FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Real-time Code Execution Analysis
Student:	Filip Touš
Supervisor:	Ing. Jiří Dostál, Ph.D.
Study program:	Informatics
Branch / specialization:	Information Security 2021
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

1. Describe a problem of pinpointing specific functions in disassembled code, emphasizing the limitations of conventional step debugging.
2. Design a method to locate functions by tracking code execution in real-time by progressively filtering the program code until the target function is isolated, based on events the reverse engineer can control or monitor.
3. Create a tool with a graphical user interface that applies the designed method specifically tailored for modern Windows operating systems.
4. Showcase practical applications of the tool and its effectiveness.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 24 January 2024 in Prague.

Bachelor's thesis

REAL-TIME CODE EXECUTION ANALYSIS

Filip Touš

Faculty of Information Technology
Department of Information Security
Supervisor: Ing. Jiří Dostál, Ph.D.
May 16, 2024

Citation of this thesis: Touš Filip. *Real-time Code Execution Analysis.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of figures

List of tables

List of code listings

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the ”Work”), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Prague on May 16, 2024

Abstract

This thesis addresses the challenge of locating functions in disassembled binary files without source code. It highlights the difficulties of this task in the absence of specialized tools and introduces a novel method to simplify the process. The method performs dynamic analysis of the software, monitoring code execution in real-time. Starting with a list of all function addresses in the binary, the reverse engineer using the tool progressively isolates the target function by triggering or avoiding specific events. A library was developed to implement this idea, adaptable by existing reverse engineering tools, with a custom executable user interface for standalone use. The thesis demonstrates the tool's practical application and effectiveness through real-world scenarios, illustrating its ability to simplify the reverse engineering process while also highlighting its limitations.

Keywords reverse engineering tool, dynamic software analysis, real-time analysis, code execution monitoring, function localization, x86-64 architecture, Windows operating system, software breakpoints, function hooking

Abstrakt

Tato práce se zabývá problémem lokalizace funkcí v disasemblovaných binárních souborech bez zdrojového kódu. Poukazuje na obtíže tohoto úkolu vzhledem k absenci specializovaných nástrojů a představuje novou metodu, která tento proces zjednodušuje. Metoda vykonává dynamickou analýzu softwaru a monitoruje provádění kódu v reálném čase. Začíná se seznamem všech adres funkcí v binárním souboru ze kterého reverzní inženýr pomocí tohoto nástroje postupně izoluje cílovou funkci tím, že vyvolává specifické události nebo se jim vyhýbá. Pro realizaci této myšlenky byla vyvinuta knihovna, kterou mohou adaptovat stávající nástroje pro reverzní inženýrství, s vlastním spustitelným uživatelským rozhraním pro samostatné používání. Práce demonstruje praktické použití a efektivitu nástroje na reálných scénářích, ilustruje jeho schopnost zjednodušit proces reverzního inženýrství a zároveň poukazuje na jeho nedostatky.

Klíčová slova nástroj pro reverzní inženýrství, dynamická analýza softwaru, analýza v reálném čase, monitorování spouštění kódu, lokalizace funkcí, architektura x86-64, operační systém Windows, softwarový breakpoint, hooking funkcí

List of abbreviations

API	Application programming interface
AVX	Advanced vector extenions
CPU	Central processing unit
DLL	Dynamic load library
EAT	Export address table
EPT	Extended page tables
GUI	Graphical user interface
IAT	Import address table
IDE	Integrated development environment
IDT	Interrupt descriptor table
IP	Instruction pointer
IPT	Intel processor trace
LBR	Last branch record
NPT	Nested page tables
OS	Operating system
TF	Trap flag
VMT	Virtual method table

Introduction

"Reverse engineering is a process where an engineered artifact (such as a car, a jet engine, or a software program) is deconstructed in a way that reveals its innermost details, such as its design and architecture. [. . .] In the software world reverse engineering boils down to taking an existing program for which source-code or proper documentation is not available and attempting to recover details regarding its design and implementation." [1]

This thesis deals specifically with what is commonly referred to as binary reverse engineering. Binary reverse engineering techniques aim at extracting valuable information from programs for which source code is unavailable. The goals of reverse engineering software can vary. As per [2, 3], some examples are:

- Achieving interoperability with proprietary systems and protocols.
- Identifying vulnerabilities, security flaws, and potential attack vectors.
- Extending or modifying behavior to enhance or change the user experience.
- Studying malware to enhance antivirus solutions or to retrospectively map the propagation of malware within a compromised system.
- Bypassing copy protection and digital rights management technologies.
- Understanding the features and designs of competing solutions.
- Detecting patent violations or intellectual property theft.

The terms software reverse engineering, reverse engineering, or reversing for short, are used interchangeably in this thesis. More specifically, the matter in this thesis is discussed in the context of reverse engineering of software created for and running on a modern Windows operating system (OS) on the x86 central processing unit (CPU) architecture.

The idea of software reverse engineering has existed for decades [4], so it is no surprise that many specialized tools and techniques have been developed, and are actively being used, for this very purpose – disassemblers, decompilers, and debuggers are essential tools in the reverse engineer's toolkit, each serving distinct yet complementary purposes.

Disassemblers, such as IDA Pro[1] and Ghidra[2], convert machine code into assembly language, providing a human-readable representation of the executable code. Decompilers, often bundled together with disassemblers, take this a step further by attempting to reconstruct high-level source code from the assembly, aiding in understanding program logic and behavior. Debuggers, such as WinDbg[3] and x64dbg[4], enable dynamic analysis by allowing users to control program execution, inspect memory, and analyze runtime behavior. Together, these tools empower reverse engineers to dissect, understand, and manipulate software systems.

Generally, reverse engineering can be split into two approaches – static and dynamic analysis. Static analysis involves examining the software without executing it, relying on disassemblers and decompilers to gain insights into the inner workings of the program and extracting useful information without the need for running the software. On the other hand, dynamic analysis involves executing the software in a controlled environment and observing its behavior in real-time, pausing the execution at will. Debuggers and other dynamic analysis tools play a crucial role in this approach, allowing reverse engineers to monitor and manipulate program execution.

Reverse engineering often involves the challenge of locating specific functions within a disassembled binary. Here, the term "function" usually refers to finding the block of instructions that get executed directly after a `CALL` instruction transfers the control flow to its starting address and ends with a `RET` instruction. Due to compiler optimizations this does not always correspond to a specific function defined in the source code and the realization of the program flow in the binary may be further altered, hence this thesis sometimes uses the terms functions and code (here referencing assembly or binary code) interchangeably for simplicity. Locating specific code is then understood as the process of pinpointing a small part of the binary as being responsible for a specific side effect to further analyze or modify this specific behavior.

This task becomes particularly daunting in large and complex software systems where the code base may span tens of thousands of lines. Suppose that a reverse engineer is tasked with analyzing such a large binary executable to understand how it handles user authentication. The engineer knows that upon entering incorrect credentials, the program displays an error message indicating "Invalid username or password." Despite the seemingly straightforward nature of the functionality, pinpointing the specific function responsible for this behavior within the vast expanse of the disassembled code can prove to be a daunting task. Without clear markers or known entry points, the engineer must navigate through countless instructions, functions, and control flow paths to locate the relevant code segment. This process can be time-consuming and labor-intensive.

However, in this particular case, the reverse engineer is fortunate to have a clear indicator in the form of the error message referenced within the binary. By tracing the execution flow backward from the point where this string is referenced, the engineer can identify the function responsible for handling authentication logic relatively easily. Unfortunately, not all reverse engineering tasks come with such obvious markers. In many instances, the absence of identifiable strings, which often prove to be the easiest way

[1]`https://hex-rays.com/ida-pro/`
[2]`https://ghidra-sre.org/`
[3]`https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/`
[4]`https://x64dbg.com/`

to locate code, complicates the process significantly. Some other artefacts that reverse engineers often use are leftover debug data, or application programming interface (API) calls. But when no such clues are present, reverse engineers face a formidable challenge in navigating the disassembled code base.

The primary objective of this thesis is to introduce and demonstrate a method aimed at addressing the challenge outlined in the previous paragraphs. While the thesis endeavors to develop a complete working tool to showcase the method, special attention is given to ensuring that its code is adaptable by existing reverse engineering tools and frameworks. It's important to note that the developed tool itself is not a fully-fledged reverse engineering tool but rather a specific feature designed to aid in the reverse engineering process.

The thesis focuses on the Windows 64-bit operating system running on x86-64[5] architecture, specifically targeting userspace program analysis to avoid the need for reverse engineers to load custom kernel drivers or hypervisors. Additionally, although a graphical user interface (GUI) is developed to help showcase the method's functionality, it's essential to emphasize that the GUI and the underlying logic are decoupled. This deliberate separation ensures that the method remains independent of any specific GUI implementation, thereby enhancing its flexibility and adaptability for integration into diverse reverse engineering environments.

In summary, the goals of this thesis are to introduce and demonstrate a method for addressing the challenge of locating specific code segments within disassembled binaries, develop a working tool to showcase the method's functionality, and ensure the tool is adaptable by existing reverse engineering tools and frameworks. The final point is achieved in two ways: by ensuring maximum code portability and by crafting the tool (sans the GUI) as a dynamic load library (DLL), rather than an executable file.

While reverse engineering is often associated with analysis of malware or other forms of software that tries to protect itself from reverse engineering using anti-debug and anti-analysis techniques, bypassing those protections is not in the scope of this thesis. Although the aim is only on developing and demonstrating the method, the detectability of the developed tool and the traces it leaves behind are still discussed.

The thesis first explains the proposed method in detail. Then, current available tools and techniques (or the lack thereof) capable of achieving the desired goal are presented. Chapter 2 deals with different methods of monitoring code execution, which are the key building blocks that this project is built on. The main part of the thesis describes in detail the design and practical implementation of the tool, including key snippets of source code. A full chapter is reserved for both the "backend", implementing the logic and program code in realization of the proposed method as a DLL (chapter 3), and the "frontend" – the graphical user interface used to control it (chapter 4). Next, special focus is granted solely on showcasing the tool in different reverse engineering scenarios, highlighting its strengths, and providing examples of its usage. It also presents shortcomings associated with the tool developed or the method itself. Afterwards the thesis discusses how the analyzed program might be able to detect the usage of the tool and how to help prevent that from happening, and lists possible future work to improve on the ideas presented in this thesis.

[5]Also called x86_64, x64, or amd64

Chapter 1

The proposed method

The core idea of this thesis revolves around the concept of real-time monitoring of executed functions during program execution, aimed at streamlining the process of locating specific functions in a disassembled binary.

Imagine a scenario where a reverse engineer could dynamically observe program execution, perform the desired action that triggers the relevant functions, and then signal that to a system recording the executed functions without pausing the execution flow. By progressively filtering out functions based on their execution status – retaining only those functions that executed when the desired action occurred and excluding those that did not (or vice versa) – the engineer could systematically narrow down the search space until only a few or even a single function remains.

This approach should accelerate the process of identifying target functions and provide a more intuitive and efficient methodology for such reverse engineering tasks. The goal of this thesis is to develop and implement a tool that realizes this idea.

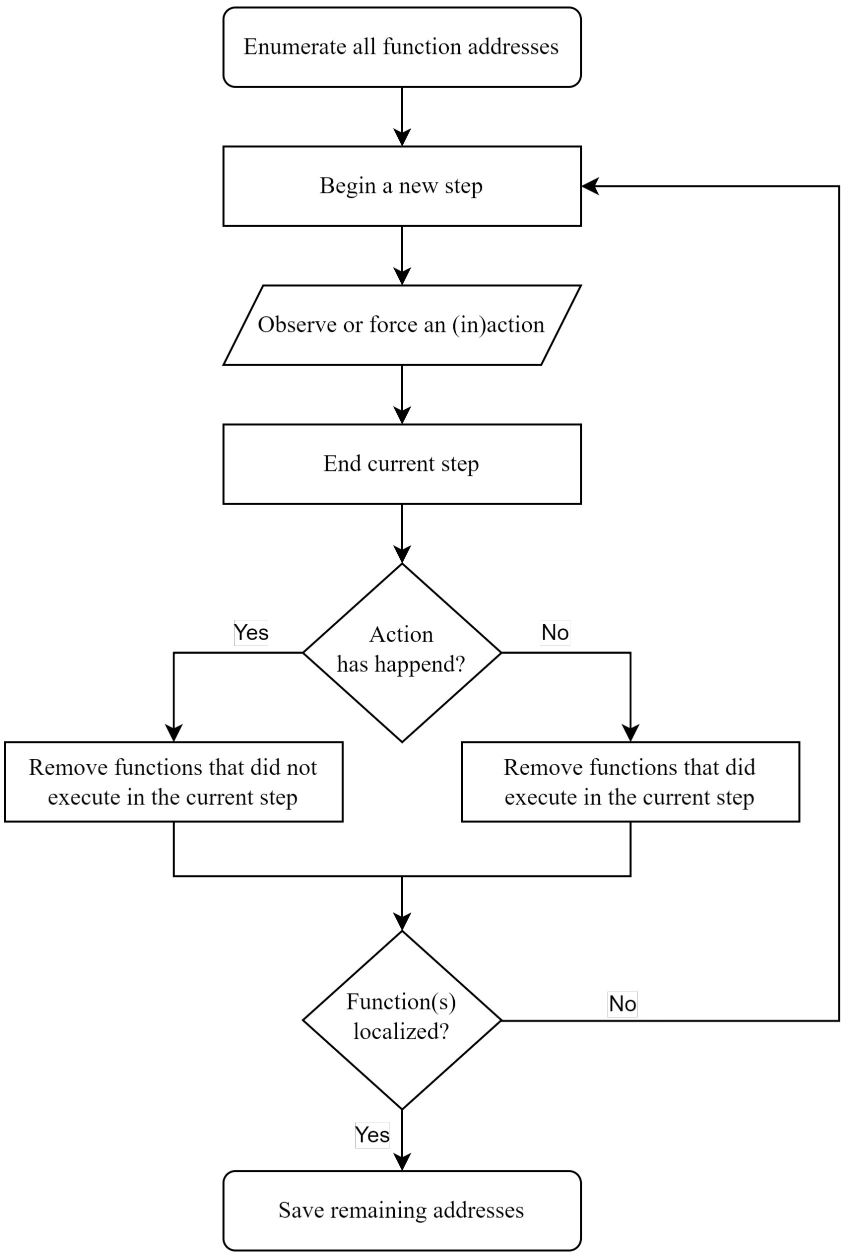
1.1 Step-based approach

In essence, the reverse engineer initiates the process with a comprehensive list of function addresses extracted from the analyzed binary. Subsequently, they commence monitoring the code execution, performing an unknown number of repeating "steps" until they achieve a satisfactory reduction in the list of remaining functions, ideally culminating in only a few addresses. This idea is illustrated in Figure 1.1.

At each step, the engineer discerns whether the desired function has been executed or not. This is made through various means, including visual or auditory analysis of the program's behavior, or deliberately inducing or refraining from certain actions that might trigger the function, provided such actions are feasible. The reverse engineer might control the tool and interact with the analysis process by various means. One option is through the use of keyboard shortcuts to signal the start and end of each step. This provides a quick and efficient way to guide the analysis in real-time.

In order to achieve truly real-time analysis, it is imperative that the process maintains minimal overhead, allowing the program to continue functioning without disruption. This requirement is particularly crucial in scenarios where pausing the execution flow or introducing significant overhead would render the analysis impractical or even impossible.

Consider applications like video games or real-time audio software, where any interruption to the execution flow can lead to degraded performance, glitches, crashes, or inability to perform the action the engineer searches for. In such contexts, the ability to seamlessly monitor executed functions in real-time without impeding the program's normal operation is paramount. Therefore, the tool developed in this thesis is designed with efficiency and low overhead as primary considerations.


Enumerate all function addresses
Begin a new step
Observe or force an (in)action
End current step
Action has happend?
Yes
No
Remove functions that did not execute in the current step
Remove functions that did execute in the current step
Function(s) localized?
No
Yes
Save remaining addresses

■ **Figure 1.1** Flowchart of the presented method

1.2 Current solutions

When static analysis methods prove insufficient for locating a function, reverse engineers turn to dynamic analysis as a viable alternative. In dynamic analysis, engineers exploit the runtime behavior of a program, sometimes relying on memory access patterns to pinpoint specific functions. For instance, identifying a memory address accessed by a function – whether for reading, writing, or both – can serve as a crucial anchor point. By halting program execution upon memory access, engineers gain insight into the precise instructions within a function responsible for accessing the memory. Subsequently, analyzing the call stack sheds light on the sequence of function calls leading up to the memory access point, potentially revealing the sought-after function (if it was not the one directly responsible for accessing the memory address).

Breaking execution on memory access can be realized through hardware breakpoints or by altering memory protection to trigger an exception upon attempted read or write. However, the efficacy of this approach hinges on the engineer's ability to identify the specific memory address accessed by the target function — an endeavor that can present considerable challenges. An ideal scenario involves locating a numeric variable whose value can be manipulated or at least observed. While such opportunities may arise, they are often rare and only applicable to a few of the executable's functions.

This approach might be viable in very specific scenarios, where the engineer can either leverage the program's intended behavior to manipulate and observe a specific value, or observe externally inputted data that a function handles, for instance:

- Identifying video game functions by tracing memory accesses related to variables representing the player's health or coordinates, which can be intentionally modified by being hit by an opponent or moving in the game world.

- Pinpointing functions responsible for parsing and handling network packets where the engineer has access to the packets' data before the program does.

When conventional approaches fall short in identifying a specific function, this thesis' method emerges as a viable solution. Existing techniques capable of achieving its goal are limited. The following subsections describe traditional methods that can be used to come close to the desired functionality presented in this thesis, ranging from traditional to specialized tools.

1.2.1 Step-debugging

Traditional step-debugging methods involve manually stepping through a program's execution, typically using a debugger. To simplify the search, the debugger can be configured to only break on function calls and returns. This way, eventually the desired function might be found given enough time, but this approach is not only time-consuming and impractical for this purpose but unfeasible for real-time systems. The process of stepping through each function not only slows down the analysis but also disrupts the program's normal operation, making it unsuitable for scenarios where uninterrupted execution is crucial [5].

1.2.2 Tracing

Tracing offers an alternative to manual step-debugging by automatically capturing program execution and potentially generating a graph of function calls. It specifically addresses one of the challenges highlighted in this thesis, as stated in the documentation of GDB[1], a popular debugger: "In some applications, it is not feasible for the debugger to interrupt the program's execution long enough for the developer to learn anything helpful about its behavior. If the program's correctness depends on its real-time behavior, delays introduced by a debugger might cause the program to change its behavior drastically, or perhaps fail, even when the code itself is correct. It is useful to be able to observe the program's behavior without interrupting it." [5]

However, tracing still requires specifying a starting point for the trace, adding a level of complexity and manual intervention that contrasts with the proposed method, where monitoring begins immediately without the need for predetermined starting points [6]. Additionally, given that the aim of tracing is to facilitate comprehensive debugging, albeit retroactively, the process of generating the trace can still notably affect the performance and speed of the program. This impact is dependent upon both the configuration of the tool and the specifics of its implementation. [7]

Conventional methods involving breakpoints and the CPU's `TRAP` flag prove inadequate for resource-intensive applications, as detailed in section 2.1. While advanced debuggers including GDB can leverage hardware assistance from certain CPU models for minimal overhead [8], this strategy does not align with the objectives outlined in this thesis, as explained in subsection 2.3.3.

Alternatively, GDB utilizes an "In-Process Agent" for a rapid debugging model, closely resembling the approach developed in this thesis (refer to section 2.2), albeit with slightly higher overhead [7, 9]. While tracing indeed offers a more streamlined approach compared to step-debugging with satisfactory performance, pinpointing a specific function among thousands can still prove immensely cumbersome. To the best of the author's knowledge and based on reviewed literature, no existing tools are explicitly designed for such operations. This contrasts with the tool developed in this thesis, which aims to significantly streamline this particular reverse engineering process.

1.2.3 Record and replay debugging

Similar to tracing, record and replay debugging provides a way to observe program behavior without interrupting its execution. However, unlike tracing, which typically generates a trace of function calls, record and replay debugging captures a more detailed snapshot of the program's state at each step of execution. This allows developers to deterministically replay the recorded execution and precisely recreate the conditions under which an issue occurred and investigate it thoroughly at an individual instruction level as if they were step-debugging a live program. [8, 10]

Record and replay debugging naturally incurs a higher performance cost compared to traditional tracing methods. However, specialized tools like rr[2] demonstrate the potential effectiveness of this approach in the context of locating functions. For instance,

[1]`https://www.sourceware.org/gdb/`
[2]`https://rr-project.org/`

a reverse engineer could manually log the exact time when a specific action occurred and then analyze the trace or recorded execution at that precise moment (assuming timestamps are saved) to narrow down the search. Conversely, they could filter out all functions executed within a certain time frame where the desired action did not occur. Nevertheless, this manual process would be laborious, and either only suitable for non-resource-intensive applications, or it would again require hardware assistance, in which case rr still reports a minimal overhead of 1.2x execution time [11].

