UCLouvain

École polytechnique de Louvain

OpenFunctionID: a collaborative database for function identification in Software Reverse Engineering

Authors: **Cyprien Janssens de Bisthoven, Zina Rasoamanana**
Supervisor: **Ramin Sadre**
Readers: **Yinan Cao, Axel Legay**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

Abstract

Software reverse engineering is the process of recovering back the source code of a program from the machine code. It has several applications, ranging from the comprehension of program operations to the analysis of viruses and malwares. Among the collections of existing reverse engineering frameworks, Ghidra stands out due to its performance, availability over open-source, and expandability through homemade plugins. This Master Thesis aims to extend FunctionID, a plugin that improves the readability of decompilation. Currently, FunctionID performs hash database fingerprinting to improve the accuracy of the commonly found functions. However, the database covers only a narrow range of functions. This work presents and evaluates OpenFunctionID, an open source collaborative database integrated with Ghidra and that intent to provide an exhaustive database of function fingerprints.

Acknowledgement

We would like to express our gratitude to our supervisor, Professor Ramin Sadre, who guided us throughout our master thesis. He consistently kept us on track, and his guidance was of considerable use to us.

Cyprien *I would like to thank every professor and assistant I encountered throughout my education journey at the Ecole Polytechnique de Louvain. They shared their passions with us and were the driving force behind the quality of teaching we received. I would express my gratitude to the International Mobility Coordination of the faculty, thanks to which I was able to go to an Erasmus and meet such enriching and wonderful people. I would like to thank Zina Rasoamanana for accepting the fact that I was participating in an Erasmus and making the coordination possible. I would also like to express my gratitude to my family and friends for the support they gave me during these years.*

Zina *I would want to thank my family, especially my brother, Tony Rasoamanana, for their unwavering support over the previous five years. In addition, I would want to thank every classmate, professor, assistant, roommate, and friend I have made during my time at UCLouvain for making it such a fantastic experience. Lastly, I would like to thank Cyprien Janssens de Bisthoven, whom I met through the realisation of my master thesis and who was a wonderful partner.*

Contents

Chapter 1

Introduction

1.1 Context

Reverse engineering is a concept that every engineer has already heard once but remains abstract for many of them. Dissecting anything to see how it works is reverse engineering. The term "reverse engineering" originated in hardware inspection, where decoding designs from produced items is common [7]. Reverse engineering is a review, not a change or replication.

Reverse engineering is used to improve or understand products by analysing them. Many fields use reverse engineering. Software, hardware, military equipment, and even gene functions can be reverse-engineered. This thesis only covers software reverse engineering, which analyses machine code to reproduce the source code.

Many tools exist to perform software reverse engineering: IDA Pro [33], Binary Ninja [1], Radare2 [3], ...

Recently, an open-source software reverse engineering suite of tools called Ghidra has been released by the NSA. Ghidra is a framework created to provide the user with a powerful range of high-end software analysis capabilities. In addition to disassembly, it provides decompilation, assembly, graphing, and scripting functionalities. Ghidra is a complete suite of tools but offers the possibility for users to easily expand it by implementing plugins.

1.2 Objective

This master thesis aims to improve the information flow surrounding reverse engineering. Specifically, the exchange of knowledge gained through the process of reverse engineering to improve the identification of functions using previously analysed functions.

With this objective in mind, a Ghidra plugin, named OpenFunctionID, was developed. When disassembling and decompiling, Ghidra contains a built-in plugin called FunctionID that attempts to identify well-known functions. OpenFunctionID is an expansion of FunctionID in that, whereas FunctionID solely uses a local database for function identification, OpenFunctionID uses a collaborative online database. The collaborative nature of OpenFunctionID's database makes it more complete, as it combines the knowledge of a community rather than a single individual.

1.3 Outline

This document is separated into three parts. The first part provides a context by discussing reverse engineering and Ghidra. The second part discusses a Ghidra plugin called FunctionID. The final part discusses OpenFunctionID, a new plugin that complements FunctionID.

Chapter 2 establishes the groundwork for reverse engineering by defining it. The chapter concludes with a discussion of real-world applications of reverse engineering to comprehend how some viruses work.

Chapter 3 outlines the history of Ghidra and describes its primary features.

Chapter 4 covers FunctionID in detail. It begins with an overview, then describes FunctionID Analyser and FunctionID databases. The section continues by discussing the various compilers and processors supported by Ghidra. The chapter concludes by discussing Hex-rays' implementation of FunctionID.

Chapter 5 describes the plugin OpenFunctionID. It starts with the objectives of the development of this plugin. Then, the chapter depicts the plugin, its additional script, and the server it relies on.

Chapter 6 explains how to integrate the plugin into a Ghidra installation. It also provides instructions to set up the lookup script and the server.

Chapter 7 dissects OpenFunctionID's security. It starts with the security of the server and the database. It continues with the user identification and tries to picture malicious usage of OpenFunctionID.

Chapter 8 begins with some analyses of OpenFunctionID. It continues by exposing some limitations and ends with some future work that can be done.

Part I

Background

Chapter 2

Reverse Engineering

2.1 Definition

Reverse engineering involves deconstructing something to figure out how it works. According to the taxonomy on reverse engineering and design recovery made by Chikofsky and Cross in 1990 [7], the term "reverse engineering" arose in the inspection of hardware, where deciphering designs from produced items is a frequent practice. Reverse engineering was defined by M.G. Rekoff [35] in 1985 as "*the process of developing a set of specifications for a complex hardware system by analysing the specimens of that system. These specifications are being prepared by persons other than the original designers, without the benefit of any of the original drawings or other documentation*". Reverse engineering does not entail modifying the subject system or developing a new system based on the reverse-engineered subject system. It is an examination process, not a change or replication process.

Reverse engineering is frequently used to improve one's own products as well as to investigate a competitor's or enemy's products in a military or national-security context. Reverse engineering can be applied in many fields. Software, physical devices, military technologies, and even biological capabilities related to gene function can be reverse-engineered.

Many types of reverse engineering exist but in this thesis only software reverse engineering will be discussed. It focuses on the machine code and tries to extract information to recreate the source code. Software reverse engineering is similar to hardware reverse engineering. The difference is that one analyses a software program and the other analyses a physical device.

According to Chikofsky and Cross [7], there are six key objectives to reverse engineering that serves the primary purpose of increasing the comprehensibility of systems.

- **Manage complexity** Researchers must create strategies to manage the sheer number and complexity of systems more effectively. Reverse engineering will enable decision-makers to control the product's system evolution by extracting pertinent information.
- **Generate alternative views** Reverse engineering enables the production of graphical representations from perspectives other than the primary designer's perspective. To facilitate the review and verification process, reverse-engineering tools can provide additional views, such as control flow diagrams, structure charts, and entity-relationship diagrams.
- **Recover lost information** Large, long-lived systems evolve, losing design information. Occasionally, modifications are not reflected in the documentation, particularly at a higher level than the code. Reverse engineering can be used for extracting information from existing systems and gaining an understanding of how their component programs interact.
- **Detect side effects** Both haphazard initial design and subsequent updates can have unanticipated effects on the performance of a system. Before people report defects, reverse engineering can identify anomalies and problems.
- **Synthesise higher abstractions** Reverse engineering necessitates methodologies and techniques for generating views at a higher level. Expert-system technology will aid in the generation of abstractions at a high level.
- **Facilitate reuse** The vast quantity of existing software assets poses a difficulty for software reusability. Reverse engineering can assist in identifying software that can be reused in current systems.

Reverse engineering can be used in many cases and for many goals. Reverse engineering may help to reuse old programs by recovering lost information. It can also be used to gain a competitive advantage by detecting side effects in a company's programmes and understanding a competitor's system. Reverse engineering may also help to simply understand how something works by giving higher abstraction views of systems. Alternatively, reverse engineering can also be used to perform security analysis because it offers alternative views that can highlight security issues or detect side effects that have security consequences.

Moreover, it can be used to understand the behaviour of a malicious program to create mechanisms of defence. Malware detection, identification of Trojan horses, and virus analysis are examples of typical duties. This last case will be dissected in depth in the Section 2.2.1 with the analysis of WannaCry using reverse engineering.

To fully understand what reverse engineering is and how it works, it is mandatory to understand the different processes that are part of reverse engineering. The tools that are needed to execute these processes and, finally, the artefacts that are produced throughout the procedure.

2.1.1 Processes

According to Treude, Filho, Storey and Salois [42], four processes can be identified in reverse engineering. They are analysing, documenting, transferring knowledge, and articulating work. All of these are equally important for achieving the goal of decompiling a program.

Analysing

The assembly code analysis process is at the heart of nearly all successful reverse engineering endeavours. Assembly code is more difficult to comprehend than source code written in high-level programming languages because it is less organised, frequently lacks meaningful symbols or data definitions, and contains gimmicks that may be used to deceive reverse engineers during their analysis attempts. As said in an exploratory study of software reverse engineering [42], it is difficult to follow the flow of data: "Understanding the data flow is a big part of understanding a program".

Documenting

Reverse engineering documentation is used to accomplish a range of tasks. There are many types of documentation that are developed to aid reverse engineers during the analysis process, and there are other types of documentation that are made to record the reverse engineers' own understanding of the code. Various more pieces of documentation are produced for distribution among team members or external stakeholders. Despite the fact that documenting source code written in high-level programming languages is difficult in and of itself, it becomes substantially more difficult when working with assembly code. During the assembly code exploration phase, most reverse engineers only write down the information needed to reproduce the result. They do not write down other options that were tried but did not work, and thus failures are not documented.

Transferring Knowledge

When it comes to reverse engineering, knowledge transfer is a difficult task. In many cases, documentation alone is insufficient to fully comprehend the work that has been accomplished by someone else; as reported in a study about reverse engineering [42]: "*I would look at a version with comments, but I'd still need to jump through to understand*". In the modern environment, information is typically communicated verbally, by email, or by chat. These procedures are not scalable beyond groups of roughly five reverse engineers, hence they are not recommended. However, workflows are not uniform across all circumstances, and the majority of workflow support tools are overly restrictive. Furthermore, rules surrounding documentation and standards for sharing information might help to enhance the reverse engineering process: Observing norms would make it easy to transition from one project to another, according to the interview of a reverse engineer in a study from 2011[42].

Articulating Work

All components required to organise a certain activity, including scheduling subtasks, recovering from mistakes, and gathering resources [18], are considered to be labouring but essential to structure the work. In regard to reverse engineering, concrete results are saved only when a line of research has been successfully explored, which has the problematic effect that unsuccessful paths are explored multiple times. This complicates the task of articulating the work. Besides that, work is often separated into sections depending on distinct pieces of hardware, different vulnerabilities, different functionalities, or different files. Then, it is extremely difficult to connect the knowledge gained from the investigation of different aspects of the problem, which makes the task of gathering resources harder.

2.1.2 Tools

Reverse engineers employ a variety of tools to accomplish their objectives. They use reverse engineering software. They mostly employ a disassembler to carry out the process of debugging and analysing of the code. In addition, they present their progress using visualisation tools that may be incorporated into their disassembler. Besides that, they employ coordination tools to coordinate the task and share knowledge.

Reverse Engineering Software

The majority of the reverse engineering effort is carried out with the aid of a disassembler. It is a piece of software that performs automated code analysis and includes interactive features to help understand the disassembly. Reverse engineers often begin with an automatically produced disassembly listing and work their way through the listing by renaming and annotating portions until they comprehend the code. Debuggers are rarely used in the early phases of malware research because sections of the code are frequently unavailable for execution or because the malware employs anti-debugging techniques that must be removed first. Many reverse engineering tools exist: IDA Pro [33], Binary Ninja [1], Hopper [4], Radare2 [3]. Although IDA Pro is the most well-known disassembler, an open-source alternative named Ghidra is gaining popularity. Ghidra is a suite of tools which includes a disassembler, but not only. Ghidra will be dissected in depth in the Section 3.2. Another important tool is a decompiler. It takes the assembly code and decompiles it into a representation written in a high-level programming language (e.g. C). The companies cited previously that develop a disassembler also have their own decompiler. Hex-Rays has its own decompiler[31]. Binary Ninja, Hopper, and Ghidra directly include a decompiler. Radare2 does not directly integrate a decompiler, but it supports plugins that are decompilers (r2dec[43]) and one of these plugins is the native Ghidra decompiler[2]. The toolbox of a reverse engineer should at least contain a decompiler and a disassembler. Then, many tools can be added: tools to reverse engineer data structures, code visualisation tools, but also more specific tools such as plugins for malware analysis.

Visualisation Tools

The overwhelming majority of documentation is generated in a normal word processing application such as Microsoft Word, Excel, or OneNote. Of course, paper is also used, most notably for workflow support, little graphs, and articulation work, among other things. UML sequence diagrams are frequently used as an additional demonstration of the comprehension of control flow. While a reverse engineer may produce these diagrams manually, some reverse engineering suites, such as Ghidra, IDA, and Hopper, can generate them automatically. The Section 3.2.1 on Ghidra's key features will go into further depth about the generation of flow diagrams.

Coordination Tools

Only the most fundamental communication methods, such as email and chat, are used. Although it is sometimes possible to interact face-to-face, the nature of the job, which can be classified or extremely sensitive, makes data exchange difficult, according to one of the reverse engineers who was interviewed for research on the subject[42]. Furthermore, reverse engineers collaborate on projects through the use of technologies such as wikis, bug trackers, source code managers (such as GitHub) and collaborative documents.

2.1.3 Artefacts

Annotations, cognitive support artefacts, and reports are examples of artefacts that are formed throughout reverse engineering processes. Artefacts are the result of a successful process of reverse engineering.

Annotations

Two sorts of annotations are supported by disassemblers (such as IDA Pro or Ghidra): repeatable and non-repeatable. In addition to the current item, a repeated annotation will be attached to any other items that reference it. Non-repeatable annotations are only shown when they are related to the current item. It is also possible to connect pre-comments and post-comments to specific lines or functions. As an illustration, Figure 2.1 is an example of annotation within the Ghidra decompiler.

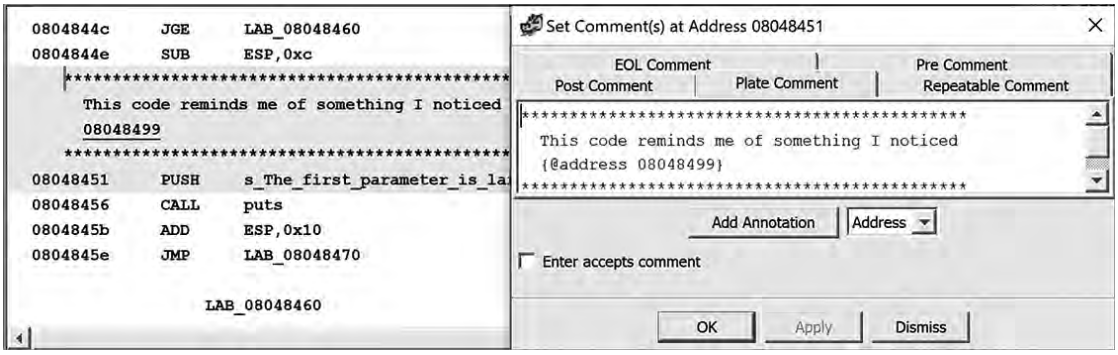


Figure 2.1: Example of annotation with Ghidra [13]

Reverse engineers use annotations for a variety of purposes, including keeping track of variables, renaming functions, documenting jumps, and noting where a specific section of code is read from or written to. However, one of the difficulties is that the annotations are always incomplete, as it is easy to forget to mention things that are evident to the writer at the time of describing the code.

Cognitive Support Artefacts

Depending on the issue they are working on, reverse engineers produce a range of documentation to help in their understanding. The following are examples of such documents: memory maps, Excel or Word tables illustrating register use and boot processes, data flow diagrams, sequence diagrams, and scripts. An engineer often needs this kind of document when he needs to keep track of all the different paths that are reviewed to understand a certain piece of code.

