Spring 5-22-2021

# Enhancing Usability Of Malware Analysis Pipelines With Reverse Engineering

Jeffrey Ching
*Syracuse University*

# Abstract

Lots of work has been done on analyzing software distributed in binary form. This is a challenging problem because of the relatively unstructured nature of binaries. To recover high-level structure, various attempts have included static and dynamic analysis. However, human inspection is often required, as high-level structure is compiled away. Recent success in this area includes work on variable-name recovery, vulnerability discovery, class recovery for object-oriented languages. We are interested in building a pipeline for user to analyze malware. In this thesis we tackle two problems central to malware analysis pipelines. The first is D3RE, an interactive querying tool that allows users to analyze binaries interactively by writing declarative rules and visualizing their results projected onto a binary. The second is Assmeblage, a tool which automatically scrapes GitHub for C and C++ repositories and builds these repositories automatically using different compilation settings to produce a variety of configurations. These two tools will enable users to get enough data to do analysis as well for them to do interactive analysis. Finally, we present future work demonstrating a possible visualization combining d3re and Ghidra along with some specific questions for future user studies.

# Enhancing Usability of Malware Analysis Pipelines with Reverse Engineering

by

Jeffrey C. Ching

B.S., SUNY, Stony Brook University, 2017

Thesis

Submitted in partial fulfillment of the requirements for

Master of science in Computer Science

Syracuse University

May, 2021

# Acknowledgements

First I want to thank my advisor professor Kris Micinski, he taught me a lot on how to do research and point me out when I am stuck. As a student without any background in programming languages and static analysis, his guidance and knowledge really encouraged me to get interested into the research. I had a good experience on working with Kris, as well as my two years at Syracuse. I also want to thank professor Shiu-Kai Chin for being the chair of the committee, and professor Endadul Hoque and Asif Salekin to be on the committee.

Of course, I couldn't done this research without the support from the HARP lab, especially with the help of Yihao, Christian, Daniel, and Davis. We worked together on the Assemblage and D3RE project, and Yihao taught me a lot on related materials and helped me a lot while doing my thesis.

Finally, I want to thank my family for supporting me, and give me freedom to do anything. It is great to have support when you are away from home, which will encourage me to keep moving forward. It has been a long journey, and the show will go on.

# Contents

# List of Figures

# List of Tables

# 1 | Introduction

Security is emerging as one of the most important cross-disciplinary focus areas within modern computing, and comprises a wide range of areas including bug finding vulnerability discovery [2], malware analysis [49], and other related tasks [25]. In the thesis, we will be working on enhancing the usability and productivity of malware analysis with a malware analysis pipeline. We define a malware analysis pipeline as the end-to-end process that involves discovering vulnerabilities in large datasets of binary code. These pipelines involve data collection, static and dynamic analysis, and ultimately human-guided reverse engineering using a reverse engineering tool. However, it is challenging because reverse engineering on binary files relies on the reverse engineers' experience and normally have to be done manually.

This thesis primarily contributes two tools that tackle foundational challenges in malware analysis pipelines. My thesis is that malware analysis pipelines may be improved via a combination of automated vulnerability discovery aided by visualization, along with tooling to build binaries to collect datasets. The first is D3RE: a declarative static analysis tool that integrated builds on a state-of-the-art logical inference backend (the Datalog solver souffle) and the NSA's Ghidra reverse engineering framework to act as a combined platform for visualization and logical inference of properties of binaries. Second, we study the problem of how to automatically scrape source code from large software repositories like GitHub and automatically build several different variants to study the impact of compilation and configuration parameters. To do this, we built a tool, Assemblage, which collects C and C++ repositories and builds binary files for future research on D3RE as well as binary reverse engineering. We expect Assemblage and D3RE to be the foundation of further research. The thesis will be the introduction as well as a preview of the project; it is an ongoing work and we are still at the beginning of the research. The work we have done will be crucial in the future extension of the project.

## 1.1 Motivation

Reverse engineering is a process of deconstructing the product to reveal the original design or architecture. Reverse engineering can be applied in different fields of engineering, including electrical engineering, electronic engineering, and software engineering. Our focus will be on software engineering since we will be discussing binary analysis, security problems, and malware analysis. Several techniques are necessary for reverse engineering, such as disassemblers, decompilers, and sometimes debuggers.

To have a deeper understanding of reverse engineering, we dedicated to do detailed analysis on binaries and develop good tools. Reverse engineering is hard and has a lot of different methods based on different needs and user's habits. In recent years, National Security Agency (NSA) released a new reverse engineering tool called Ghidra, which will be used in this project. Ghidra is an open-source tool that allows users to write plug-ins in Java and Python scripts. In the future, we want to also look at various usability aspects of Ghidra, and possibly extend it. In short, Ghidra shows the disassembled and decompiled codes of the binary file and shows lots of information through a user interface. However, Ghidra provide some tools to show relationships between functions, but still lack of tools to do more accurate analysis. Our goals are to look for and combine binary analysis tools that will be a good fit as plug-ins for Ghidra.

## 1.2 D3RE

In this thesis, we built a declarative demand-driven reverse engineering tool called D3RE which will be mentioned in chapter 3, where we proposed a new methodology for performing declarative, and demand-driven reverse engineering. In D3RE we present a vision for reverse engineers which wil enable them to do a visualization-based analysis and understand binaries while querying their own rules to perform deductive logic inference tasks. Comparing to other binary analysis tools, D3RE allows user to interactively analyze the binaries. To maximize the usage of the tool, we want to write a plug-in that will be able to show the results of binary analysis from D3RE on Ghidra. We evaluated D3RE qualitatively, by implementing several queries, and quantitatively by measuring its performance in benchmarks. To be more precise, we compared the length of the program and the run time with python script in Ghidra, which in most of the cases Datalog beats Python scripts. The detailed statistics will be shown in chapter3. For now, users will be inputting new rules through a CLI, and analyze will be processed by the Datalog engine Souffle. Currently, the results can be only viewed on the console or have limited visualization. In the future, we want to take fully advantage of Ghidra and create a user interface plugin that allows users to click on a specific line and get information on it. However, it is challenging that we have limited information on how reverse engineer will like to interact with Ghidra, thus it is difficult for us to design a user interface.

## 1.3 Assemblage

In a high-level aspect, we want to study the usability of malware analysis pipelines, which includes data collection. Besides using traditional methods for analyzing binary files, due to the popularity of machine learning in the recent computer science society, it will be interesting to apply machine learning methods to analyze binary files. However, both machine learning and malware analysis pipelines involve taking a lot of binaries and then studying them.

In order to build test the scalability and performance of the D3RE tool, we need a variety of binaries. It is hard to directly find a bunch of binaries, thus we aim to scrape C and C++ repositories on Github and build the repositories to collect the binaries. Instead of building our data scraper we take advantage of the existing database GHtorrent in this stage. GHTorrent contains a lot of open-source repositories scraped from GitHub. Since GHTorrent stopped renewing the database and we have built a prototype scraper, we want to replicate and modify GHtorrent to build our scraper. Currently, we created the coordinator and worker in ZMQ to clone and build the repositories. We have stored all the successfully built binary files and the error messages for the unsuccessful build repositories for further researches. These binaries are expected to be the test data of D3RE in the future. Also we hope to extend the tool so we can build the repositories in different compilers such as LLVM and GCC for us to study how they work in scale.

## 1.4 Visualization for Program Analysis

Recent literature suggests that many practitioners follow an iterative approach involving several rounds of hypothesis formation and validation/falsification, often assisted via a combination of static and dynamic analysis [28]. In order to get better results, during reverse engineering, we might have to switch between static analysis and dynamic analysis. Also, although decompilers play an important role while doing RE, the results from the decompiler are sometimes "special" which will be interesting to look into with D3RE and Ghidra such as redundant loops and weird structures. Despite not being a reverse engineering paper, section 6 of Battle's work[5] on data visualization structure analysis is useful for reverse engineering.

Being inspired by the idea from Battle which they mentioned approaches on structure analysis: First, the user have to define the analysis states of the research; Second, user have to take advantage of search trees, breadth, and depth analysis; Third, users will look into the Predictability and Overlap of the project. We can then mirror the above three ideas to reverse engineering, which are three phases in reverse engineering, overview, sub-component scanning, and experiment. The first two phases are most likely to be "get to know the functions" and some kind of pattern matching. We know that each part of the binary has different functions, so it will be important to know what is the purpose of each function. The third phase will most likely test the hypothesis of the engineer and prioritize some characteristics. A good static analysis algorithm is not enough with a good user interface since reverse engineering has to be practical and applicable. We want to take advantage of the ideas of the previous work and possibly design a user-friendly plugin in Ghidra that will fulfill different phases as proposed in the above statements.

## 1.5 Overview

It is necessary to have tools for analyzing binary files and programs, but it is hard and time-consuming to understand a complicated code. However, with D3RE and Assemblage, we will have a variety of data and enables the user to interactively analyze them. We can effectively point out the heart of the problem and design a more user-friendly environment. In D3RE, we introduce a new vision for reverse engineering, wherein experts can rapidly query high-performance logical inference engines to help them accomplish their day-to-day work in vulnerability discovery and other reverse engineering works. As for Assemblage, the prominent part of using open-source data from GitHub is that there will be lots of different types of binary files. Finally, we proposed an idea of visualizing the D3RE on Ghidra and plan to do a user study on the behaviors on doing static analysis, reverse engineering procedures, habits on finding vulnerabilities as well as their opinions on visualizing reverse engineering with the help of static analysis.

Overall, the thesis aims create a pipeline the will analyze the the binaries interactively with Datalog using binaries collected and built by the data collector, which will eventually be visulized on a reverse engineering tool. The figure of the pipeline is shown in figure 1.1

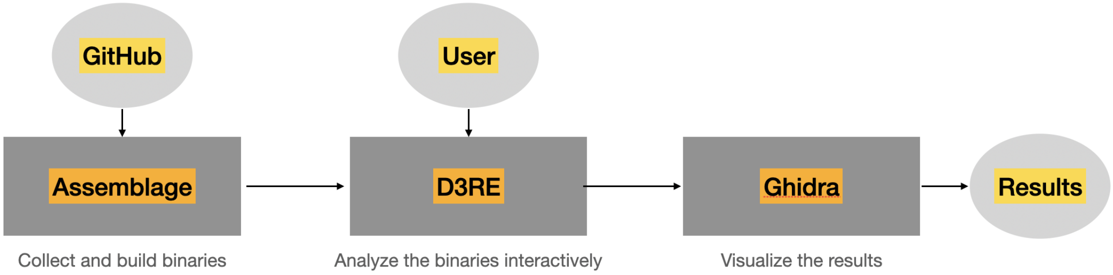

GitHub
User
Assemblage
D3RE
Ghidra
Results
Collect and build binaries
Analyze the binaries interactively
Visualize the results

**Figure 1.1:** The idea of malware analysis pipeline. Assemblage will collect and build data from GitHub. D3RE will enable interactive analysis, and the results will be visualized through Ghidra.

## 1.6 Chapters Preview and Contributions

Some binary analysis, reverse engineering tools will be introduced in chapter 2. The data collection process and future aspects of the scraper will be introduced in chapter 4. In chapter 3 the first section will be focused on the reverse engineering tool Ghidra, which gives some previews of the tool and example scripts and its usage. The prototype D3RE will be introduced in section of chapter 3, which will go through the current stage of D3RE, and the advantages by comparing with python scripts in Ghidra. Last, future work and possible proposal for a user study will be in chapter 5. The following are the contributions of the thesis:

- The idea of declarative demand-driven reverse engineering and a prototype implementation as well as experiments on testing the efficiency and quality of D3RE.
- Create a prototype, Assemblage which will collect repositories from GitHub and build with a variety of compilers for further research on the properties of binaries.
- Proposed an idea on visualizing D3RE on Ghidra with few example.

With D3RE and Assemblage integrated in the malware analysis pipeline, we believe the future is bright and promising.

# 2 Related work and Background

To get better understanding of reverse engineering, the state-of-the-art techniques and mechanics, we collect multiple papers that focus on detail-related knowledge and new open-source tools. In this chapter, we will be talking about previous academic work that has been done in reverse engineering or binary analysis tools, as well as reverse engineering for software understanding. Besides academic papers, we will also show some interesting and popular binary analysis and reverse engineering tools including Ghidra, BAP, OOAnalyzer an etc.

## 2.1 Ergonomics of Decompilers and Reverse Engineering Tools

Yakdan presents some semantics-preserving on transforming the code which make the decompiled code more readable and will help the analyst to find and target the malware [50]. There are some interesting results and founds on how they optimized the decompilers on Ghidra, which are the followings

- Through experiences of using Ghidra from the paper with some examples, the redundancy often appears on assigning variables, and if else statements. Sometimes, the decompiler will assign the variables twice, or assign unnecessary constraints.
- To optimize the efficiency of decompilers, the decompiled codes become ugly and hard to

read. For example: Switching while to if with do-while. As for do-while loops, do-while loops are harder to understand comparing to while loops, which the paper gives some examples on how they deal with this kind of problem.

- Function outlining is also pointed out in the paper, since several duplicates of the same code are spread across the program. As result, if facing duplicate codes, the user can use a transformation rule to replace the codes with functions.
- Variable naming is also important. In Ghidra, the decompilers will give names to the variables, but just with series of numbers. However, having meaningful names will be better, since it will be better to follow, and less confused. Jaffe's research on statistical renaming has been introduced throughout the discussion [22].

Beside decompilers, disassembler are also one large factor of the research in binary analysis and malware analysis, where Pang proposed a research focusing on the system and how the disassemblers work [29]. Despite the high accuracy they got on their experiments, the novelty of this research is on presenting a thorough systematization of binary disassembly from the perspective of algorithms and heuristics. The overall functionality can be divided into the following sections:

- Disassembly: For Ghidra it follows the idea of recursive descent. Starting with a given code address and performs disassembly following the control flow.
- Symbolization: Starting by identifying numerical values that are potential pointers. It will search through all the non-code regions to find the data units.

An example in operand extraction Ghidra considers a data unit to be the start of the string if it is followed by a sequence of a null-byte or ASCII bytes. Otherwise Ghidra will define the pointer of the data unit with characteristics like. The value is an address or introduction in a non-code region, the value is an address of an instruction in a known function, etc[29]. Also some other interesting founding on Ghidra are shown in Pang's research[29]:

- Comparing with other disassemblers, Ghidra can accurately identify grouped pointers like function tables.

- Ghidra does not guarantee to generate a main function, sometimes it generates the entry function as the starting point.

- There are three intersting find outs on Ghidra finding for general functions. First, GHIDRA considers the .eh_frame section to identify functions that have unwinding information. Second, Ghidra consider targets of direct calls to be function entries, and resolves certain indirect calls to determine more function entries. Third, Ghidra uses pattern-based approaches to further recover functions.

- The control flow graph is probably the most complicated part since it included lots of different function calls. The paper chose several different function calls as examples: Indirect jumps, indirect calls, tail calls, and non- returning functions.