Chapter 2
Monitoring code execution

Monitoring code execution of a different process from user space on Windows involves leveraging Windows API functions to intercept and manipulate the control flow of the target process. By gaining control over the target process's execution flow, it is possible to track the code execution in real time. The following methods are the essential building blocks of this thesis.

2.1 Software breakpoints

One of the simplest and most widely used techniques for altering code execution in user space on Windows is the use of software breakpoints. Software breakpoints involve temporarily modifying the target process's code to interrupt its execution at specific points of interest. This is achieved by replacing the instructions at the desired breakpoint location with a special instruction, such as `INT3` (interrupt 3) on x86 CPUs.

When the CPU encounters the instruction, it generates a breakpoint exception. This exception interrupts the normal flow of execution and triggers the CPU's interrupt handling mechanism. The CPU transfers control to the operating system's interrupt handler, which identifies the breakpoint exception and dispatches it to the appropriate routine within the kernel. The OS notifies any attached debuggers about the breakpoint event. The debugger gains control over the suspended process and can examine its memory, registers, and other state information. The debugger may also execute custom code or perform additional analysis as needed.

2.1.1 Handling a breakpoint exception

Depending on the debugger's configuration and the user's actions, the debugger may resume execution of the target process, either allowing it to continue normally or stepping through instructions one at a time for further analysis, usually through the use of a `TRAP` flag (TF) in the `EFLAGS` register. Since the insertion of a breakpoint requires overwriting executable memory, it is necessary to restore the original instruction that was replaced with the breakpoint opcode. This involves copying the original instruction from a backup location back into the appropriate location in the target process's address space. Additionally, to ensure the target process continues execution seamlessly after the breakpoint is hit, the debugger typically decrements the instruction pointer (IP) by the appropriate amount to reposition it at the original instruction's address before resuming execution. This meticulous restoration process ensures that the target process remains in a consistent state and can continue executing normally after the debugging operation is complete.

On the other hand, setting the `TRAP` flag for single-stepping purposes does not require overwriting executable memory in the target process. Instead, the CPU automatically generates traps after each instruction execution, providing a non-intrusive way to control the program's execution flow. When a trap occurs, the CPU again transfers control to the operating system's interrupt handler in a similar manner to the `INT3` instruction. This ultimately allows debuggers to place "persistent" breakpoints by reintroducing the software breakpoint instruction immediately after the original code gets executed.

2.1.2 Implementing a persistent breakpoint

The steps outlined below depict how a debugger could establish a persistent breakpoint at a specific address, assuming it has already enlisted itself as a debugger of the target process using an OS function such as `AttachDebugger`:

1. Retrieve the original memory contents at the designated address and store them in a backup location.
2. Replace the memory content with an `INT3` instruction opcode.
3. Flush the instruction cache at the location within the target process to prevent the CPU from executing outdated code [12].

With the breakpoint now in place, the OS will signal the debugger when it is executed. The debugger awaits this notification in an infinite loop, managing the breakpoint debug event within its dedicated handler, after ensuring that the exception was not triggered by another debugger or other outside factors:

1. Obtain the context of the thread responsible for executing the `INT3` instruction within the target process.
2. Adjust the instruction pointer of the thread context to reference the beginning of the original instruction.
3. Set the `TRAP` flag in the `EFLAGS` register of the thread context.
4. Restore the original memory contents at the breakpoint address.
5. Flush the instruction cache.
6. Finish the exception handling process and inform the OS that the thread can resume execution.

Thanks to the `TRAP` flag, the debugger promptly receives a single-step exception after the restored instruction is executed, allowing it to reintroduce the breakpoint by again replacing the memory with an `INT3` instruction opcode and flushing the instruction cache. The `TRAP` flag is cleared automatically. This approach ensures that the breakpoint is triggered every time the debugged program reaches its address.

In the context of this thesis, referencing the diagram of the proposed method (Figure 1.1), software breakpoints can be used as a tool for determining whether a function was executed during each step of the analysis process. In this approach, breakpoint persistency is not a necessity. Instead, the tool only has to ensure that breakpoints are set on all remaining function addresses at the start of each step. Conversely, if a function is filtered out during the analysis, the tool should remove the breakpoint associated with that function, even if it was not executed.

2.1.3 Overhead of software breakpoints

While software breakpoints offer a straightforward and reliable way to monitor function execution, they entail significant overhead. The process of handling a breakpoint exception on Windows involves many steps, as outlined in [13]:

1. The CPU hardware detects the exception and directs control to the appropriate kernel trap handler based on the exception's index in the interrupt descriptor table (IDT).
2. This handler generates a trap frame and an exception record containing information about the exception.
3. The exception record is sent to an exception dispatcher.
4. The exception dispatcher initially verifies whether the process responsible for the exception is associated with a debugger process. If so, it dispatches a debugger object message to the debugger process, signaling the occurrence of a breakpoint exception.
5. The debugger thread returns from a `WaitForDebugEvent` call and may proceed to handle the exception.

Even the final step alone incurs a notable CPU overhead. According to a disassembly of the function and the subsequent calls made within it, the execution sequence involves returning from the kernel mode `nt!NtWaitForDebugEvent`, then the native API function `DbgUiWaitStateChange` (implying a mode switch to user mode), and finally reaching the documented top-level Win32 API function `WaitForDebugEvent`.

Upon completion of these steps, the debugger can commence handling the exception. After finishing and calling `ContinueDebugEvent`, control returns to the kernel. The whole process concludes as the trap handler restores the thread context using the trap frame created in step 2, allowing the CPU to resume execution of the thread.

While the breakpoint only needs to be triggered once during each step of the proposed method, it imposes a significant performance penalty that amplifies with the number of breakpoints deployed in the target process. Given the thesis's focus on crafting a tool with "efficiency and low overhead as primary considerations", an alternative approach is needed for analyzing resource-intensive software.

2.2 Inline function hooking

Hooking a function alters the normal flow of execution within a program, typically to collect information or modify the behavior of the specific function. In this thesis, focus lies on the former objective. The concept revolves around intercepting the invocation of a function, redirecting the execution flow to log its occurrence, and then seamlessly allowing the original function to proceed without altering its outcome. [14]

Among the various methods of hooking, inline hooking stands out as one of the most straightforward approaches. Fundamentally, it works by overwriting code in the running program. Consequently, when the function is called, the attacker's code is executed instead of the original function. Typically, software performing inline hooking preserves the initial few bytes of the target function, which are subsequently replaced with an unconditional jump instruction, redirecting execution to the hook. The hook can then invoke the original function using the preserved bytes of the target function that were overwritten, potentially modifying the data returned by the original function. [15]

2.2.1 Preserving the original functionality

Preserving the original function with a persistent hook poses a greater challenge compared to software breakpoints and the `TRAP` flag. A common solution involves the use of a so called "trampoline" function. Once the hook completes its task, the trampoline is responsible for executing the instructions that were overwritten at the beginning of the original function and seamlessly transitioning back to continue execution from the subsequent instruction (typically the first original instruction not overwritten by the hook, but, as explained in the following paragraphs, this may vary) [15].

This approach presents two significant challenges. Firstly, the overwritten instructions within the trampoline are executed at a different address than inside the original function. While typically the beginning of the function – where the prologue is located – is overwritten, correct relocation of instructions in the trampoline might be required. For instance, if the prologue is not position independent or is shorter than the jump to the hook, careful adjustment of IP-relative instructions is necessary.

Moreover, due to the variable length of instructions, the hooking mechanism must copy the appropriate number of bytes to ensure correct interpretation by the CPU. This might be more than the amount of bytes overwritten by the jump. Ensuring proper alignment to an instruction boundary is also crucial for the transition from the trampoline to the subsequent instruction of the original function. Failure to align this transition could lead to the CPU interpreting a partial instruction as a completely different one, resulting in unexpected behavior or even program crashes.

As a result, inline hooking libraries such as PolyHook 2.0[1] necessitate bundling with a disassembler, significantly inflating the code size. Alternatively, libraries like minhook[2] or subhook[3] develop their own disassembly functionality, albeit often at the expense of not supporting all instructions and, consequently, being unable to hook certain functions.

[1] `https://github.com/stevemk14ebr/PolyHook_2_0`
[2] `https://github.com/TsudaKageyu/minhook`
[3] `https://github.com/Zeex/subhook`

Furthermore, runtime disassembly adds to the overall overhead of the hooking process (but not the hook itself).

However, since a persistent hook is not required, as the function call only needs to be detected once in every “step” of the proposed method, the implementation in the context of this thesis can be simplified, similar to the approach described in the previous chapter about software breakpoints. To preserve the original functionality, the hook can reinstate the original function’s code, akin to how a debugger replaces an `INT3` instruction with the original byte. This approach is unconventional, but it completely bypasses the need for any relocations or disassembly. The hook can simply restore the overwritten bytes and jump to the original function’s starting address.

2.2.2 Implementing the hook

The following steps outline how a non-persistent inline hook on the Windows OS might be implemented:

1. Allocate executable memory in the target process where the hook code will be placed.
2. Prepare code that will facilitate the jump from the target function to the hook code.
3. Save the original bytes of the target function that will be overwritten with the jump.
4. Develop hook code responsible for logging the function call, restoring the overwritten bytes, and jumping back to the start of the target function.
5. Write the hook code to the memory allocated in step 1.
6. Modify memory protection to allow overwriting of the target function.
7. Overwrite the beginning of the target function with the jump code and flush the instruction cache.

Once these steps are completed, the hook code is executed when the program first invokes the target function. On all subsequent calls, the jump code is no longer present in the target’s memory. The tool developed in this thesis must ensure that at the beginning of each “step” of the proposed method, the beginning of the function is again overwritten with the jump code, i.e., step 7 from the above enumeration is repeated. It is worth noting that step 6 is never reversed in this scenario. The logging of the function call can be as straightforward as setting a byte in the hook code (placed after the jump back to the original function to ensure it is never executed) to a value of 1. This byte can then be read externally by the tool to determine whether the function was or was not executed. Upon reinstating the hook (by repeating step 7), the byte must be reset to zero.

In comparison to software breakpoints, where the `INT3` instruction is a single byte, a potential risk arises in multithreaded applications with inline hooking. Depending on the implementation, the overwriting of the target function’s instructions is not guaranteed to be atomic. If a thread calls the function while another thread is in the midst of a non-atomic restoration of the original function’s instructions (from within the hook code), unforeseen consequences might occur, including a program crash due to the CPU attempting to execute invalid instructions. Although the likelihood of this happening is very low since the memory overwrite should involve only a few instructions at most, it is not entirely unrealistic depending on the design of the target application. The same issue might arise during the tool's execution of step 7 (externally overwriting the beginning of the function). Overall, the risk is application, OS, and CPU specific and challenging to predict. If it were to cause any trouble, modern x86 processors provide the means to ensure the function overwriting is atomic, so that no threads need to be suspended during the writes (which would nullify the performance gains of the hooking method compared to breakpoints). This is further discussed in chapter 3.

2.2.3 Overhead of inline hooks

Inline hooks introduce minimal performance overhead. Unlike software breakpoints, there is no need for context switching, mode switching from kernel to user space, or dispatching an exception to a debugger. The CPU seamlessly executes the instructions, and no interrupts occur. The entire jump, hook, and jump back sequence can be implemented in as few as 5 instructions. While it may be argued that the non-persistent approach could introduce unnecessary overhead due to additional memory writes, in practice, this again only results in a few extra instructions. In reality, immediately removing the jump could improve performance if the function is called often, when jumping to the hook and trampolining back every time would quickly add up to a more significant performance penalty.

A potential source of overhead is the risk of cache misses due to the jumps. Similarly, the processor's snooping logic may invalidate the cache after the hook modifies the function's code, leading to further cache misses. However, this is highly CPU-specific and is typically considered a routine aspect of execution on modern processors, and therefore not explicitly addressed in this thesis. It's noteworthy that the cache coherency mechanism in modern x86 CPUs facilitates the functioning of self-modifying code, such as the hook restoring the original instructions of the target function, without requiring explicit cache invalidation. In some scenarios, the installation of the hook itself may introduce more overhead than the hook's actual execution, especially if the function code is externally overwritten using the OS API, as implied in this and the preceding chapters. In such cases, cache flushing becomes necessary as the tool operates in a different address space than the target process. Ensuring all memory manipulations are performed from within the target process, perhaps by creating a remote thread, might be beneficial. However, this approach would incur additional overhead due to inter-process communication, and in any case, cache invalidation remains inevitable yet unpredictable. [16]

Overall, inline hooking is expected to have a negligible impact on performance compared to software breakpoints, albeit with the limitation of being unable to hook functions that are too short to accommodate the necessary jump instruction. Further details on this topic are presented in chapter 3, while chapter 5 provides insight on the real performance impact of both breakpoints and hooks.

2.3 Unsuitable methods

Apart from software breakpoints and inline hooking, other techniques also exist. However, for various reasons, these alternative methods are not deemed suitable for integration into the tool developed in this thesis. The following subsections explain the limitations and drawbacks of these alternative approaches, highlighting the reasoning behind the selection of software breakpoints and inline function hooking as the only mechanisms for tracking function execution in this thesis.

2.3.1 Hardware breakpoints

Hardware breakpoints represent an alternative approach to monitoring code execution compared to their software counterparts. While they offer certain advantages, their suitability for this thesis is minimal.

One notable advantage of hardware breakpoints is their non-invasive nature. Unlike software breakpoints, which involve patching memory with specific instructions to trigger an interrupt, hardware breakpoints utilize special CPU hardware registers, known as debug registers. This mechanism allows for monitoring code execution without directly modifying the target program's memory space.

Despite their differences in implementation, the execution flow from the CPU to the kernel dispatcher and an attached debugger remains largely consistent between hardware and software breakpoints. Therefore, the overall debugging process follows a similar trajectory and performance overhead as with software breakpoints.

One significant feature of hardware breakpoints is their ability to break on memory read or write operations, in addition to code execution. This flexibility allows developers to halt execution when specific memory addresses are accessed, as mentioned in the chapter 1.2. However, for the purposes of the tool developed in this thesis, which focuses on tracking function execution, this additional functionality is not beneficial.

Despite their advantages, hardware breakpoints have limitations that render them almost useless for function localization. One critical constraint is the limited number of available hardware registers. On x86 platforms, typically only four debug registers are available, each set up to trigger a breakpoint at a specific address. Consequently, this means that only four hardware breakpoints can be active simultaneously. This limitation becomes problematic when dealing with a large number of function addresses, as it significantly slows down the analysis process by restricting the monitoring capability to only four functions at once, in contrast to potentially monitoring thousands with software breakpoints. Moreover, since debug registers are implemented per process thread[4], setting up hardware breakpoints for function calls necessitates configuring debug registers for each thread within the target process. [17]

[4]This is OS specific, based on the implementation of multitasking and context switching. From the CPU's perspective, registers, including debug registers, are assigned per-core, or per-thread (CPU thread) in the case of hyperthreading.

2.3.2 Alternative hooking techniques and exceptions

While inline hooking offers a versatile approach for intercepting code execution, other function hooking techniques exist. However, these techniques may not be suitable for the purposes of this thesis due to their inherent limitations. The following is a list of some of the most common ones, adapted from [18]:

- Import address table (IAT) / Export address table (EAT) Hooking: This method targets functions imported from DLLs or exported by a module. It modifies the IAT or EAT to redirect calls to a hook function. However, this approach is limited to functions that are explicitly imported or exported, severely hindering its versatility.
- Virtual method table (VMT) Hooking: This technique leverages virtual functions commonly found in object-oriented programming, particularly C++. It modifies the VMT pointer (or the pointers inside of it) to redirect calls to hook functions. While powerful in applicable contexts, VMT hooking is not suitable for this thesis as it is programming language and function specific.
- Pointer swapping: This method involves replacing pointers that reference the target function with pointers to a hook function. It requires the target function to be called via a function pointer, limiting its applicability in scenarios where the target function is referenced directly, which constitutes the majority.
- Overwriting call instructions: Instead of overwriting the beginning of a function with a jump, it is possible to overwrite the call instructions to the function itself. This eliminates the need to preserve the original functionality, which is addressed in this thesis through non-persistent hooking, and alternatively trampolining. However, it comes at the significant cost of rewriting all calls to the function, which can be numerous and dynamically generated at runtime.

Similarly, software breakpoints are not the sole type of exception that can be triggered and subsequently handled. While it is feasible to induce a different exception, such as a division by zero, by modifying the code, there is no rationale for pursuing this approach in the context of this thesis. It adds unnecessary complexity compared to the relatively straightforward mechanism of software breakpoints (`INT3`). This method is sometimes termed "forced exception hooking". Another commonly used exception involves tagging a memory page with the `PAGE_GUARD` or `NO_ACCESS` flag to provoke a `STATUS_GUARD_PAGE_VIOLATION` exception. However, due to the size of the pages (4096 bytes on Windows), it becomes impractical to trigger specifically on a particular function or instruction using this method. Instead, it typically entails single-stepping until the desired address is encountered, resulting in a significant performance impact. [19]

2.3.3 Hardware-supported methods

As discussed in section 1.2, some tools leverage modern CPU functionalities to offer low-overhead methods for monitoring and analyzing execution. These features include Intel processor trace (IPT) and hypervisor and virtualization functions. However, utilizing these features would contradict one of the goals of this thesis, which is to avoid the necessity of kernel-mode or hypervisor components. Additionally, these features are CPU-model specific and not universally supported across the x86-64 architecture.

IPT is a powerful tracing tool, described by GDB's documentation as having "very low overhead" [8]. According to [20], IPT can add as little as 5% more execution time. However, it is only supported on modern Intel processors starting from the Broadwell micro architecture [21], and AMD does not offer a competitive counterpart. Alternative solutions, such as Branch trace store, are also Intel-specific and entail additional polling overhead due to the tracing data being saved in a ring buffer. While these advanced CPU features offer impressive capabilities, their lack of universal support make them unsuitable for the tool being developed in this thesis, which can be run on any modern 64-bit Windows OS in user mode.

2.4 Obtaining function addresses

To track the execution of functions based on their addresses, it is essential to first obtain them. However, when debug data such as program databases (PDB file extension) or debugging maps (MAP file extension) is unavailable to the reverse engineer, extracting the call graph and determining the addresses where functions start can become an exceptionally complex task which is still a hot topic for researchers [22, 23]. Attempting to implement this functionality is out of the scope of this thesis since it would be unnecessary and is left to existing software.

Reverse engineers using the tool developed in this thesis are expected to utilize a disassembler initially to extract the necessary addresses. To streamline this process with IDA, the de facto industry standard for static analysis and disassembly, a special script has been created. Given that modern executables are predominantly position-independent, the resulting addresses are treated relative to the base address of the image (an EXE or DLL file).

Code listing 2.1 IDA script to export function addresses and lengths

```
auto imagename, imagebase, func;
auto start, end, offset, name, prefix, length;
imagename = get_root_filename();
imagebase = get_imagebase();
func = get_next_func(0);
while(BADADDR != func)
{
    start = get_func_attr(func, FUNCATTR_START);
    end = get_func_attr(func, FUNCATTR_END);
    offset = start - imagebase;
    name = get_func_name(func);
    prefix = substr(name, 0, 4);
    length = end - start;
    if (strstr(prefix, "sub_") != -1)
    {
        msg("%s+%016X:%u\n", imagename, offset, length);
    }
    func = get_next_func(func);
}
```

The provided script from Code listing 2.1 should be executed either in IDA's command line or in the "Execute script" window (accessible via the keyboard shortcut Shift+F2). Written in IDC, IDA's proprietary C-like scripting language, the script generates output where each line represents one function. It includes the image name, offset in hexadecimal, and function length in decimal. The inclusion of length is crucial to ensure the function has sufficient space for writing the jump instruction required for the hook, when software breakpoints are not employed. The output is formatted as follows: `filename.exe+0x1234ABCD:17`. The script ignores functions that IDA already knows the name of (line 14 of the script).

Chapter 3
Implementation – DLL

This chapter offers a comprehensive exploration of the design and implementation of the primary output of this thesis: a 64-bit DLL file designed to aid reverse engineers in locating functions within a target process on Windows OS, leveraging the approach presented in chapter 1. It also documents the specific implementation of the code execution monitoring methods introduced in chapter 2 and sheds light on the technical intricacies involved in creating the tool, including its architecture, functionality, and underlying logic.

3.1 FLOC: Function Locator

The method proposed in this thesis has been dubbed "Function Locator", or FLOC for short. Throughout the code base, FLOC is frequently used as part of a name where appropriate. For instance, all exported functions are prefixed with `FLOCDLL_`. Developed in the C programming language, the DLL is crafted using the Visual Studio 2022 integrated development environment (IDE), resulting in the filename `FLOC.dll`. Additionally, the corresponding Visual Studio project, `FLOC_DLL.vcxproj`, is included in the Visual Studio solution file `FLOC.sln`.