Reports and assembly code

Companies specialising in malicious software, such as Symantec [6], publish papers on a regular basis that detail how a particular piece of malware acts. This sort of report seldom offers sufficient information to enable the reader to appreciate the malicious software's inner workings, mostly because security groups are unwilling to reveal their discoveries to malware writers. Academic papers, on the other hand, are more technical in nature and usually include annotated assembly code for functions in addition to detailed descriptions of all input and output variables.

2.2 Real World Cases

This section aims to inform about the threats that viruses represent in order to demonstrate that the analysis of them using reverse engineering is important. The final goal of such analysis is to develop solutions to protect against these viruses. In fact, it identifies the role of reverse engineering in the context of malicious program analysis.

In a 1987 essay [8], Fred Cohen coined the phrase "computer virus" to describe self-replicating programs that are meant to infect other computer programs and spread throughout the computer network. The first computer viruses were generated for scientific purposes or just for amusement, and they caused little or no damage to real-world systems at the time of creation [30]. The landscape of today represents a totally different reality from that of yesterday. Computers are widely used in illicit operations such as bank fraud, identity theft, and corporate theft, to name a few examples.

In addition to endangering national security, illegal activities in cyberspace undermine people's rights and privacy, and have far-reaching consequences on the political, economic, and social fronts. In order to achieve a variety of objectives, such as organised crime or economic gain, hackers develop harmful software, sometimes known as "malware". Malware comes in a variety of forms, including viruses, worms, Trojan horses, spyware, and ransomware, all of which can be installed on a computer without the informed consent of the computer's owner. Malware can infect a computer in a variety of ways, including through the use of email attachments.

Since the term "virus" was invented and computers became more accessible to the general public, history has witnessed various infestations and cyber attacks. Here after are some examples of the most well-known viruses.

Melissa The Melissa virus [17], which spread at a fast rate on computer networks in March 1999, sent shivers down the spines of many working in the information technology industry. Those who were affected were concerned because Melissa spread quickly through the use of infected e-mail attachments, which when opened, sent the virus to others using the contact books of those who were infected. As a result, Melissa spread fast over the world, overwhelming email servers before being brought under control. This strategy was particularly cunning since the virus Melissa used infected e-mail attachments that seemed to be from recognisable senders, giving the impression that they were. Melissa took advantage of clients' widespread and usually negligent opening of e-mail attachments that may contain viruses. Moreover, Melissa took advantage of a security weakness given by macro languages, such as Microsoft's Visual Basic for Applications (VBA), which allows users to insert executable code in documents, such as Word documents. Melissa's authors placed malicious code into a document that they sent as an email attachment.

Stuxnet Iran openly revealed in 2010 that a cyber weapon had destroyed centrifuges at the Natanz nuclear enrichment complex. Stuxnet [20], which was discovered by VirusBlokAda, was described as a "very sophisticated" and complicated program intended only for destroying uranium enrichment centrifuges operated by high-frequency converter drivers used by the Natanz nuclear enrichment plant. The virus effectively infected a large number of Iranian centrifuges, forcing more than 1,000 of them to spin out of control by targeting a certain type of industrial controller. It was surprising that the malware had reached a network that was not connected to the Internet. Individuals were continually infecting stand-alone workstations and networks by inserting malicious removable media, either intentionally or unintentionally.

WannaCry WannaCry [19] ransomware was one of the most widely distributed viruses of 2017. This global wave of cyber assaults is said to have hit over 150 nations around the world. The primary goal of ransomware is to profit from victims. By displaying a ransom screen, attackers may display crucial information such as contact information, ransom price, and method of payment. Hackers who wish to carry out any form of attack prefer to stick to their standard attack strategies and processes. The initial stage of a ransomware attack is to gain access to the target system and execute its files. Several deployment mechanisms can be used, such as phishing emails, fraudulent web sites, susceptible vulnerabilities, etc., differing from one ransomware to the next. The infection will start after the malicious payload has compromised the host. After gaining initial access to the system, the ransomware will begin its installation and seek to seize complete control of the infected host. The virus then uses the activation key to lock or encrypt data on the affected device. Finally, the behaviour of a ransomware attack is determined by the type of command-and-control system used. The malicious code on the infiltrated computer is referred as a client, while the opposite side manipulating the command-and-control system is referred to as a server. The command-and-control channels may be distinguished by metamorphic malware types and variants. These can range from basic web-based conversations using the HTTP protocol to complex Tor[21] service connections.

2.2.1 Reverse Engineering Applied to WannaCry

This section will demonstrate the utility of reverse engineering in the context of virus analysis by revealing the findings of a static study on WannaCry [19]. Studying a ransomware such as WannaCry is essential to understand how it works in order to develop solutions to protect against it.

The increasing modularity of ransomware, such as WannaCry, has compelled security experts to investigate the internal architecture of each binary module. Once the WannaCry ransomware has gained access to the system, it begins replicating and encrypting data in accordance with a predetermined pattern of behaviour. Performing a series of tasks [19] results in the creation of damage. It is organised in a cyclical manner, as is the WannaCry code. As described by Liska and Gallo [24], the architecture of a ransomware attack may be separated into four stages: deployment, installation, destruction, and command-and-control. Figure 2.2 is a schema [19] with the four stages and the execution flow is given to clarify this section.

Deployment Phase In the deployment phase, the launcher is remotely injected through famed vulnerabilities (Eternalblue vulnerability and Doublepulsar

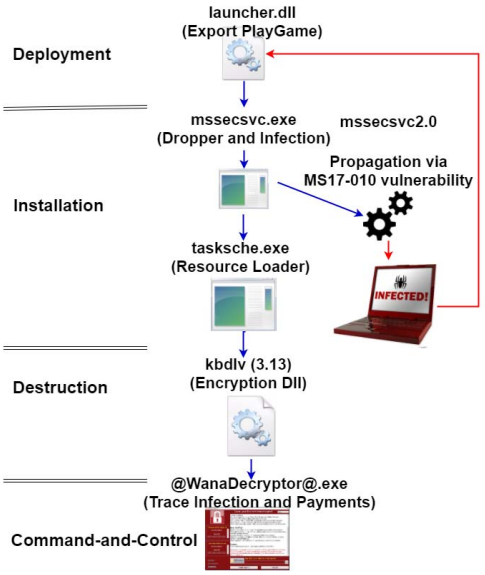


Figure 2.2: Execution flow of Wannacry [19]

backdoor) [19]. It also exports the PlayGame method to initialise the embedded "mssecsvc.exe" binary in the launcher resource area to start the second phase.

Installation Phase It consists of two components during the installation phase: "mssecsvc.exe" and "tasksche.exe". The "mssecsvc.exe" will start the "mssecsvc2.0" service to start the propagation and drop the "tasksche.exe," whereas the "tasksche.exe" is responsible for resource loading and environment configuration to continue to the destruction phase.

Destruction Phase During the destruction phase, "tasksche.exe" will load the "encryption dll" into memory, allowing to accomplish the destruction phase.

Command-and-Control Phase In the command-and-control phase, WannaCry continues to track payments and sends encrypted information to onion servers.

Once a report like the one on WannaCry is released, all companies that build security software will identify every weakness. Then, they will begin to develop solutions and safeguards against these threats. Eventually, additional protection will be added to their security software. Viruses that exploited revealed vulnerabilities will no longer be able to infect. This demonstrates the utility of software reverse engineering to identify new vulnerabilities in the process of creating new defence mechanisms.

Chapter 3

Ghidra

3.1 History

Since 2019, a powerful framework is used around the world. Its name is Ghidra and it is described as *"A Software Reverse Engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission"*[27], but what do they mean by this description?

Before going into the details of Ghidra's functionalities, it is important to know more about the history of this mysterious new framework. It was created and is maintained by the National Security Agency (NSA) Research Directorate.

3.1.1 National Security Agency (NSA)

The NSA is probably the world's most well-known intelligence agency of the Department of Defense of the United States of America. It is under the jurisdiction of the Director of National Intelligence. The National Security Agency (NSA) is in charge of global monitoring, gathering, and processing of information and data for foreign and domestic intelligence and counterintelligence purposes. The NSA is also responsible for the safeguarding of the United States communications networks and information systems that hold secret information and are utilised in the conduct of intelligence and military activities. Despite the fact that the NSA was created in 1952, its existence was not publicised until 1975. The National Security Agency (NSA) relies on a number of tools and methods to carry out its tasks, the vast majority of which are classified. The fact that the NSA is very secretive is perhaps the most important factor contributing to its widespread popularity.

(a) Seal of the U.S. National Security Agency [9]

(b) WikiLeaks logo [10]

Figure 3.1: Seals of the National Security Agency and of WikiLeaks

It often operates outside of the purview of Congress, and it is widely regarded as the most secretive of the U.S. intelligence organisations. The NSA's activities have been the subject of a number of political controversies. The most famous one was in 2013, when the former NSA contractor Edward Snowden released secret documents.

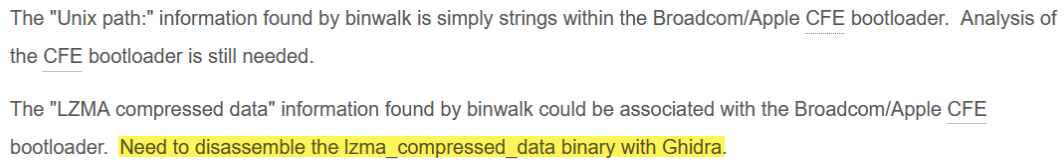
The NSA is not the only very secretive intelligence agency in the world, and some defenders of the truth regularly release classified documents. This is the case of WikiLeaks.

3.1.2 WikiLeaks

WikiLeaks is a multi-jurisdictional public non-for-profit media organisation. They are dedicated to protecting whistleblowers, journalists, and activists across the globe. They believe in open information and want to publish and share important materials so that readers and historians can see evidence of the truth. It publishes large datasets of censored documents involving war, spying, and corruption.

2017: Vault 7 On the 7$^{\text{th}}$ of March 2017, WikilLeaks released nearly 9 000 documents from the U.S. Central Intelligence Agency (CIA) [45]. This large publication has the codename "*Vault 7: CIA Hacking Tools Revealed*". It was the largest leak of confidential documents ever. These documents contained information about the hacking arsenal of the CIA, including malware remote control systems, spying techniques, Trojans, secret cyberweapons, "zero-day" exploits, viruses, etc.

The "*Vault 7*" was separated into 24 parts with codenames as "*AfterMidnight and Assassin*", "*CouchPotato*" or "*Brutal Kangaroo*". Ghidra was revealed in the first part of the "*Vault 7*" called "*Year Zero*". This part revealed many tools used by the CIA to spy on iPhone's, Android phones, Windows devices, and TVs. In this leak, Ghidra was mentioned an important number of times, with step-by-step tutorials, how to install it, and how to use it. At this time, the latest Ghidra version available was Ghidra 7.0.2. Ghidra is also referenced in other leaked documents. Figure 3.2 is from a leaked report in which the user tries to reverse engineer the Apple Airport firmware and cites Ghidra as a tool which will help him accomplish his task.


The "Unix path:" information found by binwalk is simply strings within the Broadcom/Apple CFE bootloader. Analysis of the CFE bootloader is still needed.

The "LZMA compressed data" information found by binwalk could be associated with the Broadcom/Apple CFE bootloader. Need to disassemble the lzma_compressed_data binary with Ghidra.

Figure 3.2: Leaked report where the user is trying to reverse engineer the Apple Airport firmware [44]

"*Vault 7*" leaked documents containing information about tools used by the CIA, but some weren't developed by the CIA. Ghidra, for instance, was developed by the National Security Agency Research Directorate.

3.1.3 NSA Research Directorate

At the organisational level, research involves in certain organisations a type of analysis of a given activity or technology for a specified determined period of time, followed by the production of some form of paper at the conclusion of the study. Other organisations are interested in taking a piece of technology and pushing it to its boundaries in order to generate some proof-of-concept. However, the research at the NSA Research Directorate is not exactly either one of those. They engage in what they refer to as "mission-oriented research". As a result, their purpose is to have a beneficial influence on the National Security Agency's mission in whatever period is required to accomplish this. Consequently, they will have more freedom to do what they need to do in order to make a positive difference.

The NSA Research Directorate has the freedom to step away from the NSA, and it is essential to come up with the next solutions. As Henry Ford probably said: "*If I had asked people what they wanted, they would have said faster horses*"[15]. Finally, he came with a car. Therefore, the NSA Research Directorate has the ability to step back and produce better solutions. And Ghidra is one of them.

3.1.4 RSA Conference

Two years after Wikileaks revealed Ghidra's existence, in 2019, Rob Joyce, Senior Advisor for Cybersecurity at the National Security Agency, presented Ghidra to the world for the first time at the RSA Conference [25]. The slogan of the RSA Conference was "*Where the world talks security*". Since 1991, the RSA Conference has been an international series of cybersecurity conferences. The name refers to the public-key encryption protocol developed by Rivest, Shamir, and Adleman (RSA)[36]. Each year in the United States, 45 000 people are expected to attend the world's leading information security conferences and exhibitions. Bringing everyone in the cybersecurity business together and enabling the cybersecurity industry community to take on cyber threats around the world is the objective of the RSA Conference. The 2019 RSA Conference was held in San Francisco from the 4^{th} of March 2019 to the 8^{th} of March 2019, and then the Ghidra source code was officially released on GitHub by the NSA in April 2019. The first official version was 9.0.0. Afterwards, they continued to present Ghidra at other conferences around the world [23][14].

Ghidra has been an ongoing project for years. In 2019, they said that it had been in development for around 20 years. One could wonder why they are releasing it now. Perhaps the first reason is that WikiLeaks revealed it. On the other hand, NSA spokesperson Rob Joyce stated that it would be used as a recruiting tool, which would allow users to become familiar with the tool before being accepted into the program [25]. Ghidra is also the first free tool of its size, and as such, it will aid in the training of cybersecurity defenders.

3.2 Ghidra Overview

Figure 3.3: Ghidra Logo [27]

Ghidra is a framework with the purpose of giving the user a powerful suite of high-end tools for software analysis. It includes not only decompilation, but also disassembly, assembly, graphing, and scripting.

The NSA needed a framework that could handle the scale of a reverse engineering process, a collaborative way to do it through teamwork, and be expendable. Ghidra replies to this request. Instead of having a text editor open, another program with a disassembler and again another program with a debugger, Ghidra does it all.

3.2.1 Main features

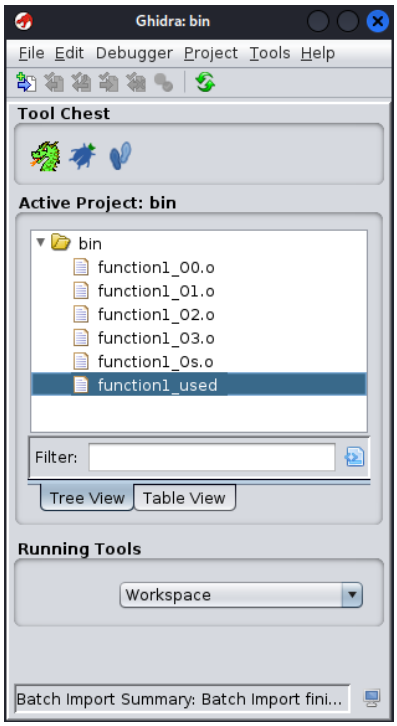
Project based Ghidra is project-based, multiple files can be imported into the same project, so the user can analyse a whole program with a lot of files. Ghidra's project window is represented in Figure 3.4a.

Listing View Once a binary code is imported, it can be opened in the **Listing View**, where all instructions in assembly language are displayed. Some annotations can be added, with data and comments as well. Ghidra's listing view window is represented in Figure 3.5a.

