0xd4], 0x0
00400611   lea     rax, [rbp-0xd0]
00400618   mov     edx, 0xc8
0040061d   mov     rsi, rax
00400620   mov     edi, 0x0
00400625   call    read
0040062a   mov     [rbp-0xd8], 0x0
00400634   jmp     0x400658

loc_0x400658:
{s_d8} = ϕ(<is_5>, <is_8>)
00400658   cmp     [rbp-0xd8], 0x13
0040065f   jle     0x400636

loc_0x400661:
00400661   cmp     [rbp-0xd4], 0xa
00400668   jne     0x40067e

loc_0x400636:
00400636   mov     eax, [rbp-0xd8]
0040063c   cdqe
0040063e   movzx   eax, [rbp+rax-0xd0]
00400646   cmp     al, 0x42
00400648   jne     0x400651

loc_0x40066a:
```

```
Type 'copyright', 'credits' or 'license' for more information
IPython 7.0.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

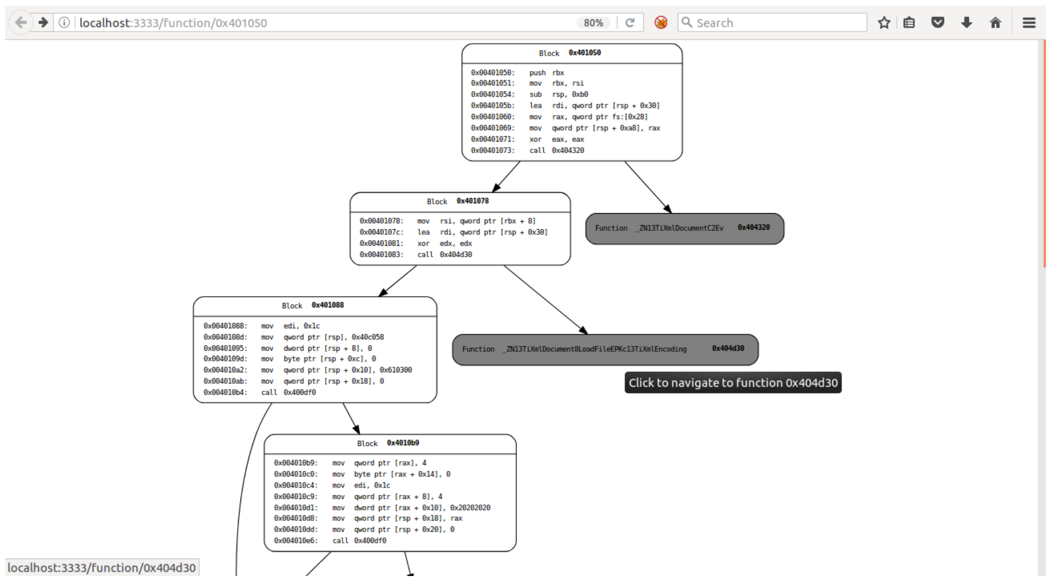
Fig. 2. Angr Management

pseudocode generation is also possible with angr-management. Consequently, cfg-utils is a utility that just visualizes the CFGs and we do not have any other functionality with it.

4.2.3 Supported File Types. Angr supports all compiled binaries and executables that can be parsed and read as a byte stream in python. We have tested this with binary files created from .c, .cpp, perl, ruby, fortran and shell files. We were also able to parse jar files and exe files using angr. Some unconventional files such as Dalvik files from android or zip files were not parsable.

4.2.4 supported Languages. Angr cannot re-parse a program written in a programming language. It can only read byte streamable compiled binary files.

