JOHANNES KEPLER
UNIVERSITÄT LINZ

Author
Alexander Voglsperger, BSc
k12005568

Submission
Institute for
System Software

Thesis Supervisor
o.Univ.-Prof. Dr. Dr. h.c.
Hanspeter Mössenböck

Assistant Thesis Supervisor
Dipl.-Ing. **Peter Feichtinger**
Dipl.-Ing. **Michael Obermüller**

June 2025

A Generic Machine Code Instrumentation Library

Master Thesis

to confer the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITÄT LINZ

o.Univ.-Prof. Dr.
Hanspeter Mössenböck
Institute for System Software

T +43 732 2468 4340
F +43 732 2468 4345
hanspeter.moessenboeck@jku.at

Secretary:
Karin Gusenbauer
Ext 4342
karin.gusenbauer@jku.at

Master's Thesis

A Generic Machine Code Instrumentation Library

Student:	**Alexander Voglsperger (12005568)**
Advisor:	Prof. Hanspeter Mössenböck
Co-Advisors:	DI Peter Feichtiger, DI Michael Obermüller
Begin:	1.10.2024

Binary instrumentation is a technique where the machine code of a running process is modified so it can be analyzed in some way. By attaching so-called hooks to specific pieces of the program, arbitrary code may be run when they are executed. This is particularly interesting for use cases such as performance measurements or monitoring, e.g. by hooking the entry points of specific functions of interest.

The goal of this thesis is to implement a generic C++ library that allows performing this kind of modification on arbitrary function entry points to attach a hook (function pointer or other callable object) provided by the user. The hook should receive all the necessary context for inspecting the program's state, such as register contents and stack pointer. Interpreting the context (e.g. reading local variables) is outside the library's scope, but rather the hook's responsibility.

One straightforward way to implement this kind of instrumentation is to use function trampolines. To instrument a piece of code, its first few machine instructions are replaced by a jump instruction to a so-called trampoline. The trampoline code takes care of preserving the necessary context, calling the actual hook, restoring context, and then executing the original instructions before returning execution to the instrumented function. To achieve this on x86, the library will have to be capable of copying machine code in memory while preserving its semantics.

The library implemented in this thesis should target x86, ideally both 32 and 64 bit. Since x86 is only one of several planned target architectures, the library should be easily extensible to support other architectures such as aarch64. The library need only support the architecture it's compiled for, i.e. running on x86_64 only needs to support hooking of x86_64 functions.

An existing library that might be of interest is Microsoft Detours [1], which implements function trampolines (and the necessary code copying) for Windows. As for a disassembler, it might be possible to use Capstone [2] rather than implementing everything from scratch.

The task requires a solid understanding of machine code and assembly language, and a commitment to implementing a user-friendly and extensible library that can be used by programmers for their daily work.

The work's progress should be discussed with the advisor at least every 2 weeks. Please follow the guidelines of the Institute for System Software when preparing the written thesis. The deadline for submitting the written thesis is 30.09.2025.

References:
[1] https://github.com/microsoft/detours
[2] https://www.capstone-engine.org

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

Dynamic machine code instrumentation is a powerful technique for analyzing and modifying running programs. This is crucial for performance monitoring, debugging and security analysis. However, with the lack of source code, creating robust instrumentation tools is challenging. It requires a deep understanding of the target architecture and calling conventions. Existing solutions are often platform-specific or lack the desired flexibility. This impacts companies such as Dynatrace that rely on such tools for application monitoring.

This thesis presents a generic `C++` library, designed to simplify the dynamic machine code instrumentation process. The goal is a simple interface that allows users to attach entry and exit hooks to functions. The library handles disassembling of the target function, relocation of original instructions while preserving semantics and the generation of trampoline code. The initial target architecture is `x86` with the ability to extend the library to support other architectures.

The resulting library lowers the barrier for developers, particularly in the context of Dynatrace, by providing a reusable and maintainable solution. The library simplifies the creation of tools that rely on instrumentation, and establishes a foundation for instrumenting further natively compiled languages such as *Rust*. This work ultimately aims to enhance the efficiency in development efforts and to improve capabilities for monitoring.

Kurzfassung

Dynamische Maschinencode-Instrumentierung ist eine mächtige Technik zur Analyse und Modifikation von laufenden Programmen. Sie ist entscheidend für Leistungsüberwachung, Fehlersuche und Sicherheitsanalysen. Ohne Zugriff auf den Quellcode ist die Entwicklung robuster Instrumentierungswerkzeuge jedoch eine große Herausforderung. Sie erfordert ein tiefes Verständnis der Zielarchitektur und der Aufrufkonventionen. Bestehende Lösungen sind oft plattformspezifisch oder bieten nicht die gewünschte Flexibilität. Dies betrifft auch Unternehmen wie Dynatrace, die zur Überwachung der Anwendungsleistung auf solche Werkzeuge angewiesen sind.

Im Zuge dieser Arbeit wurde eine generische `C++`-Bibliothek entwickelt, die den Prozess der dynamischen Maschinencode-Instrumentierung vereinfachen soll. Ziel ist eine einfache Schnittstelle, die es ermöglicht, Ein- und Austrittspunkte von Funktionen mit Hooks zu versehen. Die Bibliothek übernimmt dabei das Disassemblieren der Zielfunktion, die Verschiebung der ursprünglichen Instruktionen unter Erhalt ihrer Semantik sowie die Generierung von Trampolincode. Die erste Zielarchitektur ist `x86`, mit der Möglichkeit der Erweiterung auf weitere Architekturen.

Die resultierende Bibliothek senkt die Hürde für Entwickler, insbesondere im Kontext von Dynatrace, indem sie eine wiederverwendbare und wartbare Lösung bietet. Sie erleichtert die Entwicklung von Werkzeugen, die auf Instrumentierung setzen, und bildet die Grundlage zur Instrumentierung weiterer nativ kompilierten Sprachen wie *Rust*. Ziel dieser Arbeit ist es, die Entwicklungseffizienz zu steigern und die Überwachungsmöglichkeiten zu verbessern.

Contents

Abstract **i**

1 Introduction **1**

2 Background **3**
2.1 Machine Code and Assembly Language . . . 3
2.1.1 Machine Code . . . 3
2.1.2 Assembly Language . . . 4
2.1.3 Calling Conventions . . . 4
2.2 Methods of Function Instrumentation . . . 6
2.2.1 Source-level Instrumentation . . . 6
2.2.2 Compile-time Instrumentation . . . 7
2.2.3 Machine Code Instrumentation . . . 8
2.3 Capstone . . . 8
2.4 AsmJit . . . 10

3 Overview **12**

4 Implementation **14**
4.1 Library Interface . . . 14
4.2 Disassemble and Analyze . . . 15
4.2.1 Disassemble of Target Function . . . 15
4.2.2 Analyze Disassembly . . . 16
4.3 Trampoline Generation . . . 18
4.3.1 Memory Page Allocation . . . 19
4.3.2 Generation of Entry-Hook Call . . . 20
4.3.3 Relocation of Original Instructions . . . 22
4.3.4 Exit-Trampoline . . . 26
4.4 Hook Adapters . . . 27
4.4.1 Entry-Hook Adapter . . . 28
4.4.2 Exit-Hook Adapter . . . 28
4.5 Limitations . . . 29

5 Usage **31**

6 Evaluation **34**
6.1 Testing . . . 34
6.1.1 Unit Tests . . . 34
6.1.2 Functionality Tests . . . 35

6.1.3 Edge Case Tests 35
6.2 Benchmarks 37
6.2.1 Benchmark Setup 37
6.2.2 Benchmark Results 38
6.3 Comparative Evaluation with Existing Solutions 40

7 Summary and Future Work **42**
7.1 Summary 42
7.2 Future Work 43

8 Appendix **44**

Literature **47**

1 Introduction

In the field of software engineering and system analysis, the ability to observe, analyze and modify program behavior at run time is essential. Tasks that utilize these techniques range from debugging, profiling to security analysis. While source-level instrumentation or compile-time instrumentation are viable when source code is available, many scenarios involve analyzing pre-compiled, potentially closed-source software, thus necessitating methods that operate directly on the executed machine code.

Machine code instrumentation, specifically dynamic machine code instrumentation, provides such a capability. It involves modifying the machine code instructions of a running program. This allows developers and analysis tools to inject custom logic, typically referred to as "*hooks*". These hooks are placed at specific points in the program's execution, such as function entries and exits. Hooks can be used to gather data, measure performance, log events, or even alter the program's state. They offer a powerful insight into the program's behavior without requiring recompilation or access to the source code. Additionally, an instrumented function captures each call, no matter from where it is called. This is particularly valuable in areas such as application monitoring, where understanding the details of the software execution in production environments is crucial. For companies such as Dynatrace, simplifying the complex process of instrumenting diverse applications is a key aspect to deliver comprehensive observability solutions.

However, implementing a robust yet flexible dynamic machine code instrumentation is a complex task. It requires understanding the underlying architecture, instruction sets, Operating System (OS) internals and memory management. Existing solutions might be limited in terms of platform support, difficulty of use, or lack of desired level of abstraction. Therefore, there is a need for a more accessible, yet powerful solution that simplifies the process of instrumentation of machine code.

This thesis addresses the need by presenting the design and implementation of a generic `C++` library for dynamic machine code instrumentation. The primary goal of the library is to provide a simple and yet extensible library that allows developers to easily attach custom hooks to arbitrary functions in a running program. The library aims to handle the low-level details of disassembling machine code, managing memory, and patching of program flow, allowing developers to focus on the higher-level logic of their instrumentation tasks. A key motivation behind this generic approach is to create a reusable foundation that can simplify current instrumentation tasks. Additionally, the library serves as an entry point for the support of various natively compiled languages, such as *Rust*, or other OSs in the future.

The core technique deployed by the library is based on function trampoline jumping. When a target is instrumented, the initial machine code instructions are replaced with an unconditional jump to a generated code snippet – the "*trampoline*".

The dynamically generated trampoline is responsible for:

1. Saving the necessary execution context, such as registers.
2. Calling the user-defined hook function, and passing the captured context.
3. Restoring the original context.
4. Executing the original instructions that were replaced.
5. Adding a jump back to the target function's code, immediately after the overwritten instructions to resume normal execution.

To achieve this, particularly on architectures with variable-length instruction such as `x86`, requires several capabilities. The library must be able to correctly disassemble existing machine code, relocate the instructions while preserving their semantics, and generate new machine code that integrates seamlessly with the original program flow. The initial focus is on the `x86` architecture in both 32-bit and 64-bit variants. However, the design emphasizes extensibility in order to enable future support for other architectures such as `AArch64` and Operating Systems. While the library provides context to the hooks, the interpretation of the context, such as reading local variables, remains the task of the user (i.e., the author of the hooks).

Ultimately, this thesis aims to provide a reusable and maintainable `C++` library. It will lower the entrance barrier for dynamic machine code instrumentation and enables developers, particularly within contexts such as Dynatrace, to build sophisticated analysis and monitoring tools more easily. Additionally, it paves the way for future language support.

2 Background

Before we dive into the development of the library itself, it is important to establish a common understanding. This chapter introduces the most important concepts, terms and tools that are used throughout the thesis. Notably the concept of *machine code* and the *assembly language* in Section 2.1, and an overview of different methods to instrument a function in Section 2.2. Additionally, this chapter will introduce the *Capstone* disassembler framework in Section 2.3 and the *AsmJit* library in Section 2.4, which are used for disassembly and assembly generation, respectively.