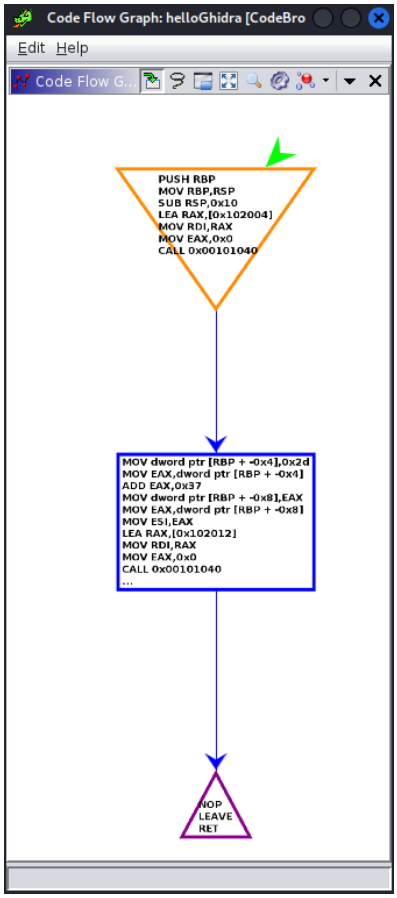
Function Graph Functions can be presented as a graph with all assembly codes linked to that function, which call which one and are called by which one. All jumps and branches are split to represent the assembly code in a more human-readable way than in **Listing View**. Ghidra's graph window is represented in Figure 3.4b.

Decompiler There is also the **Decompiler** feature, which will take the assembly code and decompile it into a representation written in the C programming language. The **Decompiler** is based on **SLEIGH** definitions of different Instruction Set Architecture (ISA). Despite the ambiguity, ISAs are called *processors* in Ghidra's documentation. This document will follow this nomenclature. Ghidra's decompiler window is represented in Figure 3.5b.

Once a programmer compiles his code, it is translated into machine language, the assembly code. However, it is different for each processor. Ghidra should support a wide range of processors to decompile the binary code. The **SLEIGH** language is used to define each architecture. **SLEIGH** was developed for Ghidra and is a processor specification language. It is used to describe processors and is used by the Ghidra disassembly and decompilation engine. It can be defined as a grammar since it provides a description of the translation from machine instructions to assembly language instructions. **SLEIGH** also shows how an instruction manipulates data, and how a machine instruction can be translated to *p-code* used by the decompiler. The use of *p-code* will be more covered in detail in section 4.1.1. As long as a processor has a **SLEIGH** description, it is supported by Ghidra.



(a) Project Based

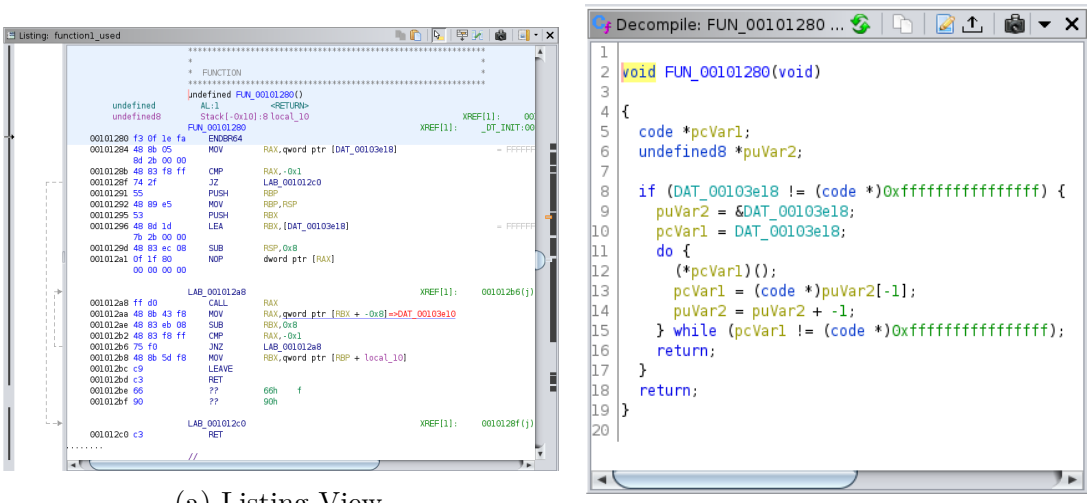


(b) Code Flow

Figure 3.4: Ghidra Features (1)

For now, the official release of Ghidra supports processors (ISAs in Ghidra language):

- x86 16/32/64
- ARM/AARCH64
- PowerPC 32/64/VLE
- MIPS 16/32/64/micro
- 68xxx
- Java / DEX byte-code
- PA-RISC
- PIC 12/16/17/18/24
- Sparc 32/64
- CR16C
- Z80
- 6502
- 8051
- MSP430
- AVR8
- AVR32
- And other variants of these processors



(a) Listing View

(b) Decompiler

Figure 3.5: Ghidra Features (2)

Script Manager The script manager is a powerful feature allowing to easily expand Ghidra with small new features as scripts. The script manager currently supports scripts written in the Java and Python programming languages, but the script manager can be extended to other languages as long as there is a link to the Java language in the new language (for example, Jython is used to support Python scripts).

Plugins The Ghidra framework is made up of plugins that are used to provide services, menu and button actions, new windows, etc. Plugins cooperate and share information within a tool. Ghidra comes with a default tool, the *Code Browser*, which is accessible via the Tool Chest. A tool is a combination of different plugins. There are also *Debugger* and *Version Tracking* in this Tool Chest. Following the expandable goal of Ghidra, the user can create new tools by combining plugins. The default tool comes pre-installed with all of the core plugins.

The ability to create new tools and plugins is one of the key aspects of Ghidra and allows it to be qualified as a robust software reverse engineering framework. By extending Ghidra with new tools and plugins, users can make it even more powerful. Since Ghidra's release, the community is growing and open source plugins are shared by independent developers. Plugins are written in Java and communicate within a tool with other plugins.

A wide range of Java classes can be extended by a plugin to provide actions, GUIs, and interact with events such as user selection, highlights, right clicks, etc. The development of a plugin will be more explained in Section 5.2.4.

Ghidra Server When working on large software reverse engineering projects, with numerous large files, team collaboration is necessary. The **Ghidra Server** allows numerous users to view and analyse the same project at the same time, allowing a team to work on a single project. It offers network storage for shared project repositories while also allowing administrators to regulate user access to such repositories.

Even free servers are made available by an association of individuals with the purpose in fostering international collaboration for software reverse engineering tasks[38].

Headless Analyser Ghidra supports a headless mode, called **Headless Analyser**. It is a command-line version of Ghidra useful when performing automated or repetitive tasks on projects.

Other Features With more than 825 pull requests (open and closed) and 1150 open issues on GitHub, Ghidra is growing every day and has many other features. Some should be mentioned, such as Debugger, FunctionID, Undo/Redo, Version Tracking, Memory Map, Symbol Table, FileSystem Browser, Entropy, and Version Control.

Part II

FunctionID: a powerful plugin in Ghidra

Chapter 4

FunctionID

4.1 Overview

When a binary file is analysed by Ghidra, a variety of distinct analysers are utilised. Each analyser can also be conducted independently, or "One shot" in Ghidra's terminology. One of them is FunctionID.

researcher

4.1.1 Decompiler

A decompiler attempts to reverse a code by understanding how a compiler operates. The compiler converts the source code to a machine-specific language. A source code can therefore be represented as composed by a lot of small blocks. A block may be an assignment, a condition block (if-else), or a loop, among other things. Each of these small blocks is then translated to machine language. Figure 4.1 illustrates how a high-level source code (as C Programming Language) is compiled to machine-specific assembly code, then the assembler translates it to machine code, creating the binary executable file.

Firstly, Ghidra uses the **SLEIGH** language specification to be able to translate machine-specific instructions into *p-code*. *P-code* is a *Register Transfer Language* designed to be machine-independent. Therefore, binary code from different processors can be translated into *p-code*. Ghidra analysers can be used on the resulting *p-code* coming from different targets without being redesigned for each processor. **SLEIGH** specification of the translation of the processor's instruction set into general *p-code* is enough to use Ghidra decompiler on all defined Instruction Set Architectures (ISAs).

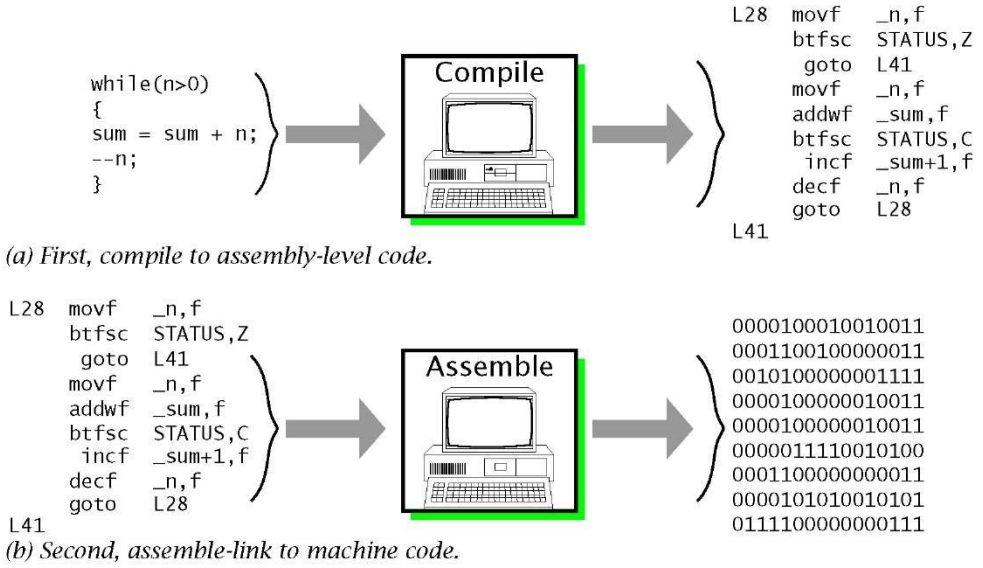


Figure 4.1: Conversion from high-level source code to machine code [22]

P-code is a representation of the execution of a program, but in a more general understandable way since it is not machine specific anymore. The *P-code* concept was not created for Ghidra, it is used by some compilers for other languages such as for Microsoft P-Code in Visual Basic or p-Machine for Pascal-P. These compilers can use it as an intermediate level before compiling a program for a specific processor or to be executed by a *p-code* machine to be run on different processors.

Ghidra's *p-code* specifies how an instruction manipulates data through registers and RAM. It is composed of *address spaces*, *varnodes*, and *operations*.

- **Address space** defines the data access space, such as RAM or registers.
- **Varnode** is a data in the *Address space*, defined by the address of the first byte and the size of this data (in bytes).
- **P-code operations** often referred as *opcode*, are operations that take one or more *varnodes* and optionally produce a *varnode*. *Opcodes* are operations similar to assembly instructions, such as *INT_ADD*, *COPY* or *BOO_XOR*.

Once generated, *p-code* is used by Ghidra for the data-flow analysis and then by the decompiler. The decompiler has to translate it to C programming language, which is way easier to do with *p-code* then with binary code.

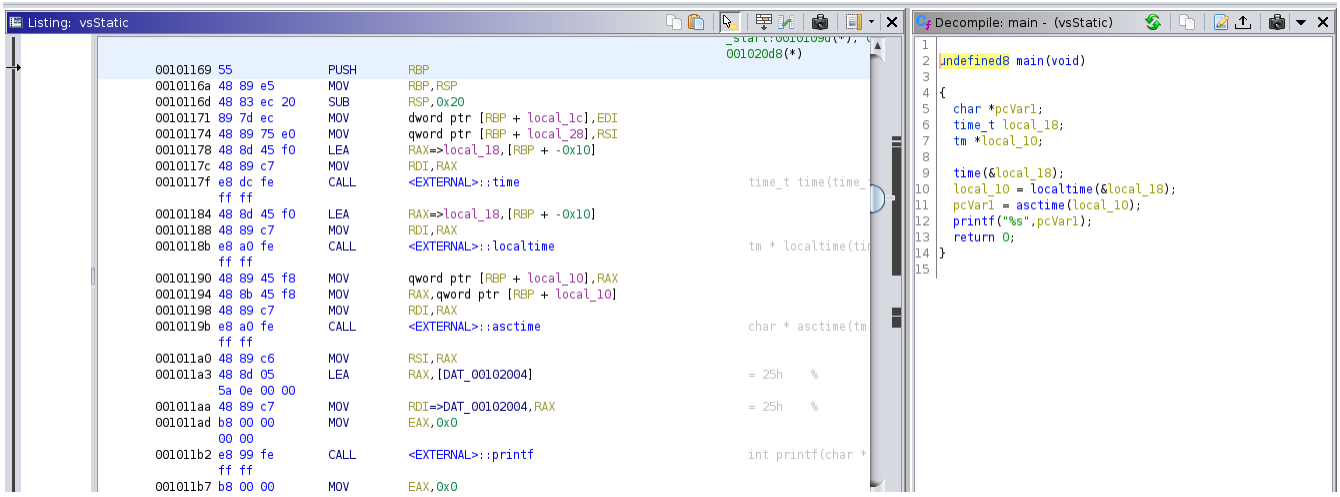


Figure 4.2: Decompilation example without FunctionID of a binary file with Ghidra (the binary file was compiled with no optimisation options and the symbol table was included in it).

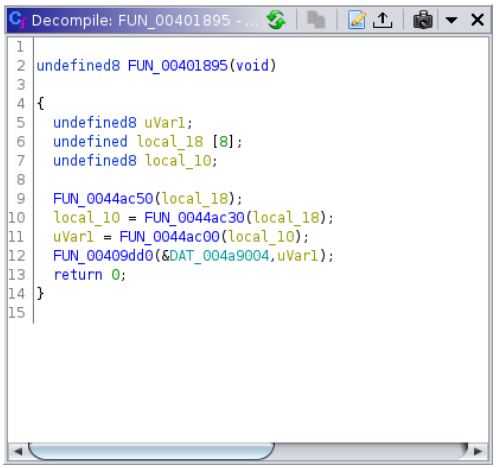
Figure 4.2 shows an example of the decompilation of a binary file of x86 64bit with Ghidra. On the left side, the *Listing View* shows the assembly code, and on the right side, there is the C code generated by the decompiler thanks to the use of the *p-code*. The decompiler is composed of several components (called *Analysers*) with different purposes such as ASCII strings and data types finders, function start search, or function name finder (FunctionID).

4.1.2 Challenge

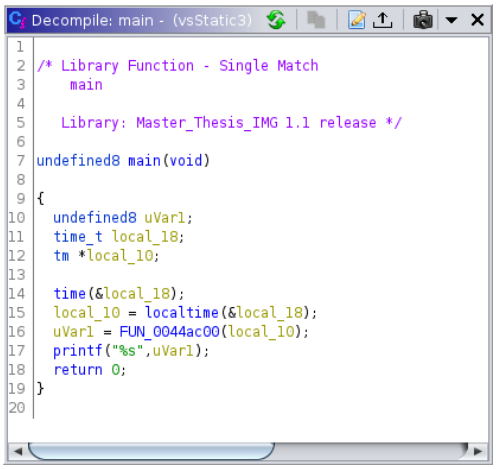
In Figure 4.2 the names of functions are well defined as *time()* or *printf()* and it is clear for the researcher that the `main` calls functions from the `time` library and then uses the famous `printf` function to print the results. If all function names are present in the decompiler view of Ghidra, that is because the binary files contained symbol names of functions, so the decompiler easily recognised it.

Often, programs and malware try to protect themselves against reverse engineering by hiding function names. Moreover, once a program is in production phase, the symbol table is often no longer needed. Therefore, symbol tables are removed to reduce the program space.

Figure 4.3a shows the output of the decompiler for the same function as in Figure 4.2, but this time the symbol table and the debugging information were removed by the compiler.



(a) Same program as in Figure 4.2, but this time the binary file was compiled with statically linked libraries and the symbol table was stripped



(b) Same binary file decompiled with Ghidra as in Figure 4.3a but this time the FunctionID Analyser was used to retrieve common function names (printf, localtime, etc.)

Figure 4.3: Ghidra Decompilation of a binary file without and with FunctionID Analyser

4.2 FunctionID Analyser

When beginning a reverse engineering project, engineers do not want to waste time reverse engineering library functions whose behaviour might be available in a `man` page, or in some source code, or online. Library functions are well known and do not require reverse engineering them. But statically linked libraries are libraries combined with the application source code in order to have only one output binary file. Differentiating application source code functions and functions from libraries can be a challenge when symbol tables are stripped. Fortunately, Ghidra has a plugin, an analyser that can recognise known functions, mark them, and use the known function signatures to help the researcher to understand the decompiled code. This plugin is called *FunctionID*.