4.2.5 Flow Graph. Control Flow Graphs(CFG) is a graph with basic blocks and edges leading to them based on direct or indirect jumps. CFGs are of two types in Angr. Static (CFGFast) and Dynamic (CFGEmulated). CFGFast generates CFGs much like any other RE tool based on the assembly code. CFGEmulated generates CFG during the program's runtime and it is more accurate but is drastically slower.
Angr does not provide any GUI to display the CFG for visualization, but we can use


localhost:3333/function/0x401050
80%
Search
Click to navigate to function 0x404d30
localhost:3333/function/0x404d30

```
Block 0x401050
0x00401050:   push  rbx
0x00401051:   mov   rbx, rsi
0x00401054:   sub   rsp, 0xb0
0x0040105b:   lea   rdi, qword ptr [rsp + 0x30]
0x00401060:   mov   rax, qword ptr fs:[0x28]
0x00401069:   mov   qword ptr [rsp + 0xa8], rax
0x00401071:   xor   eax, eax
0x00401073:   call  0x404320

Block 0x401078
0x00401078:   mov   rsi, qword ptr [rbx + 8]
0x0040107c:   lea   rdi, qword ptr [rsp + 0x30]
0x00401081:   xor   edx, edx
0x00401083:   call  0x404d30

Function  _ZN13TiXmlDocumentC2Ev  0x404320

Block 0x401088
0x00401088:   mov   edi, 0x1c
0x0040108d:   mov   qword ptr [rsp], 0x40c058
0x00401095:   mov   dword ptr [rsp + 8], 0
0x0040109d:   mov   byte ptr [rsp + 0xc], 0
0x004010a2:   mov   qword ptr [rsp + 0x10], 0x610300
0x004010ab:   mov   qword ptr [rsp + 0x18], 0
0x004010b4:   call  0x400df0

Function  _ZN13TiXmlDocument8LoadFileEPKc13TiXmlEncoding  0x404d30

Block 0x4010b9
0x004010b9:   mov   qword ptr [rax], 4
0x004010c0:   mov   byte ptr [rax + 0x14], 0
0x004010c4:   mov   edi, 0x1c
0x004010c9:   mov   qword ptr [rax + 8], 4
0x004010d1:   mov   dword ptr [rax + 0x10], 0x20202020
0x004010d8:   mov   qword ptr [rsp + 0x18], rax
0x004010dd:   mov   qword ptr [rsp + 0x20], 0
0x004010e6:   call  0x400df0
```

Fig. 3. Cfg Utils

angr-management or cfg-utils or even tools like networkX to build out the digraphs. IDA does not split basic blocks at function calls. Angr will because they are a form of control flow and basic blocks end at control flow instructions. Also, IDA will split basic blocks if another block jumps into the middle of it. This is called basic block normalization, and angr does not do it by default since it is not necessary for most static analysis.

4.2.6 Fault Tolerance. Unlike the other two tools, Angr is very poor at fault tolerance. We tried reading broken binary files using Angr and even by corrupting even one byte of a file, Angr threw 'ELFParserError' but the other tools could still read the binary and construct IRs from it.

4.2.7 Above and Beyond an RE Tool. Angr is way more than just a RE tool for binary analysis[6]. Angr can be a perfect weapon in your arsenal for cracking CTF challenges. Angr can be used for vulnerability detection[2], as a fuzzer, cracking into crackmes, backward slicing programs, and even auto creates ROP chains.