## 2.2 Reverse Engineering for Software Understanding

Reverse engineering is an important technique for scientists and engineers to solve problems. However, reverse engineering is a complex process and can be approached variously based on different users or goals. Before going deeper into research topics on reverse engineering and binary analysis, several concepts need to be introduced. While working with binary files, binary analysis tools like static analyzer and debugger will be needed. The differences between reverse engineering tools and binary analysis tools are that reverse engineering tools like Ghidra and IDA focused more on user interaction. Binary analysis tools on the other way focused more on automated analysis which might generate JSON files or binary files for further analysis using reverse engineering tools [40].

Several problems raise our attention in reverse engineering, vulnerability discovery, variable name recovery, and class recovery. We will introduce some of the previous work in these fields, as well as some interesting approaches. Lots of user studies and surveys were done in reverse engineering. In Votipka's user study, he points out some of the backgrounds on reverse engineering in the second section of his research [47]. Since malware can be considered as one of the vulnerabilities, the survey and experiences in the vulnerability section of Votipka's research will be most related to the research on building malware analysis pipeline. However, according to Votipka, ways of finding vulnerabilities and bugs can be very subjective [48]. In the research, they interviewed 25 people on finding vulnerabilities and eventually discussed how the ecosystem can be changed [47]. Similar idea has been pointed out by Shoshitaishvili, where they proposed a shift in the vulnerability analysis paradigm, from tool-assisted human-centered to human-assisted tool-centered [41].

In order to understand the binary files, Nadi points out there are two typical approaches which are highly mentioned: Static analysis and dynamic analysis [28]. These two are traditional and the most popular methods in binary analysis. Many malware binaries are stored in obfuscated form and only deobfuscated at execution time to complicate reverse engineering. This is commonly referred to as packing. Besides unpacking, reviewing the string names and API calls are also useful in identifying going forward on RE. To get more information on the techniques used in RE, the term beacon has been mentioned which is useful in related researches. Beacons are commonly schemas or patterns, which inform how developers expect variables and program components to behave. Most of the participants focused on control flow and data flow analysis to get more ideas on the files [36].

Raff pointed out some of the challenges we will be facing while doing malware analysis [33]. However, as the the techniques improve and the development of new techniques and methodologies, static analysis and dynamic analysis are no longer the only solution for binary analysis. Obviously, the most important and the hardest part is data collection. One of the notable structures in malware binaries is packing, which maybe be applied multiple times, and in different variations. Aghakhano claimed that besides dynamic analysis and static analysis, it is possible for machine learning based models to identify whether it is none malicious and malicious even when contents are packed in his research [1]. Also Raff presents it is possible to use machine learning on identifying benign and malware data [35], and claims to improved the machine learning structure in his research [34].

In the above, we are discussing about how users will start reverse engineering and binary analysis. In the following, we will point out more papers about what kinds of obstacles researchers will face while doing reverse engineering. Votipka points out one of the largest challenges in reverse engineering is it is lack of theory in the paper [47], and most of the information is from reverse engineers' experiences. Since the goal of reverse engineering is to analyze binary files, decompilers, accompany by some symbolic execution tools are necessary [10]. One of the code browser CodeSurfer created by Balakrishnan, which can eases users' reading and understanding of the code. It provides, dataflow analysis, point analysis, and call graphs. However, it has limitations that it does not have integration with IDE like Eclipse [4].

After the software is compiled, the structure of the program is fully breakdown. The relationship between classes, function inheritance, and the original name of the variables are lost. While recovering the program by decompiler, the structures and names cannot be recovered. There are works working on class recovering and variable name recoveries [22, 43]. Other researches working on relationship and flow between classes where Salah presented a new way of generating class usage scenarios and method invocations [37]. Some statistical methods can be introduced on giving reasonable names of variables. Work in these areas seems to be strongly related to machine learning since it is necessary to work with a lot of metadata. Although not for every area in reverse engineering, still the relationship between classes is important.

## 2.3 Open-Source data collection

There are some similar works in collecting and building open-source repositories from GitHub. We need lots of data while designing methods for finding vulnerabilities, solving security problems, as well as applying machine learning. As mentioned above, we need a lot of binary files, and the best way is to get the data from GitHub. There was an existing GitHub scraping tool GHTorrent [16] created by Gousios, which is widely used by researchers in software engineering, and binary analysis. With access to open-source projects, a lot of things can be done. With GitHub, users can find a large number of open projects in different stages of integration and automation, which users will be able to do some research or obtain information of different influence factors and the quality. There are some interesting works using this database such as Manes studies how often stack overflow references GitHub projects[26], Vasilescu did studies on diversity on Github[46], and Gousios user this databse to study pull-based developments[17]. Also, another interesting problem is on dependencies, while compiling a massive number of open-source repositories, the dependency of each package will also be the obstacle we have to overcome [15, 24, 46]. Along with dependencies, there are some studies on continuous integration on open sources, papers on testing, building, and user studies that can be found in the following references [6, 21].

## 2.4 Visualization for program analysis

Despite the research Battle's team are dealing with a totally different task from reverse engineering [5], their idea on visual analysis using Tableau will be related on visualization design. The nominal part is that similar to RE, the Exploratory Visual Analysis (EVA) researches are based on researchers' own experiences, and often have different thoughts and ways to solve and approach the data. They listed three processes on EVA: goals, structure, and performance, which might be the typical way of approaching problems that lack official prior works.

Most of the work in static analysis focused on developing static analysis algorithms instead of the design of the tools. Khoo presents a novel user interface toolkit called the Path Projection that helps the user visualize and understand the program paths in his study [23]. With a good UI tool, it is shown in the results that it will reduce the completion time but keep the accuracy. Other works have tried to visualize the binary file information. Han analyze malware by transforming binary information into image matrices [19]. Matzan did user studies on how VSA affects reverse engineers' behavior by monitoring with an eye-tracking system [27].

## 2.5 Tools for reverse engineering and binary analysis

In this subsection, we will give a short introduction on popular reverse engineering tools such as Ghidra and IDA. We will not be focusing on academic paper reviews and academic research tools, but on open-source reverse engineering and binary analysis tools broadly.

### 2.5.1 Interactive Disassembler (IDA)

IDA is a disassembler that will disassemble the machine codes to assembly and decompile it to C code as well. With the help of IDA, engineers will be able to analyze different types of binary files. IDA performs code analysis in which users can cross-references between code sections, function calls, and different features. However, IDA is expensive, which makes it hard to be universally accepted as a tool for reverse engineering.

### 2.5.2 Ghidra

Ghidra is an open-source reverse engineering tool developed by the NSA. Different from IDA, Ghidra is written in Java while IDA is written in C++. Fortunately, Ghidra is open-source, it is more research-friendly, which IDA costs a lot. With Ghidra, the user will be able to look into the binary executable files in the disassembled form and decompiled form base on the user's

**Table 2.1:** Comparisons between IDA and Ghidra

| | IDA | Ghidra |
|---|---|---|
| Language | C++ | Java |
| Sripts | Yes | Yes |
| Cost | 589 USD | free |
| Debugger | Yes | No |
| Number of open sources | Many | Less |

preferences. Ghidra provides multiple functions (function call graph, python script, and data flow graph, etc.) which allow the users to take advantage base on their needs. The comparisons of IDA and Ghidra can be found in Table 2.1.

## 2.6 Binary Analysis Tools

Besides reverse engineering tools, we will also introduce some binary analysis tools that are possible choices to be combined with Ghidra as plugin tools. Several good binary analysis tools are being developed already. Since Ghidra is open-source, we want to be able to integrate some of the existing tools with Ghidra in the future, to create a more user-friendly environment for reverse engineering.

### 2.6.1 Binary Analysis Platform (BAP)

Although BAP [8] have limitations on supporting formats, there are ways to solve these limitations. Like Ghidra, BAP is an open-source that can take in different kinds of plugins like analysis, disassemblers, etc. Unlike Ghidra, BAP is a binary analysis tool without user interaction reverse engineering functions. BAP is divided into front-end and back-end which both are connected by the BAP intermediate language (BIL). The front-end is responsible for lifting the binary code to the IL, while the back-end can take the BIL and do further program analysis and program verification. Since BAP can be extended, new analyses can be built on existing analyses or base on the users' needs.

### 2.6.2 GrammaTech Intermediate Representation for Binaries (GTIRB)

Grammatech intermediate representation (GTIRB) [38] is an IR designed for binary analysis and rewriting tools. Among the existing binary analysis, most of the IR are only internal, which typically specifies the instruction semantics. However, since the IRs are internal, specific IRs will not be able to communicate with other tools and projects. It is the state of the art that GTIRB is intended to be the bridge of different binary analysis tools and rewriting. With the decompiler ddisasm [12] it will be able to lift binaries to GTIRB. To ensure interoperability between tools, GTIRB does not represent any instruction semantics, which will directly store the raw machine code into IR.

The goal of Gtirb is to design a universal IR that can communicate with multiple tools since binary analysis tools have their own internal IRs. For example, BAP has BIL, and Ghidra has P-code. In order to reach the goal, the raw machine code is directly stored in the IR, and can be later decoded with the user-selected IL like BIL or P-code. For example, in our case, since we are using Ghidra, the p-code will be the IR. According to the limitation on plugging in Gtirb to Ghidra, the Gtirb file can be created from ELF files only, supported architectures IA32, ARM, X86-64, and PPC32, and the version of Gtirb extension has to correspond to the version of Ghidra.

Back to the structure of Gtirb, the very top of Gtirb is the IR element. Under the IR element is the module. The primary content of the module is Codeblock and Datablock, which respectively stores module code and data. Every Codeblock and Catablocks belong to a certain section, and the raw contents are stored in regions in vectors which are called Bytevectors. The size of the byte vector will affect the further usage of rewriting supported by Gtirb. Other structures like:

- IPCFG, is a single graph covering all the code in the IR and shows the relationship between each block. ProxyBlocks are used to represent control flow sources or targets that cannot be resolved to CodeBlocks, and the edges between CodeBlocks are the control flow in the IPCFG. The difference between CFG and IPCFG is that IPCFG only deals with the edge cases when they are needed for analysis.

- Symbols and Symbolic Expressions: Since the core structure of Gtirb is represented in symbols, these will provide symbolization information of CodeBlocks and DataBlocks.

- Auxiliary data (AuxData) tables: Each structure in Gtirb is the result of the analysis by Ddisasm, and has a unique identifier (UUID) used as the reference between each structure in the IR. Since the core structure is represented in symbols, the AuxData table is used for communication and store maps and vectors by using UUID.

### 2.6.3 CWE Checker

Bugs checking in reverse engineering is hard and time-consuming. Due to the complexity and the size of binary execution files, it is almost impossible to find all the bugs manually through tools like Ghidra. Moreover, the variety of CPU structures, including x86,x64, and ARM, have different instruction set. The main goal of reverse engineering are recovering programs, creating control flow graphs, and searching for vulnerabilities. For vulnerabilities detection, with a tool like CWE-checker, the check can be done automatically, and be classified based on common weakness enumeration (CWE) numbers. CWE-checker is based on BAP, which take the advantage of BAP intermediate representation (IR) to achieve CPU architecture independence. In general, cwe-checker will first disassemble the binary execution file. Second, after disassembling the file BAP will lift it to its IR in binary instruction language (BIL) which is defined using algebraic type. Finally, the CWE-checker will generate the reports of the analysis in the form of a JSON file. With the JSON file, the Ghidra plugin written in python will be able to integrate the cwe-checker results with Ghidra, which the results will be shown on the user interface on Ghidra in the form of comments. The CWE numbers that have been included in the checkers are in the following charts, however, users can still define their checkers.

Each CWE number are classified by the definition written in OCaml as the interface with BAP

**Table 2.2:** CWE numbers included in the cwe-checkers

| cwe Number | type | modular structure |
|---|---|---|
| CWE 190 | Integer Overflow | Static Analysis |
| CWE 215 | Information Exposure Through Debug Information | Static Analysis |
| CWE 332 | Insufficient Entropy in PRNG | Static Analysis |
| CWE 367 | Time-of-check Time-of-use Race Condition | Static Analysis |
| CWE 476 | NULL Pointer Dereference | Static Analysis |
| CWE 676 | Use of Potentially Dangerous Function | Static Analysis |
| CWE 243 | Creation of chroot Jail Without Changing Working Directory | Static Analysis |
| CWE 248 | Uncaught Exception | Static Analysis |
| CWE 426 | Untrusted Search Path | Static Analysis |
| CWE 457 | Use of Uninitialized Variable | Static Analysis |
| CWE 467 | Use of sizeof() on a Pointer Type | Static Analysis |
| CWE 560 | Use of umask() with chmod-style Argument | Static Analysis |
| CWE 782 | Exposed IOCTL with Insufficient Access Control | Static Analysis |
| CWE 215 | Out-of-bounds Read | Symbolic Execution |
| CWE 415 | Double Free | Symbolic Execution |
| CWE 416 | Use After Free | Symbolic Execution |
| CWE 787 | Out-of-bounds Write | Symbolic Execution |

as the backend. Below I will give an example on how CWE-checker worked with the definition provided in the CWE-checker GitHub and more are shown in table 2.2: For CWE-190: Integer overflow or wraparound, the CWE190 has a default symbol list that contains the memory allocation functions malloc, xmalloc, calloc and realloc. At each call to the function in the above list, it will check whether the basic block directly before the call contains a multiplication instruction. If it is true, then the call will be flagged since there is no check for overflow on multiplication before the function call. However, there are possibilities that false positives (not sure whether the result of multiplication result is used in a function call) and false negatives ( the overflow may be caused by addition or subtraction) will appear.

### 2.6.4 OOAnalyzer

While recovering binary files, not all files has simple process. Real-world projects are complicated and hard to be reversed in the correct structure. For C++ class structures, to be able to analyze at the machine code level is challenging, Schawartz and the software engineering lab at CMU came up with a tool called OOanalyzer [39] which will be able to recover classes and the relationship of methods in each function. Several background knowledge needs to be informed before introducing the concept and structure of the tool.

- Virtual Function: In C++, polymorphic methods are known as virtual functions. The entry points of each virtual function are included in the virtual function table
- Inheritance vs Composition: We say class A is inherited from class B, most variables and methods in class be will automatically take the definition of class A. While we say class A is composed of class B when class A stores an object of class B as a data member.
- RTTI: RTTI is optional metadata that is used to implement C++ type introspection features. RTTI includes information such as class name, but only polymorphic class has RTTI, and some malware programs disabled RTTI.

Most of the previous works can only recover polymorphic classes. OOanalyzer is designed so that it can deal with all kinds of methods and classes. As mentioned before, if it is a polymorphic class, it will be able to fetch information in the virtual function table. However, to solve most cases, OOAnalyzer cannot rely only on the virtual function table, which OOAnalyzer will assign the methods to the correct class base on the object pointers. Moreover, to cover ambiguous properties, XSB prolog is used to build the rules and to search for a consistent model. The reason for using XSB prolog is that it is mature, open-source, can be embedded into C/C++ programs, and has robust tabling support [40, 45]. The idea of the tool is straightforward, which it will take an execution file as the input and get the initial facts based on static analysis.