The code style heavily mirrors existing C code bases found in popular public repositories like MemProcFS[1], as well as Microsoft's Win32 API[2]. Specifically, the code adopts Hungarian notation and camel case for variables, pascal case for function names (with an underscore following a file name prefix, such as `File_ExampleFunctionName`), and macro case (also known as screaming snake case) for constants, macros, and data types. Additionally, the code emphasizes defensive programming, const correctness, and the use of "Yoda conditions"[3] to prevent inadvertent assignments in conditional statements. The complete source code and the compiled DLL are provided in the attachments accompanying this thesis.

[1]`https://github.com/ufrisk/MemProcFS`

[2]`https://learn.microsoft.com/en-us/windows/win32/stg/coding-style-conventions`

[3]See `https://en.wikipedia.org/wiki/Yoda_conditions` for an example.

3.2 Data Types

The source code introduces custom names for specific numeric data types, ensuring consistency and clarity throughout the code base while specifying the bit width of the variables. Table 3.1 outlines these types, as defined in the header file `types.h` using the `typedef` keyword in C. Additionally, the file defines three macros using the `#define` directive: `TRUE` and `FALSE`, equating to `(1)` and `(0)`, respectively, which are utilized for the `BOOL` data type. Moreover, `NULL` is defined to expand to `(0)`, serving as a zero value for pointers and handles.

Table 3.1 Custom integer data types

Defined as	True type	Size	Usage
`U64`	`unsigned long long`	8 bytes	Default integer type
`U32`	`unsigned long`	4 bytes	Where smaller size is appropriate.
`BOOL`	`int`	4 bytes	A boolean data type.
`BYTE`	`unsigned char`	1 byte	Buffers, pointer arithmetic.
`ADDRESS`	`unsigned long long`	8 bytes	Addresses in target memory.

Employing a dedicated data type for addresses within target (remote) processes offers numerous advantages over using a `void*`. Specifically, it prevents inadvertent dereferencing of the values, facilitates accurate addition or subtraction of bytes to the addresses without requiring recasting, and explicitly denotes the variable's purpose.

3.2.1 Vector

The code features a custom, rudimentary implementation of a vector (dynamically sized array) data structure. The vector itself is defined as a `struct` data type, and its components are detailed in Table 3.2. The header file `vector.h` declares four vector-specific functions:

- `Vector_Init`: Initializes the vector's variables and allocates sufficient memory.
- `Vector_AddressOf`: Retrieves the address of a specific element based on its index in the vector.
- `Vector_PushBackCopy`: Inserts a new element into the vector behind the last one by copying the element into the correct memory address. It reallocates memory if additional space is required and returns the address of the newly added element.
- `Vector_Free`: Releases the allocated memory and resets the vector's variables to zero.

3.2.2 Pool

Pools represent sections of executable memory reserved within the target process to accommodate hook code. Pools are defined as a `struct` data type, and their attributes are described in Table 3.3. Unlike vectors, pools are not expandable in size. For a deeper understanding of their usage and significance, refer to section 3.7.

■ **Table 3.2** The `VECTOR` data type

Data type	Variable name	Purpose
`void*`	`pData`	Pointer to the beginning of the allocated memory.
`U32`	`uElemCapacity`	Current total capacity of elements.
`U32`	`uElemCount`	Current number of elements saved in the vector.
`U32`	`uElemSize`	Size of one element in bytes.
`BYTE[4]`	`_padding`	Explicit alignment to 8 bytes.

■ **Table 3.3** The `POOL` data type

Data type	Variable name	Purpose
`ADDRESS`	`aStartAddress`	Addr. of the start of the allocated memory.
`ADDRESS`	`aCurrentFreeAddress`	Address of the first free byte.
`U64`	`uPoolSize`	Total size of the allocated memory.
`U64`	`uFreeSize`	Current remaining free bytes.

3.3 OS-specific functionality

All data types and functions that rely on the Windows API are consolidated within the files `os.h` and `os.c`. This not only enhances the organization of the code but also facilitates potential portability of the tool to other operating systems. These files are already structured to accommodate both Windows and Unix implementations through `#ifdef _WIN32` and `#ifdef LINUX` compiler directives, though currently only the former is implemented, aligning with the thesis' objectives.

In order to merge all OS-specific functionalities into a single file, the function naming convention is disregarded. Instead, more descriptive prefixes are utilized beyond a simple `OS_` prefix. Table 3.4 outlines the various prefixes used and their respective purposes. Within this thesis, terms like "target", "foreign", or "remote" process denote the analyzed process, while the process where the DLL is loaded is referred to as "local".

■ **Table 3.4** OS-specific prefixes

Prefix	Purpose
`Process_`	Checking or acquiring privileges of the local process.
`Memory_`	Memory allocations and transfers in the local process' address space.
`Target_`	Operations involving the target process and its address space.
`Thread_`	Support for multi-threading within the local process.

3.3.1 Privileges

The `Process_` prefix is solely linked to the `Process_CheckPrivileges` function. This function validates local process privileges to manipulate a target process, essential for `Target_` functions. It verifies the presence of the `SE_DEBUG_NAME` privilege, attempting acquisition via `AdjustTokenPrivileges`, and returns whether it succeeded or not.

3.3.2 Memory Operations

The tool incorporates functions essential for dynamic memory management within the local process. Primarily, `Memory_Alloc` and `Memory_Free` serve as wrappers for the Windows API functions `HeapAlloc` and `HeapFree`, respectively. Additionally, a custom `memcpy` implementation is included via `Memory_Copy`, which closely resembles existing public domain source code.

3.3.3 Target manipulation

The code offers several functions dedicated to interacting with a target process. These functions are listed in Table 3.5 and are further explained in relevant sections.

Table 3.5 `Target_` functions

Name (no prefix)	**Purpose**
`BreakpointAdd`	Writes `INT3` at a specified address, flushes the cache.
`B[...]RemoveDormant`	Removes a dormant b.p. (restores the byte and flushes).
`B[...]RemoveTriggered`	Removes a triggered b.p. (also decrements the IP).
`DebugBreak`	Injects a breakpoint exception in the target process.
`DebuggerAttach`	Registers the local process as a debugger of the target.
`DebuggerDetach`	Unregisters the local process as a debugger of target.
`HandleAcquire`	Acquires a handle to the target.
`HandleRelease`	Closes a handle to the target.
`Is64bit`	Returns whether a target process is 64-bit.
`IsDebuggerAttached`	Checks whether any process debugs the target.
`MemoryAllocExec`	Allocates executable memory in target at any address.
`MemoryAllocExecNear`	Like `MemoryAllocExec`, but near a specified address.
`MemoryFree`	Releases allocated memory in the target address space.
`MemoryRead`	Copies memory from target to local address space.
`MemoryUnprotect`	Sets page protection to `PAGE_EXECUTE_READWRITE`.
`MemoryWrite`	Copies memory from local to target address space.
`MemoryWriteFlush`	Like `MemoryWrite`, but also flushes the cache.
`WaitForBreakpoint`	Waits for and handles a debug event.

3.3.4 Threads

The code facilitates the creation of new threads within the local process by leveraging the Windows API. For compatibility with the `CreateThread` API function, the `os.h` header exposes a function prototype `THREAD_INIT_FUNC` (see Code listing 3.1). A function pointer of this type is passed to the DLL's `Thread_Start` function along with a pointer to a parameter that the provided function will receive upon invocation in the newly created thread. The thread initialization process first encapsulates both the function and its parameter into a `THREAD_INIT_INFO` structure (Code listing 3.1), and then the thread begins execution with the static (unexposed) function `Thread_Init`. Upon success, the `Thread_Start` function provides a handle to the newly created thread via an output parameter `pThread`. This mechanism is detailed in Code listing 3.2. Subsequently, the `Thread_Init` function invokes the `THREAD_INIT_FUNC` and passes the parameter to it. After the function returns, `Thread_Init` is responsible for deallocating the memory of the `THREAD_INIT_INFO` structure (see Code listing 3.3). The code also provides functions to wait for a thread to finish executing given a timeout in milliseconds, and to close the handle to the thread. These are functions `Thread_WaitExit` and `Thread_Close`, which are wrappers for Windows API `WaitForSingleObject` and `CloseHandle` respectively.

Code listing 3.1 Data types for thread initialization

```
typedef void (*THREAD_INIT_FUNC)(void*);
typedef struct tdTHREAD_INIT_INFO {
    THREAD_INIT_FUNC fnFunc;
    void* pParam;
} THREAD_INIT_INFO;
```

Code listing 3.2 `Thread_Start` function

```
BOOL Thread_Start(THREAD_INIT_FUNC const fnFunc,
                  void* const pParam, THREAD* const pThread)
{
    THREAD_INIT_INFO* const pInitInfo =
        Memory_Alloc(sizeof(THREAD_INIT_INFO));
    if (NULL == pInitInfo)
    {
        *pThread = NULL;
        return FALSE;
    }
    pInitInfo->fnFunc = fnFunc;
    pInitInfo->pParam = pParam;
    HANDLE const hThread = CreateThread(NULL, 0, Thread_Init,
                                        pInitInfo, 0, NULL);
    if (NULL == hThread)
    {
        *pThread = NULL;
        return FALSE;
    }
    *pThread = hThread;
    return TRUE;
}
```

Code listing 3.3 `Thread_Init` function

```
static DWORD WINAPI Thread_Init(void* lpParam)
{
    THREAD_INIT_INFO* const pInitInfo =
        (THREAD_INIT_INFO*)lpParam;
    pInitInfo->fnFunc(pInitInfo->pParam);
    Memory_Free(pInitInfo);
    return 0;
}
```

3.4 Trackers

The tool introduces the concept of a "Tracker" to represent a function in the target process whose execution is being monitored. The `TRACKER` data type encompasses all necessary information about the tracker, as detailed in Table 3.6, including a union representing both means of tracking function execution: either a software breakpoint or an inline hook. The specific data types in the union (`BREAKPOINT` and `HOOK`) are described in their respective sections. The `TRACKER_TYPE` enumeration data type is defined as either `TRACKER_TYPE_BREAKPOINT_SW` (1) or `TRACKER_TYPE_HOOK_INLINE` (2), alternatively `TRACKER_TYPE_DELETED` (0).

Table 3.6 `TRACKER` data type

Data type	Var. name	Purpose
`ADDRESS`	`aAddress`	Address of the function in the target process.
`TRACKER_TYPE`	`eType`	Number representing the tracker type.
`BOOL`	`bEnabled`	Value indicating if the tracker is currently active.
`BOOL`	`bHit`	Value indicating if the function was executed.
`BYTE[4]`	`_padding`	Explicit alignment to 8 bytes.
`union`	`u`	Union of specific tracker-type data types.

3.5 Context and multi-instancing

The tool maintains a context structure with variables that track the current analysis state, such as the target process, a vector of all trackers, and whether a step is currently active. A comprehensive overview of the corresponding `FLOC_CTX` data type is provided in Table 3.7. The tool supports the existence of multiple contexts simultaneously. While this usage scenario is not anticipated, it eliminates the need to load the DLL multiple times if any software utilizing this tool requires it. This functionality is realized by maintaining a global array of contexts called `gContexts`. By default, the maximum number of contexts is set to four.

Before initiating any operations, the exported function `FLOCDLL_Initialize` must be invoked. This function inserts a new context into the array, returning a `FLOC_HANDLE` variable to the caller via an output parameter. When interacting with the tool through every other exported function, this handle must be provided as the first parameter. The DLL interprets the handle as a pointer to the context (`FLOC_CTX`). If found in the global array, the tool proceeds with the invoked function relative to the specific context. Upon a call to `FLOCDLL_Uninitialize`, the context is removed from the array, making room for a new one.

The `FLOC_HANDLE` data type is implemented as an abstract pointer, ensuring that a handle obtained from outside the DLL's scope is never directly dereferenced. Instead, it is first validated by `FLOC_ContextGet`, which accesses the `gContexts` array, returning a valid `FLOC_CTX` pointer or `NULL`. The context data type and related functions are declared in the `floc.h` header file, which generally contains additional code for the exported functions in `flocdll.h` that did not fit within. Code listing 3.4 illustrates the implementation of this functionality, excluding `FLOC_ContextGet` – see section 3.12 for its source code. Additional functions `FLOC_ContextGetCount` and `FLOC_ContextGetMax` are employed within `FLOCDLL_Initialize` to ensure the array is not full before attempting to insert a new context.

Table 3.7 `FLOC_CTX` data type

Data type	Variable name	Purpose
`VECTOR`	`vecTrackers`	Vector of all `TRACKER`s.
`VECTOR`	`vecPools`	Vector of all `POOL`s.
`THREAD`	`thrDebug`	Handle to the thread running a debug loop.
`PID`	`pidTarget`	Identifier of the target process.
`BOOL`	`bForeignDebugLoop`	A foreign debug loop is used (true / false).
`BOOL`	`bIsStepActive`	A step is currently active (true / false).
`BOOL`	`bDbgLoopRunning`	A debug loop (any) is running (true / false).
`BOOL`	`bIsPendingReset`	Pending tracker reset / filter after step end.
`BOOL`	`bStopDebugLoop`	Flag to stop the debug loop thread.
`BOOL`	`bTargetDied`	Flag indicating the target process died.
`BYTE[4]`	`_padding`	Explicit alignment to 8 bytes.

Code listing 3.4 `FLOC_CTX` related code

```
#define MAX_CONTEXTS_COUNT 4
static FLOC_CTX* gContexts[MAX_CONTEXTS_COUNT] = { 0 };
static U32 gContextCount = 0;
typedef struct tdFLOC_HANDLE* FLOC_HANDLE;
void FLOC_ContextInsert(FLOC_CTX* const pCtx)
{
    for (U32 i = 0; i < MAX_CONTEXTS_COUNT; i++)
    {
        if (NULL == gContexts[i])
        {
            gContexts[i] = pCtx;
            gContextCount++;
            break;
        }
    }
}
void FLOC_ContextClear(FLOC_CTX const * const pCtx)
{
    for (U32 i = 0; i < MAX_CONTEXTS_COUNT; i++)
    {
        if (pCtx == gContexts[i])
        {
            gContexts[i] = NULL;
            gContextCount--;
            break;
        }
    }
}
```

3.6 Software breakpoints

Software breakpoints represent one of the two methods utilized for monitoring code execution. This section details their specific implementation. The `BREAKPOINT` structure, encapsulated within the union inside `TRACKER`, contains only one variable, `BYTE uOriginalByte`. This variable is accessed as `TRACKER.u.bp.uOriginalByte`. It holds the byte read from the function's address in the target memory, to preserve the original functionality after being overwritten with the `INT3` instruction.

3.6.1 Creating a breakpoint tracker

To create a breakpoint tracker, `FLOCDLL_TrackerAddBreakpoint` is invoked, requiring the function's address as a parameter alongside `FLOC_HANDLE`. Upon successful execution, the breakpoint remains "dormant", meaning it is inactive in the target process and ready to be enabled, and `FLOC_STATUS_SUCCESS` is returned. For more information about the return type, refer to section 3.9. The function follows this sequence of events:

1. Validates the handle (returns `FLOC_STATUS_INVALID_HANDLE` on failure).
2. Ensures a target process is set (`FLOC_STATUS_TARGET_NOT_SET`).
3. Checks for an existing tracker with this address (`FLOC_STATUS_TRACKER_ALREADY_EXISTS`).
4. Creates a `TRACKER` structure, setting `aAddress` to the specified address, `eType` to `TRACKER_TYPE_BREAKPOINT_SW`, and `bEnabled` and `bHit` to `FALSE`.
5. Acquires a handle to the target process (`FLOC_STATUS_PROCESS_HANDLE_ACQUIRE_FAIL`).
6. Reads target's memory and saves the byte at the specified address to `uOriginalByte` (`FLOC_STATUS_MEMORY_READ_FAIL`).
7. Closes the handle.
8. Adds the tracker to `vecTrackers`, the vector of trackers in the context (`FLOC_STATUS_VECTOR_PUSHBACK_FAIL`).

3.6.2 Starting the debug loop

Before enabling the breakpoint, a debugger must be attached to the target process and a debug loop started to handle the incoming debug events. This is achieved by calling `FLOCDLL_DebugLoopStart`. The function validates the following conditions:

- The context handle is valid (`FLOC_STATUS_INVALID_HANDLE`).
- A target process is set (`FLOC_STATUS_TARGET_NOT_SET`).
- The target process is alive, i.e., a handle to it can be acquired (`FLOC_STATUS_INVALID_TARGET`).

- A debug loop is not already running (FLOC_STATUS_DEBUG_LOOP_ALREADY_RUNNING).
- A debugger is not already attached to the target process (FLOC_STATUS_DEBUGGER-_ALREADY_ATTACHED, or FLOC_STATUS_TARGET_CANNOT_CHECK_DEBUGGER if the information cannot be obtained from the operating system).

If all the conditions are met, the function starts a new thread via `Thread_Start`. The `THREAD_INIT_FUNC` passed is the function `FLOC_DebugLoop`, along with a pointer to the current context as its parameter. If the thread creation succeeds, the context value `bDbgLoopRunning` is set to `TRUE`, `bForeignDebugLoop` to `FALSE` (its usage is explained in section 3.6.8), and `thrDebug` is assigned a handle to the newly created thread.

When invoked in the debugging thread, `FLOC_DebugLoop` attempts to register the local process as a debugger of the target via `Target_DebuggerAttach`. This function calls the Windows API functions `DebugActiveProcess`, and `DebugSetProcessKillOnExit` with the parameter set to `FALSE` to ensure the target is not terminated if the debugging thread exits without detaching the debugger. If successful, it then enters an infinite loop calling `Target_WaitForBreakpoint` until it is signaled to stop or the target process terminates, after which the thread finishes execution.

3.6.3 Enabling breakpoint trackers

Enabling a breakpoint tracker can be done either via `FLOCDLL_TrackerEnable`, referencing the tracker by the function's address, or through `FLOCDLL_TrackerAllEnable`. If attempted without a running debug loop, `FLOC_STATUS_ENABLING_BREAKPOINT_WITHOUT_-DEBUGGING` is returned. However, the latter function still proceeds to enable all remaining hook trackers, exempt from this requirement.

The actual activation process resides within `FLOC_TrackerEnable` to avoid redundancy between the two `FLOCDLL` functions. Upon invocation, the necessary conditions are already verified and a handle to the target process is acquired. Subsequently, it calls the appropriate function for the tracker type (`Target_BreakpointAdd`). Upon success, it marks the tracker's `bEnabled` field as `TRUE`. This function embeds the `INT3` instruction at the tracker's designated address in the target memory and flushes the instruction cache at that location using the Windows API functions `WriteProcessMemory` and `FlushInstructionCache`.

3.6.4 Breakpoint handler

This section focuses on the `Target_WaitForBreakpoint` function, integral to the debug loop. To maintain separation between Function Locator logic and OS-specific operations, a function pointer is employed. `FLOC_BreakpointHandler` is the designated function, accepting the current context pointer as a parameter. A condensed representation of the debugger functionality is provided in Code listing 3.5. It is crucial to note that when an `EXCEPTION_DEBUG_EVENT` with a code different than `EXCEPTION_BREAKPOINT` is received, the function signals that it does not handle such an exception. Otherwise, the target process might freeze due to an infinite loop scenario, where this identical exception would be received again and again since there is no other debugger to handle it, and the process would not be terminated in situations where expected [24].

■ **Code listing 3.5** Target_WaitForBreakpoint function

```
typedef BOOL (*BREAKPOINT_HANDLER_FUNC)(void*,ADDRESS,BYTE*);
BOOL Target_WaitForBreakpoint
(
    BREAKPOINT_HANDLER_FUNC const pBreakpointHandler,
    void* const pParam /* FLOC_CTX* */
)
{
/* Indentation modified. */
DEBUG_EVENT DE;
DWORD dwContinueStatus = DBG_CONTINUE;
WaitForDebugEvent(&DE, INFINITE);

/* Switch statement begin. */
switch(DE.dwDebugEventCode)
{
case EXCEPTION_DEBUG_EVENT:

if(EXCEPTION_BREAKPOINT ==
   DE.u.Exception.ExceptionRecord.ExceptionCode)
{
    ADDRESS const aAddress =
            DE.u.Exception.ExceptionRecord.ExceptionAddress;

    BYTE uOriginalByte = 0;
    BOOL bRemoveBreakpoint = pBreakpointHandler(pParam,
                                                aAddress,
                                                &uOriginalByte);
    if (bRemoveBreakpoint)
    {
        Target_BreakpointRemoveTriggered(DE.dwProcessId,
                                          DE.dwThreadId,
                                          aAddress,
                                          uOriginalByte);
    }
}
else /* Exception code != EXCEPTION_BREAKPOINT. */
{
    dwContinueStatus = DBG_EXCEPTION_NOT_HANDLED;
}

break;
/* Other debug event codes are omitted. */
}
/* Switch statement end. */

ContinueDebugEvent(DE.dwProcessId,
                   DE.dwThreadId,
                   dwContinueStatus);
/* ... */
}
```

As documented by the code, the breakpoint handler (`FLOC_BreakpointHandler`) determines whether the breakpoint should be removed, indicating if it genuinely corresponds to a tracker and was not triggered by external factors. If this condition is met and a step is currently active, it updates the relevant tracker's `bHit` field to `TRUE` and `bEnabled` to `FALSE`. Additionally, the original byte of the function saved in the tracker (`uOriginalByte`) that should replace the `INT3` instruction is passed to the caller through an output parameter. Following this, `Target_BreakpointRemoveTriggered` is called to execute this task. Its source code is shown in Code listing 3.6. The code mirrors the steps outlined in subsection 2.1.2, albeit without the single-stepping (`TRAP` flag) functionality as announced.