4.2.1 Function Fingerprints

In order to recognise common functions from assembly language, FunctionID has to form a *fingerprint* of each function. Computing function fingerprints can be made using different techniques and are studied separately from Ghidra in studies such as *BinSign: Fingerprinting Binary Functions to Support Automated Analysis of Code Executables*[26].

FunctionID uses its own method. Two different hashes are computed for each function. Both are cumulative hashes over all the machine instructions (assembly-level code) of the function:

- **Full hash** includes mnemonic, some addressing mode information and register operands from instructions. The full hash is built to be robust against changes due to linking, the linker being the program that combines one or more object files into a single file.
- **Specific hash** also includes what is used for the full hash, but adds specific values and constant operands. The specific hash is built to help distinguish between variants of a function.

Additional information is also stored, such as the parents/children functions, library name, version, etc. During the analysing process, FunctionID then compares the hashes of each function that has no user-defined name (no symbol table, or the user has not set the function name by himself yet) with its fingerprint database.

Figure 4.3b shows the same function as in Figure 4.3a but after analysis by the FunctionID Analyser. This example demonstrates the efficacy of FunctionID and how it may help many researchers speed up their analysis. Indeed, the researcher on the program from Figure 4.3a will have more work to do to understand what this program does. However, thanks to FunctionID, the researcher with the decompiled form in Figure 4.3b can directly understand that the executable requests the local time, sends it to a mysterious function, and then prints the result. In this case even the mysterious function *FUN_0044ac00()* redirects directly to another function called *asctime_internal()*. The comprehension of the aim of this short application is easier and quicker with the help of the FunctionID analyser than without.

4.3 FunctionID databases

In the example of Figure 4.3, in order to translate the function name *FUN_0044ac30()* into *localtime()*, the two hashes have to be in a database used by FunctionID, along with other useful information. These FunctionID databases are called FiDb(s) and have the *.fidb* file extension. As mentioned in Section 4.2.1, a FiDb contains a set of meta-data and hashes describing library functions or common functions. Since assembly languages are different for each processor and FunctionID hashes are computed over assembly instructions, one FiDb is processor specific.

The FunctionID Plugin allows users to create custom FiDbs, attach them, and populate them with custom functions. Library functions are useful in FiDbs since they can be used everywhere, but custom functions or custom libraries can be used several times in the same program or by the same developer for different programs or software. Therefore, it is really useful for the researcher to save previously analysed functions in FiDbs in order to not do it again next time and to rely on FunctionID Analyser to mark it for him.

The official version of Ghidra comes with 10 FiDbs with functions from libraries from Microsoft Visual Studio for the x86 processor. There are 10 FiDbs to separate different versions of Visual Studio (2012, 2015, etc.), for different architectures (32bit and 64bit) and inside these FiDbs, two variants are present (debug and production variants).

4.3.1 Populate FiDbs

Once a user has analysed a function signature, he can use the FunctionID Plugin to populate his own FiDbs. There is a populate dialog to enter the library name alongside other useful information such as the compiler that has been used. However, the populate command takes all functions inside a specific binary and adds them all to the selected FiDb. The *main* function for example will be added to the database; however, it will certainly not be used as the body of the *main* function in other programs, which will be different.

4.4 Different Compilers and Processors

Each processor supports its own assembly language, and Ghidra currently supports more than 30 different processors. Each one of these has some variants such as ARMv4, ARMv4T, ARMv5, etc. Moreover, they can be of different machines (16bit, 32bit, or 64bit) and encoded in little or big endian. On top of all this, for the same architecture (combination of processors, variants, size, and endian), different compilers can be used. For example, for the processor x86 in default variant, on 32bit with little endian, Visual Studio, gcc, clang, Delphi, and Bordland C++ are supported by Ghidra.

These combinations of architecture and compilers are called *Languages* by Ghidra. There are exactly 206 different *Languages* supported in the version 10.1.2DEV of Ghidra. Even if some assembly codes are like each other for the same architecture, the same function can be then compiled in a lot of different assembly codes.

Furthermore, each compiler has its own set of arguments. Gcc for example supports hundreds of arguments and options that can change the compilation and thus the assembly code for the same function.

FunctionID databases (FiDbs) should be large enough to support as many different languages as possible. The default FiDbs provided with the official version of Ghidra only contain a small number of common functions and are available only for 2 of the 206 *Languages* supported by Ghidra.

4.5 First Workaround

Common libraries as *time* imported with "`# include <time.h>`" in C should be packed in FiDbs. There are a lot of common libraries that should be included in FiDbs. The GitHub user threatrack[40] started a similar project and already packed a large dataset of FiDbs in the repository called `ghidra-fidb-repo`[41]. It contains static libraries from:

- **CentOS version EL6 and EL7**. Each of them compiled for `i686` and `x86_64` processors. These contains entries for `openssl-static` and `glibc-static` for example.
- **gcc**. It contains libgcc for example, compiled for `68000`, `AARCH`, `ARM`, `AVR8`, `MIPS`, `pa-risc`, `PowerPC`, `Sparc`, `SuperH4`, `x86` processors. Each of them often in multiple architecture size (32bit, 64bit,...)
- **libc** for `68000`, `AARCH`, `ARM`, `AVR8`, `MIPS`, `PowerPC`, `Sparc`, `SuperH4`, `x86` processors also with multiple architecture size.
- **libsodium** for `x86` processors for 32 and 64 bit architecture.
- **qt5** from EL7 also for `x86` processors for 32 and 64 bit architecture.
- **SDL** from EL6 and EL7 also for `x86` processor for 32 and 64 bit architecture.
- **sigmoid** containing different versions of `openssl` and for different compilers such as `vs2008` or `vs2017`. Each time for `x86` processors and in 32 and 64 bit architecture.
- **teskalabs**. Containing `openssl` source code archives compiled for Android, iOS, Windows and Windows Phone 8.1 [1]with different compilers such as `Mingw-w64` or Android NDK.

[1]These names are common names for `AARCH64`, `ARM`, `MIPS` and `x86` processors with different architecture sizes.

For more information about included libraries, see threatrack's repository[41].

There are 959.055 entries in total in these FiDbs. It is a large dataset of compiled libraries for different processors and architectures, but it does not contain all libraries yet.

4.6 Hex-rays' version of FunctionID

Hex-rays developed a tool similar to FuncionID called "IDA F.L.I.R.T. [32]" which stands for Fast Library Identification and Recognition Technology. IDA F.L.I.R.T. signatures are an attempt to generate signatures from a subset of the first bytes of a function. In fact, each function is represented by a pattern. A pattern is the first 32 bytes of a function where all variant bytes (compared to other patterns already stored) are marked. This is an attempt to solve the challenge of recognising often reused code. These signatures are obtained by building common libraries with a variety of compilers. Once a library has been generated by the compiler, IDA can extract its signatures. Over time, many signatures can be accumulated for common libraries, which will be used to recognise already met functions in future analysis. IDA F.L.I.R.T uses a distinct database of pattern for each user. In fact, each user will have his own set of signatures depending on the signatures accumulated over time.

Some time later, Hex-rays decided to create Lumina Server [34] which is based on IDA F.L.I.R.T. but this time, signatures are pushed to a server, which is used for the entire community (a non-official private Lumina server can be found on GitHub[37]). This means that users are contributing to a general database. For the moment, Lumina Server is an experimental feature and since Hex-rays develops proprietary programs, there is not a lot of information about the implementation of Lumina. Nonetheless, Hex-rays says: "*it has many shortcomings and cannot handle many real-life scenarios. For example, it currently has no handling of collisions, database poisoning, DDOS attacks, it is quite slow, uses very limited recognition methods, etc.*"[34].

Part III

OpenFunctionID: a plugin to support FunctionID

Chapter 5

OpenFunctionID

5.1 Objectives

FunctionID Analyser as well as FunctionID databases and FunctionID plugin were presented at length in Part II. The description outlines some of the system's limitations. The goal of this master thesis is to overcome some of these drawbacks.

The first big limitation is the size of the provided FunctionID databases(FiDbs). With all 206 different *Languages*, Microsoft Visual Studio FiDbs seem a bit meagre. Even in the two *Languages* supported by these FiDbs, Visual Studio does not contain a lot of common libraries. Moreover, only common libraries are included, but it can be really useful to save custom functions that program designers will reuse or reimplement in other binary files. These limited basic FunctionID databases can be a problem for someone who does not have a lot of knowledge in the field of decompilation but wants to start a new project.

The second limitation is that when populating an existing FiDb, the user populates it with all functions from his binary. At a larger scale, it generates a lot of hashes and fills large FiDbs of not useful functions. It could be useful to populate FiDbs with only user-selected functions.

In addition, as covered in Chapter 2, the diffusion of information across the community is one of the greatest obstacles that must be overcome in reverse engineering. A FiDb is a significant component of this body of knowledge, since it can be viewed as a collection of all interesting functions discovered by a researcher. When a researcher discovers an intriguing function and adds it to their personal FiDb, they are acting on their own behalf.

However, in order to provide others with access to their discoveries, they will need to provide access to their FiDb, but there is no standardised method for sharing. It is crucial to have such a channel since it provides the opportunity to share knowledge and become aware of fresh findings.

The aim of this master thesis is to overcome these limitations by developing a tool with these objectives:

- Provide a large dataset of FunctionID Databases (FiDbs) from common libraries such as *OpenSSL*, and extend the work of *threatrack* (see Section 4.5).
- Include specific FiDbs for specific functions (not coming from common libraries) that may be used several times in different SRE projects.
- Automatic generation of FiDbs for different architectures and compilers options.
- Community driven: Open source databases that can be increased by the community
- Share knowledge and information for SRE engineers
- Integrate it easily with an existing installation of Ghidra
- Allow users to select a specific function to populate a FiDb and share it to increase community knowledge.
- Help researchers to quickly understand the purpose of a decompiled function

5.2 OpenFunctionID Plugin

The extendability of Ghidra is one of its most powerful features, if not the most powerful feature overall. Researchers from every country in the world can extend Ghidra's functionalities. They can add a new processor support for Ghidra by defining the processor-specific grammar with the **SLEIGH** language for example (see Section 3.2.1 for more details). As previously mentioned in Section 3.2.1, Scripts can be easily added to Ghidra thanks to its *Script Manager*). Ghidra is also open-source, so its dissambler, decompiler, and all other tools can be modified for specific purposes.

With cybersecurity threats increasing at a world scale, the collaboration between researchers in reverse engineering is more than ever mandatory. To overcome the limitations of FunctionID, this master thesis aims to develop a tool to overcome the objectives of Section 5.1. This new tool has to be available to everyone using Ghidra, and then it should not be a custom Ghidra with a different source code from the official release. A plugin can be easily installed over each Ghidra installation and thus be used by a lot of users.

A plugin for Ghidra is like a tool attached to the framework which can interact with other tools of Ghidra, add menu items, and new GUI-windows. The plugin developed in this master thesis is called *OpenFunctionID* often shortened as *OpenFiDb*.

5.2.1 OpenFiDb Overview

OpenFiDb is a plugin that can be easily attached to an existing Ghidra installation and it allows a collaborative way to use the FunctionID functionality of Ghidra. The idea behind the plugin is to create a collaborative universal database of FunctionID Databases (FiDbs). Ideally, it would contain all ever written functions, compiled with all possible compilers with all the compiler's options over each processor. With that, the researcher would run the FunctionID Analyser and it would be able to identify each function of the binary file. To achieve it, the database has to be well designed and complete.

This database of FiDbs should then contain two different types of functions:

- **Library Functions:** As said and shown earlier, even simple common libraries are not included in the FiDbs provided in the official release of Ghidra. Static libraries, such as *OpenSSL*, for each processor should be packed into FiDbs and available on the server.
- **Users Functions:** On the other hand, one major feature of OpenFiDb is to handle not only common libraries, but also custom libraries and functions developed by companies or programmers.

5.2.2 Base FunctionID Databases

Already in 2019, the goal of *threatrack* (see Section 4.5) was to build a collaborative database with libraries from everywhere. The idea was brilliant and *threatrack* added new libraries mostly in September and October 2019, but since then no new libraries have been added. Maybe the main reason of this failure is that it has not been integrated to Ghidra, so users have to separately download FiDbs files and put them inside their Ghidra installation. Then, regularly check if a new version has been released or new libraries have been added. Moreover, Ghidra is not able to automatically add new FiDb, so users have to add them one by one through the GUI of Ghidra, or build Ghidra from source[1].

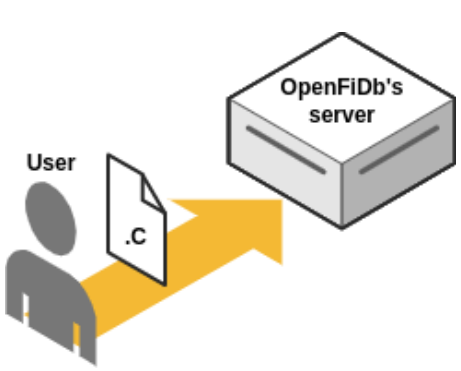
To counter these limitations, OpenFiDb is integrated in Ghidra and allows users to easily download and use these FiDbs. OpenFiDb database contains these 959.055 entries computed by *threatrack* and can be easily updated with new libraries through pull request on GitHub. One major improvement is that once the database is updated, it is immediately available to all OpenFunctionID users through the plugin in Ghidra.

5.2.3 Community FunctionID Databases

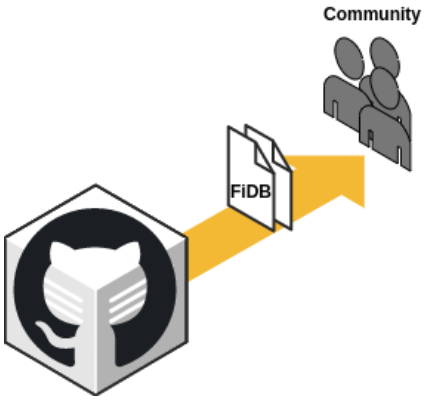
In addition to these common libraries, one powerful aspect of OpenFiDb concerns the sharing of knowledge among researchers. If a user has analysed a function in Ghidra, he can upload it to the OpenFiDb server that will process it and generate new FiDbs containing the function uploaded. The function will be compiled in several ways to generate a lot of different assembly code and supports different processors/architectures/compilers options. The whole process of the OpenFiDb Server will be explained in Section 5.4.

After analysis, a user can upload his decompiled analysed C code to OpenFiDb's server. The server will process the file, compile it for different architecture/processors/compilers, and then upload the new FiDbs to the GitHub repository. The whole community is able to download these new FiDbs thanks to OpenFiDb plugin integrated to Ghidra. This process is summarised in Figure 5.1.

[1]Two other ways exists to add new fidb to Ghidra: using a *buggy way* or using pre-unpacked FiDbs (with the .fidbf extensions as explained in the README of his repository[41]). But they are not recommended.



(a) Pushing C code from user to Server



(b) Pulling from GitHub to community

Figure 5.1: OpenFiDb Principle

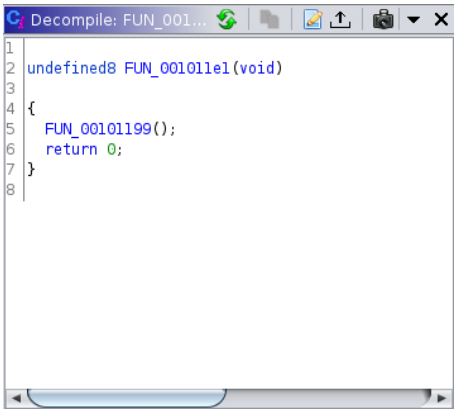
For explication purpose, here is an example with a simple harmless function. `helloGhidra()` only prints two strings. The source code is:

```c
#include <stdio.h>

void helloGhidra(){
    printf("Hello Ghidra!");
    /*
    * Here are some comments
    */
    int a = 45;
    int b = a + 55;
    printf("Here is the result: %d",b);
    return;
}

int main(int argc, char **argv){
    helloGhidra();
    return 0;
}

```

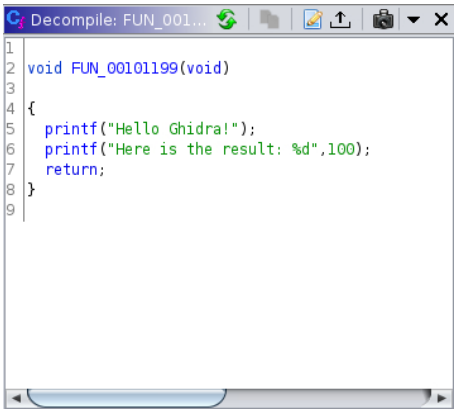
Listing 5.1: Source code to be compiled

Once this simple C code is compiled, some optimisations are done by the compiler and the function names are stripped from the binary file.