OOAnalyzer will reason about the program by matching a built-in set of rules over the facts. However, sometimes OOAnalyzer is unable to reach new forward reasoning before some properties are decided. In order to move on, OOAnalyzer will identify an ambiguous property and make a good guess to continue the process [39]. The facts emitted by forwards and hypothetical reasoning are called entity facts. Different from initial facts, which initial facts cannot be modified. The entity facts are the idea of how the tool is attempting to recover the execution file throughout the process. After OOAnalyzer finished processing the execution file, the user can choose to output a JSON file which is the information on how the tool wants to recover the classes. If users want to view it on Ghidra, the Ghidra plug-in will take the JSON file as an input and combine the information of the JSON file with the Ghidra user interface.

# 3 | D3RE

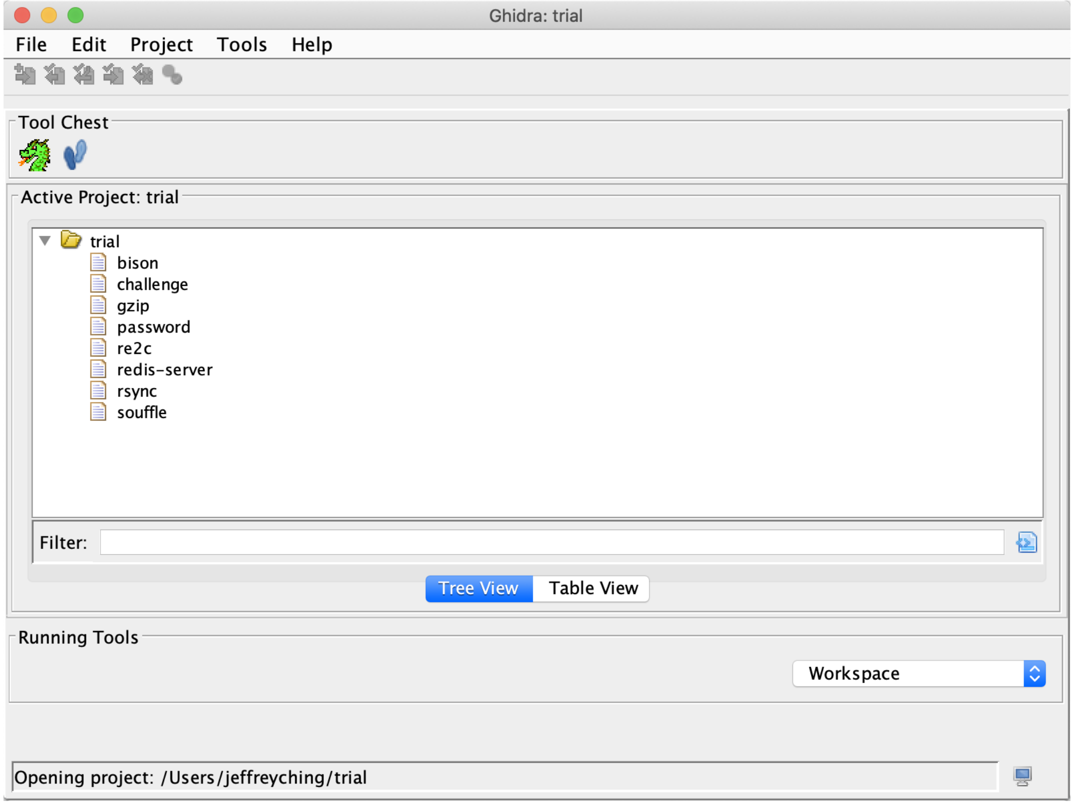
In this chapter we will talk about the reverse engineering tool Ghidra, and our static analysis tool D3RE. In the first section, we showed some basic ideas in Ghidra, and some scripts that we used in our experiments. In the second section, we will introduce our tool D3RE, which allows reverse engineers interact with a deductive database written Datalog.

## 3.1 Background: reverse engineering with Ghidra

One of the crucial parts of the malware analysis pipeline is how do we productively use Ghidra. In this section we will introduce the reverse engineering tool Ghidra, and how we would like to use it in our future works. Ghidra is a reverse engineering tool that allows the users to view the disassembled and decompiled code of a certain binary as shown in figure 3.2. In order to provide better visualization and improve user's experience, Ghidra has function tress, function call graph, and others, etc.

### 3.1.1 Loading Files and Starting Ghidra

Ghidra can be download on the following website [14]. To view a binary file, the user can simply create a new project and load the binary into Ghidra. One will then double click the binary file on the UI and Ghidra will ask you whether you want to analyze the file if it is the first time you open the current binary file on Ghidra. Once Ghidra finishes analyzing the binary file, the user will be able to view and explore the file based on their needs. An example is shown in figure 3.1.



trial
bison
challenge
gzip
password
re2c
redis-server
rsync
souffle

Opening project: /Users/jeffreyching/trial

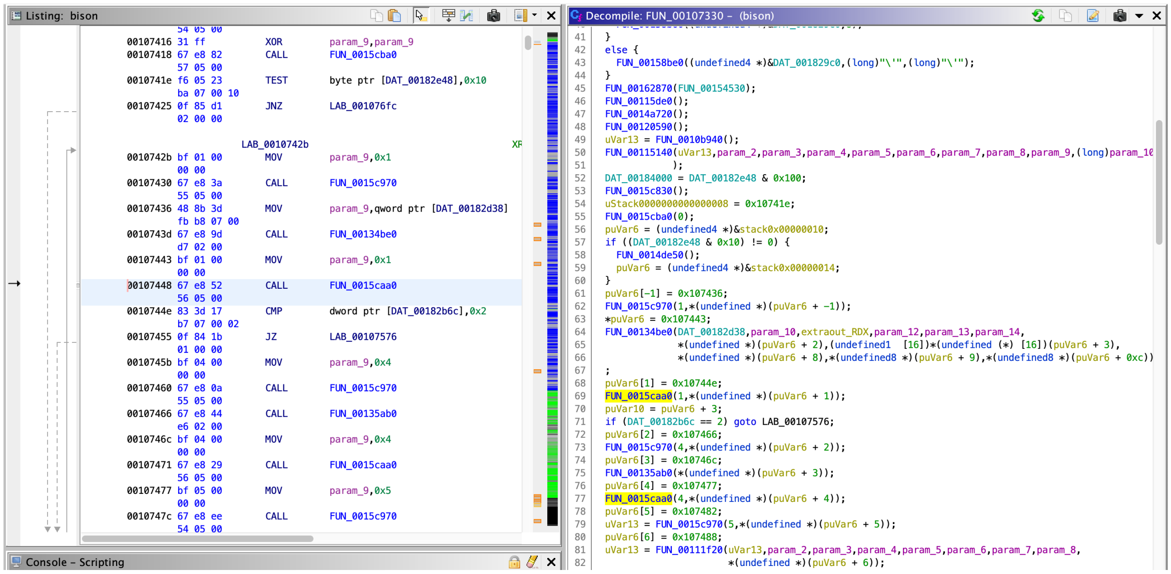
**Figure 3.1:** The figure is an example of the starting page of Ghidra.

### 3.1.2 Writing Scripts and Extensions

Ghidra is a powerful framework for open-ended exploration of properties of binaries. However, in isolation, Ghidra lacks facilities for automated analysis of binaries. Fortunately, Ghidra is an open-source tool with a rich API that allows users to extend almost every aspect of its UI.

#### 3.1.2.1 Scripts

Ghidra is written in java but includes a Python interpreter Jython that will call the java code in the back end. The current Ghidra Python scripting uses the Python 2.7 API. To start creating scripts in Ghidra, the user will click the green arrow button which is the script manager on the



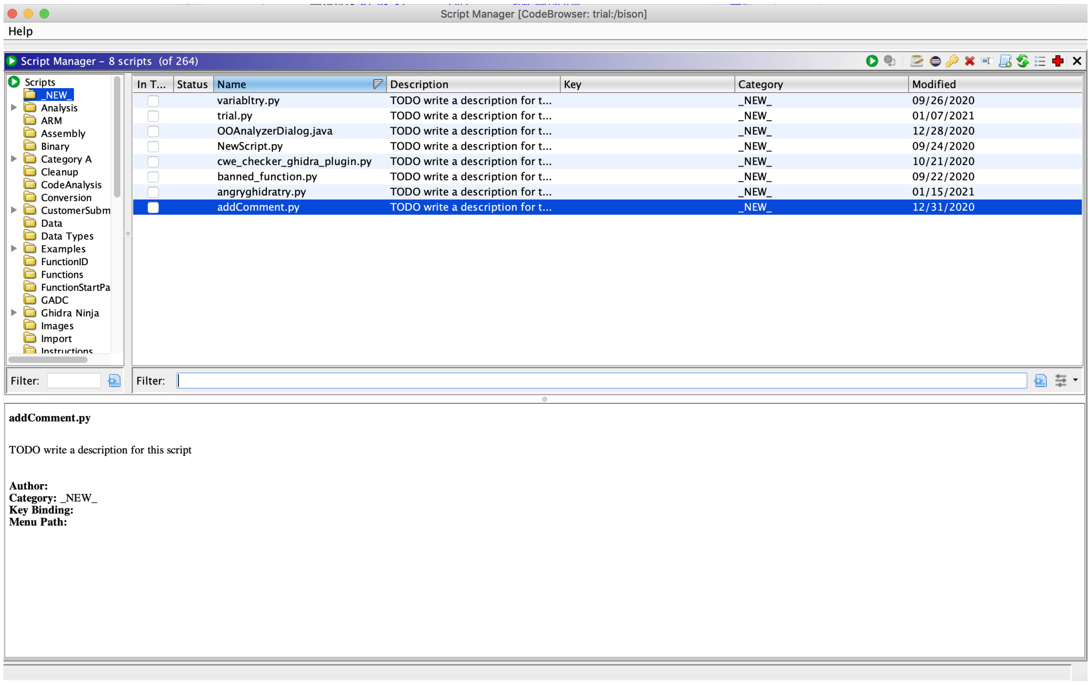
```
                    54 05 00
        00107416 31 ff          XOR      param_9,param_9
        00107418 67 e8 82       CALL     FUN_0015cba0
                 57 05 00
        0010741e f6 05 23       TEST     byte ptr [DAT_00182e48],0x10
                 ba 07 00 10
        00107425 0f 85 d1       JNZ      LAB_001076fc
                 02 00 00

                          LAB_0010742b                              XR
        0010742b bf 01 00       MOV      param_9,0x1
                 00 00
        00107430 67 e8 3a       CALL     FUN_0015c970
                 55 05 00
        00107436 48 8b 3d       MOV      param_9,qword ptr [DAT_00182d38]
                 fb b8 07 00
        0010743d 67 e8 9d       CALL     FUN_00134be0
                 d7 02 00
        00107443 bf 01 00       MOV      param_9,0x1
                 00 00
        00107448 67 e8 52       CALL     FUN_0015caa0
                 56 05 00
        0010744e 83 3d 17       CMP      dword ptr [DAT_00182b6c],0x2
                 b7 07 00 02
        00107455 0f 84 1b       JZ       LAB_00107576
                 01 00 00
        0010745b bf 04 00       MOV      param_9,0x4
                 00 00
        00107460 67 e8 0a       CALL     FUN_0015c970
                 55 05 00
        00107466 67 e8 44       CALL     FUN_00135ab0
                 e6 02 00
        0010746c bf 04 00       MOV      param_9,0x4
                 00 00
        00107471 67 e8 29       CALL     FUN_0015caa0
                 56 05 00
        00107477 bf 05 00       MOV      param_9,0x5
                 00 00
        0010747c 67 e8 ee       CALL     FUN_0015c970
                 54 05 00
```

```
  }
  else {
    FUN_00158be0((undefined4 *)&DAT_001829c0,(long)"\'",(long)"\'");
  }
  FUN_00162870(FUN_00154530);
  FUN_00115de0();
  FUN_0014a720();
  FUN_00120590();
  uVar13 = FUN_0010b940();
  FUN_00115140(uVar13,param_2,param_3,param_4,param_5,param_6,param_7,param_8,param_9,(long)param_10
              );
  DAT_00184000 = DAT_00182e48 & 0x100;
  FUN_0015c830();
  uStack0000000000000008 = 0x10741e;
  FUN_0015cba0(0);
  puVar6 = (undefined4 *)&stack0x00000010;
  if ((DAT_00182e48 & 0x10) != 0) {
    FUN_0014de50();
    puVar6 = (undefined4 *)&stack0x00000014;
  }
  puVar6[-1] = 0x107436;
  FUN_0015c970(1,*(undefined *)(puVar6 + -1));
  *puVar6 = 0x107443;
  FUN_00134be0(DAT_00182d38,param_10,extraout_RDX,param_12,param_13,param_14,
               *(undefined *)(puVar6 + 2),(undefined1  [16])*(undefined (*) [16])(puVar6 + 3),
               *(undefined *)(puVar6 + 8),*(undefined8 *)(puVar6 + 9),*(undefined8 *)(puVar6 + 0xc))
  ;
  puVar6[1] = 0x10744e;
  FUN_0015caa0(1,*(undefined *)(puVar6 + 1));
  puVar10 = puVar6 + 3;
  if (DAT_00182b6c == 2) goto LAB_00107576;
  puVar6[2] = 0x107466;
  FUN_0015c970(4,*(undefined *)(puVar6 + 2));
  puVar6[3] = 0x10746c;
  FUN_00135ab0(*(undefined *)(puVar6 + 3));
  puVar6[4] = 0x107477;
  FUN_0015caa0(4,*(undefined *)(puVar6 + 4));
  puVar6[5] = 0x107482;
  uVar13 = FUN_0015c970(5,*(undefined *)(puVar6 + 5));
  puVar6[6] = 0x107488;
  uVar13 = FUN_00111f20(uVar13,param_2,param_3,param_4,param_5,param_6,param_7,param_8,
                        *(undefined *)(puVar6 + 6));
```

**Figure 3.2:** The figure is an example of the disassembly and decompiled code for a binary file.

toolbar, and create new scripts. In the script manager window, the user can create a new script by clicking the red plus button. One will be able to choose the programming language of the script which can be either java or python. The default directory is Home/ghidra_scripts. Once the script manager is opened, the user can run the script by double-clicking the script they want to run or click the run script button shown in figure 3.3. There are some important built-ins that are accessible to all Ghidra plugins: currentProgram, which will interact with the database of the active or the current program in the code browser window, which is the same idea of current address.