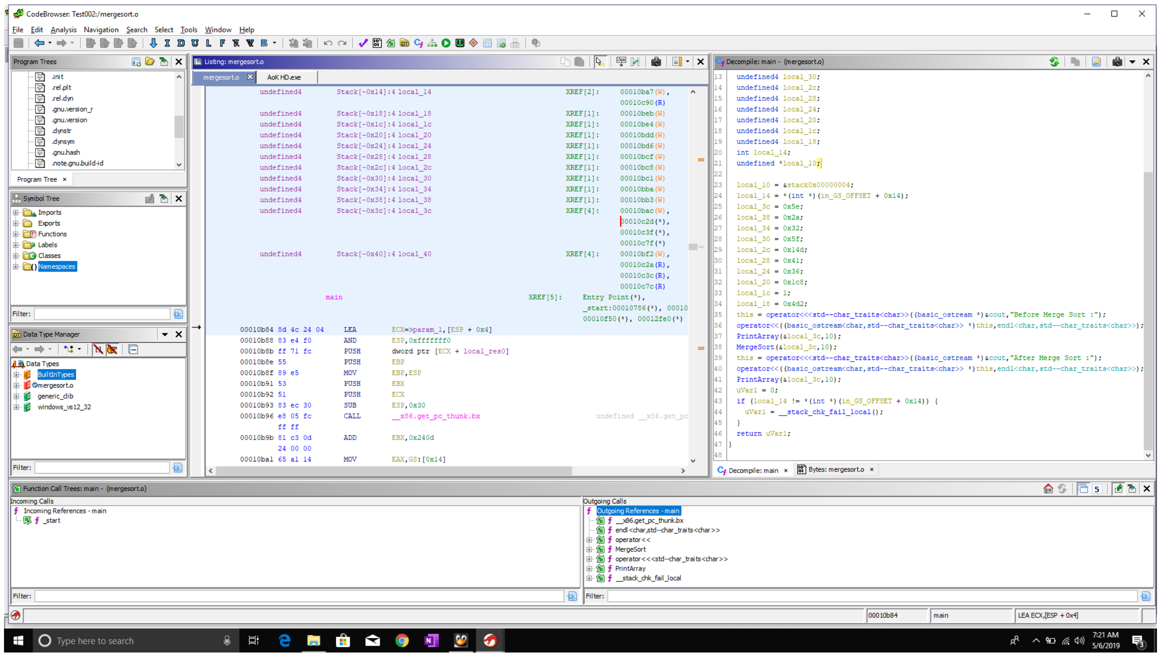
4.3 Ghidra

Ghidra is a free open source reverse engineering tool developed by the National Security Agency (NSA). It's written in C++ and Java. The binaries were released at RSA Conference in March 2019, the Sources were published a month later on GitHub. Ghidra is equipped to handle multiple scenarios as well as is apt at doing many things like analysis compiled codes on various platforms including Windows, Mac OS, and Linux, Disassembly, assembly, decompilation, graphing and scripting, work in user-interactive and automated modes. Users can develop Ghidra Plug-in components and/or scripts using the exposed APIs. Ghidra also supports collaboration and non-shared projects. This helps when multiple engineers want to work on the same code and hence have to share the code. An additional feature namely version tracking helps in maintaining synchronicity and making it possible to move between multiple versions to help diagnose the software.

4.3.1 Supported Architectures. Ghidra supports many architectures and many new are added in the newer releases everyday. Relatively new software and hence doesn't support as much architectures yet. The list of supported architectures is shown below:

- x86/Linux
- ARM and AARCH64
- PowerPC 32/64 and PowerPC VLE
- MIPS 16/32/64
- MicroMIPS
- 68xxx
- Java
- DEX bytecode
- PA-RISC
- PIC 12/16/17/18/24
- Sparc 32/64
- CR16C
- Z80
- 6502
- 8051
- MSP430
- AVR8
- AVR32

4.3.2 GUI - UI/UX. Ghidra is a very beautiful and very intuitive design [Fig 4]. The window is divided into multiple components namely,



```
             undefined4       Stack[-0x14]:4 local_14                    XREF[2]:     00010ba7(W),
                                                                                      00010c90(R)
             undefined4       Stack[-0x18]:4 local_18                    XREF[1]:     00010beb(W)
             undefined4       Stack[-0x1c]:4 local_1c                    XREF[1]:     00010be4(W)
             undefined4       Stack[-0x20]:4 local_20                    XREF[1]:     00010bdd(W)
             undefined4       Stack[-0x24]:4 local_24                    XREF[1]:     00010bd6(W)
             undefined4       Stack[-0x28]:4 local_28                    XREF[1]:     00010bcf(W)
             undefined4       Stack[-0x2c]:4 local_2c                    XREF[1]:     00010bc8(W)
             undefined4       Stack[-0x30]:4 local_30                    XREF[1]:     00010bc1(W)
             undefined4       Stack[-0x34]:4 local_34                    XREF[1]:     00010bba(W)
             undefined4       Stack[-0x38]:4 local_38                    XREF[1]:     00010bb3(W)
             undefined4       Stack[-0x3c]:4 local_3c                    XREF[4]:     00010bac(W),
                                                                                      00010c2d(*),
                                                                                      00010c3f(*),
                                                                                      00010c7f(*)
             undefined4       Stack[-0x40]:4 local_40                    XREF[4]:     00010bf2(W),
                                                                                      00010c2a(R),
                                                                                      00010c3c(R),
                                                                                      00010c7c(R)
                          main                                    XREF[5]:     Entry Point(*),
                                                                               _start:00010786(*), 00010
                                                                               00010f50(*), 00012fe0(*)
        00010b84 8d 4c 24 04     LEA        ECX=>param_1,[ESP + 0x4]
        00010b88 83 e4 f0        AND        ESP,0xfffffff0
        00010b8b ff 71 fc        PUSH       dword ptr [ECX + local_res0]
        00010b8e 55              PUSH       EBP
        00010b8f 89 e5           MOV        EBP,ESP
        00010b91 53              PUSH       EBX
        00010b92 51              PUSH       ECX
        00010b93 83 ec 30        SUB        ESP,0x30
        00010b96 e8 05 fc        CALL       __x86.get_pc_thunk.bx                   undefined __x86.get_pc
                 ff ff
        00010b9b 81 c3 0d        ADD        EBX,0x240d
                 24 00 00
        00010ba1 65 a1 14        MOV        EAX,GS:[0x14]
```