Figure 5.3 shows the `main` and `helloghidra` functions after decompilation of Ghidra. Once the researcher has analysed this function, he can set a name on the `FUN_00101199()` to identify it more easily. To help other researchers around the globe, he can upload his analysis to the community databases of OpenFiDb. The decompiled C code is then sent to OpenFiDb server, processed, and then the repository of FiDbs is updated. Server's side processing will be explained in detail in Section 5.4.



(a) Function decompiled without using OpenFiDb databases

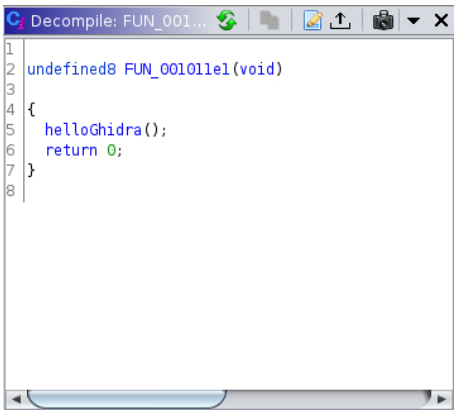


(b) Function decompiled before collaboration to OpenFiDb

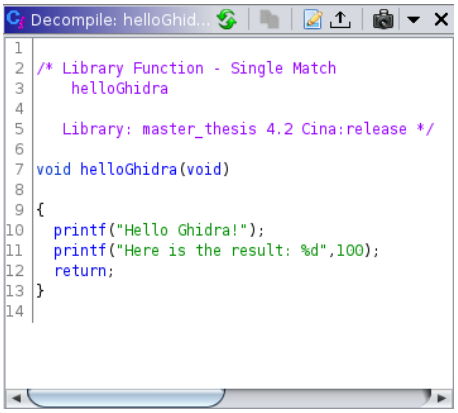
Figure 5.2: Ghidra Decompilation of a binary file before the community has updated OpenFiDb databases

In a completely different part of the world, another researcher is maybe analysing a new binary file which uses the same custom function `helloghidra()`. Using Ghidra, he will look at a function like the one decompiled in Figure 5.2a which is not understandable at all. This researcher can use the OpenFiDb plugin to download and attach community FiDbs databases using menu buttons. The way OpenFiDb is integrated with Ghidra will be presented in Chapter 6.

After applying FunctionID, Ghidra will show the code decompiled in Figure 5.3a. Thanks to this community knowledge, the researcher can move on and analyse another function. In fact, FunctionID identified the function and added comments showing to the user where it has found the name `helloghidra` as shown in the Figure 5.3b. *"master_thesis 4.2 Cina:release"* are information that the first researcher encoded to categorise and describe the function he uploaded to OpenFiDb servers. This information also helps all other users of OpenFiDb to understand where this analysis comes from.



(a) Same function as in subfigure 5.2a but using OpenFiDb database



(b) Same function as in subfigure 5.2b but after a user has uploaded `helloghidra()` to OpenFiDb

Figure 5.3: Ghidra Decompilation of a binary file after the community has updated OpenFiDb databases

5.2.4 Ghidra Development Documentation

Ghidra is written in Java, so its plugins have to be written in Java also. Ghidra was released in 2019, so 3 years ago. Since then, the community has been growing every day, with an increasing number of Youtube videos, online tutorials, and forums. Ghidra is well documented online and in Ghidra itself. It contains a help guide, explaining tools, and how to use them. As for forums, Youtube's videos, and online tutorials, but also in some books as *The Ghidra Book: The Definitive Guide*[13] or *Ghidra Software Reverse Engineering for Beginners*[11], the main goal is to present Ghidra and help users to understand how it works and how to use it.

As of this date, the community of extension/plugin/script developers for Ghidra is small. There is not much documentation on how to build plugins. There is a Telegram discussion called *GHIDRA - Development*[2] but there is only 85 members and nothing relates to FunctionID. Ghidra is extendable and was built to be. Therefore, the National Security Agency generated a JavaDoc and made it available for everyone and source code is open source on GitHub[12]. To understand how to develop a plugin, developers have to analyse Ghidra source code, understand which the purpose of main classes, how to use them, what is allowed, what is not, etc. It is a kind of reverse engineering task itself.

[2]Accessible on `https://t.me/GhidraRE_dev`

To extend Ghidra, the Ghidra Team made an Eclipse plugin that is able to build plugins integrated to Ghidra. Some slides of information are also available in the development build of Ghidra, but most of the work has to be done by developers.

5.3 Additional Script

After OpenFuncitonID has been tested on small programs, an idea came up. In fact, when a user tries to identify a function with this new plugin, it will correctly find the signature of the function (given that it is in the database) but sometimes, the name of the function is not enough to understand what it exactly does. Therefore, researchers had the reflex to lookup the function's name on the internet and try to find a manual page. It is clear that this process can be handled by a simple script that would do some research for this manual page. This is the reason it has been decided to add such script in the plugin.

How this script works is quite simple. It will get the current selected function in Ghidra and it will take its name. Then, it will try to find a manual page using the function's name. If it finds anything, it will open a browser with the corresponding manual page.

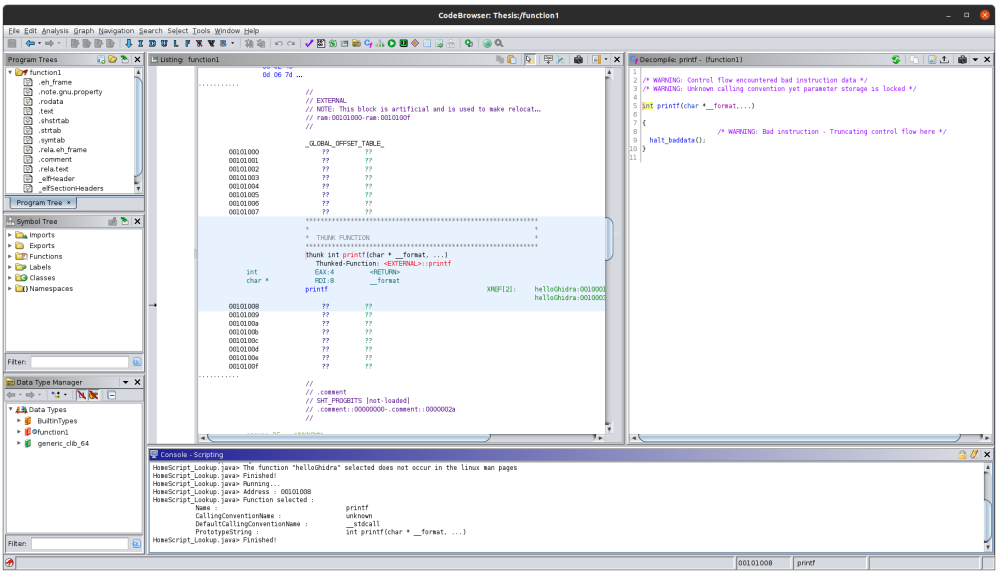


Figure 5.4: Example of lookup on the function printf

For example, in the figure 5.4, the function *printf* has been correctly identified, but let's say for the sake of this example that the behaviour of this function is unknown for the person who is doing the analysis. Thus, to try to understand what this identified function does, the researcher will launch the script that will initiate a research of a manual page for this function. This script can be launched by pressing its icon (represented as a magnifying glass, it is shown in the top menu bar of Figure 5.4) in the Ghidra GUI. More details of its usage are presented in Section 6.1.

Once the manual page has been successfully found, the script will open a browser containing it. Figure 5.5 illustrates the result of the execution of the script on the previous example.

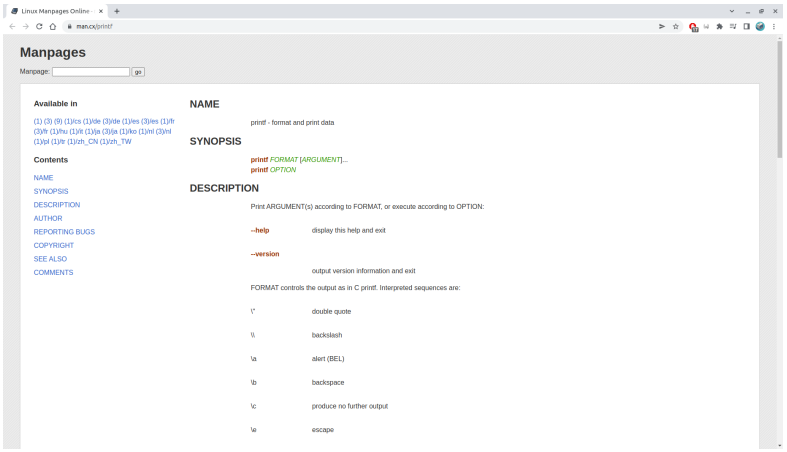


Figure 5.5: Browser opened on the manual page of printf

5.4 OpenFunctionID Server

As explained previously, the client will send a request to the server to add a new function to OpenFiDb's repository. In this section, the operation of the processing of requests will be explained, starting from the reception of the request to the update of the repository.

To correctly work, the server has some prerequisite. It needs Ghidra, java, git and 7-zip installed. The server will listen on the port 8080 and take one request at a time. First, it will decode this request and retrieve the information it contains:

- The code of the function (written in c)
- The *Language* (in Ghidra's sense, e.g. x86:LE:64:default) used for compilation
- The version of the library that the function comes from
- The library variant that the function comes from
- The id of the user

Once the request is decoded and the information extracted, the process of populating FiDbs can start. This process consists of:

1. Pulling or cloning OpenFiDb's repository
2. Compiling
3. Creating a static library
4. Unpacking library
5. Importing into a new Ghidra project
6. Verifying that everything worked
7. Populating FiDbs

5.4.1 Pulling or cloning OpenFiDb's repository

The first thing that the server should do is to update its local OpenFiDb's repository. The OpenFiDb server saves its data with the help of a third party, which also assures that it is constantly updated. This database maintains one FiDb per user in addition to a large FiDb containing all submitted functions. The third party is known as GitHub, and anybody can freely access the repository because it is public: `https://github.com/Cyjanss/OpenFiDb`.

Due to the fact that this repository is open to the general public, anybody may clone or download it without restriction. Manually or using the OpenFiDb plugin, this task can be accomplished. Nonetheless, it is impossible for any user to push anything directly to this repository. A lot of reasons contributed to these decisions:

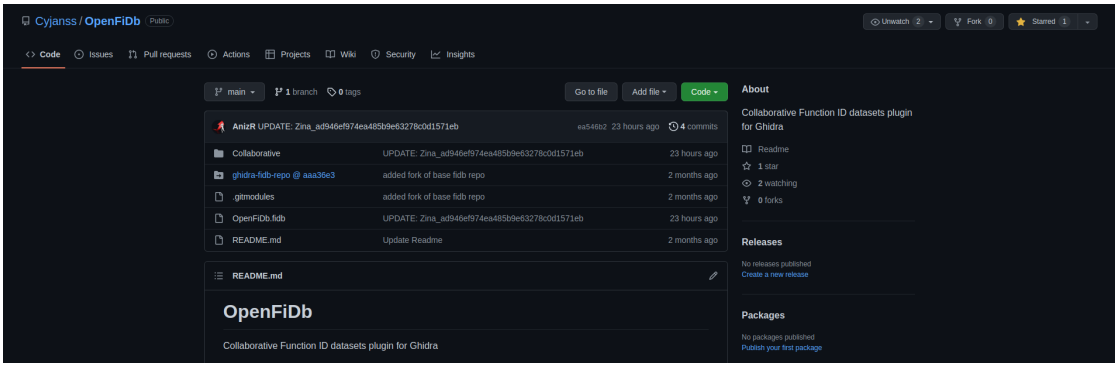


Figure 5.6: OpenFiDb's repository

Sharing knowledge As has been made clear throughout this master thesis, one of the key goals is to disseminate knowledge on the reverse engineering process. This goal necessitates that OpenFiDb is accessible to all users, which is the fundamental reason why the repository is accessible to the general public. Moreover, the channel presenting this information must guarantee that all users are aware of this channel. This clarifies why the choice to use GitHub as the communication channel was made. Due to Ghidra being an open source project hosted on GitHub[12], its community makes extensive use of GitHub.

Supervising It is vital to have a server that can act as a liaison between the repository and the users for the purpose of adding new information. It offers some supervision over what is pushed or not pushed to the repository; however, this subject will be discussed in further detail later on.

Maintaining a structure When all users have free access to a repository and may submit any file, it can be very difficult to maintain any sort of order inside the repository. However, it is essential to be structured when building a tool to encourage individuals to collaborate and share their knowledge. It was decided to assign this job to a server to ensure that the format is maintained properly. This ensures that the repository's organisational structure will always be maintained under strict supervision.

Being more complete As previously explained in Section 4.4, Ghidra is compatible with a wide range of *Languages*. This means that every C function may be compiled in a multitude of ways by only modifying the processor, the architecture, or the compiler. It implies the ability to generate several machine codes from the same set of source code. Changing the parameters of a given compiler might also cause the compiler to create different machine code. If a user wants to push one function and populate completely his OpenFiDb database, he would need substantial resources to build his function in every *Language* supported by Ghidra. The decision has been made to complete the compilation and production of FiDbs on a server. This choice was made not just to ensure proper execution, but also to facilitate user convenience. Then, this server is responsible for pushing the produced FiDbs to the repository.

5.4.2 Compiling

The second step that the server has to achieve is to compile the function received from users. As previously mentioned, a user sends the source code of a function he discovered. This function needs to be compiled and transformed into machine code to be added to a FiDb. One thing that needs to be considered during this process is that changing of processor, architecture, or compiler may result to produce a different machine code. Even changing the parameters of the compilation can change the resulting machine code. This is explained more precisely in Section 4.4. All of this implies that the server has to compile the function with as many settings as possible.

Figure 5.7 is a piece of code coming from OpenFiDb server's implementation showing the compilation using gcc and using some optimisation parameters:

```
#gcc x86_64-linux-gnu no optimization
gcc -Wall -O0 -c "$1.c" -o "$1_O0.o"
#gcc x86_64-linux-gnu optimizations 1, 2 and 3
gcc -Wall -O1 -c "$1.c" -o "$1_O1.o"
gcc -Wall -O2 -c "$1.c" -o "$1_O2.o"
gcc -Wall -O3 -c "$1.c" -o "$1_O3.o"
gcc -Wall -Os -c "$1.c" -o "$1_Os.o"
```

Figure 5.7: Example of compilation

To be totally complete, the server should be able to create machine code using every *Language* supported by Ghidra. It means 206 *Languages* for the version 10.1.12DEV of Ghidra. This has not been done yet but is considered an extension of this tool. This extension is discussed in more detail in Chapter 8.

5.4.3 Creating a static library

The third step in the server's process is to create a static library containing every compiled version of the function. This is a small step but important because it allows to use parts of scripts that will be presented later. Moreover, it allows to focus every version of the function in one place. This step is using the Linux command *ar*[16] to create an archive. This step is done to concentrate every function in one file for a reason of practicality but also to have a clear file structure when unpacking this library during the next step.