While writing scripts, there are some suggested tags in the header comments, which will help one on organizing the scripts by labeling their corresponded functions on running the scripts. The followings are the headers: category, keybinding, menupath, toolbar. Category will allow the user to store into pre-existed or create a new category. Keybinding allows users to create hotkeys. Menupath and toolbar will allow users to locate the new scripts on the drop-down menu, and change the image for the top-level toolbar button. There are some tips on adding menupath and toolbar. Since users might have more than one script, the drop-down menu might



| In T... | Status | Name | Description | Key | Category | Modified |
|---|---|---|---|---|---|---|
| ☐ | | variabltry.py | TODO write a description for t... | | _NEW_ | 09/26/2020 |
| ☐ | | trial.py | TODO write a description for t... | | _NEW_ | 01/07/2021 |
| ☐ | | OOAnalyzerDialog.java | TODO write a description for t... | | _NEW_ | 12/28/2020 |
| ☐ | | NewScript.py | TODO write a description for t... | | _NEW_ | 09/24/2020 |
| ☐ | | cwe_checker_ghidra_plugin.py | TODO write a description for t... | | _NEW_ | 10/21/2020 |
| ☐ | | banned_function.py | TODO write a description for t... | | _NEW_ | 09/22/2020 |
| ☐ | | angryghidratry.py | TODO write a description for t... | | _NEW_ | 01/15/2021 |
| ☐ | | addComment.py | TODO write a description for t... | | _NEW_ | 12/31/2020 |

**addComment.py**

TODO write a description for this script

**Author:**
**Category:** _NEW_
**Key Binding:**
**Menu Path:**

**Figure 3.3:** The UI of how to write and run scripts on Ghidra.

be overcrowded, so some design should be done while creating these headers. Also suggestions are to add comments on the scripts, so the short description will be shown in the description box. This is just for organizing, since there are lots of scripts with similar functionalities.

### 3.1.3 Eclipse

Since Ghidra is programmed in a way that everything can be extended such as loaders, importers, and etc., extensions and scripts are important. We are going to introduce the IDE, Eclipse, for writing extensions, and scripts for Ghidra. Ghidra has Eclipse integration and is an IDE typically for Python and Java, which is suggested to be the platform to work with Ghidra extensions and scripts. Also, users need to make sure they have the correct JDK version installed. For Eclipse to integrate with Ghidra, we have to install the GhidraDev plugin. In Eclipse, click Help -> Install New Software, and click Add -> Archive on the screen. Find the file GhidraDev-2.1.1.zip(version can be different) under Ghidra(your Ghidra home folder)/Extensions/Eclipse/GhidraDev, and add it. Restart Eclipse, and you should see GhidraDev on the menu bar. GhidraDev in Eclipse provides several templates for development such as Ghidra Module Project, and Ghidra script. For example, in an extension project, things like scripts, data, and native components will be stored in the correspondent folders under the extension project. When the user finishes developing the extension, they can export it and test it in Ghidra.

### 3.1.4 Examples

In this section, I will give some example Ghidra scripts we have used in our experiments in section 3.2. The scripts are scripts from either GitHub [3] or some small testing scripts we wrote.

#### 3.1.4.1 non-xor Ghidra

In this script, non-xor finds xor find instructions that are not zeroing registers. In the following Ghidra Python script, we first have to get the AddressRanges. To start iterate over all the addresses, setting the variable to begin as the starting point is necessary. The first while loop will opt out until it finds the first instruction, and go to the for loop to find the instruction that is XOR. In the for loop, it loops over instructions and checking against XOR that we want to evaluate. Although the complexity is linear, however, since it has to check the property every time, we might be able to improve the performance. In section 3.2, we use Datalog instead of python. We found that Datalog allowed us to write these not only more succinctly because of the declarative nature of Datalog but also more efficiently the Datalog engine Souffle optimally compiles input programs to efficient relational algebra kernels to avoid unnecessary loops, the code is shown in figure 3.4. The comparisons will be shown in section 3.2.3.

```
ranges = currentProgram.getMemory().getAddressRanges()
for r in ranges:
    begin = r.getMinAddress()
    length = r.getLength()
    ins = getInstructionAt(begin)
    while(ins==None):
        ins =  getInstructionAfter(ins)
    for i in range(length):
        mnemonic = ins.getMnemonicString()
        if mnemonic == "XOR":
            operand1 = ins.getOpObjects(0)
            operand2 = ins.getOpObjects(1)
            if operand1 != operand2:
                print("{}␣{}".format(ins.address, ins))
                add_bookmark_comment(ins.address, str(ins))
        ins =  getInstructionAfter(ins)
        while(ins==None):
            ins =  getInstructionAfter(ins)
```

```
.decl nonzero_xor(EA:address)
.output nonzero_xor

nonzero_xor(EA) :-
    code(EA),
    instruction(EA,_,_,"XOR",Op1,Op2,0,0,_,_),
    Op1 != Op2.
```

**Figure 3.4:** non_xor Datalog

#### 3.1.4.2 Highlight and add Comments

One of the functions that we like to take advantage of in Ghidra is highlight and add comment, which we these functions can help user doing reverse engineering. For example, in the following code, the setPreComment function will add a comment before the instruction. We then add a comment base on the type of instruction. Besides, adding comments, Ghidra can highlight specific address or instructions base on the user's need. In chapter 3, we will use this scripts to visualize the analyzed results.

```
def set_comment (address, comment):
    setPreComment(address, comment)
def add_comment(code_manager, function_iterator):
    state = ""
    for func in function_iterator:
        ins_list = get_instruction_list(code_manager, func)
        fb = func.getBody() #function bound of func
        AI = fb.getAddresses(True)#address iterator
        print(func)
        for ins in ins_list:
            addressforcomment = ins.getInstructionContext().getAddress()
            print(ins,addressforcomment)
            state = check_instruction_type(ins)
            if (state!=None):
                print("hi")
                set_comment(addressforcomment, state+"trial")
```

#### 3.1.4.3 findcrypto

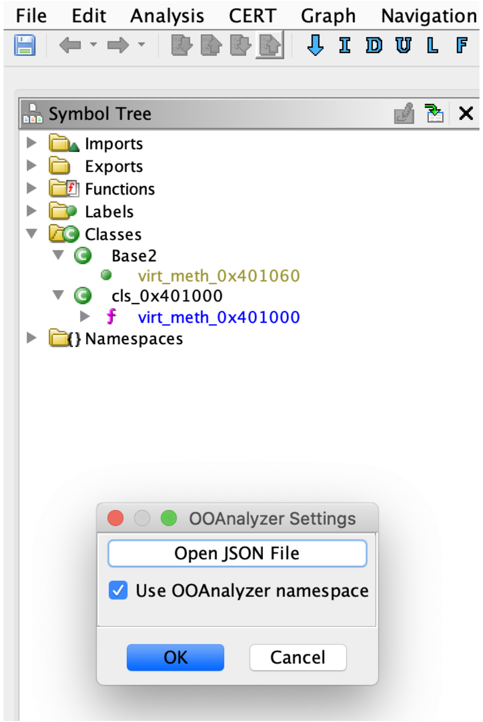
We found script that will look for common cryptographic constants[31]. The original script is in Python, which the script scans the binary for 256-segments of code. This script is also the one that we found it has runs slower than the Python scripts in Ghidra. With this example we will re-implement this script in Datalog and compared the run time and code size in section 3.2.3 as well.

#### 3.1.4.4 OOanalyzer

As introduced in section 2.6.4, we will show how to use the OOAnalyzer plugin in Ghidra. We take an example from CMU Pharos GitHub, ooex5. Before viewing it on Ghidra, we have to analyze the binary ooex5 on OOAnalyzer on docker since docker will be the best way to avoid issues caused by different working environments. We can also set docker to interactive mode, so all the new documents created during the analysis can be synchronized to the home directory. An example OOanalyzer analysis process can be found in figure 3.6. After analysis, we will get the results in the form of JSON. The OOAnalyzer extension can be found on their GitHub repository. To install the extension on Ghidra, the version of OOAnalyzer has to match the version of Ghidra. The users have to download the source file into Ghidra->extension and click install the extension in the starting user interface of Ghidra.

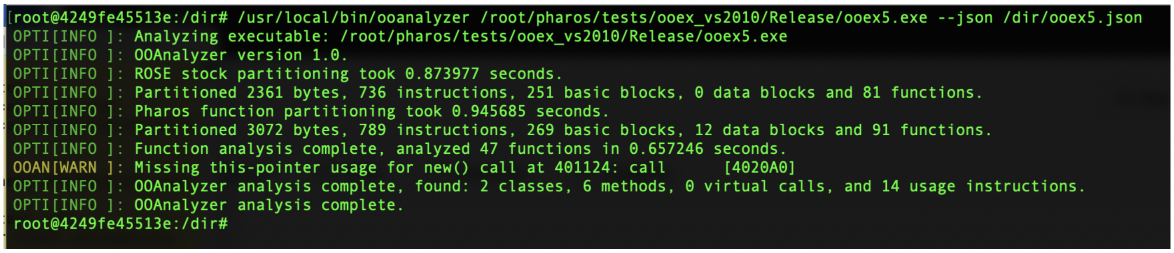
After restarting Ghidra, one can see CERT on the function tab. To view the class recovery results, the user should click CERT->OOAnalyzer, then open the JSON file produced by running OOAnalyzer on docker. Finally, click OK and the recovered class can be seen in Symbol Tree as shown in figure 3.5. The above is only a demo of how to combine the static analysis tool with

Ghidra. In conclusion, for now we have to run the analysis and visualization separately, which is more time consuming and requires more steps. However, we understand that it is hard to write a fully automated analysis tool along with visualization, in the future, we hope to come up with a more systematic way of combining analysis and visualization with simpler steps.



☑ Use OOAnalyzer namespace

**Figure 3.5:** The GUI of OOanalyzer



```
[root@4249fe45513e:/dir# /usr/local/bin/ooanalyzer /root/pharos/tests/ooex_vs2010/Release/ooex5.exe --json /dir/ooex5.json
 OPTI[INFO ]: Analyzing executable: /root/pharos/tests/ooex_vs2010/Release/ooex5.exe
 OPTI[INFO ]: OOAnalyzer version 1.0.
 OPTI[INFO ]: ROSE stock partitioning took 0.873977 seconds.
 OPTI[INFO ]: Partitioned 2361 bytes, 736 instructions, 251 basic blocks, 0 data blocks and 81 functions.
 OPTI[INFO ]: Pharos function partitioning took 0.945685 seconds.
 OPTI[INFO ]: Partitioned 3072 bytes, 789 instructions, 269 basic blocks, 12 data blocks and 91 functions.
 OPTI[INFO ]: Function analysis complete, analyzed 47 functions in 0.657246 seconds.
 OOAN[WARN ]: Missing this-pointer usage for new() call at 401124: call      [4020A0]
 OPTI[INFO ]: OOAnalyzer analysis complete, found: 2 classes, 6 methods, 0 virtual calls, and 14 usage instructions.
 OPTI[INFO ]: OOAnalyzer analysis complete.
 root@4249fe45513e:/dir#
```

**Figure 3.6:** Docker for ooanalyzer

## 3.2 D3RE

From the related work and the literature reviews in chapter 2, we noticed that reverse engineers heavily rely on reverse engineering tools like IDA or Ghidra. The process of reverse engineering can be very subjective, which means that the perspectives or methods used will be strongly related to the direction or goals the users want to achieve. Ghidra is a good solution for visualizing decompiled files and understanding the relations between function calls. However, since our goal is to "analyze" binary files, we want to add on some plugins as well as design our binary analysis tools. We want to create a fast, useful, and interactive tool, which combines the analysis along the visualization part together.

In this chapter, we would like to talk about the paper we submitted in BAR 2021 [44]. In the paper, we switched the Ghidra backend to the Datalog backend, which means the user will be able to interact with the Datalog backend and input the rules and logic based on their need. Finally, since the whole purpose of the project is to improve the user experience on reverse engineering, the results will be sent back to the Ghidra frontend and be visualized.

In the paper, we are trying to build a tool called Declarative Demand-Driven Reverse Engineering (henceforth D3RE). D3RE is designed to interact with a deductive database by user inputs. The rules in the database are inductively computed relations over facts of a certain binary. Deductive databases have also enabled several recent advances in binary analysis demonstrating both efficiency and robustness over conventional techniques. For example, the Datalog based disassembler ddisasm achieves both faster and more precise disassembly than other state-of-the-art disassemblers, and OOAnalyzer uses Prolog to enable declarative recovery of classes from compiled C++ code.

To better understand and prove our tool can reach a certain level of efficiency, we will give some examples and background on our tool D3RE. More details on the experiments and background will be introduced in the following sections.

### 3.2.1 Introduction

Binary reverse engineering is the process starting by the user input some binary files and employs various reasoning principles to explicate its behavior when executed as code. As mentioned in the related work chapter, different types of reverse engineering have been applied in researches(assisted by machine learning, static analysis, etc.). Most of the reverse engineering tasks are partially automated assisted by decompiler or disassemble, however, it is often impossible to turn full automation: the extreme semantic expressivity afforded to binaries (including encrypted code, stripped symbol tables, etc..) often necessitates open-ended exploration and the processes are different case by case.

To rapidly interact with a binary, RE practitioners often use reverse engineering tools such as Ghidra [14], IDA Pro [20], or Radare2 [32]. Since reverse engineers are expected to be expert users, and often skilled programmers, tools like Ghidra and IDA enable the engineers to plugin extensions on doing reverse engineering. Tools like Ghidra have a broad range of popular extensions that exist for several tools which perform different functionalities such as static analysis static analyses [18, 38], and interacting with debuggers. These extensions are created to make the reverse engineering process more complete and organized.

However, there are less tools that enable user to do reverse engineering interactively. Users have to switch between tools and create a lots of scripts. The goal of these tools is to allow an reverse engineer to quickly explore the binary and visualize it in an interactive way via a GUI or command-line interface(CLI).

In this chapter, we argue that Datalog, a deductive databases serve as a natural abstraction boundary between RE tools and logical inference tasks over binaries. We envision a future in which a reverse engineer interactively explores a binary using reverse engineering tools while simultaneously inputting rules by querying logical properties in a declarative style. In D3RE, a reverse engineer interacts with a deductive database by giving inputs to a rule-based deductive inference system written in a declarative language such as Datalog. Rules inductively compute relations over facts about the binary. In our vision, D3RE allows reverse engineers to interactively compute with and visualize the results of queries over these deductive rules.

We see D3RE as the state of the start of being a natural extension of several observations. First, many existing reverse engineering tools assemble databases to index various properties (e.g., addresses, symbols, etc...) of binaries for quick exploration. Deductive databases further allow reverse engineers to write arbitrary logical queries which are computed maximally efficiently via, e.g., compilation to relational algebra kernels as done in Souffle.

In the following, we describe our process on implementing a prototype tool, D3RE. D3RE allows the user to interactively define and calculate queries of arbitrary complexity over large binary files and visualize the results on Ghidra. Analogously to the indexing and analysis operations provided by Ghidra (and other RE tools), D3RE invokes ddisasm once to build an initial database.

To implement D3RE we have designed an interface, which we call the mediator, that sits between a traditional Datalog solver and an RE tool. We briefly formalize this interaction between the reverse engineering tool and logic solver in Section III of the paper [44] we submitted in BAR 2021, which we will not talk about it in this thesis. Using this formalism, we describe how D3RE readily enables a broad range of binary analyses and sketch a vision for how we believe D3RE will prove to be a natural ergonomic for reverse engineering.