2.1 Machine Code and Assembly Language

As this thesis is about instrumenting machine code, it is important to give some background on the concept of machine code and assembly language. This section will provide a brief introduction to the relevant concepts of machine code in Section 2.1.1, assembly languages in Section 2.1.2 and the concept of *calling conventions* in Section 2.1.3. Additional information and more detailed concepts will be introduced when necessary.

2.1.1 Machine Code

Machine code is the lowest-level representation of a program that is executed directly by the Central Processing Unit (CPU). It consists of a sequence of binary instructions that encode the operations to be performed and the operands to be used. Typically, those binary instructions are represented in hexadecimal code to condense the information. The available instructions are specific to the processor's architecture, such as `x86`, `ARM`, `RISC-V`, and many others. Each architecture has its own Instruction Set Architecture (ISA) that defines those instructions and their encoding.

```
1 // x86_64
2 50             // push rax;
3 48 89 d8       // mov rax, rbx;
4
5 // AArch64
6 E0 0F 1F F8    // str x0, [sp, #-16]!;
7 E0 03 01 AA    // mov x0, x1;
```

Figure 1: Comparison of move and push instructions for `x86_64` and `AArch64`.

Another crucial difference between architectures is the instruction size, and if all instructions are the same size or not. For example, the `x86` architecture has variable-length instructions, which can be up to 15 bytes long [1]. Because of this,

x86 instructions are more difficult to parse and analyze, as the length of individual instructions may vary drastically. In contrast, architectures such as ARM have fixed-length instructions, which makes them much easier to disassemble and work with, as all instructions are guaranteed to be the same size.

Figure 1 shows a comparison of a push/str, as well as a mov instruction for the x86_64 and AArch64 architectures. When comparing line 2 and line 3 it is clear that the x86_64 instructions have a variable length, where push is just one byte long, while mov is three bytes long. In contrast, the AArch64 instructions are all four bytes long. This property allows for assumptions of a fixed length, that in return can simplify certain operations.

2.1.2 Assembly Language

An assembly language is a low-level programming language that provides a human-readable representation of machine code. The language sits one layer above machine code and is still closely connected to the underlying architecture. For this reason, there is no single assembly language, but rather different types depending on the architecture. Additionally, there are different styles for assembly languages, for example from AT&T and Intel. One of the main differences between the two listed styles is the order of operands and the use of prefixes. AT&T places the destination as the last operand, while Intel places the destination as the first operand.

Figure 2 shows an example for calculating $\frac{a+b}{2}$ in x86 Intel syntax. The code shows the use of the add instructions to perform an addition of the two operands and store the result in rax. Afterwards, a division is performed in a similar way.

```
1 // rax = a, rbx = b
2 add rax, rbx // a + b
3 div rax, 2 // <a+b> / 2
```

Figure 2: Example of $\frac{a+b}{2}$ for x86_64 in Intel syntax.

For the rest of this thesis, the figures will be presented in Intel syntax, as it is the default syntax for the tools used during the development of the library.

2.1.3 Calling Conventions

In order to know how to work with the available registers, there are so-called *calling conventions*, that are applied when writing or generating code in an assembly language. A calling convention is a low-level concept in languages such as assembly, C and C++ that defines how functions are called, how their parameters are passed, and how they are returned. Additionally, a calling convention defines which registers

have to be saved by the callee, which registers can be used freely and how the stack has to be aligned. As with machine code and assembly language from the previous sections, there are different calling conventions for different architectures; they are typically defined as part of an Application Binary Interface (ABI).

There are different calling conventions, even for the same architecture. *cdecl*, *stdcall* and *fastcall* are a few examples of calling conventions that exist for the `x86` architecture. For `x86_64` the conventions are a bit more standardized, with the *System V ABI* for Unix and Unix-like systems, and the *Microsoft x64 calling convention* [2] [3]. When comparing them, both are quite similar, with the main difference being the amount of registers used for arguments.

As an example, the System V ABI specifies the following use for the General Purpose Register (GPR):

- **Order of arguments:** rdi, rsi, rdx, rcx, r8, r9

  With additional parameters being stored on the stack.

- **Return value:** rax, rdx

- **Saved by callee:** rbx, rsp, rbp, r12, r13, r14, r15

When applying the System V ABI to a function `add(int a, int b)` the assembly language program is similar to the one shown in Figure 3. The function starts with

```
1  _add:
2    // Prologue
3    push rbp;
4    mov  rbp, rsp;
5    // Actual functionality
6    mov rax, rdi;
7    add rax, rsi;
8    // Epilogue
9    mov rsp, rbp;
10   pop rbp;
11   ret;
```

Figure 3: Example of `add(int a, int b)` using the System V ABI.

a prologue in line 3 and line 4, that creates a new stack frame by storing the current base pointer `rbp` on the stack and then moving the stack pointer `rsp` into the base pointer. In some cases, an `enter` instruction may be used, which performs a similar action. However, this is rather uncommon and most compilers emit the `push` and `mov` instruction, as these subsequent instructions offer more flexibility for the compiler. The actual functionality of the add function is shown in line 6 and 7. Line 6 moves the first argument from `rdi` to the return value register `rax`. The addition is then

performed in line 7, where the second argument from `rsi` is added to the value from `rax`, and then also stored in `rax`. After the functionality is done, the function restores the stack frame in line 9 and line 10, by moving the base pointer back from the stack pointer, and then popping the original base pointer from the stack. It is also common to find the `leave` instruction, which is the counterpart to the `enter`. In contrast, the `leave` is actually emitted by compilers as it offers a simplicity benefit. Finally, the function returns to the caller in line 11.

2.2 Methods of Function Instrumentation

Function Instrumentation is the process of adding additional instructions to a function in order to collect information about its behavior at run time. Such instrumentation can be used to log function calls, measure execution time, or perform other actions. The instrumentation can be done at different levels, which depend on the availability of the source code and the used programming language.

The following sections will give an overview of the different methods of function instrumentation, such as *source-level instrumentation*, *compile-time instrumentation* and *machine code instrumentation*. This is not a comprehensive list and could be extended and structured in different ways. However, the used structure is sufficient for the purpose of an outline background explanation in this thesis.

2.2.1 Source-level Instrumentation

Source-level instrumentation is the process of adding instrumentation code directly to the source code of a program. This can be achieved my modifying the code manually or by using some tool that automatically patches the source code with instrumentation code. This type of instrumentation is quite simple, but requires access to the source code, which is not always available for closed-source software. Additionally, for compiled languages such as `C`, it requires recompilation of the program, which depending on the size of the project can become cumbersome.

The approach of source-level instrumentation is typically used for languages such as *Python* or *JavaScript*. By nature, these interpreted languages provide the source-code, which can be modified before execution.

Figure 4 shows a basic example of a function in Python that prints "*Hello <some_name>*". The function does not contain any instrumentation code, yet.

```
1 def foo(name: str) -> None:
2   print(f"Hello␣{name}!")
```

Figure 4: Example function in Python without instrumentation.

Figure 5 shows the same function with some basic execution time logging added as part of the instrumentation. In the instrumented code example, line 2 performs actions that can be classified as entry actions, with the code in line 6 and line 7 being classified as exit actions. The instrumentation code is just for demonstration purposes. Typically, the instrumentation code would be more complex or call some other function.

```
1 def foo(name: str) -> None:
2   start_time = time.time()
3
4   print(f"Hello␣{name}!")
5
6   end_time = time.time()
7   print(f"Execution␣time:␣{(end_time␣-␣start_time)␣*␣1000:.2f}␣ms")
```

Figure 5: Example function in Python with basic execution time measurement instrumentation.

2.2.2 Compile-time Instrumentation

Compile-time instrumentation is a way of instrumenting compiled languages, where the source code is available. Such instrumentation is performed by modifying the compiler using compiler plugins or specific compiler flags. These settings instruct the compiler to inject additional code into its output or an intermediate program representation during compilation. The approach is useful for natively compiled languages in larger projects, where an approach via source-level instrumentation might be cumbersome.

A well-known example of compile-time instrumentation is the GNU profiler, *gprof* [4]. Compilers such as GNU Compiler Collection (GCC) insert the additional code when a flag such as `-pg` is present [5]. With the example of gprof, the compiler adds calls at function entry and exit, which are used to record function call counts and relations. The collected data is then used to generate a call graph and perform performance analysis.

The key here is that not the developer, but the compiler adds the instrumentation code based on a configuration. Compile-time instrumentation benefits from the compiler's access to source code context or high-level intermediate representations. This can lead to more optimized and sophisticated instrumentation.

2.2.3 Machine Code Instrumentation

Machine code instrumentation can be applied to already compiled programs, no matter if source-code is available or not. This section describes the modification of binary machine code to include instructions for instrumentation.

As already mentioned in Section 2.1, the machine code is dependent on the target architecture's ISA and the OS. Because of this, instrumentation can be a bit tricky and typically involves disassembling the binary machine code into assembly code for easier manipulation. The modified assembly code is then assembled back into machine code and injected into the original place. In addition, it is important to ensure that the original program still works as intended, while still adhering to constraints of the ISA, OS and other factors.

The type of instrumentation can be classified into the following categories:

- **Static instrumentation** is performed before the program is executed, i.e., the binary file itself is modified. Therefore, the instrumentation is always present in all instances started from the instrumented binary.

  One example is *pe-afl* by L. Leong [6]. It allows fuzzy instrumentation to log the execution flow.

- **Dynamic instrumentation** is performed during the execution of a program, i.e., the binary that is loaded into memory is the only one that is modified. As a result, the instrumentation is only present during a single execution of the program. This allows for a more flexible instrumentation, as the instrumentation can be applied to certain executions only. However, such an approach requires a more complex setup, as some kind of program has to be loaded into memory in addition to the actual binary that handles the instrumentation.

  An example for a framework that provides such dynamic binary instrumentation is *Pin*, created by C. Luk et al. at Intel [7]. Pin allows for the instrumentation of binaries at run time and provides a powerful Application Programming Interface (API) for creating tools for security, emulation and performance analysis [8].

2.3 Capstone

As mentioned in Section 2.2.3, the process of machine code instrumentation typically involves disassembling the binary machine code into a higher-level representation for easier manipulation. The *Capstone* framework is a light-weight disassembler framework created by N. A. Quynh et al. [9].

The framework provides a simple and yet configurable API for disassembling various different architectures. Most crucial, it supports the common `x86` in 32-bit

and 64-bit, which are the main target architectures for the library developed in this thesis. Additionally, the framework is well maintained as of writing and provides bindings for various programming languages such as `C`, which is especially useful for the development of the library in this thesis.