■ **Code listing 3.6** `Target_BreakpointRemoveTriggered` function

```
void Target_BreakpointRemoveTriggered(PID const pidProcess,
                                      TID const tidThread,
                                      ADDRESS const aAddress,
                                      BYTE const uOrigByte)
{
    HANDLE const hThread = OpenThread(THREAD_ALL_ACCESS,
                                      FALSE,
                                      tidThread);
    if (NULL == hThread)
    {
        return;
    }
    CONTEXT threadContext;
    threadContext.ContextFlags = CONTEXT_ALL;
    if (!GetThreadContext(hThread, &threadContext))
    {
        goto ret;
    }
    threadContext.Rip -= 1;
    if (!SetThreadContext(hThread, &threadContext))
    {
        goto ret;
    }
    BYTE const byte = uOrigByte;
    HANDLE const hProc = OpenProcess(PROCESS_ALL_ACCESS,
                                     FALSE,
                                     pidProcess);
    if (NULL == hProc)
    {
        goto ret;
    }
    WriteProcessMemory(hProc,(LPVOID)aAddress,&byte,1,NULL);
    FlushInstructionCache(hProc, (LPCVOID)aAddress, 1);
    CloseHandle(hProc);
ret:
    CloseHandle(hThread);
    return;
}
```

3.6.5 Disabling breakpoint trackers

Breakpoint trackers can be disabled either individually using `FLOCDLL_TrackerDisable`, by specifying the function address, or collectively through `FLOCDLL_TrackerAllDisable`, either manually or as part of the uninitialization process in `FLOCDLL_Uninitialize`. Similar to the enabling process, the underlying functionality is encapsulated in a helper function – `FLOC_TrackerDisable`. This function then invokes `Target_BreakpointRemoveDormant`, passing the original byte as a parameter, and sets the `bEnabled` field of the tracker to `FALSE`. Removal of the breakpoint entails overwriting the `INT3` instruction with the original byte and flushing the instruction cache.

3.6.6 Halting the debug loop

To stop the debug loop, either via `FLOCDLL_DebugLoopStop` or internally through `FLOCDLL_Uninitialize`, the context value `bStopDebugLoop` is set to `TRUE`. This flag serves as a signal for the debug loop to conclude. Before raising the flag, any remaining enabled breakpoint trackers in the target process are attempted to be disabled to prevent unhandled exceptions. The following return values document the possible scenarios where the function fails to proceed:

- `FLOC_STATUS_INVALID_HANDLE`
- `FLOC_STATUS_TARGET_DIED` (see subsection 3.6.7)
- `FLOC_STATUS_DEBUG_LOOP_ALREADY_STOPPED`
- `FLOC_STATUS_DEBUG_LOOP_FOREIGN` (see subsection 3.6.8)
- `FLOC_STATUS_PROCESS_HANDLE_ACQUIRE_FAIL`

In the absence of any issues, the flag is set, prompting the infinite loop in `FLOC_DebugLoop` to break. Subsequently, `Target_DebuggerDetach` is invoked, memory allocated for the `THREAD_INIT_INFO` structure is deallocated, and the thread concludes its execution. `FLOCDLL_DebugLoopStop` waits for this to occur using `Thread_WaitExit` with a 100 ms timeout. If the thread has not yet finished within this time frame, it indicates that the debug loop might still be awaiting a return from a `WaitForDebugEvent` call (see Code listing 3.5, line 11), and thus, unable to notice the termination flag.

In order to wake up the thread, a breakpoint exception is deliberately triggered within the target process using `Target_DebugBreak`, a wrapper for the API function `DebugBreakProcess`. This action generates a debug event, affording the debugging thread an opportunity to read the `bStopDebugLoop` value. A lengthier timeout is applied, and upon completion, if the thread indeed concludes its execution, the handle to it is closed (`Thread_Close`), and the `bDbgLoopRunning` and `bStopDebugLoop` values are reset to `FALSE` in the context. Should `Target_DebugBreak` encounter failure or the second timeout expires, the function returns `FLOC_STATUS_DEBUG_BREAK_FAIL` or `FLOC_STATUS_DEBUG_LOOP_STOP_FAIL`, respectively.

3.6.7 Target termination

When the target process terminates, the debugging thread receives a notification through a debug event with either the code `EXIT_PROCESS_DEBUG_EVENT` or `RIP_EVENT` [25]. This prompts `Target_WaitForBreakpoint` to return `TRUE`, which is detected within `FLOC_DebugLoop`. Consequently, the loop is broken, and the debugger is detached, mimicking the behavior triggered by raising the `bStopDebugLoop` flag. Additionally, before the thread concludes its execution, it sets the context value `bTargetDied` to `TRUE`. This flag is subsequently checked in `FLOC_IsTargetDead` as part of the higher-level functions `FLOCDLL_DebugLoopStop`, `FLOCDLL_StepBegin`, and `FLOCDLL_StepEnd`. Essentially, it resets the context. However, unlike a full uninitialization, it does not deallocate the vector of trackers and preserves the context in `gContexts`, allowing the software utilizing this tool to retain the last state of the analysis if desired.

3.6.8 Foreign debug loop

Given that this tool is intended for integration within existing reverse engineering software and frameworks, it is probable that such software may function as a debugger and run its own debug loop for various purposes. To prevent potential conflicts between different debuggers and to ensure that the software retains full control over debug events, the DLL provides a `FLOCDLL_CallExceptionBreakpointHandler` function. This function should be invoked when a breakpoint exception occurs during debugging. The debugger running outside of Function Locator's scope is referred to as "foreign", in contrast to "native", which represents the DLL's own debugger.

Before utilizing this function, the context must be informed about the usage of a foreign debugger through `FLOCDLL_DebugLoopOverride`. This function utilizes a `BOOL` parameter to indicate the start or stop of the foreign debug loop. Most functions within the DLL do not differentiate between a native or foreign debugger, and all features of the tool are supported in both scenarios.

`FLOCDLL_CallExceptionBreakpointHandler` requires specific information that only the debugging thread possesses, namely the thread identifier where the exception occurred and the corresponding address. The function ensures that a foreign debug loop is active and then proceeds to invoke the same breakpoint handler utilized by a native debugger (`FLOC_BreakpointHandler`). If the handler returns true, indicating that the breakpoint corresponds to a tracker and should be removed, the function calls `Target_BreakpointRemoveTriggered` as well.

3.7 Inline Hooks

Hook trackers are managed similarly to breakpoint trackers from an external perspective, with the primary difference being the dedicated creation function for each tracker type – `FLOCDLL_TrackerAddHook` for hooks. Additionally, hooks do not require a debug loop to be running. The `HOOK` data type, which is part of the `TRACKER`'s union, is more complex, occupying 40 bytes. Table 3.8 provides an overview of its fields.

Table 3.8 `HOOK` data type

Data type	Variable name	Purpose
`ADDRESS`	`aHookAddress`	Address where the hook code starts.
`U32`	`uJumpBytesLen`	Length of the jump code in bytes.
`U32`	`uHitOffset`	Offset of the "hit" byte from `aHookAddress`.
`BYTE[14]`	`uJumpBytes`	Buffer for the jump code.
`BYTE[2]`	`_padding`	Explicit alignment to 8 bytes.

3.7.1 Near and far pools

As discussed in subsection 3.2.2, pools represent executable memory allocated in the target process where the hook code is placed. Utilizing pools instead of allocating memory for each hook individually optimizes long-term operations. This approach minimizes the total number of allocations and reduces overall memory overhead, as the Windows API always rounds memory allocations up to the nearest page boundary (4096 bytes) [26], even if the hook code requires only a fraction of that space.

When a new pool is created, it is added to the vector of pools (`pVecPools`) in the context. The pool fields `aStartAddress` (the starting address of the allocated memory in the target) and `uPoolSize` (the size of the allocated memory) remain constant after creation. However, other fields, namely `aCurrentFreeAddress` (the first free address after the last hook code) and `uFreeSize` (the total remaining free space), are updated each time new code is added to the pool.

The code distinguishes between "near" and "far" pools. A pool is considered near a target address if the absolute distance from the pool's `aCurrentFreeAddress` to the target address is less than 2 GB, meaning the relative distance can be represented by a signed 32-bit integer. When creating a hook, a near pool is preferred, even if it requires allocating a new pool. This preference is due to the shorter jump code (5 bytes vs. 14 bytes) required for near pools, which allows shorter functions to be hooked and reduces the hook code size (30 bytes vs. 64 bytes), saving memory and CPU instructions. Further details are provided in the next subsection. The `pool.h` file exposes a single function, `Pool_FindOrCreateBest`, which takes a pointer to the context's vector of pools, an address the pool should be near, the required free space, the near distance (±2 GB), and a handle to the target process. The "best" pool is determined based on the following criteria, ranked from most to least preferred:

1. An existing near pool with enough free space.
2. A newly allocated near pool.
3. An existing far pool with enough free space.
4. A newly allocated far pool.

For far pools, the function `Pool_CreateAnywhere` requests the OS to allocate 64 kB of memory anywhere in the target process, sufficient for 1000 far hook instances. It calls `Target_MemoryAllocExec`, a wrapper for Windows' `VirtualAllocEx` function, and sets `aStartAddress` and `aCurrentFreeAddress` to the returned value (or returns `FALSE` if it is NULL), and uPoolSize and uFreeSize to 64 kB, returning TRUE. All memory is allocated with the PAGE_EXECUTE_READWRITE protection constant.

Allocating near memory is more complex and is handled by Target_MemoryAllocExecNear, which requires the address, near distance, and minimum size to be supplied. While a desired starting address may be specified for VirtualAllocEx, it will fail if the specific pages are not free, while a different address within 2 GB might be available. Similarly, the required size might not be accommodated, but a smaller size would succeed. On the other hand, the tool might miss out on allocating more bytes than it tried to. The code uses the VirtualQueryEx API function to find the closest region of free pages before attempting allocation of its entire length.

After calculating the “near” boundary from the ± 2 GB distance around the address or the minimum and maximum application addresses obtained from GetSystemInfo, whichever is stricter, it uses the DLL’s FindPrevFreeRegion function, descending the address space and searching for a free region. It accounts for allocation addresses being rounded down to the nearest multiple of the “allocation granularity”, which is also obtained from GetSystemInfo and should be 64 kB [27]. If successful, the function returns the address of the beginning of the found free region and passes its size via an output parameter, as shown in Code listing 3.7. If no suitable region is found, FindNextFreeRegion is called to search upwards. Finally, when one is located, VirtualAllocEx is invoked to allocate the entire range of pages in the region.

■ **Code listing 3.7** FindPrevFreeRegion function

```c
static ADDRESS FindPrevFreeRegion(PROCESS const hProcess,
    ADDRESS const aAddress,       ADDRESS const aMin,
    U32 const uAllocGranularity,  U64* const puRegionSize)
{
    ADDRESS aTry = aAddress;
    aTry -= (aTry % uAllocGranularity) + uAllocGranularity;
    while (aTry >= aMin)
    {
        MEMORY_BASIC_INFORMATION mbi;
        if (!VirtualQueryEx(hProcess, (LPVOID)aTry,
                            &mbi,     sizeof(mbi)))
        {
            break;
        }
        if (MEM_FREE == mbi.State)
        {
            *puRegionSize = mbi.RegionSize;
            return aTry;
        }
        if ((U64)mbi.AllocationBase < uAllocGranularity)
        {
            break;
        }
        aTry = (U64)mbi.AllocationBase - uAllocGranularity;
    }
    return NULL;
}
```

3.7.2 Creating a hook tracker

When attempting to create a new hook tracker via FLOCDLL_TrackerAddHook, the function might return a FLOC_STATUS_HOOK_CREATE_FAIL code, in addition to the return values seen in FLOCDLL_TrackerAddBreakpoint. This occurs when the internal hook creation in Hook_Create fails, for instance, due to an insufficient function length, which must be provided to the DLL via the uFuncLen parameter. If all required conditions are met, the DLL creates a tracker of type TRACKER_TYPE_HOOK_INLINE and calls the Hook_Create function, passing pointers to the vector of pools and the tracker, a handle to the target process, and the function length.

The hook creation code first attempts to obtain a pool from Pool_FindOrCreateBest. If this function returns NULL, hook creation fails, and FALSE is returned. Otherwise, the hook creation proceeds to the appropriate function, depending on whether the pool is near (CreateHookRel32) or far (CreateHookAbs64).

The jump code is shorter when a near pool is used because the x86-64 architecture provides the JMP rel32 instruction, which can jump a maximum distance of a signed 32-bit integer, but takes only five bytes – a 0xE9 opcode and a four-byte signed displacement [28]. Therefore, if a near pool is available and the offset from the function's address to the hook code fits within a 32-bit signed integer, any function longer than four bytes can be hooked. The hook code is also shorter because fewer bytes need to be overwritten when reinstating the original function.

The function CalcSignedDisplacement32, shown in Code listing 3.8, calculates the signed 32-bit displacement between two addresses (64-bit unsigned integers). Inside the CPU, the jump instruction effectively adds the displacement to the current value of the instruction pointer ($IP = IP + rel32$), but it is important to note that when executing this instruction, the IP register already holds the address of the next instruction in memory.

Code listing 3.8 CalcSignedDisplacement32 function

```
typedef signed long I32;
static I32 CalcSignedDisplacement32(U64 const a, U64 const b)
{
    U64 const uAbsDiff = (a > b) ? (a - b) : (b - a);
    return (a > b) ? (-1) * (I32)uAbsDiff : (I32)uAbsDiff;
}
```

CreateHookRel32 is responsible for creating the shell code for both the jump and the hook. It sets the uJumpBytesLen value inside the HOOK structure to five, reads the function's original bytes, and embeds them into the hook code. The hook code takes 29 bytes of instructions, followed by the "hit" byte, which is set by the hook code to a value of one to indicate the function was executed. The offset of this byte (29) is saved to the hook's uHitOffset value.

The code is written into the pool using Target_MemoryWriteFlush, and the function's bytes are ensured to be modifiable with a Target_MemoryUnprotect call, allowing the hook to later overwrite them. If everything succeeds, the pool's uFreeSize is decremented and aCurrentFreeAddress is incremented, according to the hook code size (30 bytes). The entire jump and hook shellcode is documented in Code listing 3.9.

■ **Code listing 3.9** Near-hook pseudocode

```
; JUMP:
; Bytes: E9 78 56 34 12,
; 78 ... 12 is 32-bit displacement from RIP to hook
; Opcode: E9 cd ; Instruction: JMP rel32 (RIP = RIP + rel32)
JMP 0x12345678
;
; HOOK: Set hit byte to 1
; Bytes offset 0x0: C6 05 16 00 00 00 01
; 16 ... 00 is 32-bit displacement from RIP to hit byte (0x16)
; Opcode: C6 /0 ib ; Instruction: MOV r/m8, imm8
MOV BYTE PTR [RIP+0x16], 0x1
;
; HOOK: Restore original function (bytes 0-4)
; Bytes offset 0x7: C7 05 xx xx xx xx AA BB CC DD
; xx is 32-bit displacement from RIP to function
; AA-DD are function original bytes 0-4
; Opcode: C7 /0 id ; Instruction: MOV r/m32, imm32
MOV DWORD PTR [RIP+xx], 0xDDCCBBAA
;
; HOOK: Restore original function (byte 5)
; Bytes offset 0x11: C6 05 xx xx xx xx EE
; xx is displacement from RIP to (function + 4)
; EE is fifth original function byte
; Opcode: C6 /0 ib ; Instruction: MOV r/m8, imm8
MOV BYTE PTR [RIP+xx], 0xEE
;
; HOOK: Jump back to original function
; Same instruction as jump code, displ. from RIP to function
;
; HOOK: hit byte
; Bytes offset 0x1D: 00
```

The architecture does not support an instruction for a relative 64-bit jump, nor does it provide an absolute jump to a 64-bit immediate value [28]. When a far pool is used and the offset does not fit in a 32-bit value, the absolute value must either be loaded into a register or an indirect approach must be employed. Common methods for implementing 64-bit jumps include:

1. Using a `MOV reg / CALL reg` pair, alternativelly `MOV reg / JMP reg`
2. Pushing the address onto the stack and executing a `RET` instruction.
3. Indirect `CALL QWORD PTR [Pointer to address to jump to]` or `JMP QWORD PTR`.

With the first variant necessitating backing up and restoring the register from the stack, all three methods require 14 bytes to implement. The third approach, using an indirect `JMP` instruction, was chosen because it avoids using any registers or the stack and does not disrupt the CPU's return address branch prediction, thereby preventing unnecessary performance overhead [29]. While this approach adds two layers of indirection, in practice, the absolute address can be placed immediately after the opcode by jumping to `[RIP + 0x00]`. For the hook code, a register must be used since there is no way to transfer the bytes directly across a 64-bit distance. The `RAX` register is pushed and then popped from the stack. The complete far-hook implementation is documented in Code listing 3.10.