### 3.2.2 Overview of D3RE

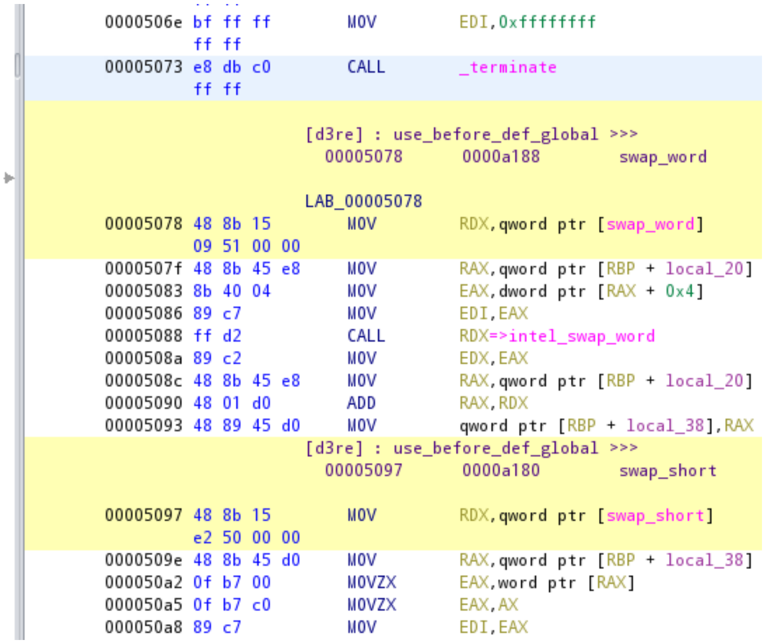
There are lots of existing reverse engineering tools that assemble databases to index various properties. Deductive databases also allow users to write arbitrary logical queries through a certain engine, and enable several recent advances in binary analysis, demonstrating both efficiency and robustness over conventional techniques.

Here, I will give an idea of the formalism, semantics, and the structure of D3RE; the high-level structure can be found in figure 3.8. In D3RE, the users will build queries based on the decompiler ddisasm. The rules will be built on top of the fact of which ddisasm created when the ddisasm decompiled the binary. Ddisasm already includes facilities to parse object files and transform them into the input databases that are required by Souffle. The current tool is a prototype written in Python, and the high-level architecture is shown in figure 4.3.In the figure, there are REPL, Ghidra, and Metadatabase. For now, the Read–Eval–Print Loop (REPL) is a CLI environment that will take in new rules, by loading new files or editing existed files.The metadatabase takes the form of a server which accepts Datalog programs to run to a fixed-point. After each run, the metadatabase will link the output facts and associate them with the program. In our experiments, we refer to this as "caching." D3RE will cache the database, so if one has a subsequent query, D3RE will find the best database with the most facts to start from. The metadatabase will track through the external database (EDB) and look for the most related database which can be used for analyzing the binaries. Broadly, we want to maintain the metadatabase, where the metadatabase is a daemon that calls out to Souffle to do the relevant work and show the results in Ghidra, along with accepting user inputs from the CLI, to allow clients incremental reuse of previously computed databases. (An example will be given in the next section.)

Since Datalog is monotonic, we can say that the users will be able to add on their own rules based on the pre-calculated result or previous rules. A more detailed formalism can be found in section III of our paper [44], which, with this formantics, will allow us to analyze a broad range of binaries, as well as visualization. Besides analysis, visualization is also part of the project. In this project, we found a "platform" that can effectively work between Datalog and the front end GUI: Ghidra. D3RE is implemented in two parts: a REPL that will communicate with the Ghidra frontend, and a background service that will manage the metadatabase and the Datalog engine. The REPL is now communicating with Ghidra using a Ghidra extension called ghidra-bridge [13], however, it has some security issues and we plan to replace it imminently with an extension using protocol buffers. The reason we need ghidra-bridge is that since Ghidra Python is using Python

2, however, some of the scripts will need Python3 packages, and it will be hard to import those packages without help from a good extension tool. The metadatabase will take in new rules from the REPL (CLI); the communication is done by protocol buffer.

In the future, we are anticipating to expand the project, which, using protocol buffer, will give us scalability during the expansion. Overall, after each analysis, the users can choose to highlight or add comments on the instructions or addresses that they want to keep track on. Although D3RE is still in the early stage of the design process, it has shown some promising results. (Figure 3.7 is an example showing how the current state of D3RE can be visualized on Ghidra.)



```
0000506e bf ff ff        MOV        EDI,0xffffffff
         ff ff
00005073 e8 db c0        CALL       _terminate
         ff ff

                   [d3re] : use_before_def_global >>>
                     00005078       0000a188        swap_word

                   LAB_00005078
00005078 48 8b 15        MOV        RDX,qword ptr [swap_word]
         09 51 00 00
0000507f 48 8b 45 e8     MOV        RAX,qword ptr [RBP + local_20]
00005083 8b 40 04        MOV        EAX,dword ptr [RAX + 0x4]
00005086 89 c7           MOV        EDI,EAX
00005088 ff d2           CALL       RDX=>intel_swap_word
0000508a 89 c2           MOV        EDX,EAX
0000508c 48 8b 45 e8     MOV        RAX,qword ptr [RBP + local_20]
00005090 48 01 d0        ADD        RAX,RDX
00005093 48 89 45 d0     MOV        qword ptr [RBP + local_38],RAX
                   [d3re] : use_before_def_global >>>
                     00005097       0000a180        swap_short

00005097 48 8b 15        MOV        RDX,qword ptr [swap_short]
         e2 50 00 00
0000509e 48 8b 45 d0     MOV        RAX,qword ptr [RBP + local_38]
000050a2 0f b7 00        MOVZX      EAX,word ptr [RAX]
000050a5 0f b7 c0        MOVZX      EAX,AX
000050a8 89 c7           MOV        EDI,EAX
```

**Figure 3.7:** Ghidra with highlights and comments declaratively specified to output results inferred via D3RE for our example

### 3.2.3 Experiments and Studies in D3RE

In this study, we want to look into whether D3RE can be the tool for professional reverse engineers to do their daily tasks. Designing a good tool is a challenging problem, which we

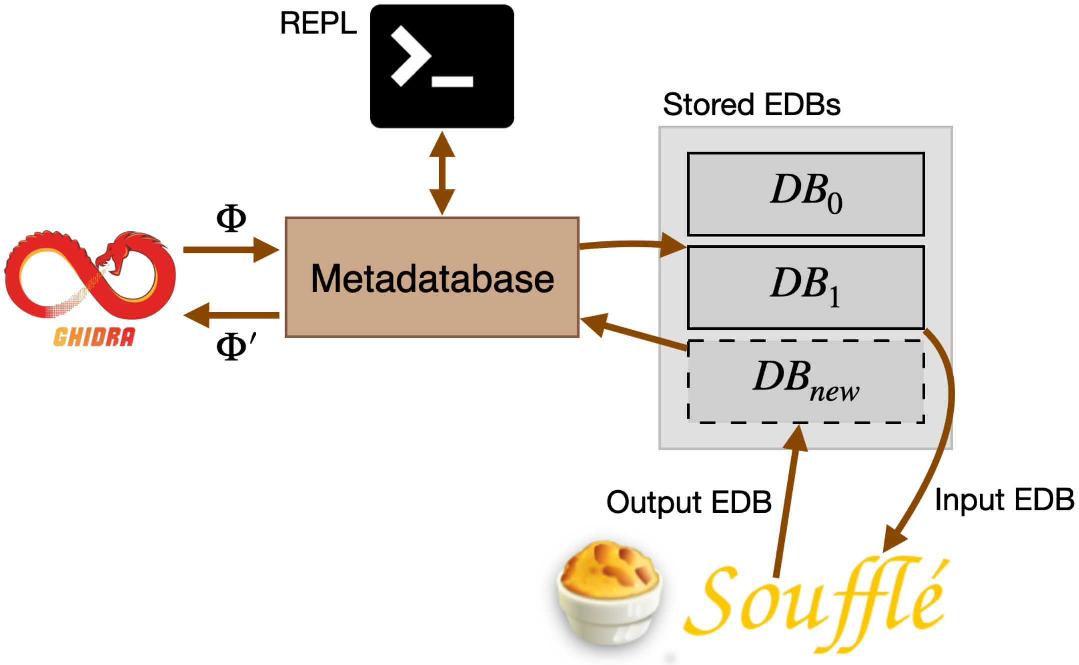


**Figure 3.8:** High-level components and their interactions in D3RE. The metadatabase will communicate with REPL through protobuff and communicate with Ghidra through Ghidra_bridge [13]. The metadatabase will track through the external database (EDB) and look for the most related database.

plan to eventually do ask experts in reverse engineers on their preferences and what kind of tools will they like to input in D3RE. We modified some scripts on the awesome-ghidra GitHub repository [3] and written some scripts in Python or Java, which most of the scripts are written in Python. The first three are relatively small and find instructions that match a specific template, e.g., non-xor finds xor instructions that aren't zeroing registers shown in the previous section, and overflow heuristically searches for potential overflows in calls to common functions such as strcpy. Our largest was findcrypto, which looks for common cryptographic constants. In the future, we plan to do user studies on experienced users on finding vulnerabilities and malware analysis with the assist of reverse engineering, binary analysis, and symbolic execution.

**Table 3.1:** Script size (lines of code) of Ghidra script (Python) vs. D3RE

| | Ghidra Python | D3RE |
|---|---|---|
| non-xor | 33 | 8 |
| basicblk | 37 | 4 |
| overflow | 60 | 18 |
| findcrypto | 166 | 45 |

#### 3.2.3.1 Experiments and Results in D3RE

In theory, it seems D3RE is promised to have good results on reverse engineering. Although D3Re is still a work in progress tool, we did several hypotheses that we want to clarify before moving on. The basic idea of these experiments is that whether D3RE can replace Ghidra scripts. First, we want to make sure D3RE is the tool that can provide the necessary building blocks to enable replacing the currently existing Ghidra scripts written in Python or Java. Second, we want to test the performance of D3RE and compare the performance with the Ghidra Python scripts. Lastly, we want to test the difference between the caching-based approach and the none-caching based approach.

Qualitative Results of our Replication Study: Since we are using Datalog and functional programming, the user will have to have a proper understanding of the context and relevant knowledge. Comparing Datalog and python, D3RE enabled us to succinctly write equivalent implementations of each Ghidra script which we rewrote the script with less Datalog code. In table 3.1, we can see the difference between the lines of codes between Python scripts in Ghidra and Datalog. The reason is that Datalog will eliminate the need for certain things that are necessary for languages like Python and Java. For example: looping over instructions and checking against a type that we found in our evaluation scripts. We believed the reason D3RE can be represented in fewer lines is that its ability to directly use relations from ddisasm.

Quantitative Results of our Replication Study: We expect that the tool D3RE will have better results compared to the Ghidra Python scripts since D3RE is based on a high-performance Datalog solver. We benchmarked Ghidra vs D3RE on six binaries from ddisasm test suite. To test the Ghidra scripts, we use Python standard time at the start and the end of the script. We evaluated the corresponding Datalog program by using Souffle's internal performance timers. In the table 3.2, it compares the runtime of each Ghidra script versus its corresponding implementation in D3RE. The single occurrence of – indicates that Ghidra did not finish within an hour. In most of the cases, the results have reached our expectation which D3RE outperformed the Ghidra Python scripts. We believe it is because that desgin of D3RE allowed us to leverage useful relations from ddisasm since it is built on top of it. Comparing D3RE and Python, we found that in the Python scripts, we need to write naive loops to loop over sets of functions or symbols in order to locate certain properties. However, on the other side, D3RE allowed us to write more succinctly which Datalog naturally aggregates results. Moreover, D3RE can execute much more efficiently, because Souffle optimally compiles input programs to efficient relational algebra kernels that loop only when necessary.

Despite the success of D3RE, there are still limitations that could cause performance issues. For example, in our last example, findcrypto looks for common cryptographic constants. The crypto script scans for the binary for 256-segments of code, D3RE is built on Souffle, which supports 64-bit primitive ints, but not 256-byte sequences. On some large binary files, Ghidra will fail to finish within an hour, which makes D3RE a great success on the matter of time of completion.

Evaluating End-to-End Behavior in Subsequent Invocations: Last, to understand the effect of caching via repeated calls to D3RE, we ran four different tests in a row using the caching-based approach and without caching. As each query builds on the previous, we expect caching to reduce the amount of work and will also reduce the runtime. We found out that caching makes a great difference according to table 3.3. Caching is expected to reduce the number of works and reduce the runtime. Overall, we found rule caching was especially important on larger binaries versus sequential runs, justifying our choice to structure the metadatabase as a graph.

**Table 3.2:** Running time of Ghidra scripts vs. equivalent implementation in D3RE (all numbers in seconds).

| | bison | souffle | gzip | re2c | redis | rsync |
|---|---|---|---|---|---|---|
| non-xor Ghidra | 3.569 | 107.5 | 2.205 | 3.903 | 10.52 | 3.050 |
| non-xor d3re | 0.518 | 6.515 | 0.097 | 0.756 | 1.306 | 0.486 |
| overflow Ghidra | 0.370 | 0.247 | 0.600 | 0.240 | 0.760 | 0.180 |
| overflow d3re | 0.617 | 0.319 | 0.051 | 0.094 | 0.095 | 0.044 |
| basicblk Ghidra | 340.6 | – | 4.664 | 472.1 | 1806 | 107.4 |
| basicblk d3re | 0.539 | 7.13 | 0.094 | 0.812 | 1.433 | 0.571 |
| findcrypt Ghidra | 0.207 | 1.033 | 0.224 | 0.214 | 0.475 | 0.289 |
| findcrypt d3re | 1.287 | 14.53 | 0.224 | 1.701 | 2.938 | 1.186 |

**Table 3.3:** Runtime of successive invocations to D3RE with (Cached, C) and without (Sequential, S) rule caching.

| | ddisasm | stack_var | heap_var | static_var | unl_static |
|---|---|---|---|---|---|
| souffle C | 170 | 11.88 | 58.35 | 5.008 | 0.039 |
| souffle S | 170 | 11.79 | 66.02 | 67.00 | 66.52 |
| bison C | 7 | 0.932 | 1.409 | 0.545 | 0.022 |
| bison S | 7 | 0.934 | 1.916 | 2.122 | 2.075 |
| re2c C | 9 | 1.457 | 4.417 | 0.704 | 0.025 |
| re2c S | 9 | 1.494 | 5.257 | 5.449 | 5.458 |
| redis C | 11 | 1.918 | 2.544 | 1.302 | 0.025 |
| redis S | 11 | 1.919 | 3.525 | 3.712 | 3.726 |
| rsync C | 8 | 0.766 | 0.908 | 0.481 | 0.028 |
| rsync S | 8 | 0.783 | 1.325 | 1.423 | 1.384 |

# 4 | Assemblage

Reverse engineering tools like Ghidra and IDA allow malware analysts to interactively discover properties of binaries in an open-ended way. However, these tools are challenging for users to interact with, and there is an inherent tension between automatability and open-ended exploration. We want to work with various types of binaries on differently built-on operating systems as well as written different types of languages. To be more precise, the purpose of assemblage is to study the properties of binaries at a scale that exists using a variety of compiler tool chains. For example, using LLVM and GCC compiler on compiling C++ and C repositories.