5.4.4 Unpacking library

The fourth step is to unpack the archive that has been created in the previous step but respecting a given structure. This is done using a small script and 7-zip [29] which is a free and open source software. The resulting structure is the following:

```
+-- \lib
|   |-- \userID
|   |   |-- \Library_Name
|   |   |   |-- \Library_Version
|   |   |   |   |-- \Library_Variant
|   |   |   |   |   |-- function_O0.o
|   |   |   |   |   |-- function_O1.o
|   |   |   |   |       ...
|   |   |   |   |   `-- function_Os.o
```

It is mandatory to use this special structure for clarity in the code but also because some parts of scripts are coming from the work of "threatrack" [39] which inspired parts of the solution proposed in this thesis. He realised a generator of FiDbs but only using libraries that are downloaded online and compiled with one specific compiler. In fact, the work that "threatrack" has done is complementary to the work that has been done during this thesis. This is why, as explained in Section 5.2 the FiDbs generated by him have been added via a link in the OpenFiDb repository. This step is done mainly to set a project structure when importing the functions in Ghidra. It allows to characterise the function in the Ghidra project using the hierarchy within the project.

5.4.5 Importing into a new Ghidra project

The fifth step starts with the creation of a temporary new Ghidra project. This project is temporary because once the server's process is finished, it will delete it. Once the project is created, the server will start importing the previously unpacked library into Ghidra. It is done using a script that runs Ghidra in *headless* mode.

The *headless* mode of Ghidra is mandatory in this case because a graphical interface is not needed and it is important to be able to interact with Ghidra using a command line. The program "analyzeHeadless" located in Ghidra (in the folder /support) allows to run Ghidra in *headless* mode which is available by default with a basic installation.

After importation, Ghidra will start a standard analysis of the functions as if it was a common project imported. This step is done to generate or to populate a FunctionID database because Ghidra needs a project with functions to add. Moreover, Ghidra needs to do some analysis to identify the functions within the project before being able to populate a FiDb.

5.4.6 Verifying that everything worked

The sixth step is to control that the previous phase was a success. In fact, during the fifth step, a log file is produced by Ghidra. This file will be browsed searching for any errors that could have occurred during the import or analysis phase. If an error is discovered, the process stops here, otherwise, it can continue with the last step.

This step has just the role of ensuring that there were not any failures during the importation of the project. It ensures that functions that were not correctly imported are not added in any FiDb. It enforces the integrity of the FiDbs.

5.4.7 Populating FiDbs

The final step is to populate FiDbs using the results of previous phases. Once more, for this step, Ghidra will be run in *headless* mode.

First of all, it will start by creating a FiDb using the user id combined with the *Language* used as name. Then, it will populate this database using the project created in the fifth step. If the database already exists, it will directly populate the corresponding database. This database is directly the database located in the folder Collaborative in the server's local repository OpenFiDb.

Then, it will populate the "general" FiDb named *OpenFiDb.fidb* using functions previously used to populate the previous FiDbs. After each populating, a small script called "RepackFid_Headless.java" will be used to repack FID databases and eliminate unused blocks and possibly make indices inside those databases more efficient.

Finally, the new FiDbs are committed and pushed to OpenFiDb's repository. Once the push is done, the server is ready for any new request and the server's process can restart.

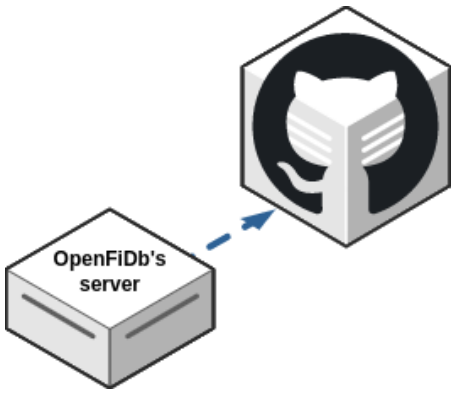


Figure 5.8: Server to git

5.4.8 Summary

Mechanisms inside the server have been described in detail, but it is important to also have an overview of the entire server. Figure 5.8 illustrates the whole process starting from the reception of the request (in yellow on the schema) and followed by every sub-process (in blue on the schema).

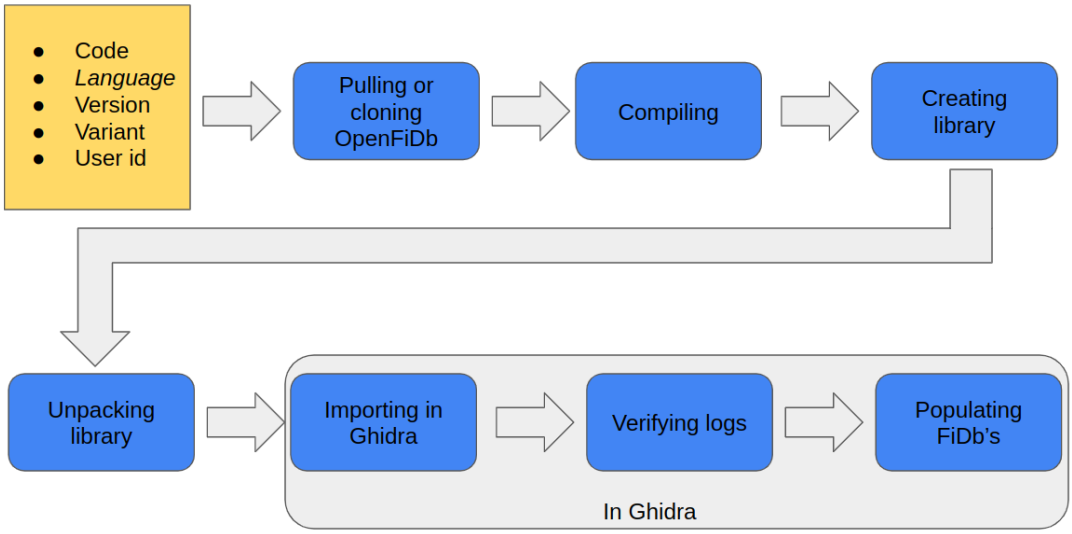


Figure 5.9: Schema of the server's process

Chapter 6

Integration

6.1 Plugin

6.1.1 Requirements

As a reminder, OpenFiDb is the shortened name for OpenFunctionID. The plugin name is thus OpenFunctionID. One feature of this plugin is that it can be easily integrated to Ghidra. However, it requires some prerequisites:

- An installation of Ghidra [1] (Version 10.1.2 or later).
- A recent installation of the tool `git` accessible in `headless` mode. Indeed, OpenFiDb uses the command line `git` to pull FiDbs from the GitHub repository.
- The latest release of OpenFunctionID plugin available on GitHub. It is a zip file with a name like *"ghidra_10.1.2_DEV_20220516_OpenFunctionID"*.
- For now, the additional script (see Section 5.3) is only available for Linux users since it calls the `man` command line tool to check if a function has a man-page.

6.1.2 Installation

For the sake of clarity and conciseness, the OpenFunctionID plugin can be integrated with an existing installation of Ghidra. Once Ghidra is launched, the project view opens as in previous Figure 3.4a. There, to install OpenFunctionID, the user has to go with menus to `File->Install Extensions...`. This shows a window where he should add the `zip` file of OpenFunctionID thanks to the green plus icon.

[1] Releases of Ghidra: `https://github.com/NationalSecurityAgency/ghidra/releases`

This window now shows a small description of the plugin as in Figure A.1. After making sure the plugin is checked, Ghidra has to be restarted to integrate the OpenFunctionID plugin.

OpenFunctionID can be integrated with all Ghidra tools. As a result, if a new tool is produced, OpenFunctionID may be included into it as well. For now, it is only useful in the *Code Browser* tool. Once opened it will prompt the user if he wants to configure the new plugin found (OpenFunctionID). Figure A.2 shows the dialog for the configuration of the plugin. It also shows menu actions that will be added to the current tool (*CodeBrowser*) such as *Pull the repo*. If this step is skipped, it can be found under the tool `File->Configure...` menu entry. It will show a dialog shown in Figure A.3. Given that OpenFunctionID is an external plugin, its configuration is under the *Experimental* section of this window.

Configure FunctionID Plugin

Figure A.3 shows multiple plugins that can be added to the current tool. *FunctionID* is one of them. The user has to select it in order to be able to create new FiDbs, populate an existing one, or choose active FiDbs. It is not mandatory to activate this plugin to use OpenFunctionID.

6.1.3 Usage

As just explained, once OpenFunctionID is activated in a tool, it adds some menu-actions. These menus are shown in Figure A.4, they can be accessed through `Tools->Function ID->OpenFiDb`.

Pull the Repository

The first thing a user of OpenFunctionID would want is to download all available FiDbs from the OpenFiDb repository to be able to use them with FunctionID. This can be done while accessing to the *Pull the repo* menu entry. Some popup windows inform the user and then all FiDbs are downloaded to the local installation of OpenFunctionID. Currently the repository contains 64 MiB of FiDbs and is expected to grow thanks to the community. Thus a loading screen is displayed while downloading. The *CodeBrowser* tool has a scripting console where some information is printed by OpenFunctionID to inform the user of the process. An example of this console after download of the repository is shown in Figure A.5. At first it will clone the repository, afterwards it will only pull incoming changes.

Once the download is finished, a popup window like in Figure A.6 appears prompting the user which FiDb he wants to use. Although all FiDbs have been uploaded, not all of them will be of interest to all researchers. They have the possibility to check Fidbs that they want to attach and use, depending on the processor for example. The name of each FiDb is composed by a name and the *Language*. After this, OpenFiDb Fidbs are used by FunctionID to recognise function names. If the user wants to select and use other FiDbs, he can use the FunctionID plugin activated (see Section 6.1.2) and use menu actions to open a window similar to the one shown in Figure A.6.

Discard Local Changes

FunctionID Plugin allows users to populate an existing FiDb. Although it is not recommended nor useful to populate an FiDb coming from OpenFiDb, it is possible. If it happens, the pull action described here before would not work since local changes are not committed. While OpenFiDb could have handled it itself and automatically discarded local changes, it was decided to not doing it without user agreement. To allow him to save elsewhere his local changes to FiDbs, another button in the menu is created and called *Discard local changes*.

Delete All OpenFiDb files

If a user wants to get rid of all the FiDbs provided by OpenFunctionID, he can choose the last menu entry called *Delete all OpenFiDb files*. It will remove the folder containing the local version of the OpenFiDb repository.

Upload function to OpenFiDb server

OpenFunctionID functionalities do not only include the use of FiDbs of the OpenFiDb repository. Another aspect of this plugin is to be able to send analysed functions to the OpenFiDb server in order to improve the quantity and quality of collaborative FiDbs. This functionality is not available as a menu entry since the user can choose which decompiled function he wants to send. In the decompiled window, OpenFunctionID adds a new menu icon as shown in Figure A.7. It will get the decompiled and analysed C code and send it to the OpenFiDb server. As explained in Section 5.4, the server needs some other information such as the library name, version, and *Language*. Therefore, a popup window like in Figure A.8 appears, prompting the user to fill some information about the function he is about to send.

Afterwards, all these information, including the C code, will be sent to the server. The server will process it and eventually create a new FiDb or populate an existing one with the new function. This FiDb will also be pushed to the OpenFiDb repository and other users will be able to use it for further analysis.

6.2 Lookup Script

The lookup script is simple to use, but it first needs to be enabled. It can be done using the toolbar and opening the script manager: `Window->Script Manager`. The script manager resumes every script available on the Ghidra installation. To enable the lookup script, the user needs to check the box on the line containing *HomeScript_Lookup.java*.

Once the script is enabled, it is ready to be used. The only thing it needs is a connection to the internet and to have a function selected in the decompiler's window. Then it will start a research on its own as briefly described in Section 5.3.

6.3 Server

The original idea was to have one server and one GitHub, but OpenFunctionID can also be used in a smaller scale. For example, for a team working on a specific project with some privacy issues regarding the code they are studying. In this case, a private server is needed. A Ghidra server and an OpenFunctionID server may seem to do the same job, but that is not the case at all. A Ghidra server has the goal to completely do the reverse engineering process on a server and not on a researcher's laptop. In fact, Ghidra is installed and running on the server. While the goal of an OpenFunctionID server is to share FiDbs produced by searchers and each one of them has their own version of Ghidra. In the next section, it will be described how to setup such OpenFunctionID server.

6.3.1 Prerequisites

In order to deploy OpenFiDb on a private server, it is recommended to use Linux since a lot of Linux functions are used. Moreover, some software prerequisites are needed:

- An installation of Ghidra [2] (Version 10.1.2 or later) because the server will use Ghidra in *headless* mode to populate FiDbs.
- A version of Java compatible with Ghidra's installed version. In fact, Ghidra's scripts that are used are written in Java.
- A recent installation of 7-zip [29]
- A recent installation of the tool *git* and set it up with an account that has access to OpenFiDb's repository or another repository if the objective is to avoid publishing functions on a public repository.

Once all these programs are installed, two environment variables have to be set on the server:

- "`GHIDRA_HOME`": this environment variable should contain the complete path to the server's Ghidra installation.
- "`GHIDRA_PROJ`": this environment variable should contain the complete path to the server's Ghidra project directory. This directory is where every project created by Ghidra will be.

The last thing to do is to clone the repository containing the code of the OpenFiDb's Server. The code running the server is in the folder "*OpenFiDbServer*".

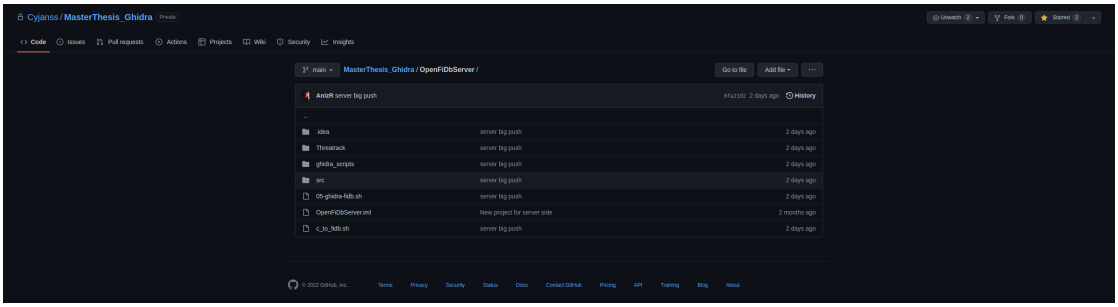


Figure 6.1: Directory containing the OpenFiDb's Server

[2] Releases of Ghidra: `https://github.com/NationalSecurityAgency/ghidra/releases`

6.3.2 Booting up

The server can almost be launched but it needs to be compiled. It can be done using the following lines (assuming that the user is at the root of the GitHub repository):

```
$ cd OpenFiDbServer
$ cd src
$ javac OpenFiDbServer.java
$ mv OpenFiDbServer.class ..

```

After having compiled the server, it can now be started by using the following command:

```
$ java OpenFiDbServer

```

By default, the server is listening on port 8080. It can be observed on the first line in Figure 6.2 representing the first lines that are produced after booting the server up and receiving a request. In this Figure, the outputs of the step that clones OpenFiDb's repository, compiles, and creates a static library are shown. As described in Section 5.4, these are the first three steps of the server's process.

```
Started on port 8080
Waiting for connection
Connection, receiving data.
HTTP-HEADER: POST / HTTP/1.1
HTTP-HEADER: Content-Type: application/x-www-form-urlencoded
HTTP-HEADER: unique_id: report_fe83cd6232024dd7afd61b526087f646
HTTP-HEADER: User-Agent: Java/11.0.10
HTTP-HEADER: Host: localhost:8080
HTTP-HEADER: Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
HTTP-HEADER: Connection: keep-alive
HTTP-HEADER: Content-Length: 428
POST-DATAI: 428
POST-DATA: rO0ABXNyABFqYXZhLnV0aWwuSGFzaE1hcAUH2sHDFmDRAwACRgAKbG9hZEZhY3RvckkACXRocmVzaG9sZHhwP0AAAA
AAAAx3CAAAABAAAAAGdAAFY29kZUN0AGsKdm9pZCBoZWxsb0doaWRyYSh2b2lkKQoKewogIHByaW50ZigiSGVsbG8gR2hpZHJhISI
pOwogIHByaW50ZigiSGVyZSBpcyB0aGUgcmVzdWx0OiAlZCIsMTAwKTsKICByZXR1cm47Cn0KCnQADmxpYnJhcnlWYXJpYW50dAAG
cmVwb3J0dAAHdXNlcl9pZHQABnJlcG9ydHQACmxhbmd1YWdlSWR0AARsYW5ndAAObGlicmFyeVZlcnNpb250AAZyZXBvcnR0ABFsa
WJyYXJ5RmFtaWx5TmFtZXQABnJlcG9ydHg=
Success!

 [Cloning OpenFiDb]


 [Generating library]

name of file: report_fe83cd6232024dd7afd61b526087f646
a - report_fe83cd6232024dd7afd61b526087f646_O0.o
a - report_fe83cd6232024dd7afd61b526087f646_O1.o
a - report_fe83cd6232024dd7afd61b526087f646_O2.o
a - report_fe83cd6232024dd7afd61b526087f646_O3.o
a - report_fe83cd6232024dd7afd61b526087f646_Os.o
```

Figure 6.2: Example of output on the server's terminal

Chapter 7

Security of OpenFunctionID

Reverse engineering is a specific subject within cyber-security. As a reminder to explain reverse engineering, this thesis began by discussing its security applications (Section 2.2). However, what kind of security issues can OpenFunctionID encounter?