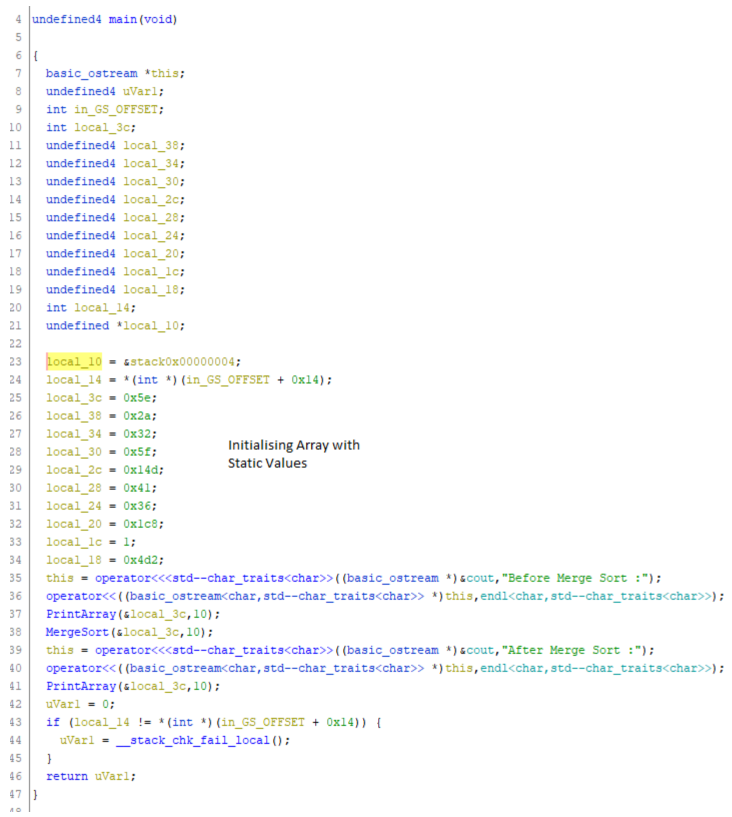
```
  undefined4 local_30;
  undefined4 local_2c;
  undefined4 local_28;
  undefined4 local_24;
  undefined4 local_20;
  undefined4 local_1c;
  undefined4 local_18;
  int local_14;
  undefined *local_10;

  local_10 = &stack0x00000004;
  local_14 = *(int *)(in_GS_OFFSET + 0x14);
  local_3c = 0x5e;
  local_38 = 0x2a;
  local_34 = 0x32;
  local_30 = 0x5f;
  local_2c = 0x14d;
  local_28 = 0x41;
  local_24 = 0x36;
  local_20 = 0x1c8;
  local_1c = 1;
  local_18 = 0x4d2;
  this = operator<<<std--char_traits<char>>((basic_ostream *)&cout,"Before Merge Sort :");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
  PrintArray(&local_3c,10);
  MergeSort(&local_3c,10);
  this = operator<<<std--char_traits<char>>((basic_ostream *)&cout,"After Merge Sort :");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
  PrintArray(&local_3c,10);
  uVar1 = 0;
  if (local_14 != *(int *)(in_GS_OFFSET + 0x14)) {
    uVar1 = __stack_chk_fail_local();
  }
  return uVar1;
}
```

Fig. 4. Ghidra User Interface

- Assembly Code - In this section we see the machine code from the compiled file. This shows us the code which helps us with finding various components and the overall logic of the execution.