■ **Code listing 3.10** Far-hook pseudocode

```
; JUMP:
; Bytes: FF 25 00 00 00 00 xx xx xx xx xx xx xx xx
; 00 ... 00 is 32-bit offset from RIP
; xx is the absolute address of hook
; Opcode: FF /4 ; Instruction: JMP r/m64 (RIP = [RIP+rel32])
; Zeroed out rel32 translates to RIP = [RIP+0] = [RIP]
JMP QWORD PTR [RIP]
;
; HOOK: Set hit byte
; Bytes offset 0x0: C6 05 38 00 00 00 01
; 38 ... 00 is offset from RIP to hit byte (0x38)
; Opcode: C6 /0 ib ; Instruction: MOV r/m8, imm8
MOV BYTE PTR [RIP+0x38], 0x1
;
; HOOK: Backup RAX register
; Bytes offset 0x7: 50
; Opcode: 50+rd ; Instruction: PUSH r64
PUSH RAX
;
; HOOK: Move original function (bytes 0-7) to RAX
; Bytes offset 0x8: 48 B8 77 66 55 44 33 22 11 00
; 00-77 are function original bytes 0-7
; Opcode: REX.W + B8+ rd io ; Instruction: MOV r64, imm64
MOVABS RAX, 0x0011223344556677
;
; HOOK: Restore original function from RAX (bytes 0-7)
; Bytes offset 0x12: 48 A3 77 66 55 44 33 22 11 00
; 77 ... 00 is absolute address of function
; Opcode: REX.W + A3 ; Instruction: MOV moffs64, RAX
MOVABS QWORD PTR [0x0011223344556677], RAX
;
; HOOK: Repeat above for original function bytes 8-F
;
; HOOK: Restore RAX register
; Bytes offset 0x30: 58
; Opcode: 58+ rd ; Instruction: POP r64
POP RAX
;
; HOOK: Jump back to original function
; Same instruction as jump code, absolute address of function
JMP QWORD PTR [RIP]
;
; HOOK: hit byte
; Bytes offset 0x3F: 00
```

Section 2.2 raised concerns about the atomicity of function overwrite operations. Although the approaches shown in the previous source codes are not atomic, testing did not reveal any problems. However, the architecture does allow for atomic implementations. For near-hooks, the five-byte jump code fits into a standard 64-bit register, making atomicity relatively easy to achieve by backing up three additional bytes. This approach must correctly handle cases where the excess bytes belong to a different function. The far-hook implementation already copies two additional bytes beyond the length of the jump code. For atomic far-hooks, AVX instructions can be used if the CPU supports them. To maximize performance, address alignment must be considered and the appropriate instruction selected. Alternatively, some processors implement the `MOVDIR64B` instruction, which does not require AVX and its registers. This instruction "[moves] 64-bytes as a direct-store with guaranteed 64-byte write atomicity from the source memory operand address to the destination memory address specified [...] in the register operand", which is sufficient to handle both hook sizes atomically. [30]

3.7.3 Enabling hook trackers

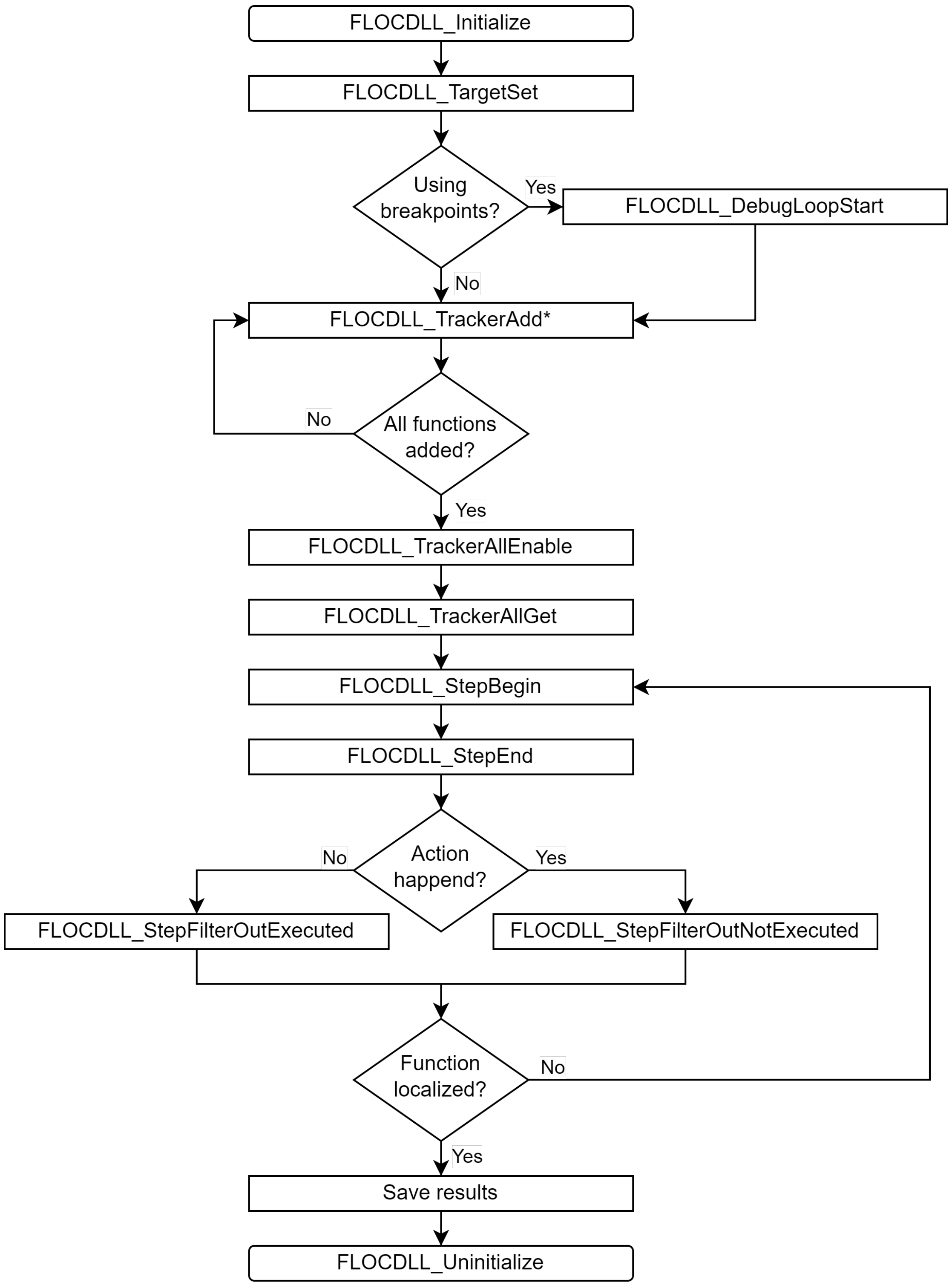
After a hook tracker is created, it is not immediately active, similar to breakpoints. The hook code is prepared in the pool, and the jump code is prepared in the `HOOK` field `uJumpBytes`, but it is not yet written into the target's memory. When enabling a hook, either directly or through `FLOCDLL_TrackerAllEnable`, the function `FLOC_TrackerEnable` calls `Hook_Enable` and sets the `bEnabled` field to `TRUE` if the operation succeeds. Similar to breakpoints, the enabling process is straightforward, but before overwriting the function with the jump code, the "hit byte" (accessed via `TRACKER.u.hook.aHookAddress + TRACKER.u.hook.uHitOffset`) must be reset to zero. Additionally, when a step is ended (`FLOCDLL_StepEnd`), the tracker's `bHit` value must be manually updated by reading the byte from target memory, because unlike breakpoints, there is no handler to do it immediately after function execution.

3.7.4 Disabling hook trackers

When a hook tracker is disabled, the only action taken is setting the `bEnabled` field to `FALSE`. There are no functions to remove the hook from the target. If it is currently active, the jump code will be removed on the first execution of the function. Thus, the tracker can be safely kept in the target's memory even after Function Locator is closed. Manually replacing the jump code with the original bytes would incur more overhead than allowing the hook to handle it autonomously. Similarly, the hook code and the pool's allocated memory are never released. These resources are intentionally leaked because there is no need to release them. It would only add unnecessary complexity and overhead to the tool. Unlike breakpoints, a leftover hook does not pose any risk.

3.8 Workflow

The standard workflow when using this tool is depicted in the flowchart in Figure 3.1. Before starting the analysis, the reverse engineer must obtain the function addresses, possibly using the IDA script provided in section 2.4, and launch the target process.



■ **Figure 3.1** FLOC.dll workflow

After loading the DLL, initializing a context, and setting the target PID to a running process, the reverse engineer is expected to load the function addresses (create the trackers), start the debug loop if breakpoints are used, and begin analysis by enabling all trackers and starting the first step. After observing the desired action or intentionally triggering other actions, the step is ended and the trackers are filtered. This process continues by re-enabling all remaining trackers and starting another step until a satisfactory reduction in the number of functions is achieved. The engineer then saves the resulting addresses from the vector and uninitializes the tool. Chapter 5 provides tips for achieving optimal results more quickly.

While the flowchart shows `FLOCDLL_TrackerAllGet` being called only once, it is important to remember that after each new tracker is added, the vector might be reallocated. Therefore, this function should be called again, and the previously reported memory should not be accessed. The flowchart does not cover alternative workflows, such as using a foreign debug loop where `FLOCDLL_DebugLoopStart` is replaced with `FLOCDLL_DebugLoopOverride`, or bypassing tracker filtering before a new step, i.e., ignoring the step and calling `FLOCDLL_TrackerAllReset`. Another option not shown is enabling only selected trackers to minimize the number of active trackers at any given time, thus reducing overhead at the cost of a longer analysis, if maximum performance is required.

3.9 Exported Functions

This section provides a summary of each exported function and its respective purpose. The return value for all functions is a `FLOC_STATUS` data type, which translates to a 32-bit unsigned integer. The value `FLOC_STATUS_SUCCESS` is defined as zero, while any other value indicates a failure. The file `status.h` defines a total of 45 distinct error codes to specify the reason for function failure. If a relevant status is not defined, a generic `FLOC_STATUS_FAILURE` (1) is returned. With the exception of `FLOCDLL_Initialize`, all exported functions require a valid `FLOC_HANDLE` as the first parameter. If an invalid handle is provided, `FLOC_STATUS_INVALID_HANDLE` is returned.

All exported functions are declared within the file `flocdll.h`, which is partially segmented between Windows and Unix implementations using `#ifdef` directives. However, this segregation is solely to ensure the functions are exported. The `FLOC_EXPORT` directive is utilized for this purpose. In Windows, where the usage of MSVC is expected, this directive is not translated to anything, as a module definition (`.def`) file is provided to the linker instead. Additionally, the file includes an `#error` directive to prevent compilation on architectures other than x86-64.

3.9.1 FLOCDLL_Initialize

Initializes a new context. Upon successful initialization, it provides the caller with a handle to the new context via an output parameter (a pointer to a `FLOC_HANDLE` data type). Unlike other functions, this one does not expect a valid `FLOC_HANDLE` to be passed. Initialization fails under the following conditions: if the process cannot acquire the required privileges from the OS, if the global array of contexts is full, or due to an internal error such as memory allocation or vector initialization failure.

3.9.2 FLOCDLL_Uninitialize

This function disables all active trackers (by invoking `FLOCDLL_TrackerAllDisable`) and attempts to halt the debug loop (via `FLOCDLL_DebugLoopStop`) if it is currently running and not foreign. If no errors occur, it proceeds to free all allocated memory and clears the context from the global array.

3.9.3 FLOCDLL_TargetSet

If a target process has not already been set, and the process exists and is 64-bit, this function sets the target in the context to the provided value, which is of `PID` data type. Once set, the target cannot be changed until the context is uninitialized and reinitialized again.

3.9.4 FLOCDLL_DebugLoopStart

This function attempts to initiate a debug loop under the following conditions:

- The debug loop is not already running.
- A foreign debug loop is not utilized.
- The target process is set.
- The target process is running.
- The OS confirms that no other debugger is currently attached to the target process.

If successful, a new thread is created to initiate a debug loop, responsible for handling all debug events, including breakpoint exceptions triggered by breakpoint trackers.

3.9.5 FLOCDLL_DebugLoopStop

This function attempts to halt the debug loop if it is currently active and not foreign. It proceeds only when all active breakpoint trackers are successfully disabled, as they could otherwise trigger unhandled exceptions, leading to a program crash. It signals the debug thread to cease operation by modifying a value in the context, accessible to the thread. However, the thread becomes aware of this change only after returning from a `Target_WaitForBreakpoint` call, which occurs after the OS dispatches a debug event to the debugger. If the wait for the thread to finish times out, a debug break is enforced in the target process, followed by another wait.

3.9.6 FLOCDLL_DebugLoopOverride

This function serves to notify the context when a foreign (caller-owned) debug loop has been initiated or terminated. It verifies the caller's assertion with the assistance of a `Target_IsDebuggerAttached` call. Table 3.9 outlines all scenarios handled by the function, with only the first two specific situations considered valid.

Table 3.9 Overriding the debug loop

Present	Foreign	Running	Override	Success	Foreign debugger is:
✓	×	×	✓	✓	Confirmed to be activated.
×	✓	✓	×	✓	Confirmed to be deactivated.
×	∗	∗	✓	×	Claimed to be turned on but none was found.
✓	×	✓	✓	×	Claimed to be turned on but a native loop is running.
✓	✓	✓	×	×	Claimed to be turned off but a debugger is still attached.

Present is a boolean value indicating whether any debugger is attached to the target process according to the OS. *Foreign* represents the current value stored in the context, denoting whether a foreign debug loop is being utilized or not (`bForeignDebugLoop`). *Running* is another context value indicating the current status (running or stopped) of the loop, regardless of whether it is foreign or not (`bDbgLoopRunning`). *Override* is the parameter passed to the function (`BOOL` data type), asserting the current status of a foreign debug loop (either running or stopped). The symbol $*$ serves as a wildcard. Any case not explicitly mentioned in the table is handled generically, returning a `FLOC_STATUS_ILLOGICAL_OVERRIDE` value. As the tool lacks control over the caller's debug loop, it does not verify the presence of any remaining active breakpoints in the target process. The responsibility for removing them before halting the loop lies with the caller.

3.9.7 FLOCDLL_CallExceptionBreakpointHandler

When utilizing a foreign debug loop, this function should be invoked each time the loop encounters a breakpoint exception. Its purpose is to enable the tool to appropriately handle triggered breakpoint trackers. First, it verifies that a foreign debug loop is presently active within the context. If confirmed, it redirects execution to the same handler utilized by the tool's native debug loop and removes the breakpoint if its address corresponds to a tracker. The caller is responsible for supplying all necessary parameters: the process identifier (`PID`), the thread identifier (`TID`), and the address where the exception occurred (`ADDRESS`).

3.9.8 FLOCDLL_TrackerAddBreakpoint

This function attempts to establish a new software breakpoint tracker for the specified address (parameter of `ADDRESS` data type). It fails if there is already an existing tracker associated with this address, if a handle to the target process cannot be acquired, if the byte at the address cannot be read from the target process, or due to an internal error (such as a failure to allocate memory or an unsuccessful vector pushback operation). After invoking this function, the breakpoint remains in an inactive state.

3.9.9 FLOCDLL_TrackerAddHook

This function attempts to create a new inline hook tracker for the specified address (parameter of `ADDRESS` data type). It fails if there is already an existing tracker associated with this address, if a handle to the target process cannot be acquired, if the creation of the hook fails (as indicated by a `Hook_Create` call), or due to an unsuccessful vector pushback operation. Additionally, it requires the provision of the function length via a `U32` parameter. Upon calling this function, the hook remains inactive.

3.9.10 FLOCDLL_TrackerRemove

Attempts to remove a tracker specified by its address (parameter of `ADDRESS` data type). If applicable, it removes the associated breakpoint from the target process. However, hooks, even if enabled, are disregarded, as the hook will safely remove its own jump code eventually. The function fails if the tracker is not found. Upon removal, the `eType` field of the tracker is set to `TRACKER_TYPE_DELETED`, ensuring it will be ignored in all `FLOCDLL` functions, while other fields are zeroed out.

3.9.11 FLOCDLL_TrackerEnable

This function attempts to enable a tracker specified by its address (parameter of `ADDRESS` data type). For a software breakpoint, this involves writing the `INT3` instruction into the target process memory. For hooks, the beginning of the function is overwritten with the pre-prepared jump code from a previous `FLOCDLL_TrackerAddHook` call. If successful, it sets the field `bEnabled` of the tracker to `TRUE`. However, it fails if a handle to the process cannot be acquired, if the tracker is a breakpoint and the debug loop is not running, or if the tracker is not found. If the tracker is already enabled, `FLOC_STATUS_SUCCESS` is immediately returned.

3.9.12 FLOCDLL_TrackerDisable

This function attempts to disable a tracker specified by its address (parameter of `ADDRESS` data type). For a software breakpoint, this entails restoring the original byte at the function address, which was overwritten with an `INT3` instruction beforehand. Hooks are left within the target process, even if currently enabled, as the jump code will eventually be automatically removed by the hook code. If successful, it sets the field `bEnabled` of the tracker to `FALSE`. However, it fails if a handle to the process cannot be acquired or if the tracker is not found. If the tracker is already disabled, `FLOC_STATUS_SUCCESS` is immediately returned.

3.9.13 FLOCDLL_TrackerAllGet

This function allows the caller to access the vector of trackers (`vecTrackers`), providing awareness of their status. Instead of copying the vector to a caller-owned destination, it provides the address of the vector in memory via an output parameter (implemented as a double pointer to a `VECTOR const` data type).

3.9.14 FLOCDLL_TrackerAllReset

This function sets the `bHit` field of all trackers and the context value `bIsPendingReset` to `FALSE`. By doing so, it enables the caller to start a new step without filtering out any trackers using `FLOCDLL_StepFilterOutExecuted` or `FLOCDLL_StepFilterOutNotExecuted`. It fails if a step is currently active.

3.9.15 FLOCDLL_TrackerAllEnable

This function attempts to enable all trackers. It fails if a handle to the target process cannot be acquired. It returns `FLOC_STATUS_SUCCESS` even if some trackers fail to be enabled, except when attempting to enable a breakpoint tracker while a debug loop is not running. In this case, the function still attempts to enable all hook trackers, but it returns `FLOC_STATUS_ENABLING_BREAKPOINT_WITHOUT_DEBUGGING`.

3.9.16 FLOCDLL_TrackerAllDisable

This function attempts to disable all trackers. It fails if a handle to the target process cannot be acquired. It returns `FLOC_STATUS_SUCCESS` even if some trackers fail to be disabled.

3.9.17 FLOCDLL_StepBegin

Initiates a new step by setting the context value `bIsStepActive` to `TRUE`. Fails if a step is already active or if the target process has terminated. If `bIsPendingReset` is `TRUE` (indicating that trackers were not filtered out or reset), the function manually calls `FLOCDLL_TrackerAllReset`. If the reset operation succeeds, `bIsPendingReset` is set to `FALSE`; otherwise, `FLOC_STATUS_TRACKER_RESET_FAIL` is returned.

3.9.18 FLOCDLL_StepEnd

Concludes an active step by setting the context values `bIsStepActive` to `FALSE` and `bIsPendingReset` to `TRUE`. Fails if the step is already stopped, the target process has terminated, or a handle to the target process cannot be acquired. Updates the `bHit` value of all hook trackers by reading the byte in the hook code where the value resides. The `Hook_IsHit` function, which is responsible for this task, also sets the tracker value `bEnabled` to `FALSE` if the function was executed. If `bHit` is true, it indicates that the hook removed itself as part of the hook code. This mechanism differs from breakpoints, which automatically update their `bHit` and `bEnabled` values in the breakpoint handler.

3.9.19 FLOCDLL_StepFilterOutExecuted

Removes all trackers that were triggered (executed) during the previous step. Resets the remaining trackers (`bHit` is set to `FALSE`) in preparation for the next step. Sets the context value `bIsPendingReset` to `FALSE`. Fails if a step is currently active (`FLOC_STATUS_STEP_ACTIVE` is returned).

3.9.20 FLOCDLL_StepFilterOutNotExecuted

Removes all trackers that were enabled but not triggered (not executed) during the previous step. Resets the remaining trackers (`bHit` is set to `FALSE`) in preparation for the next step. Sets the context value `bIsPendingReset` to `FALSE`. Fails if a step is currently active (`FLOC_STATUS_STEP_ACTIVE` is returned).

3.10 File Structure

Table 3.10 provides an overview of all the source files comprising the DLL and their respective purposes. Disregarding file extensions (`.c` and `.h`), the C header and implementation files are merged for simplicity. The "`.h` only" field in the table indicates when only the header file is present. Not included are `flocdll.def`, which contains the names of exported functions for the linker, as well as the Visual Studio project and solution files.

Table 3.10 Summary of source code files

File	`.h` only	Purpose
floc	×	`FLOC_CTX` and helper functions for flocdll.
flocdll	×	x86-64 arch. compilation guard and all exported functions.
hook	×	Creation and management of inline hooks.
os	×	All OS-specific functionality (Process, Memory, Target, Thread).
pool	×	Location or creation of a (preferably near) pool for hooks.
status	✓	`FLOC_STATUS` data type and all status code definitions.
tracker	✓	`TRACKER_TYPE` enum, `TRACKER` and `BREAKPOINT` data types.
types	✓	Custom integer data types, definitions of `TRUE`, `FALSE`, and `NULL`.
vector	×	`VECTOR` data type implementation.

3.11 Portability and compilation

One of the primary objectives of this thesis is to ensure the tool's adaptability by existing reverse engineering tools and frameworks and achieving maximum code portability. To accomplish this, the tool has been designed with minimal dependencies. With the exception of `os.c`, which includes the `Windows.h` header file, the tool relies on absolutely no external libraries, not even the C runtime. In total, the DLL imports three functions from the `ADVAPI32` library, and 28 functions from the `KERNEL32` library.

This independence is achieved through the `/NODEFAULTLIB` linker switch. This change, however, leads to the linker not finding the `_DllMainCRTStartup` symbol, the expected entry point of the DLL. To address this, a custom entry point named `DllMain` is specified using the `/ENTRY:"DllMain"` switch. This function mirrors the example user-defined entry function from Microsoft's documentation and is invoked on DLL load and unload, as well as when a thread is created or terminated in the process [31]. The only function of this custom entry point is to disable these unnecessary thread calls, as illustrated in Code listing 3.11, which is a potential optimization strategy [32].

Code listing 3.11 Custom entry point function

```
BOOL WINAPI DllMain(HANDLE const hHandle,
                    DWORD const dwReason,
                    LPVOID const lpReserved)
{
    (void)lpReserved;
    if (DLL_PROCESS_ATTACH == dwReason)
    {
        DisableThreadLibraryCalls((HMODULE)hHandle);
    }
    return TRUE;
}
```

The code is intended to be compilable with an ISO C99[4] conforming compiler. However, this was not directly tested as the project is compiled using what Microsoft terms the "MSVC legacy" standard. This mode implements ANSI C89 but includes certain Microsoft extensions, some of which align with ISO C99 standards. However, it does not fully support C99 either, making strict conformance to C99 impossible [33]. In any case, C11 and C17 extensions are not necessary for compilation. The code is compilable by both C and C++ compilers, facilitated by the `#ifdef __cplusplus` directive and `extern "C"` declarations where necessary.

The code is not compatible with ISO C89 due to aspects such as declaration and statement mixing, as well as control variable declaration within for loop statements. While these issues could be addressed through rewriting, it would come at the expense of const correctness and scope limiting. However, the more critical factor is its dependency on the `(unsigned) long long int` data type, which was introduced in ISO C99 [34].