7.1 Server Security

Firstly, what is the security of the OpenFunctionID Server? Can a hacker hack the server ? The first thing to note is that the server is not in the production phase. OpenFunctionID is a prototype but the server is not yet online. It is possible to deploy OpenFiDb server on a dedicated server (see Section 6.3), but at that time the communication protocols should be reviewed. For now, all communications are done locally with *HTTP POST* requests on port 8080. All information is encapsulated into a JAVA *HashMap*, then an encoder (called `encodeSerializable()` in the project) encodes it to a `byte` array. This array is sent via an *HTTP* connection to the server. The server has an antivalent decoder called `decodeDeserialize()` that decodes the *byte* array to a JAVA `HashMap`. There is no security behind this connection. Some improvements can be made to the connection such as:

- Add an `SSL` support and communicate through `HTTPS` against *man-in-the-middle* attacks. It requires a server certificate and that was not relevant for the current state of the OpenFunctionID project since all communications were done locally.
- Add a cryptographic function. For now, the encoder and decoder do not use any form of cryptography, but they could have one in order to make the `byte` array sent unreadable to anyone who does not have the specific decoder.
- Other protections, such as against DDoS attacks are often provided by website hosts.

Moreover, C code is processed as it comes from users. They may be able to send harmful code executable. The server does not execute C code coming from users, but it saves and compiles it. These files are temporary, they are deleted once the FiDb has been populated, but hackers could always find a way to grant them access to the server. For example, the inputs such as *Library name* written by the user in the send window (Figure A.8) are not checked and a hacker could use it to input some harmful code.

To summarise, the server is not yet deployed in production and thus is not protected against cybercriminals.

7.2 Database Security

The security issues in the previous section were server side. It is important to look about security issues that can be found in the open databases of FiDbs. OpenFiDb database is considered as open since everyone can download its content, and its data comes from the community. However, the data does not come directly from OpenFunctionID users, there is the OpenFiDb server between users and the database.

As explained before, the server receives C code to be able to use it and compile it in different ways. But another purpose of the server is to guarantee the integrity of the database. No one can push a harmful file hidden behind a *.fidb* file extension on the FiDb database. Only the server has the ability to push new files on it and FiDbs are generated thanks to the server's Ghidra instance.

As a result, files on the database will only be true FiDb containing hashes of compiled version of user's C source code. In malware analysis, functions analysed are often harmful, but the server will only store hashes of these harmful functions. A user of OpenFunctionID plugin that downloads the repository of FiDbs will not download harmful files that can hack his computer.

Something that a hacker can do is hide his harmful code behind a trusted function name. It would be beneficial for him to upload to OpenFiDb his function with a name that could be interpreted as a safe function by researchers. Therefore, his malware could take more time to be analysed by cybersecurity experts. However, function names identified by FunctionID will always have automatic comments on top of it to inform the user that the function has been recognised thanks to a FiDb and showing library names alongside other information.

Figure 5.3b shows the comment generated by FunctionID. There, the user identification name is *Cina*. The user identification is not yet implemented on OpenFiDb but is a future work that should be done before releasing it into production. It is discussed in Section 7.3. Once implemented, a researcher should be able to identify from whom the function name proposition is coming from and thus be aware that this could be wrong. Even for not harmful functions, one can submit incorrect data while believing he is doing the right thing.

Small unintended mistakes or intentional hiding between trusted function names should be fixed by the community itself thanks to the *Multiple Matches* functionality of FunctionID. Indeed, if another researcher uploads the same function with another function name, two names for the same function will be available on OpenFiDb FiDbs. When analysing, FunctionID will not be able to narrow down to a single function name. In this case, it will show the different function name matches and it will be noted *Multiple Matches* in the generated comments above the function. Consequently, the hacker cannot hide his function forever and could be reported as a not trusted user in a future version of OpenFunctionID.

7.3 User Identification

The OpenFunctionID authentication method is not safe and can be improved. In fact, the userID is accepted as an input when a new function is pushed and is not verified. This can be dangerous since a user can use another user's identity and send any function to the server by impersonating that user. In Chapter 8, methods for enhancing the security of user identification are described.

7.4 Malicious Usage

When developing a programme like OpenFunctionID, it is evident that it may be exploited for malevolent purposes. Malware developers, for instance, may utilise OpenFunctionID to determine if their application can be decompiled and the functions it contains easily identifiable. They can attempt to use obfuscation tactics, but as detailed in Chapter 8, these methods are not particularly effective. Finally, the quality of a tool depends solely on what its final user accomplishes with it.

Chapter 8

Analysis, Limitations and Future Work

8.1 Analysis

8.1.1 Server Load

Since the OpenFiDb server should be available to all OpenFunctionID users, it should have enough computational power, RAM, and storage space. In further implementation of the server, it should also be able to support multiple connections, through queue implementations, for example.

For now, the server's process described in Section 5.4 takes 21 seconds on average on an Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz. Since the OpenFiDb server has to go to multiple steps to populate or create an FiDb, going by compiling with multiple optimisations, it takes a lot of computational load and time. This average time for a request will be a lot increased when the server will compile for different processor architectures with different compilers and different compiler options. It is not a big issue since it is not required to have a database updated instantly. OpenFunctionID aims to be used by a large community, but even with it, there should not be 1 000 requests of populate a day. Even with 1 000 requests a day, if the average time for processing a request is still 21 seconds, it takes less than 6 hours to process. Therefore, the OpenFiDb database should always be up to date daily even with heavy use.

8.1.2 Lumina and OpenFiDb

As described in Section 4.6, Hex-rays is developing a server known as Lumina, which is comparable to OpenFunctionID but for IDA Pro. It may be accurate in the sense that they both aim to solve the difficulty of recognising reused code, but the implementation of both techniques differs significantly. Moreover, the realisation of OpenFiDb has been done independently from Lumina since it is relatively new and not yet widespread in the community.

Firstly, OpenFunctionID does numerous compilations and stores the various compiled versions of the function, whereas Lumina simply stores the user's compiled version of the function. This distinction means that if the identical function is recompiled with a different compiler, OpenFunctionID will be able to identify it directly whereas Lumina would not.

Secondly, Lumina stores information about the function, whereas OpenFunctionID maintains information about the user who pushed the code. It impacts the traceability of the stored functions and indicates if the identification of a given function can be trusted.

8.2 OpenFunctionID's limitations

8.2.1 Obfuscation

Code obfuscation is an information management technique designed to obscure the meaning of source code while maintaining its operation. Obfuscation is used to protect the intellectual property of a software. However, since it is mainly used against reverse engineering, it is often used by malware authors. Obfuscation is seen as an *"anti-reverse engineering"* method. Several different obfuscation techniques are used on different levels[28]. At the source code, intermediate (such as *p-code*) or at the machine language level.

Obfuscation aims at creating difficult code for humans to understand, but powerful diassemblers and decompilers such as Ghidra or IDA Pro are able to correctly understand program flow even with some obfuscating techniques. Compilers optimisations also can understand the flow of a program and thus remove unused data, thus remove some obfuscation. However, like the reverse engineering, obfuscation techniques evolve, some are specially designed to not be ripped off by compilers[5].

Obfuscation is a challenge for reverse engineering, OpenFiDb is not excluded. Hashes stored in FiDbs are generated by decompiled code from Ghidra. Therefore, they could be different from the initial code compiled with obfuscation. The same function analysed even after having updated FiDbs could not be identified if it has obfuscated code in it.

8.2.2 Compile from Decompiled Functions

As explained in Section 5.2, functions transmitted to the OpenFunctionID service come from the decompiler of Ghidra. Due to the imprecision of decompilation, it is occasionally the case that the original function and the decompiled function differ slightly. It implies that the hashes generated by each function are distinct. Consequently, the original version of the function will not be identified. It is an issue due to the imperfect nature of decompilation. With some empiric tests, about 90% of the functions were identified, but these tests are not representative of all possible functions. Defining the proportion of functions that can be identified is tricky because OpenFunctionID has some restrictions as explained in the next subsection, and every test would be made up since the source code and the machine code are needed. However, the majority of functions are still identifiable since the compiled version of the original code and the compiled version of the decompiled code are often identical.

8.2.3 Independent Function

One objective of OpenFunctionID is to allow users to only push one function at a time and it has been established to limit the amount of data added to the database. The inability to compile functions that use objects or functions not defined within the function is a consequence of this particularity. Therefore, functions that cannot be independently compiled are not added to OpenFunctionID databases.

To address this issue, one workaround would be to populate OpenFunctionID FiDbs with the whole program, like the *Populate* dialog already implemented in FunctionID does. However this workaround assumes that the source code of all functions of the program has been well decompiled and all pieces put together in one file could be compiled. Which is rarely the case.

A first improvement for OpenFunctionID could be to allow the user to modify the C code before sending it to the OpenFiDb Server, allowing him to include some libraries or define used objects and functions needed for the compilation of the target function.

8.3 Future work

8.3.1 Server deployment

The OpenFiDb server is currently only available locally. Before releasing OpenFiDb for the community, the server implementation should be improved.

- A better request handler should be implemented, to support multiple requests, as explained in Section 8.1.1.
- Server security, multiple areas for improvement are described in Section 7.1.

Once hosted, it should be available non-stop, and maybe require some maintenance.

8.3.2 Git Portability

The OpenFunctionID plugin uses the `git` command line to download FiDbs from the repository and update them. It is required for all users to have a `git` version on their computer. Having an external requirement for a plugin for Ghidra is restrictive. To increase the portability of the plugin, a Java implementation of `git` could be used such as `JGiT`[1].

8.3.3 User identification

User identification in OpenFunctionID is an area of improvement. The objective is to verify that a user is who he or she claims to be. Using, for instance, Gitub's username (because OpenFunctionID utilises GitHub) or creating a whole user identification system would be possible, but would require more resources.

The identity verification of the user enables greater control over who pushes data to the server. It would permit banning users who are attempting to corrupt the database and removing functions submitted by these users. Additionally, it would allow users to only select the FiDbs of other users in which they have greater confidence.

8.3.4 Common libraries

An area of improvement concerns the addition of common used libraries to the OpenFunctionID repository. An idea is to try to automatically scrap every library available online and compile it with every *Language* that Ghidra contains. It would extend the work done by threatrack [41]. Moreover, these libraries should automatically be kept updated.

[1] Available on `https://www.eclipse.org/jgit/`

Chapter 9

Conclusion

This master thesis was about developing a plugin to improve Ghidra's functionalities. The FunctionID Analyser was described at large in Chapter 4 and objectives to improve it were presented in the first section of Chapter 5.

The plugin developed, OpenFunctionID, provides access for Ghidra's users to a large database of FunctionID Databases (FiDbs) containing information to identify function names based on the function's body instructions. This database can be updated and improved by Ghidra's users directly inside Ghidra. OpenFunctionID fills FiDbs thanks to users' contributions with multiple hashes for different compiler options.

The analysis of this plugin showed that it is a strong base but it would welcome some future work, such as improvement of the server's security, implementation of user identification and generating more FiDbs from common libraries.

Bibliography

[1] Vector 35. Binary ninja. https://binary.ninja/.

[2] Sergi Alvarez and the community. r2ghidra. https://github.com/radareorg/r2ghidra.

[3] Sergi Alvarez and the community. Radare2. https://rada.re/.

[4] Cryptic Apps. Hopper. https://www.hopperapp.com/.

[5] Sandrine Blazy and Stéphanie Riaud. Measuring the Robustness of Source Program Obfuscation - Studying the Impact of Compiler Optimizations on the Obfuscation of C Programs. In *Fourth ACM Conference on Data and Application Security and Privacy - SIGSAC ACM CODASPY 2014*, San Antonio, United States, March 2014.

[6] Broadcom. Symantec. https://securitycloud.symantec.com/. "[Online; accessed 04-MAY-2022]".

[7] Elliot J. Chikofsky and James H Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[8] Fred Cohen. Computer viruses: theory and experiments. *Computers & security*, 6(1):22–35, 1987.

[9] Wikimedia Commons. File:seal of the u.s. national security agency.svg — wikimedia commons, the free media repository. https://commons.wikimedia.org/w/index.php?title=File:Seal_of_the_U.S._National_Security_Agency.svg&oldid=606712472, 2021. [Online; accessed 22-April-2022].

[10] Wikimedia Commons. File:wikileaks logo.svg — wikimedia commons, the free media repository. https://commons.wikimedia.org/w/index.php?title=File:Wikileaks_logo.svg&oldid=635262340, 2022. [Online; accessed 22-April-2022].

[11] A.P. David. *Ghidra Software Reverse Engineering for Beginners: Analyze, identify, and avoid malicious code and potential threats in your networks and systems.* Packt Publishing, 2021.

[12] National Security Agency Research Directorate. Ghidra software reverse engineering framework. `https://github.com/NationalSecurityAgency/ghidra`, 2019.

[13] Chris Eagle and Kara Nance. *The Ghidra Book: The Definitive Guide.* no starch press, 2020.

[14] Emteere and ghidracadabra. Recon mtl 2019. `https://recon.cx/2019/montreal/`.

[15] B Ford. Transcript of q4 2005 ford motor company earnings conference call [congressional quarterly transcriptions], 2006.

[16] Free Software Foundation. ar(1) - linux man page. `https://linux.die.net/man/1/ar`. "[Online; accessed 14-MAY-2022]".

[17] Lee Garber. Melissa virus creates a new type of threat. *Computer*, 32(06):16–19, 1999.

[18] Elihu M Gerson and Susan Leigh Star. Analyzing due process in the workplace. *ACM Transactions on Information Systems (TOIS)*, 4(3):257–270, 1986.

[19] Shou-Ching Hsiao and Da-Yu Kao. The static analysis of wannacry ransomware. In *2018 20th International Conference on Advanced Communication Technology (ICACT)*, pages 153–158. IEEE, 2018.

[20] Emilio Iasiello. Cyber attack: A dull tool to shape foreign policy. In *2013 5th International Conference on Cyber Conflict (CYCON 2013)*, pages 1–18. IEEE, 2013.

[21] The Tor Project Inc. The onion router (tor). `https://www.torproject.org/`.

[22] Sid Katzen. The quintessential pic microcontroller. In *Computer Communications and Networks*, 2001.

[23] Brian Knighton and Chris Delikat. Black hat usa 2019. `https://www.blackhat.com/us-19/briefings/schedule/index.html#ghidra---journey-from-classified-nsa-tool-to-open-source-16309`.

[24] Allan Liska and Timothy Gallo. *Ransomware: Defending against digital extortion.* " O'Reilly Media, Inc.", 2016.

[25] Lily Hay Newman. The nsa makes ghidra, a powerful cybersecurity tool, open source. https://www.wired.com/story/nsa-ghidra-open-source-tool/, March 2019. [Online; accessed 28-April-2022].