Figure 6 shows an example of how to use the Capstone framework to disassemble some machine code. Line 4 to line 7 show the initialization, where the architecture and the mode are set, which in the case of the example are set to disassemble instructions compiled for the `x86_64` architecture. Line 10 triggers the disassembly process, starting with the machine code at the address of `CODE`. The framework will try to disassemble `sizeof(CODE) - 1` bytes from the passed address and assumes that those bytes start at `0x1000`. The `0x1000` is especially useful, as Capstone is able to automatically provide the correct absolute address of disassembled instruction.

```
1  int main() {
2    constexpr uint8_t CODE[] = "\x55\x48\x89\xE5";
3
4    csh handle;
5    if (cs_open(CS_ARCH_X86, CS_MODE_64, &handle) != CS_ERR_OK) {
6      return -1;
7    }
8
9    cs_insn *instr;
10   size_t amount = cs_disasm(handle, CODE,
11                             sizeof(CODE)-1, 0x1000, 0, &instr);
12
13   for (size_t i = 0; i < amount; i++) {
14     printf("%s\t%s\n", instr[i].mnemonic, instr[i].op_str);
15   }
16 }
```

Figure 6: Example of using the Capstone framework to disassemble `x86_64` machine code.

As a result of the disassembly step, the framework will store a pointer to the first disassembled instruction into `instr`, and then returns the amount of disassembled instructions. This pointer is used to access the dynamically allocated array of instructions, which can be looped over, as shown starting line 13. The output of the code shown in Figure 6 is shown in Figure 7, which is a typical prologue of a function in `x86_64` assembly language.

```
1 push rbp
2 mov  rbp, rsp
```

Figure 7: Output of the Capstone example code.

By default, Capstone does not generate any detailed information about the disassembled instructions, such as semantic groups or architecture-dependent information. However, it is possible to enable the additional information via an option as shown in Figure 8.

```
cs_option(handle, CS_OPT_DETAIL, CS_OPT_ON);
```

Figure 8: Enable detailed information in Capstone.

The more detailed information allows more specific checks and processing of the disassembled instructions. The code shown in Figure 9 provides an example function that checks if an instruction is in a specific group of instructions. As an example, the function can be used to check if an instruction is of any jump instruction kind, such as `jmp` or `je`.

```
bool IsInGroup(cs_insn *insn, cs_group_type group) {
  for (uint8_t i = 0; i < insn->detail->groups_count; i++) {
    if (insn->detail->groups[i] == group) {
      return true;
    }
  }
  return false;
}
```

Figure 9: Example for a check if an instruction is in a specific group.

2.4 AsmJit

In addition to disassembling machine code, in this thesis it is also necessary to generate new machine code. For that task, the original plan was to use the *Keystone* assembler, which is a partner project to Capstone, and was also created by N. A. Quynh et al. [10]. However, due to a lack of activity on the GitHub repository and the fact that the latest release is version 0.9.2 from June 2020, the choice was already questionable. Additionally, the license of Keystone for commercial use is not ideal for the intended use of the library.

For this reason, we decided to use *AsmJit* created by P. Kobalíček et al.[11]. AsmJit is a lightweight `C++` library that allows the generation of machine code for `x86`, `x86_64`, and `AArch64` architectures. The library provides a simple API for generating machine code by calling functions that correspond to the assembly instructions of a specific ISA. This is especially useful as these calls are compile-time checked, which allows for a more robust way of generating machine code. Additionally, the library

is well maintained and provides a more permissive license, which is more suitable for the use in the scope of this thesis.

Figure 10 shows the use of AsmJit to generate a callable function from assembly instructions. The code shows the initialization of the library starting from line 3 to line 8. In those lines, the `Environment` is set to target `kX64`, which is AsmJit's way of specifying `x86_64`. Line 11 moves the value `1` into `rax`, that holds the return value based on the ABI *System V* and *Microsoft x64*. Line 12 simply instructs the CPU to return from the function. Line 15 attaches the assembled instructions to the `Func fn`, which can then be called just like any other function as shown in line 20.

```
1  typedef int (*Func)();
2  int main() {
3    JitRuntime rt; CodeHolder ch; Environment env;
4    env.setArch(Arch::kX64);
5    ch.init(env);
6
7    x86::Assembler assm;
8    ch.attach(&assm);
9
10   // SystemV and Microsoft x64 expect return value in RAX
11   assm.mov(x86::rax, 1);
12   assm.ret();
13
14   Func fn;
15   if (Error err = rt.add(&fn, &ch)) {
16     printf("AsmJit␣error:␣%s\n", DebugUtils::errorAsString(err));
17     return 1;
18   }
19
20   printf("Result:␣%d\n", fn());
21 }
```

Figure 10: Example of using the AsmJit library to generate some `x86_64` machine code.

3 Overview

The goal of this thesis is to provide a user-friendly library for instrumentation, enabling the user to add hooks to the entry and exit of arbitrary functions. The current implementation of the library supports `x86` and `x86_64` architectures, with the design allowing for easy extension to other architectures.

The current implementation provides a single-function interface, which allows the user to instrument a function by providing a function pointer, an entry-hook and some configuration, as shown in Figure 11.

```
1 void instrument(void *target_function, void *entry_hook, Config &
      conf);
```

Figure 11: Interface of the library.

To insert an exit hook, the entry-hook must return the address of the exit-hook. This design decision was made to allow for flexibility in the use of the library. The primary reason is that the use case at Dynatrace may not always require an exit-hook. However, it is not always possible to determine this at the point of instrumentation. Instead, this needs to be determined at the run time of the function. This approach enables the library to offer greater flexibility in such cases.

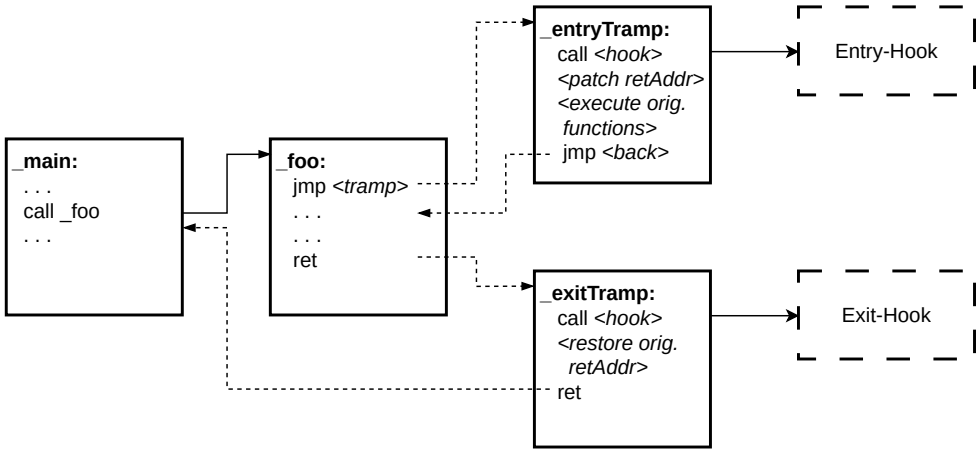

Figure 12: Overview of the instrumentation process.

The objective of the library is to modify the program flow such that the original function calls an entry-hook and an exit-hook, as outlined in Figure 12. The figure illustrates the interception of the original `_foo` by jumping to a trampoline that calls the entry-hook. After the entry-hook call the trampoline patches the return

address on the stack to point to another trampoline – the exit-trampoline. This patched address causes the program flow to jump to the exit-trampoline instead of the original caller of `_foo`. The exit-trampoline then calls the exit-hook and restores the original return address on the stack. The key point is that the instrumentation should be executed in a way that ensures the original function is performed without any undesired side effects. In other words, the original function should be executed as if it was not instrumented in the first place.

To instrument a function, the library must insert instructions to call the entry-hook at the beginning of the target function. However, the insertion of these instructions can be more complex than it may seem. The reason for this is that inserting instructions into an existing and compiled function would require moving all subsequent instructions. The primary issue is that subsequent functions may not be separated by padding after the target function. This would result a scenario where the subsequent function would also necessitate the movement of instructions, thereby initiating a chain reaction of instruction movement. Moving instructions is not only inefficient, but also very error-prone. The library would have to know the layout of the entire binary in order to move the instructions correctly without breaking any jumps or relative addresses.

Therefore, rather than inserting the instructions directly into the target function, the library relocates certain instructions and replaces them with custom ones. The library relocates the first few instructions of the target function and replaces them with an unconditional jump to a trampoline. The trampoline then calls the entry-hook, executes the relocated instructions, and then jumps back to the original function. This practice enables execution of additional instructions before the original function without loss of instructions and without the need to move all subsequent instructions.

In addition to the entry-hook, the library also provides an exit-hook call. The execution of this operation is comparable to the entry-hook, as it utilizes a trampoline. However, rather than relocating some instructions before each return instruction, the library updates the return address of the function to point to the trampoline for the exit-hook. While this approach may not be the most straightforward, it is undoubtedly efficient, as it eliminates the need to relocate instructions.

4 Implementation

The subsequent sections provide a comprehensive overview of the implementation details. As illustrated in Figure 13, the instrumentation is shown in high abstraction and is divided into the primary steps. These steps include the disassembly of the target function and the analysis of the disassembly in Section 4.2, and the generation of hook calls and the relocation in Section 4.3. Additionally, Section 4.4 introduces the usage of adapters in the library to counter limitations caused by the nature of machine code instrumentation.

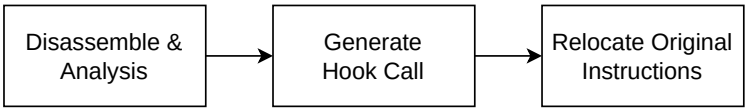

Figure 13: Overview of an instrumentation run.

The limitations of the library will be discussed in Section 4.5 at the end of the implementation. This includes the limitations of the current implementation, such as those imposed by the target architecture and OS. Additionally, the section outlines restrictions imposed by the generic nature of the library, such as limited support for certain programming languages.

4.1 Library Interface

Before starting with the core implementation, it is essential to provide a more thorough explanation of the library's interface. All public user-facing functions are located in the `LibHop.h` header, so the library is referred to as *LibHop* throughout the rest of this thesis. The header includes the `instrument` function, as previously referenced in Section 3.

As illustrated in Figure 14, this function serves as the primary interface for the user, performing the instrumentation of the target function. It expects the address of the function to instrument, i.e., the start address of the function. The next step is to pass the address to the entry-hook. In addition to the first two arguments, the user is required to provide a configuration. The configuration in lines 4 to 7 currently supports the selection of registers used for temporary storage, as well as an optional guaranteed minimum size for the target function.

At last, Figure 14 shows the `DataStruct`, which contains all GPR available on the target architecture. For example, in `x86` the struct has eight register entries; for `x86_64` it has 16 register entries. The struct is used to provide an easy way to interact with the registers when the hook is called.

```
void instrument(void *target_function,
                void *entry_hook, Config &conf);

struct Config {
  Register scratchRegs;
  std::optional<uint8_t> guaranteedMinFunctionSize = {};
}

struct DataStruct {
  // All general purpose registers available on the target
      architecture
}
```

Figure 14: Interface of the library.

4.2 Disassemble and Analyze

As previously mentioned in Section 3, the instrumentation of a function requires the relocation of some original instructions. Specifically, these are the instructions that will be overwritten by the new custom instructions of the library. Given the variable instruction size of machine code on certain architectures, such as `x86`, the library cannot simply relocate arbitrary machine code bytes. Instead, the library needs to disassemble parts of the target function, as presented in Section 4.2.1, in order to obtain the full instructions and their actual size. In Section 4.2.2, LibHop performs a quick analysis of the assembly to determine its suitability for instrumentation.