3.12 Optimization

The compiler settings are configured to prioritize maximum optimization (`/O2` switch) and speed over code size (`/Ot`). Inline function expansion is set to "Any Suitable" (`/Ob2`), which is the default for `/O2`. Despite these settings, the resulting file size is a modest 18 kB. However, the most aggressive inlining switch (`/Ob3`) cannot be utilized because it leads to the compiler emitting `memcpy` calls despite the unavailability of the C runtime.

The compiler inlines nearly all non-exported functions anyways, except for certain cases such as the debug loop and breakpoint handler code, which are executed from a different thread and the former is passed around with a function pointer. Functions responsible for installing hook code, a segment of the pool allocating code, and the `Vector_PushBackCopy` function are also excluded from inline expansion. As expected, the compiler does not employ Advanced vector extensions (AVX) instructions in the generated code, so their generation is disabled.

The only case of source code micro-optimization is observed in the `FLOC_ContextGet` function, which is invoked within every exported function requiring a `FLOC_HANDLE`. It ensures that the handle corresponds to a valid `FLOC_CTX` pointer stored in the global array of contexts (`gContexts`). Since it is anticipated that multi-instance use of the tool

[4]An informal name for the the ISO/IEC 9899:1999 C programming language standard.

would be rare, the context is expected to be located in the first element of the array (`gContexts[0]`). Additional compiler-specific or standard-specific keywords would be required to hint the compiler about this behavior, which led to a manual rewriting of the function to explicitly achieve the desired assembly, as depicted in Code listing 3.12. This approach yielded the fastest code path for the expected scenario when compared to several alternative variants. However, it still results in a redundant `NULL` comparison (corresponding to the validation of `FLOC_ContextGet`'s return value in the caller function), as the compiler fails to optimize it away. Although manually inlining the function to facilitate immediate return of `FLOC_STATUS_INVALID_HANDLE` from the caller upon exiting the loop was considered, it was deemed too extreme.

Code listing 3.12 Optimized `FLOC_ContextGet` function

```c
FLOC_CTX* FLOC_ContextGet(FLOC_HANDLE const hHandle)
{
    if (gContexts[0] == (FLOC_CTX*)hHandle)
    {
        return (FLOC_CTX*)hHandle;
    }
    for (U32 i = 1; i < MAX_CONTEXTS_COUNT; i++)
    {
        if (gContexts[i] == (FLOC_CTX*)hHandle)
        {
            return (FLOC_CTX*)hHandle;
        }
    }
    return NULL;
}
```

3.13 Interface

For a program to effectively utilize the functions provided by the DLL, it must define all fields listed in Table 3.11 to ensure compatibility. Understanding all the different `FLOC_STATUS` codes is not mandatory. Interpreting any non-zero value as a failure is sufficient. While not all member variables of the `TRACKER` and `VECTOR` structures need to be known, it is expected that the structure data types will be defined in their entirety. Afterwards, all the function declarations can be adapted from the file `flocdll.h`.

Additionally, the program must be able to accurately interpret the vector of trackers. This requires either the `TRACKER`'s binary size or the vector's `uElemSize` field to be defined, and a function to read each element of the vector must be implemented. This function can adhere to the original C code as depicted in Code listing 3.13, which includes boundary checking. Alternatively, it can use the following equation, assuming that the first element (index 0) is represented by $N = 1$:

$$\text{Address of } N\text{th element} = \texttt{pData} + [(N-1) \cdot \texttt{uElemSize}]$$

Table 3.11 FLOC.dll compatibility requirements

Type	FLOC.dll name	Equivalent
Data type	`FLOC_HANDLE`	64-bit unsigned integer
Data type	`ADDRESS`	64-bit unsigned integer
Data type	`U32`	32-bit unsigned integer
Data type	`PID`	32-bit unsigned integer
Data type	`TID`	32-bit unsigned integer
Data type	`TRACKER_TYPE`	32-bit unsigned integer
Data type	`FLOC_STATUS`	16-bit unsigned integer
Data type	`BOOL`	32-bit signed integer
Constant	`TRACKER_TYPE_DELETED`	0
Constant	`TRACKER_TYPE_BREAKPOINT_SW`	1
Constant	`TRACKER_TYPE_HOOK_INLINE`	2
Constant	`FLOC_STATUS_SUCCESS`	0
Struct. size	`sizeof(TRACKER)`	56 bytes
Struct. member	`TRACKER.aAddress`	Offset: 0, type: `ADDRESS`
Struct. member	`TRACKER.eType`	Offset: 8, type: `TRACKER_TYPE`
Struct. member	`TRACKER.bEnabled`	Offset: 12, type: `BOOL`
Struct. member	`TRACKER.bHit`	Offset: 16, type: `BOOL`
Struct. member	`VECTOR.pData`	Offset: 0, type: 64-bit pointer
Struct. member	`VECTOR.uElemCount`	Offset: 8, type: `U32`
Struct. member	`VECTOR.uElemSize`	Offset: 12, type: `U32`

Code listing 3.13 `Vector_AddressOf` function

```
void* Vector_AddressOf(VECTOR const * pVec, U32 const uIndex)
{
    BYTE* const pRes = ((BYTE*)pVec->pData
                     + ((U64)uIndex * pVec->uElemSize));
    BYTE const * const pBegin = (BYTE*)pVec->pData;
    BYTE const * const pEnd =
        pBegin + ((U64)pVec->uElemCapacity * pVec->uElemSize);
    BOOL const valid = pRes >= pBegin && pRes < pEnd;
    return valid ? pRes : NULL;
}
```

Chapter 4

Implementation – GUI

This chapter introduces an executable program featuring a graphical interface to control the `FLOC.dll` module. Named `Function Locator.exe`, it is accompanied by a custom icon shown in Figure 4.1. The GUI executable aims to offer a user-friendly approach to utilizing the developed tool and achieving the objectives outlined in the thesis. Moreover, it is designed to facilitate the utilization of all features provided by the DLL. Developed using Visual Studio 2022 and written in the C# programming language, it leverages the Windows Forms graphical class library included as part of Microsoft's .NET Framework. The project file `FLOC_GUI.csproj` is included in the same Visual Studio Solution file as the DLL, `FLOC.sln`.

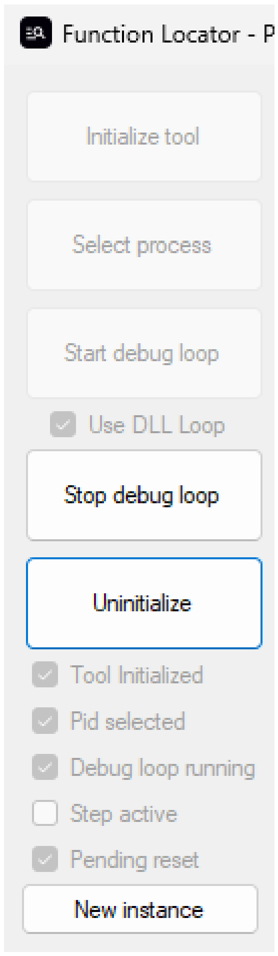
The executable requires the `FLOC.dll` file to be present in the folder where the tool is being executed from, and the .NET framework to be installed on the system. Rather than focusing on the source code, which is relatively straightforward and does not adhere to specific code conventions, this chapter aims to present the practical design and usage of the GUI.

Figure 4.1 Function Locator icon

4.1 Main form

This section focuses on specific components of the main window visible to the user. For a comprehensive overview of the entire graphical user interface, please refer to Appendix A for a screenshot. While the DLL itself safeguards against illegal operations, such as attempting to initiate a step without selecting a target, the GUI incorporates a simplified version of the context to intelligently disable buttons when they would lead to an illegal action. It is worth noting that encountering a failure status from the DLL is quite uncommon and typically indicates an internal error like target termination, rather than any fault on the user's part.

4.1.1 Left-hand side

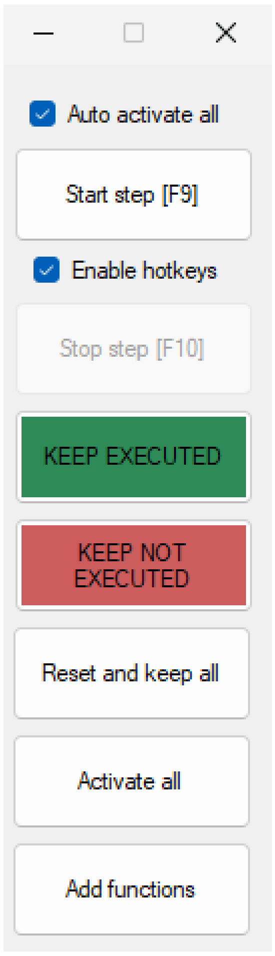


■ **Figure 4.2** Function Locator GUI, left-hand side

On the left-hand side of the GUI, as depicted in Figure 4.2, are the primary controls for managing the context. These controls include buttons for initializing the context, selecting the target process, initiating and halting the debug loop, and uninitializing the context. Beneath these buttons are greyed-out (unclickable) checkboxes serving as status indicators for the current context.

If the “Use DLL loop” checkbox is unchecked before commencing the debug loop, the GUI’s own (foreign) debugger is activated and utilized. Once a debug loop is started, regardless of type, this checkbox becomes disabled. Additionally, the button at the bottom allows for creating a new instance of the main form, facilitating multi-instancing. Behind the scenes, there is an invisible “Owner form” that manages all instances of the main form, ensuring that closing the first instance does not terminate all subsequently created instances derived from it.

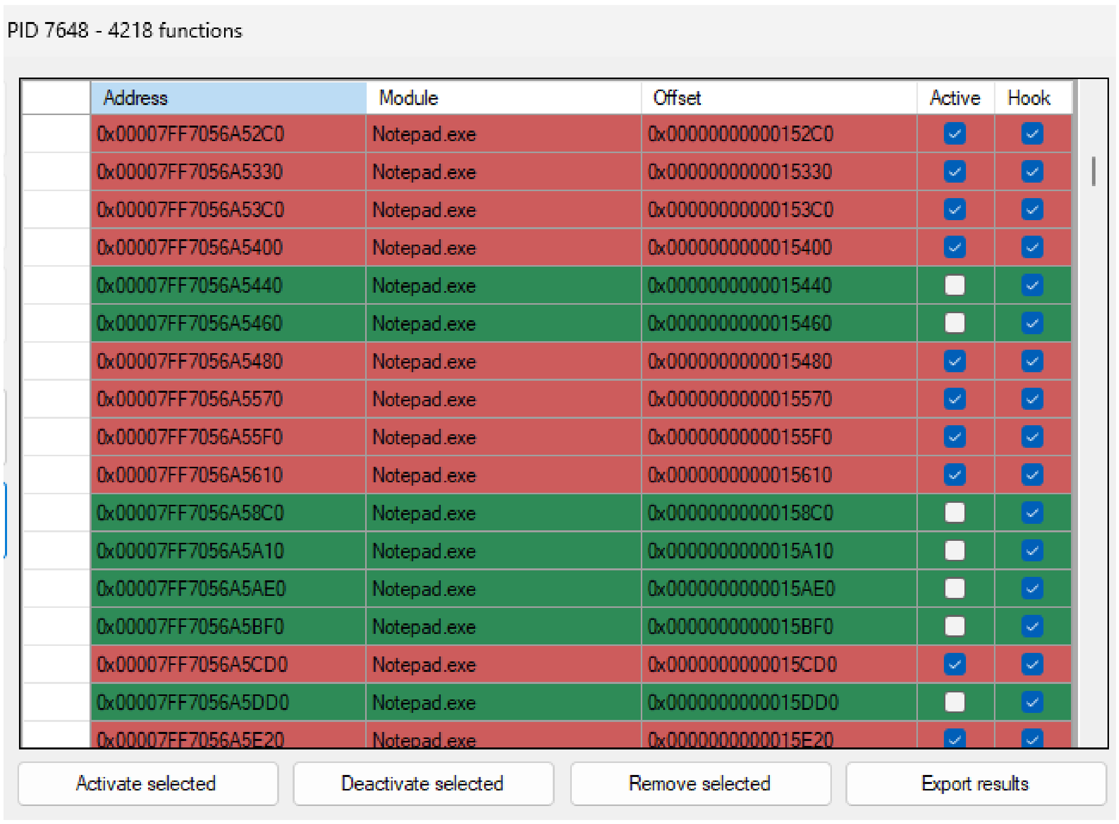
4.1.2 Right-hand side



■ **Figure 4.3** Function Locator GUI, right-hand side

On the right-hand side, depicted in Figure 4.3, users wield controls to manage the analysis process. They can add new functions (trackers), commence or conclude a step, filter trackers based on the last step results, or reset their hit status and ignore the previous step. Users can activate all trackers with the “Activate all” button, or opt for automatic activation via the “Auto activate all” checkbox. As a reminder, trackers are in a disabled state after being triggered, requiring reactivation for the next step. For added convenience, users can initiate and conclude steps using keyboard shortcuts (F9 to start and F10 to end), particularly useful when analyzing full-screen applications. In this situation, the GUI provides auditory feedback, signaling the initiation or termination of a step. The filtering buttons are color-coded; green signifies the retention of executed functions and removal of unexecuted ones, while red indicates the opposite, serving as a visual aid to prevent the user from selecting the wrong option.

4.1.3 Middle section



PID 7648 - 4218 functions

	Address	Module	Offset	Active	Hook
	0x00007FF7056A52C0	Notepad.exe	0x00000000000152C0	✓	✓
	0x00007FF7056A5330	Notepad.exe	0x0000000000015330	✓	✓
	0x00007FF7056A53C0	Notepad.exe	0x00000000000153C0	✓	✓
	0x00007FF7056A5400	Notepad.exe	0x0000000000015400	✓	✓
	0x00007FF7056A5440	Notepad.exe	0x0000000000015440		✓
	0x00007FF7056A5460	Notepad.exe	0x0000000000015460		✓
	0x00007FF7056A5480	Notepad.exe	0x0000000000015480	✓	✓
	0x00007FF7056A5570	Notepad.exe	0x0000000000015570	✓	✓
	0x00007FF7056A55F0	Notepad.exe	0x00000000000155F0	✓	✓
	0x00007FF7056A5610	Notepad.exe	0x0000000000015610	✓	✓
	0x00007FF7056A58C0	Notepad.exe	0x00000000000158C0		✓
	0x00007FF7056A5A10	Notepad.exe	0x0000000000015A10		✓
	0x00007FF7056A5AE0	Notepad.exe	0x0000000000015AE0		✓
	0x00007FF7056A5BF0	Notepad.exe	0x0000000000015BF0		✓
	0x00007FF7056A5CD0	Notepad.exe	0x0000000000015CD0	✓	✓
	0x00007FF7056A5DD0	Notepad.exe	0x0000000000015DD0		✓
	0x00007FF7056A5E20	Notepad.exe	0x0000000000015E20	✓	✓

■ **Figure 4.4** Function Locator GUI, middle section

The central section features a scrollable list of all remaining[1] functions. Each line corresponds to a single tracker or function, displaying its address, current activation status (under the "Active" column), and whether it represents a hook or a breakpoint. The GUI enhances this information by adding details about the module to which the function belongs and the offset from the module's base address, information that the DLL does not possess. The GUI achieves this through its own module enumeration function. The color of each row provides crucial insight – following a step's conclusion, rows are colorized based on the `bHit` field of the tracker: green if executed and red

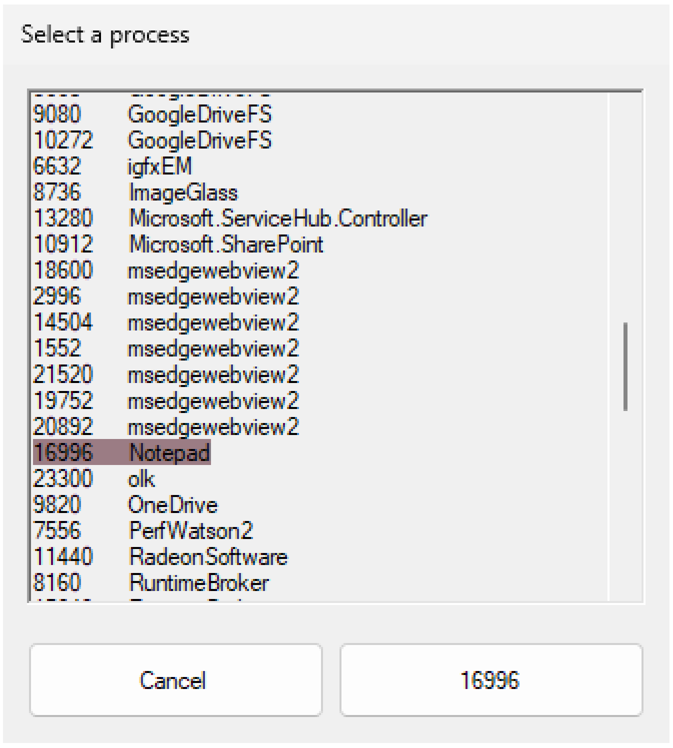
[1] "Remaining" are those trackers not marked with the type `TRACKER_TYPE_DELETED` due to being filtered out or explicitly removed.

if not. This color scheme corresponds with the color-coded buttons, reinforcing their functionality – clicking the green button retains the green (executed) functions while removing the red (non-executed) ones.

Users can select one or more rows using familiar methods such as clicking, dragging, or using the keyboard, akin to popular office applications. Actions on the selected trackers are then executed via the buttons at the bottom of the form, leveraging the one-by-one functions of the DLL in a loop. The “Export results” button serves as a standout feature, enabling users to copy and paste the remaining function addresses in a format reminiscent of the one offered by the IDA script in section 2.4, once they are content with the list’s size. The window’s title follows the format “Function Locator - PID X - Y functions” indicating to users the number of remaining unfiltered functions.

4.2 “Select a process” form

When the user clicks the “Select process” button, a new form emerges, presenting a list of all 64-bit processes currently running on the system, as illustrated in Figure 4.5. This list displays the PID value alongside the corresponding executable names, enabling users to select their desired target process.

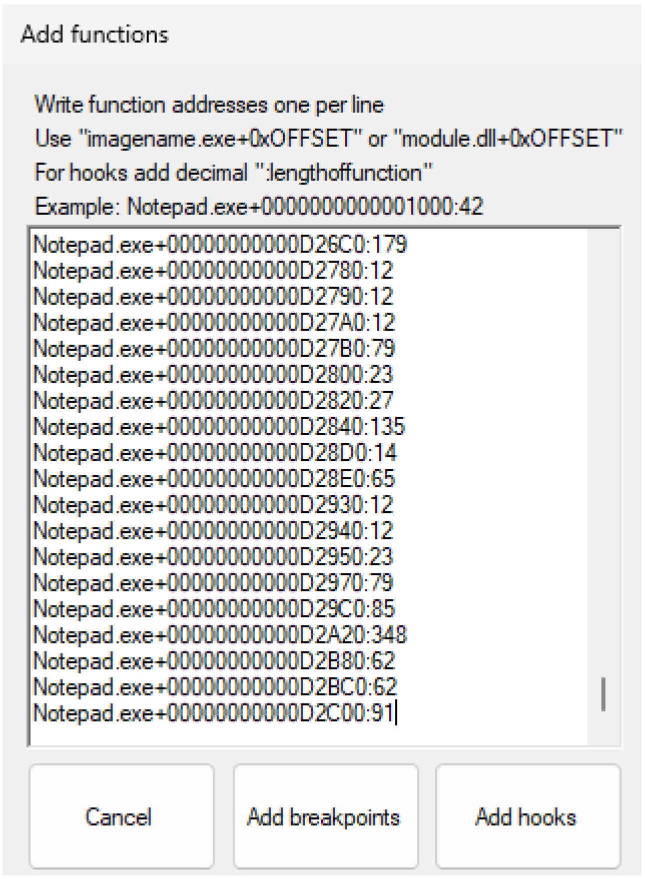


9080	GoogleDriveFS
10272	GoogleDriveFS
6632	igfxEM
8736	ImageGlass
13280	Microsoft.ServiceHub.Controller
10912	Microsoft.SharePoint
18600	msedgewebview2
2996	msedgewebview2
14504	msedgewebview2
1552	msedgewebview2
21520	msedgewebview2
19752	msedgewebview2
20892	msedgewebview2
16996	Notepad
23300	olk
9820	OneDrive
7556	PerfWatson2
11440	RadeonSoftware
8160	RuntimeBroker

■ **Figure 4.5** Function Locator GUI, “Select a process” form

4.3 “Add functions” form

Upon clicking the “Add functions” button, users are presented with a scrollable text box, as depicted in Figure 4.6. Here, they are prompted to input or paste function addresses following the same format generated by the IDA script, although the inclusion of function length is optional for breakpoints. Within this interface, users have to make the choice whether the entered functions should be added as breakpoint or hook trackers.