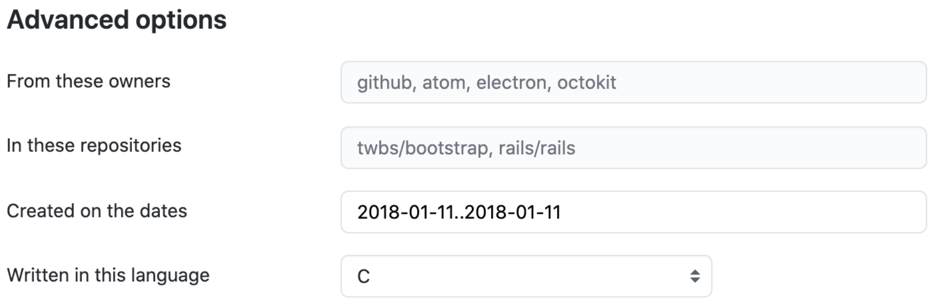
To go further on the research, collecting open-source repositories online massively and build them will be necessary. Thanks to modern software tools like GitHub, GitLab, and Bitbuckets, we can collect a variety of open-source projects for our data sets. Down the line, we are considering using this dataset as input to a machine learning algorithm or more, but that is explicitly out-of-scope for this thesis. However, we do validate that Assemblage is working and found several interesting insights of it as well as some problems to be solved while expanding Assemblage. The purpose of Assemblage is to collect a variety of binaries. Since we are creating a analysis tool D3RE, we need a data provider to test D3RE. Assmeblage will play the data provider role in the pipeline, which feed D3RE data to do analysis.

## 4.1 Methodologies and ideas

To understand the process of reverse engineering, and simulate certain situations in real-world circumstances we need a lot of data that we can compile them using various compilers and different strategies. To achieve this goal, we found a database GHTorrent which has scraped a lot of repositories from GitHub and will fulfill our need in the current stage. GHTorrent not only gets the URLs, but also push, pull, fork, and other related information in GitHub repositories. However, since GHTorrent is written in ruby and can be improved, in long term, we have the idea to build our scraper based on GHTorrent with modifications. The idea of the scraper is to search for repositories that fulfill our requirements and perform corpus compilation where we will most likely use LLVM or GCC compilers. Finally, we want to list all the compiled repositories with its corresponded binaries on a web front end for further research purposes.

## 4.2 Github API scraping

The goal of the task is to collect, binary files, so we aim to collect as many C and C++ open-source repositories as possible. We figure out that GitHub will be the best source to collect data, which will have a good amount of C and C++ programs and examples. There are a variety of ways on building the scraper, including using text related strategies and API. To make our work easier, we choose to build the scraper in python, which provides a large amount of scraping packages and more important there are GitHub API written in Python. The scraper will be mostly assisted by pygithub using GitHub API, which will provide us advanced searching methods within GitHub. GitHub has provided a good amount of tools for searching data, including time constraints, languages, and ratings. The web version of the GitHub API is shown in figure4.1. In the web version, we will be able to do some testings and check out how the search works, and understand how to use the GitHub API for scraping the repositories we need.



Advanced options

From these owners: github, atom, electron, octokit

In these repositories: twbs/bootstrap, rails/rails

Created on the dates: 2018-01-11..2018-01-11

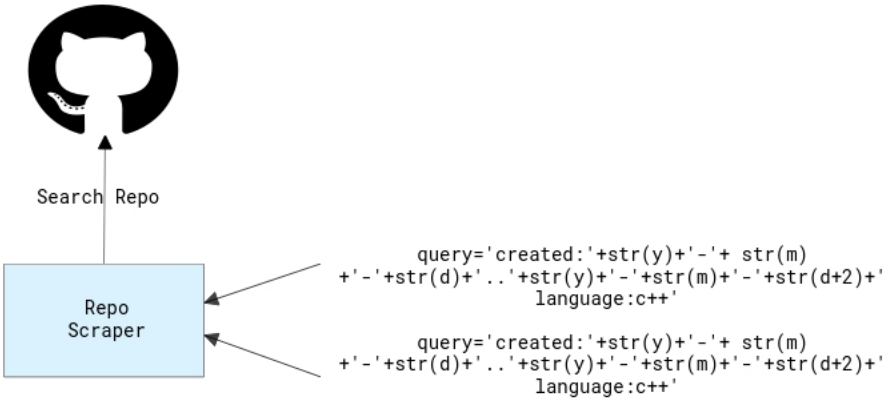
Written in this language: C

**Figure 4.1:** The figure is an example how github advanced search works in the form of web user interface.

However, while GitHub API brings convenience, it also brings challenges. Github API will make the whole searching process a lot easier, but it has rate limits. Github allows 30 searches per minute, and only 40 repositories can be found during each search. Also, the API only shows 1000 results per search to prevent traffics we believe, which means we have to divide our search into parts. For example, a user wants to search all the repositories written in C in 2018, which has about 235,000 repositories in total. The user would not be available to get everything in one search. To get all the repositories, the user will have to shorten the time range. With some researches, the repositories created in a day in language C or C++ is about 1000 which is about the maximum we can take in one search. In order to "play" with the rate limit, and most efficiently to use the rate limits, the code will be weird. For example, since writing the scraped repositories take time, the rate limit might be restored, so we can switch to the new rate limit to avoid possible sleep. The idea of coding this scraper will be to replace or sleep whenever it almost reaches the rate limit. Although this seems to be on the strategy, we still hope to find other more optimal ways on avoiding rate limits.

## 4.3 Pipeline

The pipeline consists of three different stages, web scraping, repositories cloning, and building files. Since we are scraping both C and C++ repos, we have to provide two queries. As shown in figure 4.2 (which will iterated by for loops), the queries consist of the date and the language we are searching. The current pipeline is based on using the database GHTorrent.



```
      query='created:'+str(y)+'-'+ str(m)
+'-'+str(d)+'..'+str(y)+'-'+str(m)+'-'+str(d+2)+'
             language:c++'

      query='created:'+str(y)+'-'+ str(m)
+'-'+str(d)+'..'+str(y)+'-'+str(m)+'-'+str(d+2)+'
             language:c++'
```

**Figure 4.2:** The idea of the repo scraper. We take in the query with specific dates and languages, and search the repos in gothub.

### 4.3.1 Methodology

The web scraper allows only 30 searches in one minute, and 40 repositories in one search. In order to avoid the rate limit, the scraper will wait in two circumstances: reach 25 searches, or reach the limit of extracting repositories after one search. as shown in figure4.3. To use the time efficiently, all the functions in the scraper and cloner will be written in the async function, which the asyncio will schedule the works for us. However, since we are using GHTorrent right now, it will temporarily solve the issue of rate limits since we already got all the URLs we need.

### 4.3.2 GH torrent

According to the GitHub home page, there are currently 100+ million repositories and 56+ million developers. Due to a large amount of data, GitHub has certain rate limits. According to the paper, GitHub allows 5,000 requests per hour, and the generation rate is already higher than

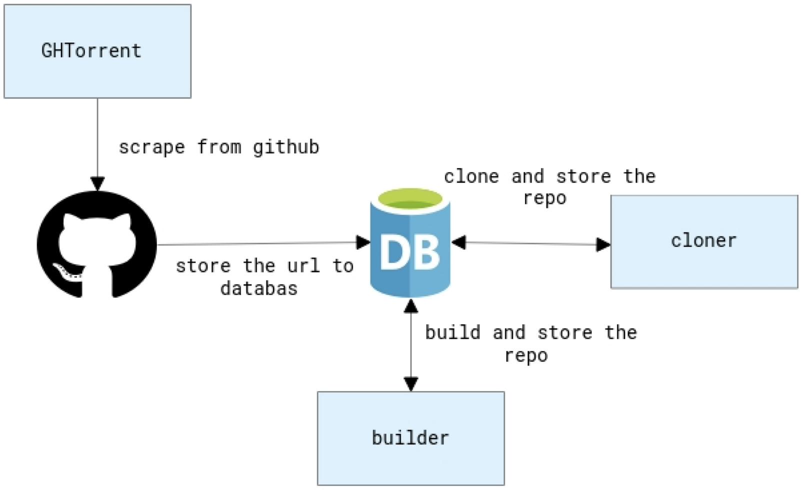


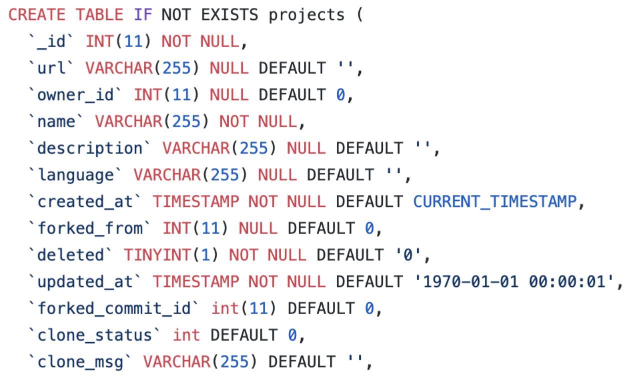
**Figure 4.3:** The ideal pipeline for scraping, cloning, and building the repositories.

the rate allowed. Eventually, we have to enable multiple users to retrieve data in parallel.

While designing our web scrapper, and open-source collector, there was an existed project called GHTorrent which seems pretty successful. We decide to take a look into the project and take the data they collected in this stage, and hopefully to modify GHTorrent and have our scraper. Although both GHTorrent and our ideas are targeting to scrape GitHub, we have different perspectives on scraping Github. They focused on getting all the events from including forks, pull requests, commits, and other information. They provide a database schema in the paper to show how they collect the data and store the data. GHTorrent will cache the data into MongoDB database However, we only want the repository data and the language they are written with. We have a relatively easier task on scraping, but the big ideas are similar. By fetching URL in the database, we can clone the repositories. However, since the database is old, certain repositories are either removed or switched to private. This is also another interesting problem to look into as well. How many repositories are still existed, and how many repositories can be successfully built with simply make commands.

### 4.3.3 Sqlite Schema

After cloning the repositories from GitHub, we will store all the repositories in the database. For future extensions, we have designed a database schema that we think will be the most appropriate and useful. At this stage, we probably will stick to the database schema from GHTorrent. The database schema is shown in figure 4.4. The most important items we need are _id, URL, name, language, created_time, and clone_status. We create the _id: management purposes in the future, URL and name: cloning repositories, language: since we have C and C++, we need to identify what the repositories are written with, created_time: tracking purposes, clone_status: for tracking whether the repository is cloned or not, buil_status: for tracking the current stage on building repositories and error messages. The error messages are crucial and important, while the unsuccessful building might be caused by missing packages, or we need specific commands to builds. By collecting these messages, we can tackle these problems by switching our design schema by adding more patterns.



```
CREATE TABLE IF NOT EXISTS projects (
  `_id` INT(11) NOT NULL,
  `url` VARCHAR(255) NULL DEFAULT '',
  `owner_id` INT(11) NULL DEFAULT 0,
  `name` VARCHAR(255) NOT NULL,
  `description` VARCHAR(255) NULL DEFAULT '',
  `language` VARCHAR(255) NULL DEFAULT '',
  `created_at` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `forked_from` INT(11) NULL DEFAULT 0,
  `deleted` TINYINT(1) NOT NULL DEFAULT '0',
  `updated_at` TIMESTAMP NOT NULL DEFAULT '1970-01-01 00:00:01',
  `forked_commit_id` int(11) DEFAULT 0,
  `clone_status` int DEFAULT 0,
  `clone_msg` VARCHAR(255) DEFAULT '',
```

**Figure 4.4:** The database schema of GhTorrent

### 4.3.4 Building Repositories

With the database GHtorrent, we can scrape the repositories base on the URL. Currently, we tried to build the repositories by running the make command. We build a coordinator and worker with ZMQ, which the cooedinator will distributed works to the worker, and the worker can clone and build the repositories. When we get a new repository we will check if a makefile (Makefile) exists, if it exists, it will simply execute the make command. If the build succeeds, it will return 0, or it will return error messages and stored back to the database. The build process is expected to be expanded in the future, which might be able to analyze the readme file to have a better understanding of how to proceed with the build process.

## 4.4 Insights of Assemblage

As we use GHTorrent as our database for Assemblage, we have successfully clone a number of repositories. Until now the lab (with one PhD and other undergraduate students) has Assemblage start working and reach a number of clones and builts. Total repositories cloned is 1872, and total built repositories is 27. We organized some reasons that clone and built failed.

Since GHTorrent is an old database, some repositories have already been closed or turned private which leads to clone failed. We want avoid this kind of errors as little as possible. On the other hand, since we are only checking whether the repositories contains makefile (Makefile), it is often that the building process will fail. For now, we clone all the URLs included in the GHTorrent, and lots of them are not C or C++ files. In the future, we want to improve pre-analysis to determine when they're C or C++. In such cases, and other buildable cases that we're missing. As for those failed building with makefile (Makefile), it is likely cased by missing pacakges or customized makefiles. Besides these more strategy related error, still other errors including syntax errors in code or makefiles, etc.

We are still adding more functions on Assemblage, and make the built process more generalized. Our next next goal will be on revising the GHTorrent database, for example: remove homework repositories, and outdated repositories, to lower the fail rate of cloning process, and maybe analyze the readme file to improve the building process.

## 4.5 Combining D3RE and Assemblage

The environment that we are creating are the foundation to study how reverse engineering will perform their work across various compilers, which in C and C++ are gcc and clang. As mentioned in chapter 4, we cloned the repositories and build. In the future, we will build the repositories in gcc and clang. After that, we can take the binary files, run D3RE and study the results. We expect to see different results and study what can we do with the results. Although it seems to be straightforward, building open-source repositories won't be a easy task. Also since D3RE is using gtirb as the IR, we will need a gtirb extension on Ghidra in order to load the file. However, the version will always be the challenge part of combining different open-source tools, which gtirb and Ghidra will have to be the same version. Currently, D3RE and Assemblage haven't been combined , since there are some version differences between the open-source tools that will cause problems that we can't fix yet. Also Assemblage is still in the early stage, we have limited data and are still running the scraper to get more.

# 5 Future and user studies

Since the project is still in an early stage, there will be a lot of future work waiting. Here, we plan to separate the future goal into two different categories, the repo scraper and building our own analysis tool. As for building the tool, we propose an idea of visualization, and possibly do user studies. The user studies will be on how our tool can improve the experience of reverse engineering and static analysis as well as how the experienced user do reverse engineering and how specific information can help them.