4.2.1 Disassemble of Target Function

In order to get the original function to jump to the trampoline, the library needs to insert a jump instruction into the target function. In `x86` and `x86_64`, this is done via a `movabs` instruction, which stores the address of the trampoline in a register. Afterwards, a `jmp` instruction is used to jump to the address stored in this register. In order to place these instructions, some original instructions have to be moved to the trampoline. It is important to note, that not all instructions of the function get moved, but only those that get overwritten by the `movabs` and `jmp` instructions. This is done to keep the original function mostly intact, and to avoid the need to relocate the entire function to the trampoline, which would be inefficient and unnecessary.

To achieve the disassembly part, LibHop utilizes the already presented Capstone framework from Section 2.3. The framework is configured to provide additional details for the disassembled instructions. This information will become important for the relocation of the original instructions to the trampoline and will be presented in more detail in Section 4.3.3.

The disassembly process starts immediately upon the call of the `instrument`

function. As shown in Figure 15, the disassembler is invoked to process the initial instructions of the target function. The length of the segment is twice the value of `MAX_INSTR_SIZE`, where `MAX_INSTR_SIZE` is a constant that defines the maximum size of a single instruction. Given that this thesis focuses on `x86` and `x86_64` first, the constant is set to 15 bytes for this example. Therefore, the disassembler will attempt to disassemble 30 bytes from the target function. The two-instruction depth ensures sufficient space to capture the necessary instructions, even when the function starts with smaller instructions. It is important to note, that not all disassembled instructions are necessarily used, instead only those that require relocation are considered further in the process.

```
1  size_t disassembled_size = cs_disasm(
2    handle, target_function,
3    MAX_INSTR_SIZE * 2, target_function,
4    0, &instr);
```

Figure 15: Disassembly of the target function.

Additionally, the target function is not only passed as the start of the code to disassemble, but also as the start address of the code. This argument ensures that all relative instructions are automatically adjusted by Capstone to include their absolute address. While the absolute address may appear irrelevant, it simplifies the conversion of relative instructions to absolute ones.

4.2.2 Analyze Disassembly

This section explains the analysis of the disassembled instructions. Specifically, the analysis checks if the disassembly created in Section 4.2.1 is large enough to be instrumented, and searches for relative instructions within the disassembly.

To verify that the function is sufficiently long to be instrumented, the library searches for a `ret` instruction within the instructions that have be relocated to the trampoline. To be more precise, if any return instruction is present in the first N bytes, it will be identified as they cannot be instrumented. In the `x86` architectures the N indicates the amount of bytes reserved for the unconditional jump. In this case the N is consists of a `movabs` with 10 bytes and a `jmp` with 3 bytes, therefore N is 13 bytes.

As shown in Figure 16, this synthetic example contains a return instruction early on. The example will cause the library to abort instrumentation and throw an error because the return instruction is within the 13 byte range. The objective of abort is to avoid the overwriting of instructions during the actual instrumentation process. These instructions are no longer part of the target function and, consequently can

lead to undefined behavior if overwritten.

As already mentioned, the library requires 13 bytes of space for the jump to the trampoline. In the case of the example, the `ret` instruction is the twelfth byte, which means that the function would have to overwrite one additional byte of code that is not part of the function anymore. The library is conservative in that regard and assumes that functions might be tightly packed in memory without padding. Therefore, writing beyond the detected end of a function is prohibited to prevent potential corruption of subsequent, unrelated code or data segments.

```
1 _add:
2   0x55             // push rbp
3   0x48 0x89 0xE5   // mov  rbp, rsp
4
5   0x48 0x89 0xF8   // mov  rax, rdi
6   0x48 0x01 0xF0   // add  rax, rsi
7
8   0xC9             // leave
9   0xC3             // ret
```

Figure 16: Disassembly of the target function that causes an instrumentation abortion due to short length.

To detect cases as presented in Figure 16, the library can simply iterate over the disassembled instructions until a sufficient size has been reached. If the library encounters a `ret` instruction before enough instructions have been checked it aborts the instrumentation process.

As shown in Figure 16, the example is straightforward, with a single return instruction and a linear program flow. However, the library also needs to handle more complex cases, such as early returns in branches. In such cases, a simple iteration over the instructions is no longer sufficient, as the function may still continue after the `ret` instruction as shown in Figure 17. The sample is very synthetic and does not include the prologue or epilogue, but it shows the problem very well. The function branches at line 3, which potentially skips the `ret` instruction in line 4. Therefore, the function does not end at the `ret` instruction in line 4, but rather at a subsequent `ret` instruction in line 7.

In order to identify such cases, it is necessary for the library to analyze the function's control flow in greater detail, as the function does not end at the first `ret` instruction. To obtain this information the library also uses a method similar to the example in Figure 16, where the library iterates over the disassembled instructions. However, if a jump is detected, the library ignores any `ret` instructions until the target of the jump is reached. The detection is based on the assumption that a function will never jump outside itself. As shown in Figure 17, the jump instruction

```
1 _foo():
2   0x48 0x39 0xF7  // cmp  rdi, rsi
3   0x7E 0x01       // jle  _after_ret
4   0xC3            // ret
5   _after_ret:
6   ...
7   0xC3            // ret
```

Figure 17: Disassembly of the target function that has an early return in a branch.

in line 3 is detected, and the library ignores any `ret` until line 5 is reached. While this assumption generally holds for most compiler-generated code, it can present a potential point of failure for highly obfuscated code or specific hand-crafted assembly. However, such cases are considered beyond the scope of typical usage of this library.

In addition to checking for `ret` instructions, the library also scans for any instructions that are relative to the current instruction. In essence, the library scans if an instruction is a relative jump or if any of the operands is the Instruction Pointer (IP) register. The check is quite simple since relevant information is directly provided by Capstone, and group comparisons such as in Figure 9 simplify these checks even more. This check for relative instructions is performed for a minor optimization reason. Due to this optimization, the library is able to bypass any transformation during the relocation of the original instructions to the trampoline. For more information about the relocation process see Section 4.3.3.

4.3 Trampoline Generation

The previous section outlined the disassembly and analysis of the target function, which is the initial step of the instrumentation process. This section details the generation of a trampoline. A trampoline is an additional memory location where the entry-hook is called, and the copied instructions are placed. For each instrumentation of a function, a new trampoline is generated. This is because each trampoline contains target-specific instructions, i.e., the instructions that have been relocated.

The library allocates a memory page for the trampoline, which is used to store the additional custom instructions and relocated instructions as described in Section 4.3.1. The contents of the trampoline are outlined in Figure 18. The trampoline begins with the entry-hook call, which prepares the structure for the registers and cleans up afterwards. This process is presented in Section 4.3.2. Subsequently, the initial instructions from the original function are to be relocated to the trampoline, as outlined in Section 4.3.3. At the end of the trampoline, the library inserts an unconditional jump to the next untouched instruction of the target function. Finally, in Section 4.4 the thesis presents the usage of adapters to serve as an intermediate

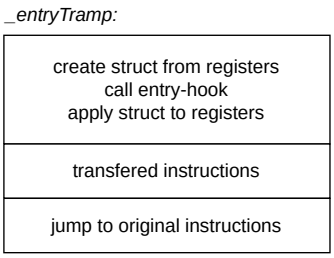

Figure 18: Overview of the trampoline to generate.

layer between the trampoline and the actual hook. The approach with the adapter is intended to simplify some code and to improve maintainability.

4.3.1 Memory Page Allocation

In order to place the additional and relocated instructions on the trampoline, an essential step for the library is the allocation of additional memory. The process of memory allocation is OS-dependent, meaning that different methods must be employed depending on the specific OS. By default, any method allocates three continuous memory pages, with the middle one being used for the trampoline. The first and last pages are designated as guard pages, which are marked as non-readable and non-writable. This approach utilizes more memory, but aids in the detection of memory underflow and overflows, thereby helping library developers to identify and prevent bugs.

For systems using some form of Linux, memory page allocation is performed via `mmap`. The library incorporates this and related function within an `allocate`, as shown in Figure 19. The function determines the size of a memory page by referencing the `sysconf(_SC_PAGESIZE)` setting, as specified in line 2. Line 4 attempts to allocate the memory with triple the amount to account for the guard pages. Additionally, the protection is set to `PROT_NONE`, which sets the pages to non-readable and non-writable. The remaining arguments are irrelevant to this thesis and are therefore omitted in this section. The return value of the function call may be a valid address or an error code. Therefore, it is essential to check the return value. If the allocation was successful, the address of the actually usable page is calculated in line 10. As shown on line 11 the library attempts to mark the center page as readable, writable and executable. In the event of failure, the library must deallocate the already allocated page, as shown in line 12. Finally, the address of the middle memory page is returned in line 16.

For systems using Windows, the memory allocation process is quite similar to

```
char *allocate() {
  const size_t page_size = sysconf(_SC_PAGESIZE);

  void *pages = mmap(NULL, 3 * page_size,
                     PROT_NONE, MAP_PRIVATE | MAP_ANON, 1, 0);
  if (pages == MAP_FAILED) {
    throw std::runtime_error("Failed to allocate memory");
  }

  void *middle_page = ((char *)pages) + page_size;
  if (mprotect(middle_page, page_size, PROT_READ | PROT_WRITE |
      PROT_EXEC) != 0) {
    munmap(pages, 3 * page_size);
    throw std::runtime_error("Failed to set main page to RWX");
  }

  return (char *)middle_page;
}
```

Figure 19: Memory page allocation with guard pages on Linux.

the one on Linux. However, the library employs `VirtualAlloc` and its associated functions instead of utilizing `mmap`.

4.3.2 Generation of Entry-Hook Call

With the memory page allocated, this chapter focuses on the generation of the entry-hook call. It handles the process of preparing the registers for the hook, the hook call and the post-processing of the passed registers.

To pass the registers to the hook, the library uses a struct on the stack, which contains all available GPR of the target architecture. An example of such a struct is shown in Figure 20 for `x86_64`. In this example, the registers `rax` to `r15` with the 64-bit type `int64_t` are used. The specific arrangement of the struct is determined by the target architecture, given the number and types of available registers.

```
struct DataStruct {
  int64_t rax;
  int64_t rbx;

  ...

  int64_t r15;
}
```

Figure 20: Reference `DataStruct` for `x86_64`.

To allocate the `DataStruct` on the stack and populate the struct with data from the registers, the library must generate the necessary instructions. These instructions are generated during instrumentation time using *AsmJit* and then positioned at the start of the trampoline. As the preparation and call of the hook can be viewed as a distinct function, this hook call must also be addressed in a similar manner. Therefore, the library generates the typical prologue of a function at the start of the trampoline and the epilogue before calling the original instructions.

As illustrated in Figure 21, the instructions for allocating and filling the struct on the stack are generated. Line 1 shows the allocation of the struct itself, which uses a 16 byte aligned version of the size of the struct. The alignment is necessary to maintain a 16 byte alignment as required by most `x86` ABIs. The 16-byte alignment is primarily necessary for Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) instructions that operate on *XMM* and *YMM* registers.

```
1 assm.sub(x86::rsp, <aligned-DataStruct-size>);
2
3 assm.mov(qword_ptr(x86::rsp, offsetof(DataStruct, rax)), x86::rax);
4 ...
5
6 // Adjust stored rsp register to contain original rsp
7 assm.add(qword_ptr(x86::rsp, offsetof(DataStruct, rsp)),
8          <aligned-DataStruct-size> + 8);
```

Figure 21: Stack allocation and filling of `DataStruct`.