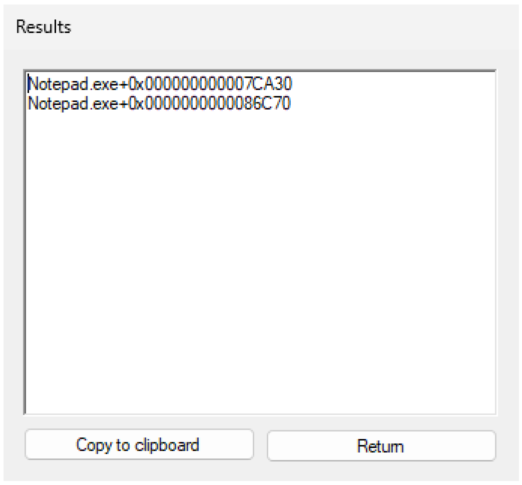
Write function addresses one per line
Use "imagename.exe+0xOFFSET" or "module.dll+0xOFFSET"
For hooks add decimal ":lengthoffunction"
Example: Notepad.exe+0000000000001000:42

```
Notepad.exe+00000000000D26C0:179
Notepad.exe+00000000000D2780:12
Notepad.exe+00000000000D2790:12
Notepad.exe+00000000000D27A0:12
Notepad.exe+00000000000D27B0:79
Notepad.exe+00000000000D2800:23
Notepad.exe+00000000000D2820:27
Notepad.exe+00000000000D2840:135
Notepad.exe+00000000000D28D0:14
Notepad.exe+00000000000D28E0:65
Notepad.exe+00000000000D2930:12
Notepad.exe+00000000000D2940:12
Notepad.exe+00000000000D2950:23
Notepad.exe+00000000000D2970:79
Notepad.exe+00000000000D29C0:85
Notepad.exe+00000000000D2A20:348
Notepad.exe+00000000000D2B80:62
Notepad.exe+00000000000D2BC0:62
Notepad.exe+00000000000D2C00:91
```

■ **Figure 4.6** Function Locator GUI, “Add functions” form

4.4 “Results” form

Upon reaching a satisfactory selection of function addresses, users can click the “Export results” button to access a small form, as shown in Figure 4.7. This form features a text box populated with the remaining functions, formatted identically to the output generated by the IDA script, albeit without the function lengths. For added convenience, a button labeled “Copy to clipboard” transfers the contents of the text box to the user’s Windows clipboard.



```
Notepad.exe+0x000000000007CA30
Notepad.exe+0x0000000000086C70
```

Figure 4.7 Function Locator GUI, "Results" form

4.5 FLOCDLL functions

Table 4.1 GUI buttons and their corresponding `FLOC.dll` functions

Form	Button name	FLOC.dll function
Main	Initialize tool	`FLOCDLL_Initialize`
Main	Uninitialize	`FLOCDLL_Uninitialize`
Main	Start debug loop	`FLOCDLL_DebugLoopStart`
Main	Stop debug loop	`FLOCDLL_DebugLoopStop`
Main	Start / Stop debug loop	`FLOCDLL_DebugLoopOverride`
Main	Activate all / Start step [F9]	`FLOCDLL_TrackerAllEnable`
Main	Start step [F9]	`FLOCDLL_StepBegin`
Main	Stop step [F10]	`FLOCDLL_StepEnd`
Main	Keep executed	`[...]StepFilterOutExecuted`
Main	Keep not executed	`[...]StepFilterOutNotExecuted`
Main	Reset and keep all	`FLOCDLL_TrackerAllReset`
Main	Activate selected	`FLOCDLL_TrackerEnable`
Main	Deactivate selected	`FLOCDLL_TrackerDisable`
Main	Remove selected	`FLOCDLL_TrackerRemove`
Add functions	Add breakpoints	`FLOCDLL_TrackerAddBreakpoint`
Add functions	Add hooks	`FLOCDLL_TrackerAddHook`
Select a process	Select	`FLOCDLL_TargetSet`

Table 4.1 provides an overview of which buttons correspond to specific functions exported by `FLOC.dll`. The GUI integrates all functions except `FLOCDLL_TrackerAllDisable`, as the DLL automatically invokes it during uninitialization. `FLOCDLL_CallExceptionBreakpointHandler` is invoked by the foreign debugger when in use. The GUI's debug loop essentially mirrors the functionality of the `Target_WaitForBreakpoint` function, rewritten in C# with the breakpoint handler replaced by the DLL's export. Additionally, the GUI invokes `FLOCDLL_TrackerAllGet` following any context alteration/

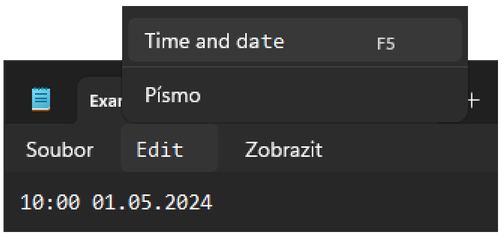
Chapter 5

Demonstration

This chapter provides a comprehensive demonstration of the usage and effectiveness of the tool in real-world scenarios, highlighting its capabilities and addressing some of its limitations. Through a series of select examples, it explores how the tool performs in both basic and complex situations. These demonstrations aim to give a clear understanding of how to leverage the tool, providing tips and strategies for optimizing its usage.

5.1 Notepad example

Windows Notepad, a simple text editor bundled with the operating system, was used during development for testing. Although it is a basic application with no significant performance demands, it sufficiently tests all features of the tool. This example provided valuable insights on how to use the tool most effectively. The target was the code responsible for inserting the current time and date into the text. This feature was chosen for two reasons: it is easy to trigger or avoid, and it can be activated either by clicking a button or using a keyboard shortcut. Additionally, searching for this code is not entirely unrealistic since Notepad does not offer built-in options to alter this behavior. The user interface and the function's effect are shown in Figure 5.1[1]



10:00 01.05.2024

Figure 5.1 Notepad "Time and date" function

The IDA script exported over 4,200 different function addresses. All of them could be added as breakpoint trackers, and only nine failed to be added as hooks due to their lengths. During an example run, after starting the first step, clicking the "Time and date" button, and immediately stopping it, 141 functions remained after filtering out

[1]The image is modified to provide an English translation. Does not represent actual software.

the not executed ones. This highlights an important lesson about the logic employed to make the analysis process as efficient as possible. When clicking the Notepad's button, many other functions are inadvertently triggered. The following list provides examples of events likely causing a function call in the first step, even though it was not explicitly desired:

- Window focus change.
- Mouse click.
- Opening the "Edit" menu.
- Mouse movement and button highlight.
- Adding and pasting new text.
- Moving the cursor.
- Character count change.

Since the target code should be specific and not dependent on whether the button or keyboard shortcut was used, a logical second step could involve pressing F5 and retaining the executed functions, ideally using Function Locator's F9 and F10 shortcuts to bypass the window focus change. This reduces the remaining functions to 113. To significantly reduce this number further, the next step should involve triggering as many functions as possible, except for the one being searched for.

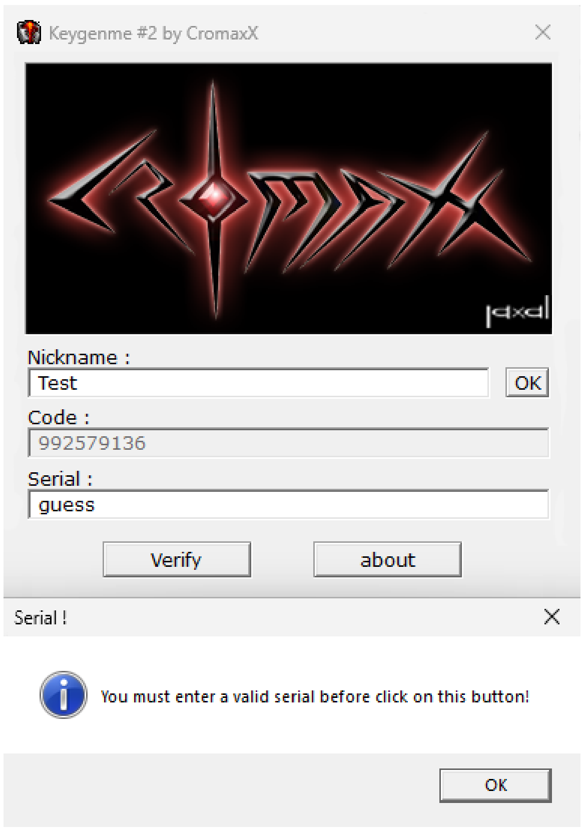
Ultimately, the "ideal" strategy was determined: only two steps are needed. First, clicking the "Time and date" button and retaining the executed functions. Then, clicking a different button, using a different keyboard shortcut, moving and minimizing the window, pasting text from the clipboard, writing, deleting, and selecting text, and retaining the not executed functions. This leaves only two functions, which is the optimal state of the analysis in this scenario. Both functions are called whenever the correct button is clicked or F5 is pressed, and otherwise, they are never triggered. A recording of Function Locator being used in this specific scenario is provided in thesis' attachments as a video example.

Further analysis reveals that only the first function contains the desired code, although the second one is invoked by the first and could therefore be considered part of the correct result. In any case, this level of localization is satisfactory, as manually analyzing only two functions is quick. In this specific version of Notepad, the functions are `Notepad.exe+0x7CA30` and `Notepad.exe+0x86C70`. Static analysis in IDA reveals that the former includes `GetLocalTime`, `GetLocaleInfoW`, `GetUserDefaultUILanguage`, `GetDateFormatW`, and `GetTimeFormatW` Windows API calls and text formatting code. The latter function was analyzed dynamically yet its purpose was not researched further as it did minimal work—performing one comparison and returning, skipping almost all remaining instructions.

The strategy of first performing a "Keep executed" step followed by the opposite proved effective in other use cases as well, provided it is feasible. Additionally, an event not mentioned here but common in many applications, especially those with infinite loops running in the background (such as video games, video players, and audio software), or those implementing some sort of heartbeat or polling, can cause functions to be invoked frequently just by being run. In such cases, it might be worthwhile to let the program idle for a bit and filter out these functions before executing the desired actions.

5.2 Simple “crackme”

A “crackme” is a file designed to test and improve reverse engineering skills by presenting various tasks, usually created by reverse engineers themselves [35]. This example aims to illustrate a scenario similar to the one described in the thesis’ introduction while also discussing some shortcomings of the tool. A crackme with a graphical interface was selected, featuring a button to click and an error message for incorrect keys, as depicted in Figure 5.2.


Keygenme #2 by CromaxX

Nickname : Test

Code : 992579136

Serial : guess

Serial !

You must enter a valid serial before click on this button!

Figure 5.2 Crackme

One initial challenge was quickly evident: the executable file was packed, and since Function Locator cannot extract function addresses directly from the target’s memory, it relies on the reverse engineer’s skills to obtain them. In this simple scenario, dumping the executable from memory allows IDA to correctly analyze the binary and provide the function addresses.

Finding the relevant function responsible for validating the key is straightforward with static analysis, and Function Locator makes this task even simpler. However, a different crackme was chosen to highlight another deficiency, which is more relevant to the specific design of the GUI rather than anti-analysis methods. This crackme only provides a command line (console) interface and allows the user only one attempt at inputting the correct password before immediately terminating. The GUI does not offer any means to automatically re-add the remaining functions into a new instance of the process. Since the functions are treated as image-relative, new addresses can be recalculated easily, but the DLL does not offer a method for efficiently "retargeting" the tool. This requires reinitializing the context before continuing with a new PID. Consequently, in situations where the reverse engineer can perform only a limited number of steps before needing to restart the application, the current design of the tool necessitates exporting intermediate results from the GUI and reinserting them to continue the analysis. While this does not critically impede functionality, it can prove to be an inconvenience. Both crackmes were obtained from [36].

5.3 Image processing software

Unlike the previous examples, this section presents a real-world reverse engineering and debugging scenario. Despite originating from the author's own needs, it serves as a perfect example of yet more complexity: a multi-module scenario within a large and complex application. Additionally, it implements the aforementioned challenge of having only one chance to trigger the event per each start of the program (further explained below). The analyzed software is Adobe's Photoshop Lightroom Classic[2], an image organization and processing tool. Excluding OS and shared libraries, the software loads a total of 75 DLL files, with the desired functionality potentially residing in any of these modules or the executable itself. The tool does not distinguish between different modules and is capable of supporting all of them simultaneously. The GUI tracks the modules and provides correct module-relative results to the reverse engineers, making multi-module analysis feasible.

The targeted functionality is the loading of "Enhance data" after pressing the "Enhance" button in the "Photo" drop-down menu, as depicted in Figure 5.3. This process occasionally causes a full system freeze of the author's computer when hardware acceleration is enabled, presenting a challenging problem to debug due to the lack of logs and the need for a forced system reset after encountering this problem. Identifying the responsible functions in the extensive code base would enable step-debugging to analyze the issue and locate potential problems. This could involve tracing or record-and-replay debugging to capture the freeze event (which, due to the system halting, would require a remote analysis approach or a tool capable of saving intermediate records to disk immediately).

It is expected that "loading the enhance data" will invoke numerous functions, with the goal being to identify the initiation of the process. Despite starting with 38,000 breakpoint trackers, the target application remained highly responsive. This large number of trackers only burdened the GUI, which is understandable given its code. This resulted in a brief period of unresponsiveness while the trackers were loaded from the DLL's vector into the middle section of the GUI. However, the underlying `FLOCDLL` operations remained very fast.

[2]`https://www.adobe.com/products/photoshop-lightroom-classic.html`

Despite the software's complexity, the initial step filtered the number of functions down to a remarkable 28. This outcome demonstrates the efficacy of this method compared to existing methodologies. Although the program required a restart to force a reload of the enhance data, the analysis process remained swift. Hook trackers were also tested to ensure they did not cause any problems, and the test was successful. The process involved starting a step with the F9 key press, clicking the "Enhance" button, and immediately stopping the step with F10. After the 28 remaining functions were exported from the GUI, the process continued with a fresh restart of Lightroom Classic and a reinitialization of the context.

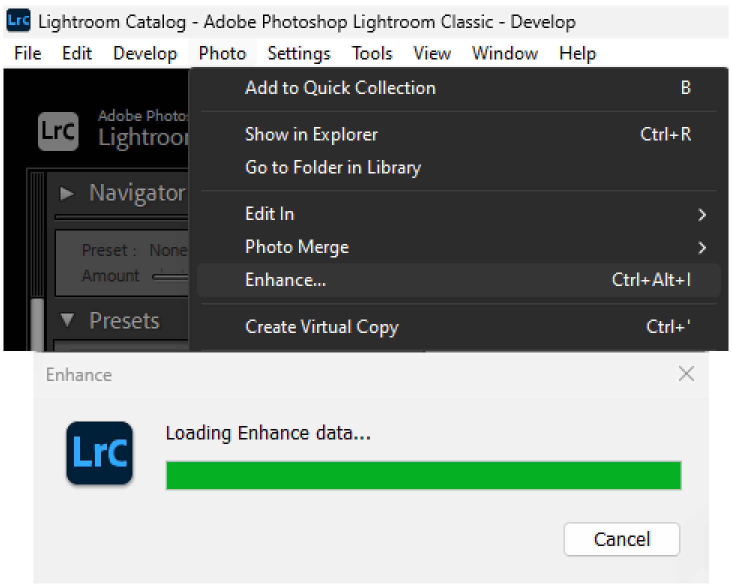

Lightroom Catalog - Adobe Photoshop Lightroom Classic - Develop
File Edit Develop Photo Settings Tools View Window Help
Add to Quick Collection B
Show in Explorer Ctrl+R
Go to Folder in Library
Edit In
Photo Merge
Enhance... Ctrl+Alt+I
Create Virtual Copy Ctrl+'
Navigator
Presets
Enhance
Loading Enhance data...
Cancel

Figure 5.3 Image processing software – Enhance button

5.4 Virtual piano

Virtual musical instruments, samplers, and synthesizers are essential in music production and live performance. These programs run on consumer operating systems and are often compiled as Virtual studio technology plugins, with file extensions `.dll` or `.vst3`, to be used inside digital audio workstations, or as standalone executables. Due to strict latency requirements, audio buffers at a 44.1 kHz sample rate can be as short as 64 samples, necessitating audio rendering every 1.5 milliseconds or faster to avoid pops, cracks, and glitches. This makes it an ideal scenario to showcase the tool's ability to identify functions executed repeatedly and rapidly.

The selected virtual instrument is Pianoteq[3], a virtual piano software. It was configured to play a MIDI file and produce audio in real-time. When using breakpoint trackers, occasional audio glitches were heard, especially as more breakpoints were triggered in a short time frame. However, the first step concluded successfully, reducing the initial 31,000 functions to about 1,600.

[3] `https://www.modartt.com/pianoteq_overview`

Soon, the challenge of not being able to filter out non-executed functions became evident. Without a “Keep not executed” filter, the process slowed significantly, reducing the remaining functions by only about 200 per step. The software simply had too many functions invoked repeatedly with each audio buffer iteration. Some functions were executed so quickly that the GUI could not reload the list in time after enabling them – the trackers were immediately reported as inactive because of the breakpoint handler being faster than the GUI. They would correctly show up green after a step was concluded, but since coloration is not applied beforehand it created the illusion that the trackers were never enabled. At 279 functions, it became clear that all these remaining trackers were triggered immediately after being enabled. Although this test did not target any specific code, reducing from 31,000 to under 300 functions provides some assistance, but the reverse engineer would still have work to do.

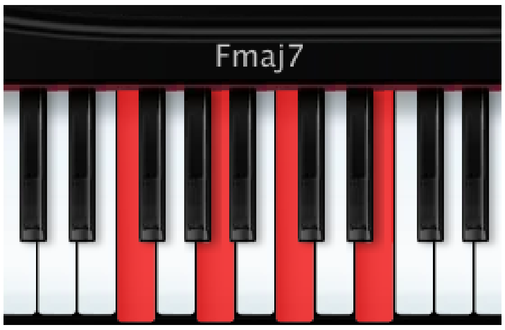


Figure 5.4 Pianoteq chord identification

A real function target was then selected so as to avoid the aforementioned problem but still add some complexity. Pianoteq’s interface can show the name of the currently played chord on the keyboard, as shown in Figure 5.4. The goal was to locate the code performing this auto-detection of chords from pressed keys. Seeing the chord name is less straightforward than pressing a button, but not playing any notes can avoid triggering the function.

The analysis utilized the strategy suggested at the end of section 5.1. Initially, the program ran by itself, and functions running constantly without any user action were filtered out. Subsequently, the usual “executed, then not executed” steps were performed. A chord was played, the name displayed, and only executed functions were kept, narrowing the search to about 1,000 functions. In the next step, single notes were played (but not chords), and only non-executed functions were kept, resulting in 43 functions. After a few more steps, only three functions remained. It is essential to remember that assumptions about a function’s behavior can lead to early mistakes in filtering. If the chord recognition function triggered periodically and without any notes being played, it would have been mistakenly filtered out.

When attempting to use hooks, the application crashed. Based on the last status of all trackers, potential culprits can be identified, or trackers that did get executed simply replaced with breakpoints in subsequent attempts until only functioning hooks remain. This approach was used, and the issue was not further investigated. The GUI does not explicitly support this kind of crash analysis. Adding such a feature could be highly beneficial.

5.5 Video game

The ultimate test aimed to gauge the tool's performance overhead under demanding conditions, using Counter-Strike 2, a popular video game released in 2023. The analysis focused solely on the game's executable `cs2.exe` and the module `client.dll`, known repositories of game-specific code. With a total of 70 thousand addresses to sift through, expectations for performance were tempered.

Initial trials with breakpoint trackers revealed significant problems. Attempting to enable all trackers simultaneously resulted in game freezes and crashes, necessitating a more cautious approach. Enabling trackers in smaller batches proved more manageable, allowing the game to remain functional albeit with occasional stutters and freezes, particularly noticeable when multiple breakpoints were triggered simultaneously.

Transitioning to inline hooks, the experience was notably smoother despite encountering a few crashes. The chosen non-persistent hook approach, emphasized in this thesis, demonstrated its value. Once triggered, trackers remained essentially invisible, their impact on performance non-existent, which was expected. Utilizing smaller batches of trackers at the beginning further mitigated stutters and other issues, with the tool facilitating such selective activation through both the GUI's tracker selection feature and the DLL's provision of one-by-one tracker enabling functions `FLOCDLL_TrackerEnable`.

Chapter 6

Detectability and future work

The tool does not attempt to conceal its manipulation of the target process. When using breakpoints, the simplest anti-analysis method the target might employ is querying the Windows API to check if a debugger is attached. This can be bypassed by hooking the API calls, although numerous and advanced anti-debugging techniques exist, as detailed in [37]. For greater stealth, hooks should be preferred, although they too leave many traces.

With inline hooks, the most conspicuous evidence is the use of Windows API functions to manipulate the target's memory. These functions might be hooked by the target, or open handles to the target could be analyzed or blocked. If the reverse engineer cannot hide API usage, they might directly invoke syscalls. Ideally, these operations should be performed from the kernel instead.