## 5.1 Visualization vs Current

In chapter 3, we introduced a declarative way of analyzing the binary files. We want to try some other examples to list out strategies for improving the visualization of D3RE on Ghidra. With the results of the 3 short experiments (qualitative, quantitative, and cache) in the D3RE paper, it seems d3re has a promising future in analyzing binary files. However, we want to have reverse engineers interactively explore a binary using a reverse engineering tool while simultaneously querying their own rules into the CLI of D3RE in a declarative style.

During our installation of D3RE, particularly ddisasm, multiple errors will pop out while working on different working environments (Linux, MacOSX). Some of the errors will need some tricks to tackle down, and others just take plenty of effort. In order to avoid these burdens, we create a dockerfile to run all the required packages for ddisasm on docker. Besides writing actual rules in Datalog on D3RE, we will combine and modify some existing scripts or plugins on Ghidra In the D3RE paper, our goal is to test some methodologies for designing the user interface of the tools.

In D3RE, the user can start their analysis based on a set of Datalog rules built above ddisasm, a Datalog-based disassembly engine. Since Datalog is monotonic, we can evaluate the program base on the results of the previous program. Moreover running ddisasm once allows pre-populating a large set of relations for which has a lot of interesting facts of the binary file.

Besides the rules based on ddisasm, the user can input their own rules. This allows the user to input rules based on their own needs. For example, the user can input rules targeting uninitialized variables, possible buffer overflow, etc. In D3RE, the communication between the logical rules and the state of the reverse engineering tool is reconciled by input and output tables reverse engineers can write queries that consume the state of the reverse engineering tool such as currentAddress or the currently-selected address as input relations, perform logical inference. Currently, our REPL allows loading rules by loading new files. If the user wants to add a new rule, the user will create a new file, which can be edited in the future as well. In the current settings, D3RE can highlight or add comments which will be displayed on the data marked on Ghidra.

As for the search range of D3RE, the user can search through the binary file. However, if the user wants to focus on a certain function or address set, the user can set the range of the code along with the rules in the same uploaded file.

In the following experiment, we used one of the CGC Challenge Binaries as an example. The CGC repository has a lot of examples for testing binary vulnerabilities, which in this case, I chose CROMU_00038. I the user wants to find all the possible overflow functions in the binary, the user can input the example Datalog program as follow. The user will add a file with the new rule, and if the user only wants to search for a certain function of the file, the user can add the code_in_range rule in the file as well. Finally, the user can highlight and add comments to the function call that over possibly will appear.

```
.decl overflow_sink(SymName: symbol)

overflow_sink("strcpy").
overflow_sink("gets").
overflow_sink("getpw").
overflow_sink("memcpy").
overflow_sink("sprintf").
overflow_sink("printf").
overflow_sink("strcat").
overflow_sink("vsprintf").

.decl overflow_target(EA:address, Index:operand_index)
.output overflow_target
overflow_target(EA,Index) :-
    overflow_sink(Sym),
    symbolic_operand(EA,Index,OverflowFuncAddr,_),
    got_reference(OverflowFuncAddr,Sym).
```

**Figure 5.1:** possible buffer overflow functions

```
.decl code_in_range(from: address, to: address)
code_in_range(19210,28707)
```

**Figure 5.2:** code range

However, in the current version of D3RE the user can only add comments and highlight instructions to assist them on doing reverse engineering. With some experience in Ghidra, we believe there are more to be done on visualizing reverse engineering using static analysis. We propose a new UI that can create some function graphs and flow relations along with highlights as well as comments that will help the user on finding possible overflow functions with reverse engineering.

In the above example, D3RE will be able to highlight or add comments on related function calls. However, we believe there is still more to be done. Since we are using Ghidra, we should take advantage of the visualization tools provided in Ghidra. In the CROMU_00038 example, we run the script to find possible buffer overflow functions. In the function, we modified the function so it will list out the address of the possible buffer overflow functions, as well as the actual actress of where this particular function is declared. We want to create an idea of taint analysis, in which the particular functions, registers, or variables will be tainted for the user to do further analysis. For example, when we run the script on the binary file CROOMU_00038, it will print out possible buffer overflow functions can be highlighted. One of the functions is vsprintf. In this case since the size is declared 4096 in the stack if the size is greater than 4096, a bug will appear.

The purpose of creating the UI is not only to show the bug in GUI but also to improve the user experiences. An idea is shown in figure 5.3. We come up with this idea from experiences from previous Ghidra scripts and some of out own opinions. The following is the reason why it is important and plans for the next stage of developing plugins or extensions for D3RE.

Our REPL is now a CLI, in which the user can input the code range along with the declared rules. We have two ideas on the REPL, the first which is keeping the original idea on the REPL, which the user will have to input the address range manually. The second will be when the user clicks on a particular instruction on Ghidra, the Ghidra will get the address of the instruction and which function does it belongs to. In this way, it will be more user-friendly, but also increase the difficulty of connecting d3re and Ghidra (it is now connected by Ghidra bridge [13]).

In our CROMU_00038 example, after the possible overflow function is found, the user might want to know more about the information flow, so they can keep track of where the possible vulnerability travels. We're suggesting the user to right-click the instruction. Then the user can find out the dataflow graph, function call graph, and call graphs. The other proposal will be, list out all the possible vulnerabilities in a pop-out, so the user won't have to search one by one. In figure 5.4, it showed a sample idea on how the extension will work. By right-clicking on the instruction, the user can view the graph based on their need. In the future, we hope to integrate the ideas from the following bullet points on designing UIs

- What kind of plugins do you wish to see in Ghidra while doing RE?
- What kind of user interaction do you wish to see in GHidra while doing RE (ex: select a region of binary to add rule in GUI)?

- How do you wish the tool to help you visually on doing RE (commands, highlights, control flow graph, function call relation etc)

These questions are created based the current situation of D3RE, and the future expectation of D3RE according to our need to vision building the malware analysis pipeline.

Overall, we are thinking about right-click, enter in a combobox, and other GUI interface that will help the user the most. It is hard to design a good UI since we are not expert in reverse engineering. According to our experiences, we have collect some of the possible cases, and functions that will fit the experts' need.

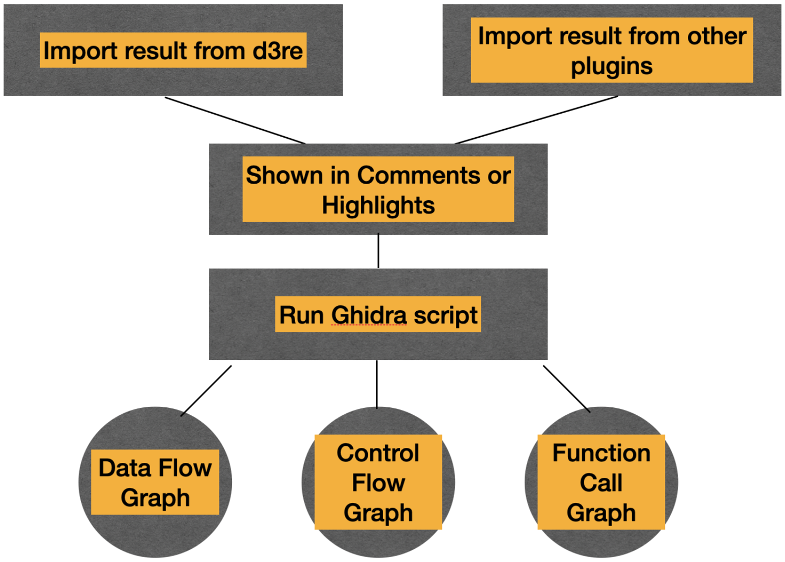
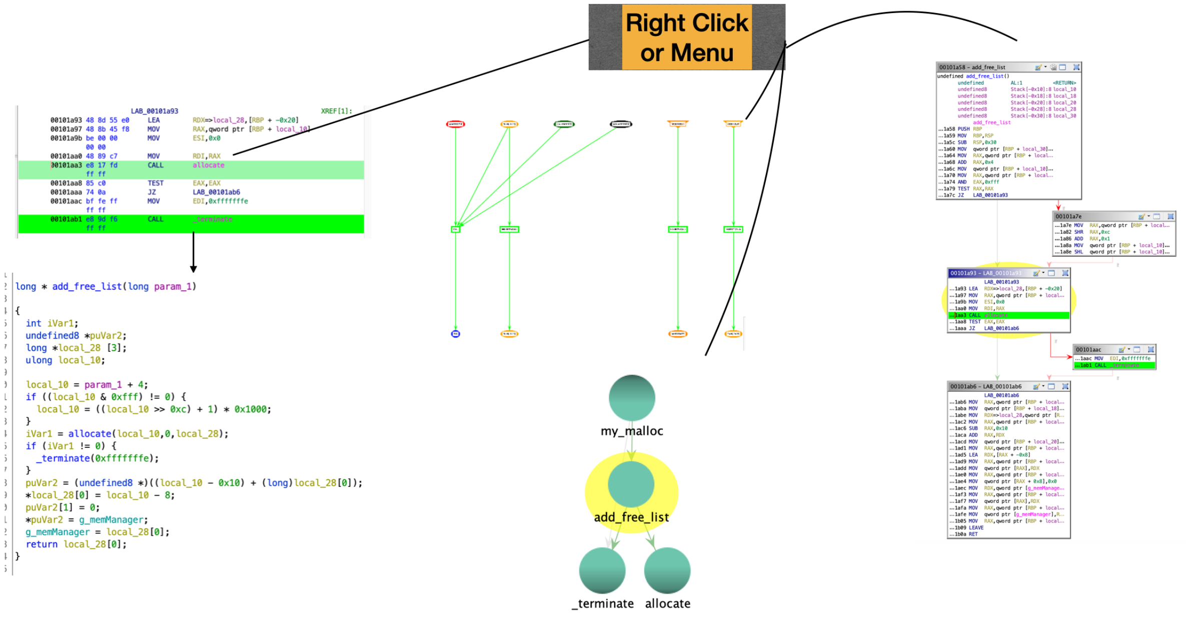


**Figure 5.3:** Idea for the design

## 5.2 Future Work and User Studies

With the examples above, we can say that we have a start. However, to continue, we believe user studies are necessary. The user studies will be related to D3RE and future works in visualizing reverse engineering. In D3RE and the above sample, we believe the idea is applicable, but can be more crisp. In the paper [42], it pointed out that while focusing on security vulnerabilities



```
                         LAB_00101a93                              XREF[1]:
00101a93 48 8d 55 e0     LEA        RDX=>local_28,[RBP + -0x20]
00101a97 48 8b 45 f8     MOV        RAX,qword ptr [RBP + local_10]
00101a9b be 00 00        MOV        ESI,0x0
         00 00
00101aa0 48 89 c7        MOV        RDI,RAX
00101aa3 e8 17 fd        CALL       allocate
         ff ff
00101aa8 85 c0           TEST       EAX,EAX
00101aaa 74 0a           JZ         LAB_00101ab6
00101aac bf fe ff        MOV        EDI,0xfffffffe
         ff ff
00101ab1 e8 9d f6        CALL       _terminate
         ff ff
```

```
long * add_free_list(long param_1)

{
  int iVar1;
  undefined8 *puVar2;
  long *local_28 [3];
  ulong local_10;

  local_10 = param_1 + 4;
  if ((local_10 & 0xfff) != 0) {
    local_10 = ((local_10 >> 0xc) + 1) * 0x1000;
  }
  iVar1 = allocate(local_10,0,local_28);
  if (iVar1 != 0) {
    _terminate(0xfffffffe);
  }
  puVar2 = (undefined8 *)((local_10 - 0x10) + (long)local_28[0]);
  *local_28[0] = local_10 - 8;
  puVar2[1] = 0;
  *puVar2 = g_memManager;
  g_memManager = local_28[0];
  return local_28[0];
}
```

```
00101a58 - add_free_list
undefined add_free_list()
      undefined        AL:1             <RETURN>
      undefined8       Stack[-0x10]:8 local_10
      undefined8       Stack[-0x18]:8 local_18
      undefined8       Stack[-0x20]:8 local_20
      undefined8       Stack[-0x28]:8 local_28
      undefined8       Stack[-0x30]:8 local_30
           add_free_list
...1a58 PUSH RBP
...1a59 MOV  RBP,RSP
...1a5c SUB  RSP,0x30
...1a60 MOV  qword ptr [RBP + local_30]...
...1a64 MOV  RAX,qword ptr [RBP + local...
...1a68 ADD  RAX,0x4
...1a6c MOV  qword ptr [RBP + local_10]...
...1a70 MOV  RAX,qword ptr [RBP + local...
...1a74 AND  EAX,0xfff
...1a79 TEST RAX,RAX
...1a7c JZ   LAB_00101a93
```

```
00101a7e
...1a7e MOV  RAX,qword ptr [RBP + local...
...1a82 SHR  RAX,0xc
...1a86 ADD  RAX,0x1
...1a8a MOV  qword ptr [RBP + local_10]...
...1a8e SHL  qword ptr [RBP + local_10]...
```

```
00101a93 - LAB_00101a93
           LAB_00101a93
...1a93 LEA  RDX=>local_28,[RBP + -0x20]
...1a97 MOV  RAX,qword ptr [RBP + local...
...1a9b MOV  ESI,0x0
...1aa0 MOV  RDI,RAX
...1aa3 CALL allocate
...1aa8 TEST EAX,EAX
...1aaa JZ   LAB_00101ab6
```

```
00101aac
...1aac MOV  EDI,0xfffffffe
...1ab1 CALL _terminate
```

```
00101ab6 - LAB_00101ab6
           LAB_00101ab6
...1ab6 MOV  RAX,qword ptr [RBP + local...
...1aba MOV  qword ptr [RBP + local_18]...
...1abe MOV  RDX=>local_28,qword ptr [R...
...1ac2 MOV  RAX,qword ptr [RBP + local...
...1ac6 SUB  RAX,0x10
...1aca ADD  RAX,RDX
...1acd MOV  qword ptr [RBP + local_20]...
...1ad1 MOV  RAX,qword ptr [RBP + local...
...1ad5 LEA  RDX,[RAX + -0x8]
...1ad9 MOV  RAX,qword ptr [RBP + local...
...1add MOV  qword ptr [RAX],RDX
...1ae0 MOV  RAX,qword ptr [RBP + local...
...1ae4 MOV  qword ptr [RAX + 0x8],0x0
...1aec MOV  RDX,qword ptr [g_memManage...
...1af3 MOV  RAX,qword ptr [RBP + local...
...1af7 MOV  qword ptr [RAX],RDX
...1afa MOV  RAX,qword ptr [RBP + local...
...1afe MOV  qword ptr [g_memManager],R...
...1b05 MOV  RAX,qword ptr [RBP + local...
...1b09 LEAVE
...1b0a RET
```

**Figure 5.4:** Layout of the ideal plugins.

problems, we can not only consider the potential attacks, but also how the software works and the related resources.