- Decompiler Window - In this section we see a pseudo-code representation of the functions in C language. This helps us get information about the variables used, magic numbers, data-types as well as logic the being used.[Fig 5]



```
undefined4 main(void)

{
  basic_ostream *this;
  undefined4 uVar1;
  int in_GS_OFFSET;
  int local_3c;
  undefined4 local_38;
  undefined4 local_34;
  undefined4 local_30;
  undefined4 local_2c;
  undefined4 local_28;
  undefined4 local_24;
  undefined4 local_20;
  undefined4 local_1c;
  undefined4 local_18;
  int local_14;
  undefined *local_10;

  local_10 = &stack0x00000004;
  local_14 = *(int *)(in_GS_OFFSET + 0x14);
  local_3c = 0x5e;
  local_38 = 0x2a;
  local_34 = 0x32;
  local_30 = 0x5f;
  local_2c = 0x14d;
  local_28 = 0x41;
  local_24 = 0x36;
  local_20 = 0x1c8;
  local_1c = 1;
  local_18 = 0x4d2;
  this = operator<<<std--char_traits<char>>((basic_ostream *)&cout,"Before Merge Sort :");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
  PrintArray(&local_3c,10);
  MergeSort(&local_3c,10);
  this = operator<<<std--char_traits<char>>((basic_ostream *)&cout,"After Merge Sort :");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
  PrintArray(&local_3c,10);
  uVar1 = 0;
  if (local_14 != *(int *)(in_GS_OFFSET + 0x14)) {
    uVar1 = __stack_chk_fail_local();
  }
  return uVar1;
}
```

Fig. 5. Decompiled Code

- Program Tree - This tree helps us get a bird's eye view of the whole machine code by dividing the code into its segments such as bss, dynamic, data, etc.
- Symbol Tree - We can see all the kinds of files such as Imports, Exports, Functions, Labels, Classes, and Namespaces. All the files and libraries being imported and exported. We can see a list of functions both user created as well as thunk functions. You can also see the labels and classes that are being used in the program. We can also look at all the Namespaces used in the software being reverse engineered.
- Data Type Manager - In this section, we can take a look at all the various data types used as well as the types of data-types such as built-in, generic dib, etc.

- Function Call Tree - This section helps us visualize function calls in a tree structure. This section is further divided into 2 parts namely,
  - Incoming Calls - All the incoming calls to the current function are shown here. The incoming call for the main function will be the start function which is in turn called by the program (like OS) starting this current software which is being reverse engineered.
  - Outgoing Calls - All the functions called by the current functions are shown as well as a cascading of all the functions the called function calls as well in order of their calls.

4.3.3 Flow Graph. Ghidra allows you to work on Flow Graph and code browser at the same time. It is awesome that the Function graph window and the code browser window are synchronized to each other. Using two screens, we can go through the code flow in the graph mode in one screen and keep the other screen for assembly code which is useful when I need details in the assembly code or to add comments and rename labels, etc. We can also trace the links between functions and how control is being passed around; also the existence of loops and recursions is made easy.

4.3.4 Supported File Types. Ghidra supports a wide range of file formats as mentioned for Angr and IDA Pro.

4.3.5 Fault Tolerance. Ghidra performs extremely well in case of faults, we corrupted the compiled file and tried to reverse engineer the file using Ghidra. The results were surprising as a file reduced from 7096 KB to 116 KB yield great results as the file is not only opened but also analyzed. The analysis for corrupted file and the original file is similar and hence we can conclude that fault tolerance is very good.