In line 3, the library moves the value of the `rax` register into the struct at the offset of `rax` in the `DataStruct`. Instructions similar to this `mov` are generated for all available GPRs. The `rsp` value captured in the struct must reflect the stack pointer as it would be seen by the original function, prior to the hook's own manipulation. As the trampoline has its own prologue that allocates space on the stack, the value must be adjusted. Therefore, the library adds the size of the aligned `DataStruct` and additional 8 bytes to account for the pushed `rbp`.

With the struct being allocated and populated, the next step is to call the hook. For simplicity reasons, the following explanation are based on the *System V ABI*, though similar principles apply to the currently supported *Microsoft x64 calling convention*. Figure 22 shows the instructions necessary for the hook call. Line 1 moves the address to the start of the struct into `rdi` to be used as the first argument. This is the necessary part to allow the hook to access the registers via a `DataStruct` pointer. Line 2 moves the address of the entry-hook into a user-defined scratch register and line 3 finally calls the entry-hook.

Finally, the library needs to clean up the stack when the hook returns. In most cases, this would be a simple `leave` instruction. However, the desired functionality

```
assm.mov(x86::rdi, x86::rsp);
assm.mov(<scratch-register>, <entry-hook-addr);
assm.call(<scratch-register>);
```

Figure 22: Call of the entry-hook using the System V ABI.

is for the hook to be able to modify the `DataStruct`'s registers and for those changes to be applied to the original function. Consequently, the library must perform the opposite of the fill step and move the values from the struct back to the registers as shown in Figure 23. The restoration process does not apply to `rsp`, `rbp` or `rip`, because modifying them could cause more issues. The final instruction is the `leave` instruction in line 4.

```
1 assm.mov(x86::rax, qword_ptr(x86::rsp, offsetof(DataStruct, rax)));
2 assm.mov(x86::rbx, qword_ptr(x86::rsp, offsetof(DataStruct, rbx)));
3 ...
4 assm.leave();
```

Figure 23: Dismantle of the `DataStruct` to registers and deallocation.

4.3.3 Relocation of Original Instructions

This section describes how to relocate the original instructions to the trampoline to retain the target function's original functionality. The relocation process must preserve the semantics of the original instruction sequence, even when instructions must be transformed.

During the relocation of the original instructions to the trampoline, some instructions may need to be transformed, resulting in a different instruction length. Figure 24 shows a `mov` instruction in line 2 that accesses memory relative to the IP register. The transformation process converts this relative access into an absolute address, as shown in line 6. The machine code is presented in hexadecimal format below the two instructions. The original instruction is 7 bytes long, while the transformed instruction is 8 bytes long.

The library manages the varying lengths internally during the construction by placing each instruction sequentially onto the trampoline. After writing an instruction, the write-offset for the next instruction is increased by the size of the just-written instruction. This process ensures that instructions are placed contiguously and that the size differences between the original and transformed versions are managed correctly. The approach taken for instruction relocation largely depends on whether the instructions to be relocated contain relative instructions.

```
// Original instruction
0x1000:  mov  rax, QWORD ptr[rip + 0x10];
         => 0x48, 0x8B, 0x05, 0x10, 0x00, 0x00, 0x00

// Transformed instruction
0x1000:  mov  rax, QWORD ptr[<0x1000 + 0x10>];
         => 0x48, 0x8B, 0x04, 0x25, 0x10, 0x10, 0x00, 0x00
```

Figure 24: Example of a transformed instruction.

The simpler scenario is when the analysis in Section 4.2.2 determines that all the relocatable instructions are free of relative instructions. This knowledge allows the library to perform a straightforward memory copy of the original machine code to the trampoline. This is possible because no adaptation is necessary, and the library knows the size of the machine code for the relocatable instructions. While not strictly necessary, this simplifies the actions taken by the library in such cases. Additionally, it makes debugging during development easier. With this single large copy step, the library has completed the relocation process and can add the jump to the next untouched instruction of the target function.

However, it is more common for the analysis to find at least one relative instruction that needs to be relocated. A direct copy would lead to incorrect behavior when a relative instruction is executed from a different memory location in the trampoline. Therefore, as shown in Figure 25, the library iterates over all disassembled instructions. This iteration continues until either enough instructions have been relocated to provide sufficient space in the original function, or the end of the disassembly is reached. It is noteworthy that the second part of this condition is merely a safeguard because the library already checked for enough instructions in the analysis step.

```
for (size_t idx = 0, copied_bytes = 0;
     idx < disassembled_size && bytes_copied < MOV_AND_JMP_SIZE;
     ++idx) {

  if (!isRelativeInstruction(instr[idx])) {
    // 1:1 copy of instruction
  } else {
    // Adapt instruction
  }
  copied_bytes += instr[idx].size;
}
```

Figure 25: Relocation of original instructions.

Within the loop, the library examines each instruction to identify relative operations. If the current instruction being checked is not relative, a memory copy

similar to the first case is performed, but at the level of a single instruction. If the instruction is relative, the library needs to adapt it.

Adapting instructions is slightly more complex because the library must know how to transform each relative instruction into an absolute instruction. Depending on the ISA, these transformations are not always trivial, and the library needs to adapt a different amount of instructions. This is especially true for architectures that belong to the Complex Instruction Set Computer (CISC) family, such as `x86` and its variants. These architectures provide many instructions, some of which are relative instructions that need to be transformed. Additionally, it may not be possible to write generic transformation functions for the bigger group of instruction, which creates the need for many independent transformation functions.

The explanation of these transformation functions begins with instructions that use the IP register as an operand for a relative operation. The IP register case is simple because the library can replace the register with the instruction's original address, as shown in Figure 24. After this transformation, the instruction will access the same memory location as before, but with an absolute address. The transformed instruction can be relocated to an arbitrary location such as the trampoline without causing a side effect. This principle applies to `add`, `sub`, `lea`, and many other instructions common at the start of a function. Exceptions are the relative `jmp` instruction and conditional jumps, such as the `je`.

A `jmp` instruction can also be transformed to use an absolute address. However, it cannot directly use the absolute address. Instead, it expects the absolute address to be in a register or accessible via a relative address inside the machine code. To avoid undesired side effects from overwriting an unknown value in a register, the library stores the absolute address as part of the machine code. Figure 26 shows an example of such a transformation. The transformed `jmp` instruction uses the address

```
1 // Original instruction
2 0x1000:  jmp  0x14;
3
4 // Transformed instruction
5 0x1000:  jmp  QWORD ptr[rip + 0x00];
6 0x1006:  <0x1000 + 0x14>;
```

Figure 26: Transformation of relative `jmp` instruction.

stored in the machine code right after the jump instruction. Keen readers may have noticed that the `jmp` instruction now includes a relative address to the IP register. This is not a problem in the case of this transformation, because the instruction is already relocated.

Finally, the library needs to handle conditional jumps. This is not as simple

as replacing something with an absolute address. This is due to a limitation of the currently supported `x86` ISA, which does not allow conditional absolute jump instructions. Therefore, the library must transform the single conditional jump instruction into a sequence of instructions that perform the same operation with an absolute jump.

Figure 27 shows the transformation of a conditional jump instruction. In this example, the instruction is a *jump if equal*. Since there are no absolute conditional jumps, the library must be able to skip the transformed absolute jump. The library can achieve this by jumping over the absolute jump with a conditional jump that checks for the inverse condition. In line 5 of the example, the inverse conditional jump allows the program flow to skip the absolute jump. The absolute jump in line 6 is only executed if the condition is not met. This transforms the original instruction into a sequence that retains the same functionality but now has an absolute address.

```
1 // Original instruction
2 0x1000:  je  0x14
3
4 // Transformed instruction
5 0x1000:  jne _skip;
6 0x1002:  jmp QWORD ptr[rip + 0x00];
7 0x1008:  <1000 + 0x14>;
8 0x1006:  _skip:
```

Figure 27: Transformation of conditional jump instruction.

These are the main transformations that the library uses to translate relative instructions into absolute ones. The presented transformations are not the only ones needed. Due to the large number of existing instructions, however, the library and therefore this thesis cannot cover them all. For now, the library only supports transformations for instructions that are more commonly present at the start of a function. If the library encounters an unsupported instruction, an error is thrown and the instrumentation process is aborted. This prevents any undefined behavior or side effects that may occur when the library tries to transform an unsupported instruction.[1] Nevertheless, the library is designed to allow for the easy extension of transformation functions.

Although the library can transform individual IP-relative instructions, it is important to note that the library does not yet retarget relative jumps between instructions. This means that if both the jump and the target are relocated, the library

[1]Supported relative instructions include IP-relative addressing for `mov`, `movabs`, etc., and relative forms of `jmp`, `je`, etc. A complete list can be found in Figure 41.

does not adjust the jump to point to the relocated target. This is a major limitation that is also mentioned in Section 4.5.

4.3.4 Exit-Trampoline

Finally, this section presents the trampoline for the exit-hook. Unlike the trampoline for entry-hooks, the exit-trampoline is created only once, at the time of the first instrumentation, for the entire library. As mentioned in Section 3, the exit-trampoline is also different in that it is not jumped to by replacing instructions. The reason for this is that a function may contain multiple exit points, which would require the library to replace them all with jumps. This approach would be inefficient and would require the library to identify all exit points in functions of any size. Instead, the library uses a different approach to get to the exit-trampoline from the target function. Rather than replacing return instructions, the library patches the return address of the caller to point to the exit-trampoline. This approach allows the library to avoid unnecessary instruction updates and can handle multiple exit points with a single change.

Figure 28 shows the outline of the exit-trampoline. The target function's return instruction directs the program flow to the exit-trampoline and pops the return address from the stack. For this reason, the library must reallocate space for the return address on the stack to store the original return address. This reallocation is performed by a single `sub` instruction, which moves the stack pointer down by 8 bytes as shown in line 2. With the space reserved, the library can restore the original return address. Then, the library performs the steps for calling the hook, as previously described in Section 4.3.3. These steps are the same as those for the entry-hook and include preparing the `DataStruct` and the call of the exit-hook. Finally, the trampoline exits with a `ret` instruction, which uses the popped return address from the stack to return to the original caller. With this final step, the program flow returns to the original caller.

```
1 _exit_trampoline:
2   sub  rsp, 8;
3   // Restore original return address
4   ...
5
6   // Preprocessing, exit-hook call, postprocessing
7   ...
8
9   ret;
```

Figure 28: Outlines of the exit-trampoline.

4.4 Hook Adapters

So far, the thesis has presented the library in such a way that a trampoline calls the hooks directly. However, some more complex tasks are necessary that are difficult to implement in raw assembly. For this reason, the library uses two `C` functions, that work as an intermediate layer between the trampoline and the actual hooks. This allows those complex tasks to be implemented once, after which the compiler takes care of the rest.

But what are those complex tasks? As mentioned briefly in Section 3, Dynatrace's use of the library requires on-demand exit-hooks, meaning an exit-hook call is not always necessary. The entry-hook decides if a call necessary. For this reason, the library expects the entry-hook to return either an address to the exit-hook or a `nullptr`. With this information, the library can decide whether to replace the return address on the stack with the address of the exit-trampoline. However, relaying the exit-hook address and the original return address to the exit-trampoline is not as easy as it sounds. The library cannot simply store the addresses on the stack because the stack must remain untouched during execution of the original function. Additionally, storing the addresses on the trampoline's memory page is not possible because the instrumented function is shared across multiple threads.