Multiple analysis can be done in reverse static analysis and reverse engineering. However, it is very hard to automatically analyze the binary files especially in static analysis. Some semi-automatically analyze work were done in dynamic analysis, and symbolic execution. However since the definition of static analysis is analyze the program without running the program, two kinds of strategies were suggested in multiple papers which are heuristics and declarative.

Reverse engineering can be totally different based on different goals. Since we are focusing on malware and vulnerabilities, we hope to focus on reverse engineers who mainly focus on vulnerabilities discovery and malware analysis. Since the area of vulnerabilities are large, I am going to give the examples and introduce some previous works in detecting buffer overflow as shown in the previous section.

We summarized the problems and obstacles that might be solved by user studies. Overall two questions are raised from the D3RE paper:

- Determine whether D3RE could realistically be used to accomplish the kinds of tasks that reverse engineers face on a day-to-day basis.
- Should we allow the user to select a region of the binary and build a rule that applies only to that region build a rule specific to callers of that function.

Some previous work have been done is related fields. Not all of the work are about visualization, but some of them point out certain characteristics that will be useful on createing UIs. Matzan showed that VSA might have help, and did an eye tracing user study, and proved that the more detail the VSA is, the better the experts will perform [27]. Similar researches showed that user declared graph and user assisted automated analysis might help [9]. Moreover, some Wang proposed it is possible to include AI and machine learning techniques on finding buffer overflows. Besides visualization, Zitser and Feng found some of the circumstances that buffer overflow will exists [51] [11].

These kind if papers are useful because they provided conditions and information that a buffer overflow might appear. According to these previous work, we can narrow down to certain properties that we will be looking for which are: Path classification: What kind of typical path will it lead to a buffer overflow; Heuristics:What kind of circumstances will lead to buffer overflow; How does while loops and if-else statements affect the appearances of buffer overflow? Declarative Base on our ultimate goals we have listed out two main topics we will be focusing on the future user studies:

- While our ultimate goal is to design a user interface working with Datalog backend, the design is important.
- Due to the purpose of the project, we want to design the interface closely to reverse engineers who focused on malware and finding vulnerabilities.

However, although the goal is clear, every experts in reverse engineering have their own habits, we would like to know some background information about them. We listed out some background problems that will be important for us to know while designing the UI for D3RE.

- Have you ever programmed extensions in Ghidra (what tools or packages are you using Gtable, Gtree, or etc…)?
- Have you regularly use Ghidra to discover malware?
- Have you regularly discover vulnerabilities with Ghidra?
- How well are your functional programming skills (Datalog etc)? Since we are inputting the rules in Datalog.

Since we are designing malware analysis pipeline, we want to focus only on malware analysis experts. We chose these questions based on our research and experiments own D3RE and Ghidra. Ghidra provides lots of user interface tools, but we figure out that not all the extension developers will use these packages we would like to know why. Also since. Also, we realized we are using Datalog as the programming language while declaring rules, we want to know how comfortable experts is on coding in Datalog. We have also created some general questions, some of them are from user studies on reverse engineering. Most of the questions are related to control flow and call information [42].

- Where is the method being called?
- How can I get calling information?
- Who can call this?
- Are all calls coming from the same class?
- What gets called when this method gets called?

For now we have some ideas on targeting buffer overflow problems, and we know finding vulnerabilities in buffer overflow is difficult in binary files. This chapter is still under experiment and hypothesis. We have a lot of thoughts and ideas on how our UI for D3RE to be, but we are not sure it will meet the public expectation since we are not experts. In the future, we hope to generalize the idea we have now and some prototypes to do a user study in visualizing reverse engineering.

Moreover, while there are a broad range of plugins for Ghidra and IDA Pro to load the results of static analyses, we believe D3RE is the first to focus on the combination of open-ended deductive logical inference with a reverse engineering tool front end. We believe the most closely related work is Ponce [30], which enables GUI-based symbolic execution. We plan to integrate symbolic execution into D3RE as a long-term goal, inspired by the recent work of Formulog [7]. Visualization-based tools such as Ghidra are of immense value in understanding a binary, but have fundamentally different design considerations than high-performance logical inference engines, where a good UI will solve the problem.

# 6 | Conclusion

Reverse engineering plays a big roll in our malware analysis pipeline. Since reverse engineering in binaries required users to inspect manually, it makes it challenging but fascinating. Throughout the literature review, we've found a good amount of work have been done in using static analysis and dynamic analysis on recovering binaries. However, since the task is too subjective and lack of standard procedures, there haven't been a specific or a perfect way on tackling reverse engineering. We hope that with this project, we can bring up two important part of reverse engineering and static analysis. First, D3RE the tool of visualising the results of static analysis. With D3RE we've shown that it's possible to use Datalog in a declarative style to analyze binary files, and we hope to take advantage of the visualization functions provided in Ghidra to improve users' experience on reverse engineering. Second, collecting repositories from GitHub and build the open-source projects with different compilers. Last but not least, by integrating assemblage and D3RE, we hope to extend the project even further. We hope to do user studies on user experiences on reverse engineering, and how to design D3RE's plugin on Ghidra. Overall, the project is still on en early stage, it looks promising, but of course a lot of work are still waiting for us.

# Bibliography

[1] Hojjat Aghakhani et al. "When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features". In: Jan. 2020. DOI: 10.14722/ndss.2020.24310.

[2] Omar Alhazmi and Yashwant Malaiya. "Modeling the Vulnerability Discovery Process". In: *2010 IEEE 21st International Symposium on Software Reliability Engineering* 0 (Nov. 2005), pp. 129–138. DOI: 10.1109/ISSRE.2005.30.

[3] *awesome ghidra*. https://github.com/AllsafeCyberSecurity/awesome_ghidra.

[4] Gogul Balakrishnan et al. "CodeSurfer/x86—A Platform for Analyzing x86 Executables". In: vol. 3443. Apr. 2005, pp. 250–254. ISBN: 978-3-540-25411-9. DOI: 10.1007/978-3-540-31985-6_19.

[5] Leilani Battle and Jeffrey Heer. "Characterizing Exploratory Visual Analysis: A Literature Review and Evaluation of Analytic Provenance in Tableau". In: *Computer Graphics Forum* 38 (June 2019), pp. 145–159. DOI: 10.1111/cgf.13678.

[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. "Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub". In: May 2017, pp. 356–367. DOI: 10.1109/MSR.2017.62.

[7] Aaron Bembenek, Michael Greenberg, and Stephen Chong. "Formulog: Datalog for SMT-Based Static Analysis". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428209.

[8] David Brumley et al. "BAP: A binary analysis platform". In: vol. 6806. Jan. 2011, pp. 463–469. DOI: 10.1007/978-3-642-22110-1_37.

[9] Cristina Cifuentes and Bernhard Scholz. "Parfait - Designing a Scalable Bug Checker". In: (June 2008). DOI: 10.1145/1394504.1394505.

[10] Robin David et al. "BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis". In: Mar. 2016, pp. 653–656. DOI: 10.1109/SANER.2016.43.

[11] Chao Feng and Xing Zhang. "A Static Taint Detection Method for Stack Overflow Vulnerabilities in Binaries". In: July 2017, pp. 110–114. DOI: 10.1109/ICISCE.2017.33.

[12] Antonio Flores-Montoya and Eric Schulte. "Datalog Disassembly". In: (June 2019).

[13] *Ghidra Bridge*. https://github.com/justfoxing/ghidra_bridge. Accessed: 2020-01-10.

[14] *Ghidra released by National Security Agency*. https://ghidra-sre.org/.

[15] Georgios Gousios. "The GHTorent dataset and tool suite". In: May 2013, pp. 233–236. ISBN: 978-1-4799-0345-0. DOI: 10.1109/MSR.2013.6624034.

[16] Georgios Gousios and Diomidis Spinellis. "GHTorrent: Github's data from a firehose". In: *IEEE International Working Conference on Mining Software Repositories* (June 2012), pp. 12–21. DOI: 10.1109/MSR.2012.6224294.

[17] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. "Work practices and challenges in pull-based development: the contributor's perspective". In: May 2016, pp. 285–296. DOI: 10.1145/2884781.2884826.

[18] Grammatech. *Gtirb*. https://github.com/GrammaTech/gtirb-ghidra-plugin.

[19] KyoungSoo Han, Jae Lim, and Eul Gyu Im. "Malware analysis method using visualization of binary files". In: Oct. 2013, pp. 317–321. DOI: 10.1145/2513228.2513294.

[20] Hexray. *Hex-rays:The IDA Pro disassembler and debugger.*

[21] Michael Hilton et al. “Usage, costs, and benefits of continuous integration in open-source projects”. In: Aug. 2016, pp. 426–437. DOI: 10.1145/2970276.2970358.

[22] Alan Jaffe et al. “Meaningful variable names for decompiled code: a machine translation approach”. In: May 2018, pp. 20–30. ISBN: 978-1-4503-5714-2. DOI: 10.1145/3196321.3196330.

[23] Yit Khoo et al. “Path projection for user-centered static analysis tools”. In: Nov. 2008, pp. 57–63. DOI: 10.1145/1512475.1512488.

[24] Riivo Kikas et al. “Structure and Evolution of Package Dependency Networks”. In: May 2017, pp. 102–112. DOI: 10.1109/MSR.2017.55.

[25] Zhenmin Li et al. “CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code”. In: Jan. 2004, pp. 289–302.

[26] Saraj Manes and Olga Baysal. “How Often and What StackOverflow Posts Do Developers Reference in Their GitHub Projects?” In: May 2019, pp. 235–239. DOI: 10.1109/MSR.2019.00047.

[27] Laura Matzen, Michelle A Leger, and Geoffrey Reedy. “Effects of Precise and Imprecise Value-Set Analysis (VSA) Information on Manual Code Analysis”. In: Jan. 2021.

[28] Sarah Nadi et al. “Mining configuration constraints: static analyses and empirical results”. In: (May 2014). DOI: 10.1145/2568225.2568283.

[29] Chengbin Pang et al. “SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask”. In: (July 2020).

[30] *Ponce (IDA Pro Plugin)*. https://github.com/illera88/Ponce. Accessed: 2020-01-10.

[31] *py-findcrypt-ghidra*. https://github.com/AllsafeCyberSecurity/py-findcrypt-ghidra.

[32] *Radare2*. https://github.com/radareorg/radare2.

[33] Edward Raff and Charles Nicholas. “A Survey of Machine Learning Methods and Challenges for Windows Malware Classification”. In: June 2020.

[34] Edward Raff et al. “Classifying Sequences of Extreme Length with Constant Memory Applied to Malware Detection”. In: Dec. 2020.

[35] Edward Raff et al. “Malware Detection by Eating a Whole EXE”. In: (Oct. 2017).

[36] Atanas Rountev, Olga Volgin, and Miriam Reddoch. “Static control-flow analysis for reverse engineering of UML sequence diagrams”. In: *ACM SIGSOFT Software Engineering Notes* 31 (Jan. 2006), p. 96. DOI: 10.1145/1108768.1108816.

[37] Maher Salah et al. “Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences”. In: vol. 2005. Oct. 2005, pp. 155–164. ISBN: 0-7695-2368-4. DOI: 10.1109/ICSM.2005.78.

[38] Eric Schulte et al. “GTIRB: Intermediate Representation for Binaries”. In: (July 2019).

[39] Edward Schwartz et al. “Native ×86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring”. In: Aug. 2013, pp. 353–368.

[40] Edward Schwartz et al. “Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables”. In: Oct. 2018, pp. 426–441. DOI: 10.1145/3243734.3243793.

[41] Yan Shoshitaishvili et al. “Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance”. In: Oct. 2017, pp. 347–362. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134105.

[42] Justin Smith et al. “Questions developers ask while diagnosing potential security vulnerabilities with static analysis”. In: Aug. 2015, pp. 248–259. DOI: 10.1145/2786805.2786812.

[43] Venkatesh Srinivasan and Thomas Reps. “Recovery of Class Hierarchies and Composition Relationships from Machine Code”. In: Apr. 2014, pp. 61–84. DOI: 10.1007/978-3-642-54807-9_4.

[44] Yihao Sun, Jeffrey Ching, and Kristopher Micinski. "Declarative Demand-Driven Reverse Engineering". In: Jan. 2021.

[45] Terrance Swift and David Warren. "XSB: Extending Prolog with Tabled Logic Programming". In: *Theory and Practice of Logic Programming* 12 (Dec. 2010). DOI: 10.1017/S1471068411000500.

[46] Bogdan Vasilescu et al. "Quality and productivity outcomes relating to continuous integration in GitHub". In: Aug. 2015, pp. 805–816. DOI: 10.1145/2786805.2786850.

[47] Daniel Votipka et al. "An Observational Investigation of Reverse Engineers' Process and Mental Models". In: Apr. 2019, pp. 1–6. ISBN: 978-1-4503-5971-9. DOI: 10.1145/3290607.3313040.

[48] Daniel Votipka et al. "Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes". In: May 2018. DOI: 10.1109/SP.2018.00003.

[49] Carsten Willems, Thorsten Holz, and Felix Freiling. "Toward Automated Dynamic Malware Analysis Using CWSandbox". In: *Security & Privacy, IEEE* 5 (Apr. 2007), pp. 32–39. DOI: 10.1109/MSP.2007.45.

[50] Khaled Yakdan et al. "Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study". In: May 2016, pp. 158–177. DOI: 10.1109/SP.2016.18.

[51] Misha Zitser, Richard Lippmann, and Tim Leek. "Testing static analysis tools using exploitable buffer overflows from open source code". In: vol. 29. Nov. 2004, pp. 97–106. DOI: 10.1145/1041685.1029911.

# JEFFREY CHING

## EDUCATION

**Syracuse University** *August 2019 - Present*
Master in Science
Department of Electrical Engineering and Computer Science
Advisor: Kristopher Micinski

## RESEARCH EXPERIENCE

**Syracuse University, NY** March 2020-Present
*Research*

- The research is on surveying the reverse engineering, and working with the tool Ghidra.
- Integrating the results of binary analysis tools with ghidra for further analysis.
- Write Ghidra plugins using java and python for analyzing the binary files.

**Syracuse University, NY** March-August2020
*Research(IOT)*

- The project mainly focus on analyzing and prediction of walking data and walking speed collected with accelerometer.
- Build up CNN models classifier to identify whether a participant is walking or not.
- Build up a CNN based prototypical neural network (few shot learning), to identify the participants walking speed.

## TECHNICAL STRENGTHS

| | |
|---|---|
| **Programming Languages** | C, C++, Python, Java |
| **Software & Tools** | Matlab, Github, Docker |

## WORK AND CLASS PROJECTS

**Athletics academic tutoring** December 2019-Present
*Syracuse athletics department*

- I tutor D1 athletics student on mathematics(statistics, Calculus1,2,3), physics and chemistry.

**AI Checkers game** December 2019
*Introduction to Artificial Intelligence Syracuse University*

- A class project of Introduction to artificial intelligence, working on build a AI checker game with pytorch using Monte Carlo tree search and CNN.

**Obligatory Military Service,Taiwan** May-August2018
*Military Service*

- Served four months obligated military service in army ROC.