5 DISCUSSIONS

5.1 Strengths

All these three tools are very powerful and versatile tools for reverse engineering and binary analysis purposes. However, each has its own strengths that make them unique and desirable for the very specific task you would want to accomplish.

5.1.1 *IDA Pro.*

IDA has a debugger whereas Ghidra does not.- If your application needs a debugger then you should prefer IDA over Ghidra.
Lots of available plugins (i.e. IDAngr: Use Angr in the IDA debugger) - IDA has been around for a while and so we have a huge number of plugins readily available.
Support for a massive amount of platforms; IDA Pro supports more than 50 different architectures which is very impressive comparing to the other 2 tools we analyzed.
Great support from the developers and their forums because it has been used for a longer time than Angr or Ghidra.

5.1.2 *Angr.*

Angr creates accurate CFGs in CFGEmulated mode as the CFG is dynamically built over runtime. But other than that Angr isn't the greatest RE tool. It performs other functions like fuzzing, brute forcing and vulnerability checks much better than the other two tools.

5.1.3 *Ghidra.*

Ghidra is a free and open source software - This makes it easy to access for everyone. We can add personal modules to better help us.
Ghidra allows multiple reverse engineers to share a project - This makes reverse engineering a collaborative approach and hence helps in continuous monitoring as well as updating of the binaries.
Has Undo option - This option was added posterior into Ghidra, this allows undoing certain parts of the file so as to allow dynamic editing. Version control helps in better maintaining the binary files.
Ghidra allows multiple files to be added to the same project.
Ghidra is noticeably fast for files over 1GB.

5.2 Limitations

5.2.1 *IDA Pro.*

IDA Pro doesn't have an Undo button - Undo button helps in making dynamic changes to the file without worrying about corrupting the binary file. Another major limitation to IDA Pro is its price. This tool is very pricey in comparison to its new competitor, Ghidra, which comes at no cost.

5.2.2 *Angr.*

Angr isn't the best RE Tool out there. It does not handle faulty files, and it is much slower than the other tools as the dynamic analysis is done alongside the runtime execution of the program. And with a lack of GUI, the user experience and usability is the least favorable compared to the other two tools.

5.2.3 Ghidra. Ghidra doesn't have a debugger like IDA Pro. Also, Ghidra is fairly a new tool, therefore, unlike IDA Pro, there aren't yet many plugins developed for it. Also, the documentation is not fully developed for it, which results in a delay in learning Ghidra.

6 CONCLUSION

From our evaluations and from using each of the tools we can say that no single tool is the best among them. Each of them has their own strengths and weaknesses. All succeed in being a good Reverse Engineering tool, but each of the tools has something more to offer that the others don't. In addition, we have many more platforms and languages being used in many of the real world applications that we didn't have the chance to analyze since they did not fit our timing constrains of this project. However, our efforts in this project was to deliver an unbiased comparison among these three tool using a short list of comparison metrics, which we successfully achieved.

REFERENCES

[1] B Bellay and H Gall. 1997. A Comparison of four reverse engineering tools. *Proceedings of the 4 Working Conference on Reverse Engineering*, 2–11. https://doi.org/10.1109/WCRE.1997.624571

[2] Ronny Chevalier, Stefano Cristalli, Christophe Hauser, Yan Shoshitaishvili, Ruoyu Wang, Christopher Kruegel, Giovanni Vigna, Danilo Bruschi, and Andrea Lanzi. 2019. BootKeeper: Validating Software Integrity Properties on Boot Firmware Images. In *CODASPY*. ACM, 315–325.

[3] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation.. In *PLDI*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 89–100. http://dblp.uni-trier.de/db/conf/pldi/pldi2007.html#NethercoteS07

[4] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[5] Jacob Springer and Wu-chang Feng. 2018. Teaching with angr: A Symbolic Execution Curriculum and CTF. In *ASE @ USENIX Security Symposium*. USENIX Association.

[6] Fish Wang and Yan Shoshitaishvili. 2017. Angr - The Next Generation of Binary Analysis. In *SecDev*. IEEE Computer Society, 8–9.