For this reason, the library uses a different approach for storing addresses. The library uses Thread Local Storage (TLS) to store addresses on a per-thread basis. The library uses a `std::stack` to correctly handle scenarios where instrumented functions are called in a nested fashion within the same thread. For example, if an instrumented function A calls an instrumented function B, the stack's *last-in first-out* nature ensures that B's exit-hook is called before A's exit-hook. TLS allows the library to store addresses without affecting the stack or the trampoline's memory page. However, interacting with TLS directly from assembly code is not trivial because the TLS implementation varies depending on the OS, architecture and model specified by the ABI.

```
1 struct TlsAddressStorage {
2   uintptr_t origRetAddr;
3   uintptr_t exitHookAddr;
4 }
5
6 thread_local std::stack<TlsAddressStorage> addressStack;
```

Figure 29: Storage of addresses in TLS stack.

Now, coming full circle, the library uses the two `C++` functions in order to easily work with TLS and safely store addresses during the execution of the original function. As shown in Figure 29, the addresses are stored in a `std::stack` instance

declared as a TLS variable using the `thread_local` keyword. Since the addresses are used in pairs, they are combined into a single struct, which reduces the number of TLS variables and the likelihood of two separate stacks becoming out of sync.

4.4.1 Entry-Hook Adapter

This section presents the adapter function, which sits between the entry-trampoline and the entry-hook. Figure 30 shows the adapter function, which is called by the entry-trampoline. Similarly to the entry-hook, the adapter receives the `DataStruct` pointer as the first argument. However, the adapter also receives the address of the entry-hook as the second argument. This second argument is necessary because the adapter itself has no context of the actual instrumentation and would therefore not be able to determine, which hook to call. The adapter function begins by calling the entry-hook in line 3. The return value may be the address of the exit-hook, therefore, a check is performed in line 4. If the return value is a `nullptr`, the adapter has finished its task and can return without adjusting exit-hook.

```
1  using EntryHook = void *(*) (DataStruct *)
2  extern "C" void asmMethodEnter(DataStruct *ds, EntryHook entryHook){
3    void *exitHookAddr = entryHook(ds);
4    if (exitHookAddr == nullptr) {
5      return;
6    }
7
8    // Store original return address and exit-hook address
9    auto rsp = (uintptr_t *)(ds->rsp);
10   addressStack.push({ rsp[0], (uintptr_t)exitHookAddr });
11
12   // Patch the return address
13   rsp[0] = (uintptr_t)getExitTrampolineAddr();
14 }
```

Figure 30: Adapter function between entry-trampoline and entry-hook.

Lines 9 and 10 use the TLS stack to store the original return address and the exit-hook. Finally, line 13 updates the return address to point to the exit-trampoline. The `getExitTrampolineAddr()` function shown is a simple getter that returns the address of the exit-trampoline, which is created during the first instrumentation.

4.4.2 Exit-Hook Adapter

This section introduces the exit-hook adapter, the counterpart to the entry-hook adapter presented in the previous section. The exit-hook adapter is called by the exit-trampoline and is responsible for restoring the original return address and calling

the exit-hook. The adapter function is shown in Figure 31 and begins by checking whether the TLS stack contains any entries. If the stack is empty, then `asmMethodExit` was called without a preceding `asmMethodEnter`, and therefore aborts.

```
1  using ExitHook = void (*) (DataStruct *)
2  extern "C" void asmMethodExit(DataStruct *ds) {
3    if(addressStack.isEmpty()) {
4      throw std::runtime_error("Exit␣adapter␣called,␣but␣stack␣is␣
           empty!");
5    }
6    auto [origRetAddr, exitHookAddr] = addressStack.top();
7    addressStack.pop();
8
9    // Restore original return address
10   auto rsp = (uintptr_t *)(ds->rsp);
11   rsp[0] = origRetAddr;
12
13   // Call exit-hook
14   auto exitHook = (ExitHook)exitHookAddr;
15   exitHook(ds);
16 }
```

Figure 31: Adapter function between exit-trampoline and exit-hook.

If there is still data on the stack, the library retrieves the original return address and the exit-hook address. Using the `DataStruct` pointer again, the library restores the original return address in line 11. Finally, the exit-hook is called in line 15. This completes the exit-hook adapter. The flow will then return to the exit-trampoline, which utilizes the original return address to exit to the original caller.

4.5 Limitations

As with any library, this library has its inherent limitations. Although the library was designed to be as generic as possible, there are still some areas in which it falls short. This chapter aims to present these limitations, and offers a transparent view of the library's scope and potential areas for future development.

The following list provides an overview of known limitations:

- **Handling of Relative Jumps within Relocated Instructions:** Currently, the library has a major limitation in how it handles relative jumps within a relocated target. When the target of a relative jump instruction also resides within relocated instructions, the library does not adjust the target address correctly. Instead of the new position in the trampoline, the target will be the absolute address of the original position. This error would to undefined

behavior when the instrumented function is executed. Therefore, the library does not instrument such functions for now.

- **GoLang Support:** The intricate nature of the *GoLang* runtime environment, particularly its dynamic stack management, poses significant challenges to the current instrumentation strategy. The main issue is the potential lack of stack space for the trampoline, which requires space to push the `DataStruct`. Currently, the library does not support unique stack handling mechanisms. This is because the library would require a more sophisticated approach to instrumentation, as well as specific code to perform allocations via the GoLang runtime. This is not feasible for the current version of the library, so it does not support GoLang applications. Applications written in other languages without such stack limitations should work without issues.

- **Architecture Support:** The current version of the library focuses on the most commonly used `x86` and `x86_64` architectures. Other architectures, such as `AArch64` and `RISC-V`, are not supported. However, the library's modular design makes it possible to add support for other architectures with reasonable effort.

- **Operating System Support:** The current version of the library is designed to work on Linux and Windows. Other OS such as *macOS* and *FreeBSD*, are not supported. This is simply a matter of limiting the scope of this thesis and library. However, given the library's modular design, it should be possible to add further support with reasonable effort.

- **Memory Overhead:** The current placement of each trampoline on a separate memory page (surrounded by guard pages) ensures isolation and simplifies memory management. However, this approach introduces a degree of memory inefficiency. This is especially true when many functions are instrumented and therefore many memory pages are allocated for trampolines. Optimizing trampoline placement to reduce memory consumption is a potential area for future development.

- **Instruction Coverage:** The library provides transformations for many of the more commonly used relative instructions within the `x86` and `x86_64` ISA. However, not all such instructions are currently supported. The library is designed to be extensible, allowing for the addition of new transformations as needed. However, the current version may encounter unsupported instructions, resulting in to instrumentation failures. This limitation is particularly relevant for less frequently used instructions or those with complex operand structures.

5 Usage

To demonstrate the practical application of the library created for this thesis, we will provide a simple usage example. This section covers the steps necessary to instrument a C++ function using LibHop.

The process of getting the library recognized by the compiler depends on the build system used and the installation method. This process is beyond the scope of this thesis and will not be covered in detail. However, it should be similar to the process for other libraries. Therefore, consult the documentation of the used build system for further information.

```
1  int fibonacci(const int n) {
2    if (n <= 1) {
3      return n;
4    }
5    return fibonacci(n - 1) + fibonacci(n - 2);
6  }
7
8  int main() {
9    int result = fibonacci(4);
10   std::cout << "Result:␣" << result << std::endl;
11   return 0;
12 }
```

Figure 32: Example program used as a base for the explanation of the usage.

Figure 32 presents a sample that is used to explain the usage. It is a simple `C++` program that calculates the Fibonacci number of a given integer and prints the result to the standard output. In the case of the example, the program calculates the Fibonacci number for 4, which is 3.

```
1  int main() {
2    hop::LibHop lh;
3    std::array scratchRegs{hop::Register::R11};
4    lh.instrument((void *)fibonacci, (void *)entryHook, {scratchRegs})
         ;
5
6    int result = fibonacci(4);
7    std::cout << "Result:␣" << result << std::endl;
8    return 0;
9  }
```

Figure 33: Instrumentation call to instrument the function `fibonacci`.

To instrument the `fibonacci` function using the LibHop library, the source file must include the `LibHop.h` header file. The example in Figure 33 shows an updated

`main` function. Line 2 creates a LibHop instance, and line 3 creates an array that holds scratch registers. Starting on line 4, the `instrument` function is instructed to instrument the `fibonacci` function in order to call the `entryHook` function before executing the original function. With this call, the entry-trampoline is created, and the machine code of the `fibonacci` function is modified in memory.

```
1 void *entryHook(hop::DataStruct *ds) {
2   std::cout << "Input: " << ds->rdi << std::endl;
3   return (void *)exitHook;
4 }
5
6 void exitHook(hop::DataStruct *ds) {
7   ds->rax += 1;
8 }
```

Figure 34: Entry-hook and exit-hook function for the instrumentation of `fibonacci`.

Figure 34 shows the sample hook functions provided by the user of the library. In this example, the entry-hook prints the value of the input parameter. Since the `DataStruct` has no knowledge of the used calling convention, the user must ensure that the correct registers are accessed. In the example shown, the target system uses Ubuntu and the System V ABI, therefore, the first parameter is passed in the `rdi` register. Finally, the address of the `exitHook` function is returned to the library, instructing it to call the exit-hook. On the other hand, the exit-hook modifies the return value of the function. This modification does not serve any practical purpose, but demonstrates the ability to alter values of the original program via the `DataStruct`. In the case of ABI the return value is stored in the `rax` register.

```
1  Input: 4
2  Input: 3
3  Input: 2
4  Input: 1
5  Input: 0
6  Input: 1
7  Input: 2
8  Input: 1
9  Input: 0
10 Result: 12
```

Figure 35: Output of the example program.

The program can be compiled with the instrumentation of the function. Calling the `fibonacci` function will redirect the control flow to the entry-hook function via a trampoline and adapter. Then, the original function is executed. With any return call of the original function, the exit-hook is called via a trampoline and adapter.

The exit-hook then returns to the original caller of the functions. Figure 35 shows the output of the program including the input values of the `fibonacci` function and the results of modifying the return values.

This section demonstrated the basic usage of the library by walking through a simple example of instrumenting a `C++` function. It showed how to instrument a target function using entry-hook and exit-hook functions to interact with the function's execution flow. Figure 39 in the appendix shows the complete code for this example program.

6 Evaluation

This section presents the evaluation of the implemented instrumentation library, focusing on functional correctness and performance impact. The testing strategy focused on validating the libary's functionality and behavior against various inputs and edge cases, as discussed in Section 6.1. Additionally, Section 6.2 measures the performance overhead introduced by the instrumentation through a synthetic benchmark. Finally, we compare the LibHop library to Microsoft *Detours*, the de facto standard for machine code instrumentation on Windows. Together, these assessments provide a thorough understanding of the library's capabilities and its practical impact on application performance.

6.1 Testing

A proper testing strategy is a critical component of the development process to ensure trust in the correctness and reliability of the instrumentation library. For testing purposes, the development environment uses the *Google Test* framework [12]. The test approach includes unit tests for individual components, integration tests for core processes, and functional tests to validate the behavior of the entire instrumented functions. Tests were primarily conducted in an `x86_64` Linux environment.