[26] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. BinSign: Fingerprinting Binary Functions to Support Automated Analysis of Code Executables. In Sabrina De Capitani di Vimercati and Fabio Martinelli, editors, *32th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC)*, volume AICT-502 of *ICT Systems Security and Privacy Protection*, pages 341–355, Rome, Italy, May 2017. Springer International Publishing. Part 6: Applied Cryptography and Voting Schemes.

[27] National Security Agency (NSA). Ghidra sre official web page. https://ghidra-sre.org/. "[Online; accessed 04-May-2022]".

[28] Toshio Ogiso, Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 86(1):176–186, 2003.

[29] Igor Pavlov. 7-zip download page. https://www.7-zip.org/. "[Online; accessed 14-MAY-2022]".

[30] TF Peterson. A history of hacks and pranks at mit, 2011.

[31] Hex Rays. Decompiler. https://hex-rays.com/decompiler/.

[32] Hex Rays. Ida f.l.i.r.t. https://hex-rays.com/products/ida/tech/flirt/in_depth/.

[33] Hex Rays. Ida pro. https://hex-rays.com/ida-pro/.

[34] Hex Rays. Lumina server. https://hex-rays.com/products/ida/lumina/.

[35] Michael G Rekoff. On reverse engineering. *IEEE Transactions on systems, man, and cybernetics*, (2):244–252, 1985.

[36] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978.

[37] Synacktiv. Lumina local server. https://github.com/synacktiv/lumina_server.

[38] Intriguing Systems. Ghidra server. https://www.ghidra-server.org/.

[39] Threatrack. Ghidra fid generator. https://github.com/threatrack/ghidra-fid-generator. "[Online; accessed 18-MAY-2022]".

[40] Threatrack. Ghidra fid generation. https://blog.threatrack.de/2019/09/20/ghidra-fid-generator/, Sep 2019. "[Online; accessed 18-MAY-2022]".

[41] Threatrack. Ghidra function id dataset repository. https://github.com/threatrack/ghidra-fidb-repo, 2019. "[Online; accessed 18-MAY-2022]".

[42] Christoph Treude, Fernando Figueira Filho, Margaret-Anne Storey, and Martin Salois. An exploratory study of software reverse engineering in a security context. In *2011 18th Working Conference on Reverse Engineering*, pages 184–188. IEEE, 2011.

[43] Wargio. r2dec. https://github.com/wargio/r2dec-js.

[44] WikiLeaks. Firmware reverse engineering. https://wikileaks.org/ciav7p1/cms/page_15728683.html, March 2017. [Online; accessed 22-April-2022].

[45] WikiLeaks. Vault 7 : Cia hacking tools revealed. https://wikileaks.org/ciav7p1/index.html, March 2017. [Online; accessed 22-April-2022].

Appendix A

Manual

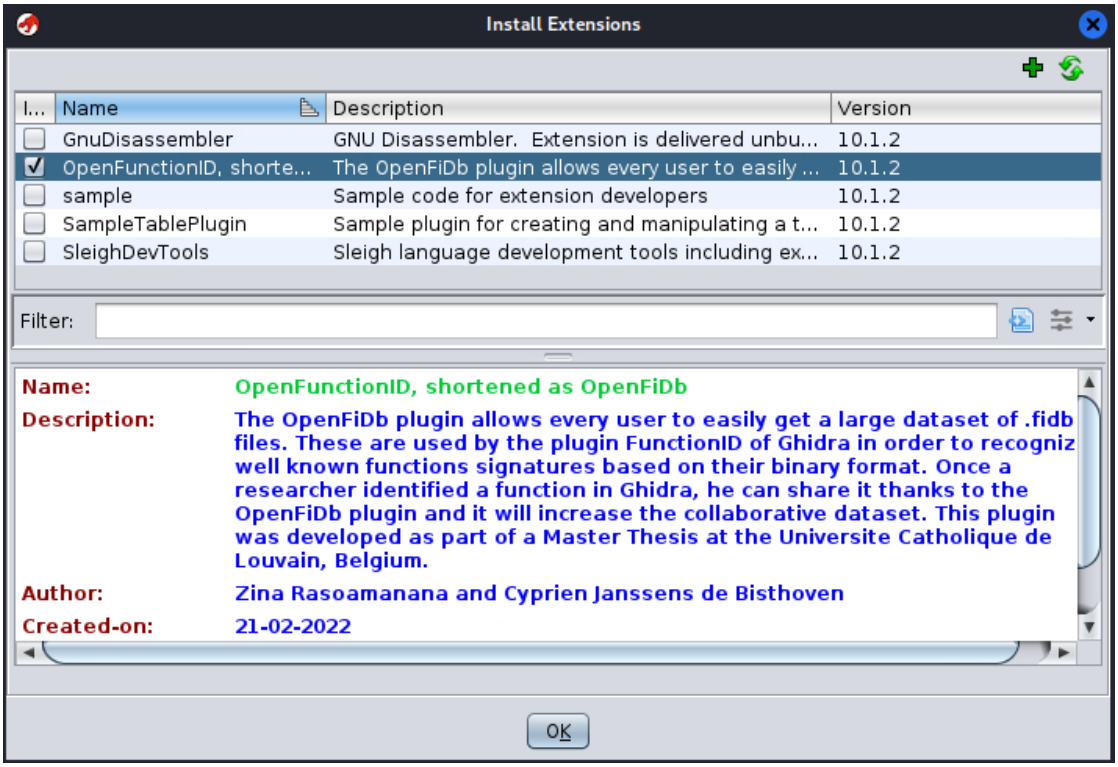


Figure A.1: Install OpenFunctionID Plugin

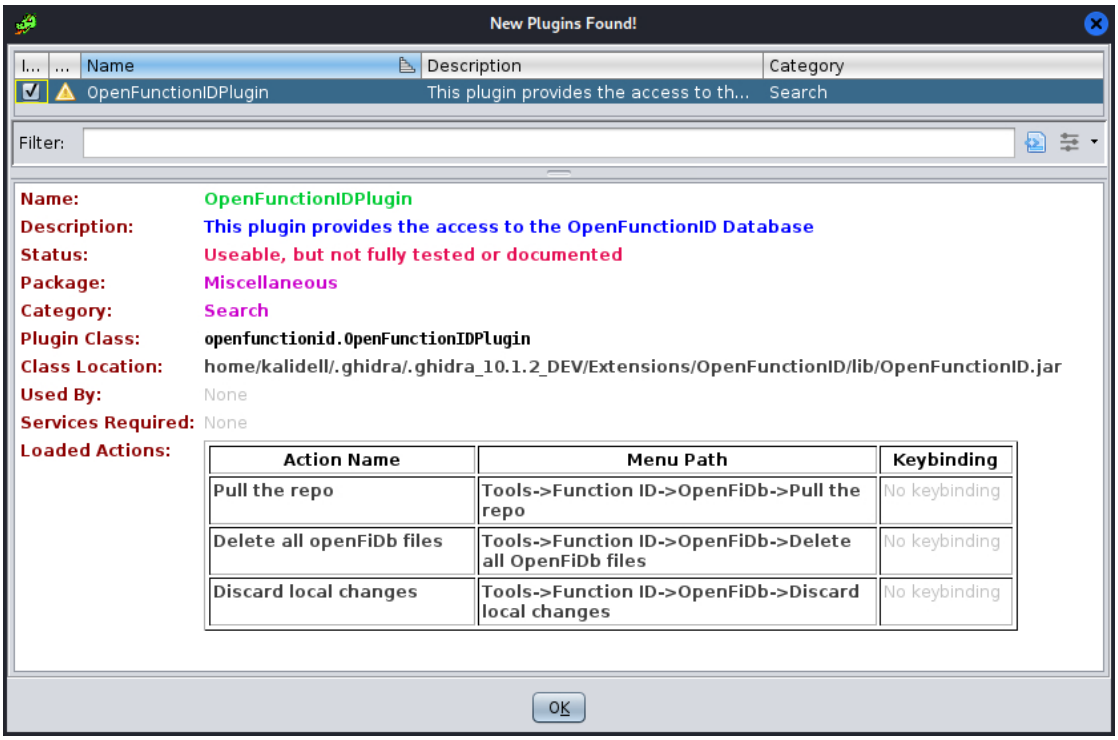


Figure A.2: Configure OpenFunctionID Plugin

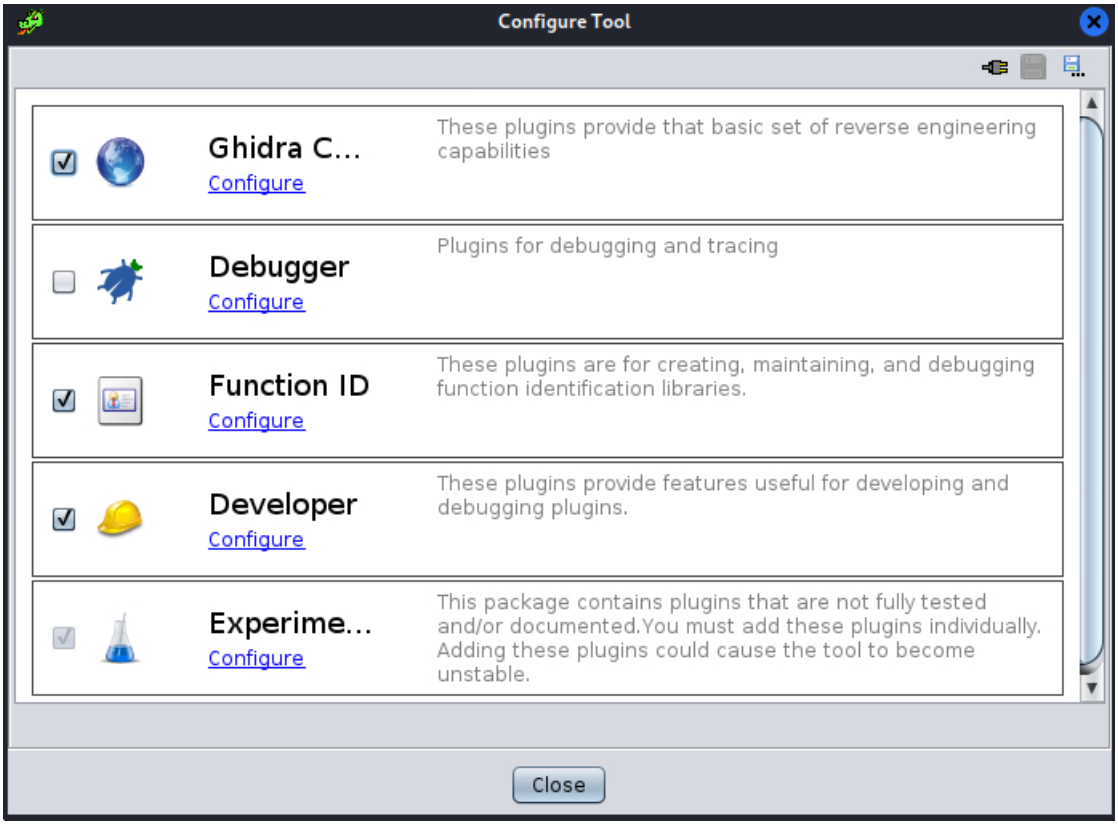


Figure A.3: Configure Tool

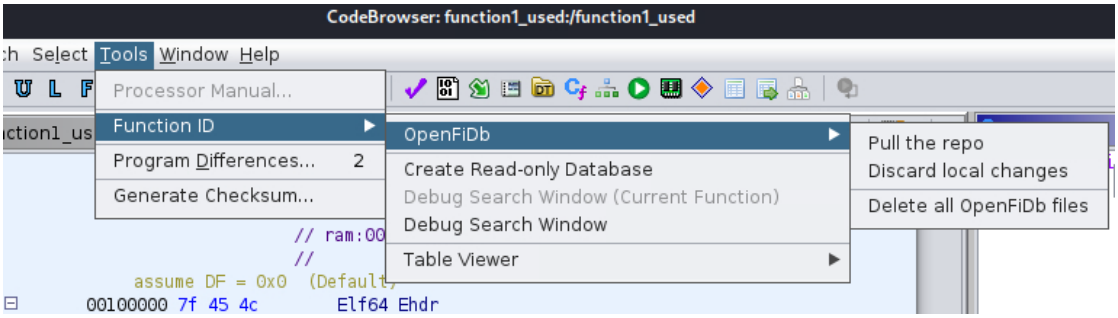


Figure A.4: OpenFunctionID Menu entries

Figure A.5: Console output after cloning OpenFiDb repository

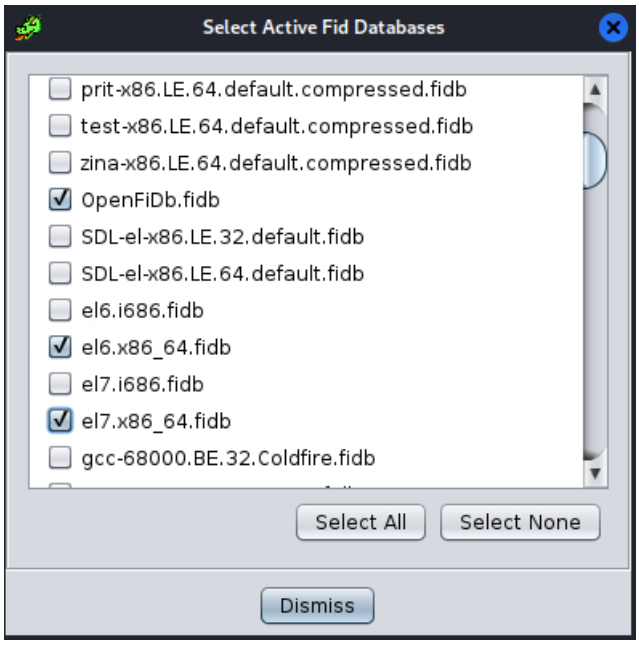


Figure A.6: Select FiDbs to use after pulling OpenFiDb repository

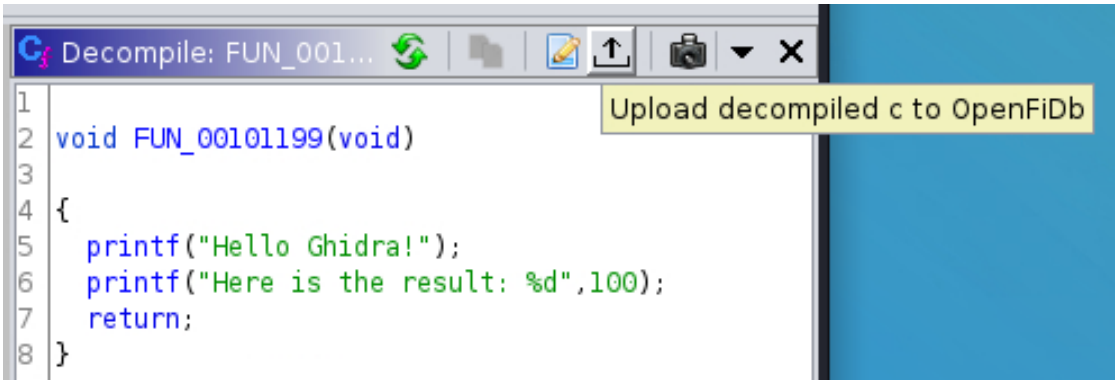


Figure A.7: Upload analysed function to OpenFiDb server

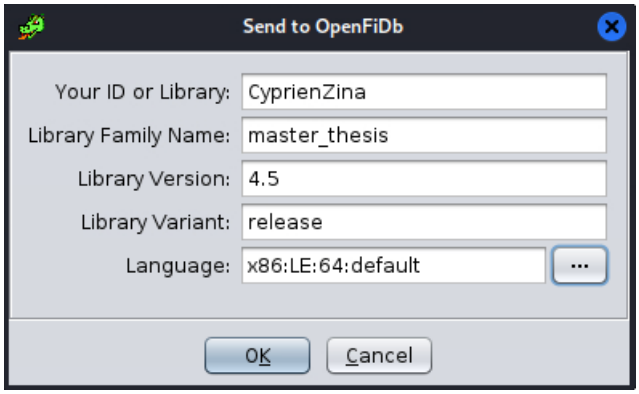


Figure A.8: Popup dialog when sending a function to OpenFiDb server

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**