Memory modifications still present a challenge, even if done from the kernel. The target might detect pool allocations, memory protection changes, or perform integrity checks on executable sections to identify instruction changes. If directly attacking the target's defense mechanisms is impractical, advanced methods could be employed. One possibility is to use a hypervisor and leverage the CPU's Second level address translation – referred to as "Extended page tables" (EPT) on Intel and "Nested page tables" (NPT) on AMD – to split memory between read/write and execute access. This technique is sometimes called EPT hooking [38, 39].

In these extreme cases, the tool's effectiveness diminishes because bypassing detections requires creating a specialized environment where kernel access and virtualization are necessary and any real-time software running is unexpected. Consequently, the tool is not designed for analyzing protected software. Moreover, heavily virtualized code, such as that protected by VMProtect[1], renders the concept of "locating functions" obsolete. Disassemblers will not generate a sensible list of addresses, and the whole idea of functions is undermined as the target may self-modify anyways.

As a user-friendly debugging tool or a tool for reverse engineering generic software, it is highly effective, even allowing real-time analysis of resource-intensive applications like video games. However, it is unexpected that a malware analyst would find this tool very useful.

[1] `https://vmpsoft.com/`

6.1 Shortcomings

The demonstrations in chapter 5 reveal certain shortcomings of the tool in its current state, which could be addressed in future updates. These issues can mostly be resolved through improvements to the GUI. For instance, the tool currently lacks the functionality to seamlessly continue analysis after a target process is restarted by automatically recalculating the module-relative addresses. Additionally, it cannot launch the target under debugging conditions and can only attach to a running process. This limitation means that functions executed only during initialization cannot be monitored without using a third-party debugger to launch the process in a suspended state.

In section 5.4, the inline hooks caused the target to crash. The vector of trackers was subsequently analyzed to identify potential problematic addresses. Adding this functionality and additional crash support to the GUI could enhance the tool's robustness. Furthermore, both the GUI and the DLL encounter issues with processes started with administrator rights. Although running Function Locator at the same privilege level resolves this, the tools do not provide feedback indicating this requirement. Specifically, the GUI fails to display the process in the “Select a process” form, and the DLL only returns an “invalid target” status code because it cannot acquire a handle to the process.

Perhaps the most severe problem currently is the GUI's handling of a large number of trackers. The process of loading trackers from the vector into the scrollable list should be done in the background, allowing more critical operations, such as controlling the analysis, to proceed without interruption. At present, the GUI can freeze for several seconds during this loading process.

6.2 Long-term plans

In the long term, several additional features could enhance the tool's capabilities. First, bundling the DLL with a disassembler library such as Zydis[2] or Capstone[3] would eliminate the need for reverse engineers to rely on third-party tools to obtain function addresses. Additionally, adding support for Unix-based operating systems could broaden the tool's reach, as the code is already prepared for such an extension. Furthermore, adapting the tool to target 32-bit applications would require minimal changes, and the hooking process would be simplified since a 32-bit relative jump would always suffice.

Addressing the atomicity of memory writes within hook code and during the installation of jump code is another potential improvement. While the thesis raised concerns and suggested solutions, it did not address the issue further. For near-hooks, optimizing the hook code by using a single 8-byte move instruction, rather than the current 4 + 1 variant, would enhance performance while ensuring atomicity.

[2]`https://zydis.re/`
[3]`https://www.capstone-engine.org/`

Chapter 7
Conclusion

This thesis has introduced a method for efficiently locating specific functions within disassembled binary files, addressing a challenge in software reverse engineering. The method streamlines the process and provides a simplified approach to the task.

First, the thesis thoroughly explored the problem and identified the shortcomings of current solutions in the field. Subsequent chapters focused on the technical aspects of real-time code execution monitoring, highlighting software breakpoints and inline function hooking as the chosen solutions.

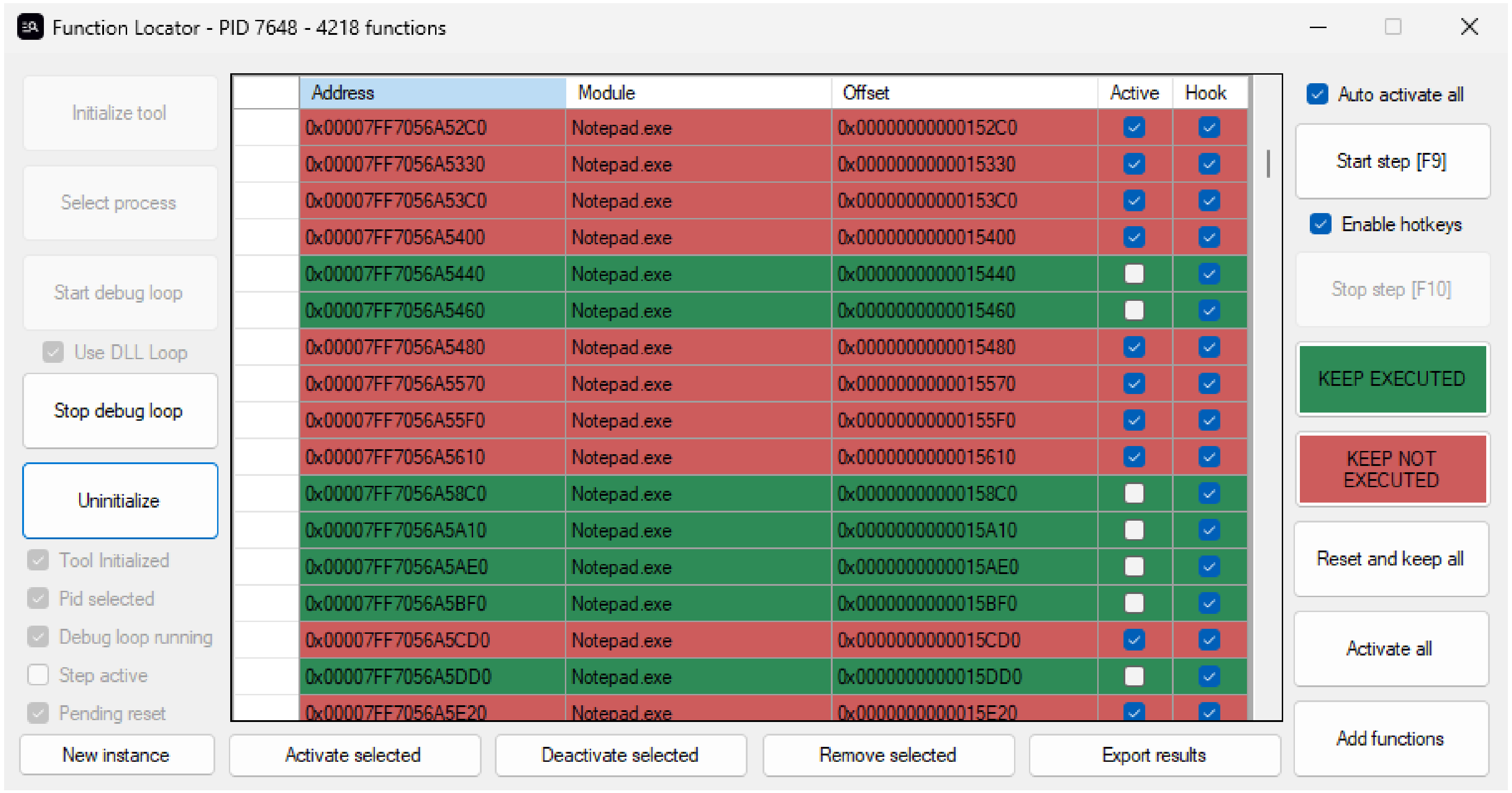
The primary aim of the thesis was to develop a tool compatible with all modern 64-bit Windows systems, avoiding the need for hypervisors or kernel access. Moreover, the tool was designed to allow seamless integration with existing reverse engineering frameworks and to be able to effectively analyze resource-intensive software like video games in real-time.

All goals of the thesis have been met. The tool was divided into two parts: a robust and portable dynamic link library written in the C programming language providing the core functionality, and a user-friendly graphical interface written in C# to harness the DLL's features. The method proved to be valuable as the tool performed beyond expectations, swiftly filtering tens of thousands of functions down to a few dozen in a matter of moments and demonstrating real-time analysis capabilities even with demanding applications such as modern video games. While the tool has shown promising results, its effectiveness varies in certain scenarios, and opportunities for future enhancements exist.

In conclusion, this thesis contributes to the field of reverse engineering by providing a solution to a specific task. The thesis, along with its associated source code and binaries, is freely available under the thesis' specified license, encouraging further development.

Appendix A

Function Locator GUI


Function Locator - PID 7648 - 4218 functions

	Address	Module	Offset	Active	Hook
	0x00007FF7056A52C0	Notepad.exe	0x00000000000152C0	✓	✓
	0x00007FF7056A5330	Notepad.exe	0x0000000000015330	✓	✓
	0x00007FF7056A53C0	Notepad.exe	0x00000000000153C0	✓	✓
	0x00007FF7056A5400	Notepad.exe	0x0000000000015400	✓	✓
	0x00007FF7056A5440	Notepad.exe	0x0000000000015440		✓
	0x00007FF7056A5460	Notepad.exe	0x0000000000015460		✓
	0x00007FF7056A5480	Notepad.exe	0x0000000000015480	✓	✓
	0x00007FF7056A5570	Notepad.exe	0x0000000000015570	✓	✓
	0x00007FF7056A55F0	Notepad.exe	0x00000000000155F0	✓	✓
	0x00007FF7056A5610	Notepad.exe	0x0000000000015610	✓	✓
	0x00007FF7056A58C0	Notepad.exe	0x00000000000158C0		✓
	0x00007FF7056A5A10	Notepad.exe	0x0000000000015A10		✓
	0x00007FF7056A5AE0	Notepad.exe	0x0000000000015AE0		✓
	0x00007FF7056A5BF0	Notepad.exe	0x0000000000015BF0		✓
	0x00007FF7056A5CD0	Notepad.exe	0x0000000000015CD0	✓	✓
	0x00007FF7056A5DD0	Notepad.exe	0x0000000000015DD0		✓
	0x00007FF7056A5E20	Notepad.exe	0x0000000000015E20	✓	✓

✓ Use DLL Loop
✓ Tool Initialized
✓ Pid selected
✓ Debug loop running
☐ Step active
✓ Pending reset
✓ Auto activate all
✓ Enable hotkeys

Bibliography

1. EILAM, Eldad. *Reversing: Secrets of Reverse Engineering.* Indianapolis, Indiana: Wiley Publishing, 2005, p. xxiv. ISBN 978-0-7645-7481-8.
2. EILAM, Eldad. *Reversing: Secrets of Reverse Engineering.* Indianapolis, Indiana: Wiley Publishing, 2005, chap. 1, pp. 4-9. ISBN 978-0-7645-7481-8.
3. QUARKSLAB. *Reverse engineering: a threat to intellectual property of innovations.* Online. In: Quarkslab. Offensive and Defensive Security Solutions. N. d. Available from: `https://www.quarkslab.com/article-reverse-engineering-threat-to-intellectual-property-innovations/`. [cited 2024-05-11].
4. CHIKOFSKY, Elliot and CROSS, James. *Reverse engineering and design recovery: a taxonomy.* In: IEEE Software. 7. Los Alamitos, California: IEEE Computer Society Press, 1990, pp. 13-17. ISSN 0740-7459. Available from: `https://doi.org/10.1109/52.43044`. [paywall]. [cited 2024-05-11].
5. STALLMAN, Richard M.; PESCH, Roland and SHEBS, Stan. *Debugging with GDB: The GNU Source-Level Debugger.* Online. Tenth edition. Boston, Massachusetts: Free Software Foundation, 2024, chap. 13. ISBN 978-0-9831592-3-0. Available from: `https://sourceware.org/gdb/current/onlinedocs/gdb.pdf`. [cited 2024-05-11].
6. SIKORSKI, Michael and HONIG, Andrew. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* San Francisco, California: no starch press, 2012, chap. 9, pp. 192-194. ISBN 978-1-59327-290-6.
7. SUCHAKRA. *Fast Tracing with GDB.* Online. In: Tuxology. 29 June 2016. Available from: `https://suchakra.wordpress.com/2016/06/29/fast-tracing-with-gdb/`. [cited 2024-05-11].
8. STALLMAN, Richard M.; PESCH, Roland and SHEBS, Stan. *Debugging with GDB: The GNU Source-Level Debugger.* Online. Tenth edition. Boston, Massachusetts: Free Software Foundation, 2024, chap. 7. ISBN 978-0-9831592-3-0. Available from: `https://sourceware.org/gdb/current/onlinedocs/gdb.pdf`. [cited 2024-05-11].
9. FREE SOFTWARE FOUNDATION. *FastTracepoints.* Online. In: GDB wiki. Last modified 18 September 2013. Available from: `https://sourceware.org/gdb/wiki/FastTracepoints`. [cited 2024-05-11].

10. HONARMAND, Nima and TORRELLAS, Josep. *Replay Debugging: Leveraging Record and Replay for Program Debugging.* Online. In: ISCA '14: Proceeding of the 41st annual international symposium on Computer architecuture. Minneapolis, Minnesota, 14-18 June 2014. IEEE Press, 2014, pp. 445-447. ISBN 978-1-4799-4394-4. Available from: `https://dl.acm.org/doi/abs/10.5555/2665671.2665737`. [paywall]. [cited 2024-05-11].

11. *rr: lightweight recording & deterministic debugging.* Website. Available from: `https://rr-project.org/`. [cited 2024-05-11].

12. MICROSOFT. *FlushInstructionCache function (processthreadsapi.h).* Online. In: Microsoft. Microsoft Learn: Build skills that open doors in your career. Last modified 22 February 2024. Available from: `https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-flushinstructioncache`. [cited 2024-05-11].

13. RUSSINOVICH, Mark; SOLOMON, David A. and IONESCU, Alex. *Windows Internals Part 1.* 6th ed. Redmont, Washington: Microsoft Press, 2012, chap. 3, pp. 123-126. ISBN 978-0-7356-4873-9.

14. QUINN, Sam. *Function Hooking for Recon and Exploitation.* Online. Musarubra US, 2022. Available from: `https://www.trellix.com/assets/docs/atr-library/tr-function-hooking-for-recon-and-exploitation.pdf`. [cited 2024-05-11].

15. HOGLUND, Greg and BUTLER, James. *Rootkits: subverting the Windows kernel.* Stoughton, Massachusetts: Pearson Education, 2005, chap. 4, pp. 74–76. ISBN 0-321-29431-9.

16. INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3A: System Programming Guide, Part 1.* Online. Intel, 2024, chap. 12. Last modified March 2024. Available from: `https://cdrdv2.intel.com/v1/dl/getContent/671190`. [cited 2024-05-11].

17. MICROSOFT. *Processor Breakpoints (ba Breakpoints).* Online. In: Microsoft. Microsoft Learn: Build skills that open doors in your career. Last modified 29 August 2023. Available from: `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/processor-breakpoints---ba-breakpoints-`. [cited 2024-05-11].

18. ECKELS, Stephen. *PolyHook 2.0: C++20, x86/x64 Hooking Libary v2.0.* Online. In: GitHub. Last modified 10 May 2024. Available from: `https://github.com/stevemk14ebr/PolyHook_2_0`. [cited 2024-05-11].

19. BUI, Hoang. *Vectored Exception Handling, Hooking Via Forced Exception.* Online. In: Medium. Medium – Where good ideas find you. 13 January 2019. Available from: `https://medium.com/@fsx30/vectored-exception-handling-hooking-via-forced-exception-f888754549c6`. [cited 2024-05-11].

20. BAKHVALOV, Denis. *Enhance performance analysis with Intel Processor Trace.* Online. In: Denis Bakhvalov. Easyperf. 23 August 2019. Available from: `https://easyperf.net/blog/2019/08/23/Intel-Processor-Trace`. [cited 2024-05-11].

21. INTEL. *Which Intel Processor Models Support Intel Processor Trace (Intel PT)?*. Online. In: Intel. Intel Support. Last modified 7 August 2021. Available from: `https://www.intel.com/content/www/us/en/support/articles/000056730/processors.html`. [cited 2024-05-11].

22. HARRIS, Laune C. and MILLER, Barton P. *Practical analysis of stripped binary code*. In: ACM SIGARCH Computer Architecture News. 2005, vol. 33, pp. 63-68. ISSN 0163-5964. Available from: `https://doi.org/10.1145/1127577.1127590`. [paywall]. [cited 2024-05-11].

23. ANDRIESSE, Dennis; CHEN, Xi; VEEN, Victor van der; SLOWINSKA, Asia and BOS, Herbert. *An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries*. In: 25th USENIX Security Symposium (USENIX Security 16). Austin, Texas: USENIX Association, 2016, pp. 583-600. ISBN 978-1-931971-32-4. Available from: `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse`. [cited 2024-05-11].

24. MICROSOFT. *ContinueDebugEvent function (debugapi.h)*. Online. In: Microsoft. Microsoft Learn: Build skills that open doors in your career. Last modified 13 October 2021. Available from: `https://learn.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-continuedebugevent`. [cited 2024-05-11].

25. MICROSOFT. *Debugging Events*. Online. In: Microsoft. Microsoft Learn: Build skills that open doors in your career. Last modified 7 January 2021. Available from: `https://learn.microsoft.com/en-us/windows/win32/debug/debugging-events`. [cited 2024-05-11].

26. MICROSOFT. *VirtualAllocEx function (memoryapi.h)*. Online. In: Microsoft. Microsoft Learn: Build skills that open doors in your career. Last modified 27 July 2022. Available from: `https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex`. [cited 2024-05-11].

27. CHEN, Raymound. *Why is address space allocation granularity 64KB?*. Online. In: Microsoft. The Old New Thing. 27 July 2022. Available from: `https://devblogs.microsoft.com/oldnewthing/20031008-00/?p=42223`. [cited 2024-05-11].

28. INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-L*. Online. Intel, 2024, chap. 3, pp. 556-564. Last modified March 2024. Available from: `https://cdrdv2.intel.com/v1/dl/getContent/671199`. [cited 2024-05-11].

29. WONG, Henry. *Microbenchmarking Return Address Branch Prediction*. Online. In: Henry Wong. Blog. 18 April 2018. Available from: `http://blog.stuffedcow.net/2018/04/ras-microbenchmarks`.

30. INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, M-U*. Online. Intel, 2024, chap. 4, pp. 63-76. Last modified March 2024. Available from: `https://cdrdv2.intel.com/v1/dl/getContent/671241`. [cited 2024-05-11].

31. MICROSOFT. *Dynamic-Link Library Entry-Point Function*. Online. In: Microsoft. Microsoft Learn: Build skills that open doors in your career. Last modified 7 January 2021. Available from: `https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-entry-point-function`. [cited 2024-05-11].

32. MICROSOFT. *DisableThreadLibraryCalls function (libloaderapi.h)*. Online. In: Microsoft. Microsoft Learn: Build skills that open doors in your career. Last modified 22 February 2024. Available from: `https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-disablethreadlibrarycalls`. [cited 2024-05-11].

33. MICROSOFT. */std (Specify Language Standard Version)*. Online. In: Microsoft. Microsoft Learn: Build skills that open doors in your career. Last modified 22 February 2024. Available from: `https://learn.microsoft.com/en-us/cpp/build/reference/std-specify-language-standard-version?view=msvc-170`. [cited 2024-05-11].

34. ISO/IEC 9899:1999. *Programming languages – C*.

35. ARNOUD, Stanislas. *FAQ*. Online. In: Stanislas Arnoud, Crackmes. N. d. Available from: `https://crackmes.one/faq`. [cited 2024-05-11].

36. ARNOUD, Stanislas. *Crackmes*. Website. Available from: `https://crackmes.one/`. [cited 2024-05-11].

37. CHECK POINT RESEARCH. *Anti-Debug Tricks*. Website. Available from: `https://anti-debug.checkpoint.com/`. [cited 2024-05-11].

38. TANDA, Satoshi. *SimpleSvmHook*. Online. In: GitHub. Last modified: 18 February 2021. Available from: `https://github.com/tandasat/SimpleSvmHook`. [cited 2024-05-11].

39. HYPERDBG. *Design of !epthook*. Online. In: HyperDbg. HyperDbg Documentation. Last modified 2022. Available from: `https://docs.hyperdbg.org/design/features/vmm-module/design-of-epthook`. [cited 2024-05-11].

Contents of the attachment

```
Release .......................... directory with a compiled version of the tool
  ├── FLOC.dll ........................... DLL implementation of the method
  └── Function Locator.exe ................... executable GUI using the DLL
├── Source ........................ directory with source code of the complete tool
  ├── DLL ................................ directory with source code of the DLL
  └── GUI ................................ directory with source code of the GUI
├── Example.mp4 ......................... video with an example usage of the tool
├── FLOC.sln ........................................ Visual Studio Solution file
├── IDA script.txt ........................... plain text version of the IDA script
└── Thesis.pdf ....................................... this thesis in PDF format
```