6.1.1 Unit Tests

Small, isolated tests are used to check the correctness of the individual components and functions within LibHop. A table-driven approach is used for many of these tests. For each piece of code under test, a list of test cases is created. Each case has a specific input and a known correct output. These test cases are then executed in a loop, and the results are compared to the expected output. This approach allows for the easy addition of new test cases.

For example, one core component of the library that is tested this way is the *relative-to-absolute instruction translator*. Unit tests cover various tasks, such as determining whether an instruction is relative, i.e., whether it uses the IP register, or whether it is a conditional or unconditional relative jump. These tests also check that the input instruction is correctly translated into an absolute instruction.

Another critical set of unit tests focuses on the disassembly and copying of instructions. These tests receive a sample of machine code as input and check various aspects. For instance, they check that the initial instructions are disassembled correctly and that only the necessary instructions are relocated to the trampoline. To cover as many scenarios as possible, the tests utilize various instruction combinations.

6.1.2 Functionality Tests

In addition to testing individual components, the entire instrumentation process is tested to ensure that the library functions as intended. These tests cover the entire instrumentation process by instrumenting an actual `C++` function. Target functions range from simple arithmetic operations such as *addition* and *subtraction*, to comparison functions such as *minimum* and *maximum*, to more complex functions, such as *isPrime* and *fibonacci*.

The hooks do not perform any real actions. Instead, they simply log the entry and exit of the function. This step primarily serves to check that the instrumentation process itself is performed correctly and that the hooks are called as expected. These tests also check that the original function executes correctly and that no unexpected behavior or side effects occur.

The same way, the tests utilize hooks that modify the behavior of the target function by altering the `DataStruct` data. These tests check that the instrumentation applies the modified struct data to the target function and that the target function behaves as expected with the modified data. For instance, the *isPrime* function is instrumented and modified to always return *true* regardless of the input value. This test checks that the instrumentation correctly modifies the target's behavior and that it returns the expected result. Another example is a test that instruments the *subtraction* function in so that it always returns *zero*, regardless of the input values.

6.1.3 Edge Case Tests

Special attention is given to situations that could lead to undefined behavior or crashes. These edge case tests are crucial for ensuring the robustness of the instrumentation library. The following tests are examples of those tests used for edge cases that arose during the library's development.

- **Short Functions:** These are functions with a very limited size. Examples include empty functions and a function with only a few instructions. They are tested to ensure that instrumentation fails as expected instead of producing undefined behavior by overwriting more than the function's size.

  An additional version of the test is available, in which the user explicitly guarantees that the function is of a certain length. This guarantee is provided via the `guaranteedMinFunctionSize` member of the `Config` that is passed to the `instrument` function. If this optional value is set, the library trusts the specified size and ignores other length checks. The test checks that the advanced user feature works as expected.

- **Functions with Early Returns:** The size of a function is detected based on any encountered return instruction. However, if the return instruction is inside a conditional block that can be skipped, then the function has not ended. These tests ensure that different combinations of conditional blocks and early returns are handled correctly. This includes proper termination if the function is too short and proper continuation if the function is long enough to be instrumented despite any early returns.

Taking a systematic approach to testing with synthetic input and actual functions helps to build confidence in the correctness of the instrumentation library. Additionally, the tests demonstrated that the library can reliably instrument functions while ensuring predictable, consistent behavior.

6.2 Benchmarks

To evaluate the performance impact of this library, we conducted a synthetic benchmark designed to isolate the constant overhead of the instrumentation mechanism. In the benchmark, we first executed a target function in its original form, and then executed the same function with instrumentation applied. We deliberately implemented the entry-hook and exit-hook without any logic to ensure that the measured overhead reflects only the cost of the instrumentation itself. The following sections present the benchmark setup and the evaluation of the results

6.2.1 Benchmark Setup

The benchmark is implemented using Google's *Benchmark* library [13], which enables precise, statistically sound measurements of function execution time. The benchmark is designed to be run in a release build configuration with adaptions to prevent compiler over-optimization.

In Figure 36, lines 1 to line 5, shows the two hook functions used in the benchmark. The only action of the entry-hook is to return the address of the exit-hook. The target function, which starts in line 6, is a simple *power* function. The function starting in line 16 is a plain wrapper that calls the instrumentation only once. The instrumentation process itself is similar to the one shown in Section 5.

```
1  __attribute__((noinline)) void exitHook(hop::DataStruct *ds) {}
2
3  __attribute__((noinline)) void *entryHook(hop::DataStruct *ds) {
4    return (void *)exitHook;
5  }
6
7  __attribute__((noinline)) void int64_t power(int64_t b, int64_t e) {
8    int64_t result = 1;
9    for (int i = 0; i < e; ++i) {
10     result *= b;
11   }
12   return result;
13 }
14
15 std::once_flag instrumentOnceFlag;
16 __attribute__((noinline)) void instrument() {
17   hop::LibHop lh;
18   array:std::array scratchRegs{ hop::Register::R11 };
19   lh.instrument((void *)power,
20                 (void *)entryHook, { scratchRegs });
21 }
```

Figure 36: Hook functions and target function used in the benchmark.

Figure 37 shows the implementation of the benchmark, which consists of a `BM_original` and a `BM_instrumented` function. The first benchmark is for the original target function, which only calls the `power` function. The second benchmark instruments the target function once and then performs the same operation as the first benchmark. To avoid compiler optimization of the function calls, the call uses values from an array of integers and stores the result in a volatile variable.

```
std::array vals {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
static void BM_original(benchmark::State &state) {
  volatile int i = 0;
  for (auto _ : state) {
    benchmark::DoNotOptimize(power(vals[i % vals.size()], 3)); ++i;
  }
}

static void BM_instrumented(benchmark::State &state) {
  std::call_once(instrumentOnceFlag, instrument);
  volatile int i = 0;
  for (auto _ : state) {
    benchmark::DoNotOptimize(power(vals[i % vals.size()], 3)); ++i;
  }
}
```

Figure 37: Functions used to benchmark the original and instrumented target function.

To ensure statistically robust and precise overhead measurements, each benchmark was configured to execute 1'000'000 iterations per repetition, repeating this process 50 times. These settings were chosen to minimize the impact of noise and fluctuations in execution time, allowing for a more accurate assessment of the overhead introduced by the instrumentation. The complete benchmark source code is presented in Figure 40 in the appendix.

6.2.2 Benchmark Results

This section presents the results of the benchmark and attempts to evaluate the overhead introduced by the instrumentation. The benchmark was conducted on a system with an Intel Core i9-13950HX CPU operating at a maximum frequency of 5.5 GHz. The host OS was Ubuntu 24.04 LTS, running Linux kernel version 6.11.0. Because the benchmark workload is single-threaded and has minimal memory consumption, specifications such as CPU core count and total system memory are omitted as they are considered to have an insignificant influence.

To minimize the impact of the OS, the system was configured to run with the *performance* CPU governor, and unnecessary processes were stopped. The bench-

mark was started using the command shown in Figure 38. To avoid any context switches and to ensure that the benchmark runs on the came CPU core every time, the benchmark was pinned to the first CPU core.

```
$ taskset -c 0 ./benchmark
```

Figure 38: Command used to start the benchmark.

Table 1 shows the performance evaluation of the benchmark. The reported *Time* and *CPU* values represent the average for a single `fibonacci` iteration. The table provides a direct comparison of performance characteristics between the original baseline function and the instrumented version. This comparison is crucial for understanding the performance impact introduced by the instrumentation technique itself.

Benchmark	**Time**	**CPU**	**Repeats**
BM_original/iterations:1'000'000_mean	0.930 ns	0.930 ns	50
BM_original/iterations:1'000'000_median	0.921 ns	0.921 ns	50
BM_original/iterations:1'000'000_stddev	0.038 ns	0.038 ns	50
BM_original/iterations:1'000'000_cv	4.10 %	4.09 %	50
BM_instrumented/iterations:1'000'000_mean	15.8 ns	15.8 ns	50
BM_instrumented/iterations:1'000'000_median	15.7 ns	15.7 ns	50
BM_instrumented/iterations:1'000'000_stddev	0.613 ns	0.615 ns	50
BM_instrumented/iterations:1'000'000_cv	3.87 %	3.89 %	50

Table 1: Benchmark results of the original and instrumented target function.

The most notable difference between the two benchmarks is their execution time. The original function has a mean execution time of *0.930 ns* per iteration. This extremely low execution time is to be expected, since the function is simple, and the compiler can optimize it very effectively. In contrast, the instrumented version has a mean execution time of *15.8 ns* per iteration. The overhead comparison can be interpreted in the following ways:

- **Absolute Overhead:** The difference in mean execution times suggests an average overhead of approximately *15 ns*.
- **Relative Overhead:** The instrumented version is about *17 times slower* than the original. However, this value is not very meaningful because the original function is extremely fast, and the introduced overhead is not linear.

A direct comparison of the benchmark results highlights the following key aspects:

- **Inherent Costs of Function Calls:** The approximately 15 ns of overhead is primarily due to the inherent costs of jumps and function calls introduced by trampolines and adapter functions.
- **Sensitivity to Baseline Performance:** The 17-fold increase in execution time is a result of the extremely low execution time of the original function. The same 15 ns overhead would be negligible in a more complex function that takes milliseconds or seconds to execute.
- **Predictable Overhead:** Similar coefficients of variation suggest that instrumentation consistently adds the same amount time to each call rather than causing unpredictable performance.

In essence, this benchmark provides the essential data necessary for designing and applying the instrumentation system. It clearly defines the consistent, fundamental cost per hook, enabling users to make informed decisions about the trade-offs between performance and instrumentation needs. This foundational understanding is a crucial step toward effectively leveraging the instrumentation technique in real-world scenarios.

6.3 Comparative Evaluation with Existing Solutions

Microsoft *Detours* [14] is a prominent example of a widely adopted library to instrument machine code. This section qualitatively evaluates the developed LibHop library against Detours, focusing on key areas such as platform and architecture support, API design, and extensibility.

Support for multiple platforms and architectures is a critical aspect of any instrumentation library. According to its documentation, Microsoft Detours supports the more common architectures such as `x86`, `x86_64`, and `ARM`. Detours' most common application is the interception of *Win32* API calls. Although multi-architecture support is a strong point of Detours, it is typically deployed in the context of Windows API interception. The LibHop library, which was initially created to support `x86` and `x86_64` architectures on Linux and Windows, was designed to be OS-agnostic and cross-architecture-compatible. This design incorporates specialized frameworks, such as Capstone and AsmJit, to address the nuances of different architectures. This approach aims to simplify future extensions to other architectures and to enhance compatibility across various platforms.

The API design also reflects these different areas of focus. Detours, with its long history, offers a mature and powerful API that gives users precise control over instrumentation process. This often involves working directly with function pointers and

specific detour macros. In contrast, the LibHop library aims to provide a smoother user experience with a simple interface that handles common tasks internally. A key feature is its built-in ability to handle dynamically chosen exit-hooks. This feature enables flexible instrumentation with less user involvement than one might experience when piecing together such a feature using the fundamental building blocks provided by Detours.

In terms of extensibility and stability, Detours provides reliable functionality and stability within supported architectures, with a particular focus on Windows. Its internals are well-tailored to these environments. In contrast, the LibHop library approaches its extensibility by leveraging a more modular composition. Core tasks such as disassembly and JIT compilation are delegated to specialized external libraries. This design choice enables easier adaptation and extensibility such as support for new instruction sets or optimizations for specific architectures. This design paves the path for evolving the created library to address a wider range of instrumentation requirements as the need arises.

In summary, Detours is a powerful, multi-architecture solution that is well-suited for and widely used in the interception of function calls, such as the Win32 API on Windows. While LibHop shares core concepts such as trampolines and hooks, it is developed with a different focus. It prioritizes a simple API and foundational architecture to offer a generic, highly extensible `C++` library.

7 Summary and Future Work

This section concludes the work presented in this thesis. First, Section 7.1 summarizes the project's motivation, implementation, and achievements. Then, Section 7.2 explores potential enhancements and suggests directions for the library's future research and development.

7.1 Summary

This thesis addressed the inherent complexities of dynamic machine code instrumentation. The primary objective was to develop a generic `C++` library that facilitates platform-independent machine code instrumentation. The library aims to simplify the process of analyzing and modifying native program behavior at run time. It is particularly valuable for applications such as application monitoring, as applied by Dynatrace. Furthermore, the library establishes a foundation to support instrumentation of other natively compiled languages.

The core contribution is LibHop, a simple, user-friendly and extensible library. With this library, users can attach `C++` function hooks to the entry and exit points of any functions in a running process. The library uses trampolining to replace the start of the original function with an unconditional jump to the trampoline. The trampoline preserves the context, calls the entry-hook, executes the relocated instruction, and continues with the untouched part of the original function.

To accomplish this, the library uses Capstone to disassemble and analyze the initial instructions of the target function. For supported architectures, the library relocates variable-length instructions to the trampoline while ensuring semantic preservation. AsmJit generates the trampolines and other new instructions. In addition to the entry-hook, the library supports dynamic exit-hooks. This support is implemented by patching the return address on the stack, which redirects the control flow to the exit-trampoline.

The developed library abstracts away certain details of machine code manipulation and provides the necessary execution context for user hooks. However, interpreting the context, such as accessing specific local variables, remains the responsibility of the hook's logic. Although the focus of the library is on the `x86` architecture, it was designed with modularity in mind. This design choice will facilitate the addition of support for other architectures in the future.

In essence, this work provides a practical, reusable library that streamlines the process of instrumenting machine code during execution. This allows developers to create more sophisticated analysis and monitoring solutions. Furthermore, the library represents a significant step toward achieving broader language instrumentation capabilities.

7.2 Future Work

The current version of the library provides a solid foundation for instrumenting machine code. However, there are several areas that offer opportunities for enhancement and expansion, as well as addressing the known limitations presented in Section 4.5.

Future work could focus on the following aspects:

- **Additional Architecture Support:** The library currently supports `x86` in both 32-bit and 64-bit modes. Therefore, extending support to other architectures, such as `AArch64` and `RISC-V`, would significantly increase the library's applicability.
- **Expand OS Compatibility:** Currently, the OS support is limited to Linux and Windows. Supporting other OS platforms, such as macOS, would make the library more versatile and usable across different platforms.
- **GoLang Support:** Investigate solutions for GoLang's dynamic stack to identify potential interaction points with the runtime. Alternatively, use different methods for context passing.
- **Efficient Trampoline Management:** In the current version, a new trampoline is allocated for each instrumented function. This can lead to increased overhead because to three pages are used per trampoline. One possible research focus could be implementing trampoline pooling or compaction to combine multiple trampolines into a single memory block.
- **Full Instruction Coverage:** For `x86`, the library can transform only 24 out of more than a thousand instructions, which poses a significant limitation. However, there is potential to extend the library to support a broader set of instructions and extensions.
- **Improved Context Helpers:** Currently, the hook must interpret the contents of a `DataStruct`. Future work could focus on creating helper functions that allow the user to access specific data based on the underlying calling convention.

The above list is not exhaustive, but serves to illustrate the amount of potential future research directions and enhancements that can be made to the library. However, any improvement would significantly increase the library's utility, performance and scope, thus making it an even more effective library for machine code instrumentation.

8 Appendix

```cpp
#include <iostream>
#include <LibHop.h>

void exitHook(hop::DataStruct *ds) {
  ds->rax += 1;
}

void *entryHook(hop::DataStruct *ds) {
  std::cout << "Input: " << ds->rdi << std::endl;
  return (void *)exitHook;
}

int fibonacci(const int n) {
  if (n <= 1) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
  hop::LibHop lh;
  std::array scratchRegs{hop::Register::R11};
  lh.instrument(
    (void *)fibonacci,
    (void *)entryHook,
    {scratchRegs});

  int result = fibonacci(4);
  std::cout << "Result: " << result << std::endl;
  return 0;
}
```

Figure 39: Complete example program used in Section 5.

```
__attribute__((noinline)) void exitHook(hop::DataStruct *ds) {}
__attribute__((noinline)) void *entryHook(hop::DataStruct *ds) {
  return (void *)exitHook;
}
__attribute__((noinline)) void int64_t power(int64_t b, int64_t e) {
  int64_t result = 1;
  for (int i = 0; i < e; ++i) {
    result *= b;
  }
  return result;
}

std::once_flag instrumentOnceFlag;
__attribute__((noinline)) void instrument() {
  hop::LibHop lh;
  array:std::array scratchRegs{ hop::Register::R11 };
  lh.instrument((void *)power,
                (void *)entryHook, { scratchRegs });
}

std::array vals { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };
static void BM_original(benchmark::State &state) {
  volatile int i = 0;
  for (auto _ : state) {
    benchmark::DoNotOptimize(power(vals[i % vals.size()], 3)); ++i;
  }
}
static void BM_instrumented(benchmark::State &state) {
  std::call_once(instrumentOnceFlag, instrument);
  volatile int i = 0;
  for (auto _ : state) {
    benchmark::DoNotOptimize(power(vals[i % vals.size()], 3)); ++i;
  }
}

BENCHMARK(BM_original)->Iterations(1'000'000)->Repetitions(50)
  ->ReportAggregatesOnly(true);
BENCHMARK(BM_instrumented)->Iterations(1'000'000)->Repetitions(50)
  ->ReportAggregatesOnly(true);
BENCHMARK_MAIN();
```

Figure 40: Full source code of the benchmark used in Section 6.2.

- **Relative (Conditional) Jumps:** `jmp`, `je`, `jne`, `ja`, `jae`, `jb`, `jbe`, `jg`, `jge`, `jl`, `lje`, `jo`, `jno`, `js`, `jns`, `jp`, `jnp`, `jecxz`
- **IP Relative:** `mov`, `movabs`, `sub`, `add`, `cmp`, `lea`

Figure 41: List of all supported instructions that can be transformed.

List of Figures

1 Comparison of move and push instructions for `x86_64` and `AArch64`. 3
2 Example of $\frac{a+b}{2}$ for `x86_64` in Intel syntax. 4
3 Example of `add(int a, int b)` using the System V ABI. 5
4 Example function in Python without instrumentation. 6
5 Example function in Python with basic execution time measurement instrumentation. 7
6 Example of using the Capstone framework to disassemble `x86_64` machine code. 9
7 Output of the Capstone example code. 9
8 Enable detailed information in Capstone. 10
9 Example for a check if an instruction is in a specific group. 10
10 Example of using the AsmJit library to generate some `x86_64` machine code. 11
11 Interface of the library. 12
12 Overview of the instrumentation process. 12
13 Overview of an instrumentation run. 14
14 Interface of the library. 15
15 Disassembly of the target function. 16
16 Disassembly of the target function that causes an instrumentation abortion due to short length. 17
17 Disassembly of the target function that has an early return in a branch. 18
18 Overview of the trampoline to generate. 19
19 Memory page allocation with guard pages on Linux. 20
20 Reference `DataStruct` for `x86_64`. 20
21 Stack allocation and filling of `DataStruct`. 21
22 Call of the entry-hook using the System V ABI. 22
23 Dismantle of the `DataStruct` to registers and deallocation. 22
24 Example of a transformed instruction. 23
25 Relocation of original instructions. 23
26 Transformation of relative `jmp` instruction. 24
27 Transformation of conditional jump instruction. 25
28 Outlines of the exit-trampoline. 26
29 Storage of addresses in TLS stack. 27
30 Adapter function between entry-trampoline and entry-hook. 28
31 Adapter function between exit-trampoline and exit-hook. 29
32 Example program used as a base for the explanation of the usage. . . 31
33 Instrumentation call to instrument the function `fibonacci`. 31

34 Entry-hook and exit-hook function for the instrumentation of `fibonacci`. 32
35 Output of the example program. 32
36 Hook functions and target function used in the benchmark. 37
37 Functions used to benchmark the original and instrumented target function. 38
38 Command used to start the benchmark. 39
39 Complete example program used in Section 5. 44
40 Full source code of the benchmark used in Section 6.2. 45
41 List of all supported instructions that can be transformed. 46

List of Tables

1 Benchmark results of the original and instrumented target function. . 39

References

[1] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on 06/09/2025).

[2] H.J. Lu et al. *System V Application Binary Interface*. AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models). 2025. URL: https://gitlab.com/x86-psABIs/x86-64-ABI (visited on 06/09/2025).

[3] Microsoft Corporation. *x64 calling convention*. 2025. URL: https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention (visited on 06/09/2025).

[4] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. "Gprof: A call graph execution profiler". In: *SIGPLAN Not.* 17.6 (1982), pp. 120–126. ISSN: 0362-1340. DOI: 10.1145/872726.806987. URL: https://doi.org/10.1145/872726.806987.

[5] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection - For GCC version 15.1.0*. Free Software Foundation, Inc. 2025. URL: https://gcc.gnu.org/onlinedocs/gcc-15.1.0/gcc.pdf (visited on 06/09/2025).

[6] Lucas Leong. *pe-afl*. 2019. URL: https://github.com/wmliang/pe-afl (visited on 06/09/2025).

[7] Chi-Keung Luk et al. "Pin: building customized program analysis tools with dynamic instrumentation". In: PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005. ISBN: 1595930566.

[8] Intel Corporation. *Pin – A Dynamic Binary Instrumentation Tool*. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html (visited on 06/09/2025).

[9] Anh Quynh Nguyen. *Capstone – The Ultimate Disassembler*. 2013. URL: https://www.capstone-engine.org/ (visited on 06/09/2025).

[10] Anh Quynh Nguyen. *Keystone – The Ultimate Assembler*. 2016. URL: https://www.keystone-engine.org/ (visited on 06/09/2025).

[11] Petr Kobalíček. *AsmJit – Low-Latency Machine Code Generation*. 2024. URL: https://asmjit.com/ (visited on 06/09/2025).

[12] Google LLC. *GoogleTest – Google Testing and Mocking Framework*. 2008. URL: https://github.com/google/googletest (visited on 06/09/2025).

[13] Google LLC. *Benchmark – A microbenchmark support library*. 2013. URL: `https://github.com/google/benchmark` (visited on 06/09/2025).

[14] Microsoft Corporation. 2016. URL: `https://github.com/microsoft/Detours` (visited on 06/09/2025).

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.