Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Elena Falkenberg Nordmark

The fundamentals of debuggers and the challenges of Reverse Debugging

Bachelor's project in Computer Science
Supervisor: Ole Christian Eidheim
May 2019

NTNU
Kunnskap for en bedre verden

Elena Falkenberg Nordmark

The fundamentals of debuggers and the challenges of Reverse Debugging

Bachelor's project in Computer Science
Supervisor: Ole Christian Eidheim
May 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Sammendrag av Bacheloroppgaven

Tittel:	**Det fundamentale ved debuggere og utfordringene med Revers Debugging**
Oppgavenummer:	118
Dato:	17.05.2019
Forfatter:	Elena Falkenberg Nordmark
Veileder:	Ole Christian Eidheim
Oppdragsgiver:	NTNU IDI AIT
Nøkkelord:	Avhandling, Bachelor, Debugging, Reverse debugging, Backwards debugging, Software
Antall sider:	15
Tilgjengelighet:	Åpen kildekode iht. lisens GPLv3

Sammendrag:

Per i dag er ikke begrepet *revers debugging* velkjent. Formålet med denne oppgaven er å undersøke hvorfor det er slik, hvordan debuggere kan bevege seg baklengs, samt hvorfor det ikke er bedre implementert. I tillegg vil prosjektet dokumentere opprettelsen av en simpel debugger.

Summary of Bachelor project

Title:	**The fundamentals of debuggers and the challenges of Reverse Debugging**
Project number:	118
Date:	17.05.2019
Author:	Elena Falkenberg Nordmark
Supervisor:	Ole Christian Eidheim
Employer:	NTNU IDI AIT
Keywords:	Thesis, Bachelor, Debugging, Reverse debugging, Backwards debugging, Software
Pages:	15
Availability:	Open source licensed under GPLv3

Abstract: As of today the term *reverse debugging* is not widely known. The purpose of this project is to research why that is, how debuggers can run backwards, as well as why they are not better implemented. In addition the project will document the making of a simple debugger.

Foreword

This thesis is in a sense a continuation of an assignment I worked on during the autumn of 2018, in the subject *TDAT3023 3D-grafikk med prosjekt*. The assignment then was to visualize information given by the debugger, using liblldb. During a conversation with my teacher, Ole Christian Eidheim, he mentioned the concept of reverse debugging and that it could make for an interesting subject to research. I, having gotten a taste of debuggers, found it interesting and decided to continue down that road. Throughout my work on the thesis, Eidheim has been a good help as my dedicated supervisor from NTNU.

I would like to thank Erlend Tobiassen for all the support, from late night quesadillas to helping me debug my debugger. You have a way of explaining foreign concepts in ways that strangely make sense.

Thanks to Heather Nieto for dealing with my questions involving grammar and sentence structure, for proofreading the final thesis, and for motivating me to finish the project in time.

Trondheim, 17.05.2019

Elena Falkenberg Nordmark

Mission statement

When debugging a process, a debugger moves forward either by running to a breakpoint or stepping through the code one line at a time. If one were to pass the line of interest, one would typically have to restart the process in order to get back to the part that was already run. Perhaps if there was a way to step backwards, one could avoid restarting the entire process.

The original description of the assignment, translated from norwegian.

> "The purpose of the assignment: The student will research how to best move backwards in the code during debugging. (…) Currently reverse debugging is not widespread, so in order to see earlier events in the program to be debugged, one has to start the process anew. The intention of reverse debugging is to avoid starting this process all over again, and instead jump directly backwards in the code. Because this is not yet widespread, it is interesting to research the best possible way to do this."

As the project started up, the focus remained on reverse debugging, but the question changed to: if reverse debugging is possible, why is it not widely implemented? Part of the assignment became researching reverse debugging, while the other part became a software development process; creating a basic debugger. The purpose of the development being a learning process in order to have an informed base understanding for the research part.

Summary

Debugging is a vital part of most software development processes, and a lot of time is spent looking for and trying to solve bugs. If there is a possibility for speeding up the bug finding process, it should be explored. In this paper I take a closer look at what reverse debugging is and what it is capable of, exploring how it can be implemented and attempt to answer why it is not better implemented on a wider scale.

In order to understand reverse debugging, one must first understand what debugging is. A significant portion of this paper is therefore dedicated to explaining the fundamental functionality of debuggers, as well as describing the work done in attempting to make a debugger from scratch, mainly using built-in functions from Linux with some help from libdwarf.

Some of the challenges of reverse debugging are how to record information about the debuggee, and how to store this data. Programs can run differently from one time to another, depending on factors like user input and file readings. It can prove challenging to deal with the complexity of some debuggees, but it is possible. If one can provide enough resources, computer power and memory space, reverse debugging can be done.

Uncertainty when it comes to when reverse debugging is worth it seems to be the reason why it is not widely implemented. When the process is expensive, it is a matter of comparing the cost of the resources to the cost of the time spent debugging. In some cases it might be worth it, while in others it might not. When reverse debugging is worth it is not covered in this paper but could be an interesting topic for future research.

Table of contents

Abbreviations/symbols

NTNU	Norwegian University of Science and Technology
OS	Operating System
ptrace	Process trace
RIP	64-bit Instruction Pointer
API	Application Programming Interface
JVM	Java Virtual Machine
ODB	Omniscient Debugger
UI	User Interface
CPU	Computer Processing Unit

Chapter 1: Introduction and relevance

Debuggers are very helpful tools for developers when writing code. A lot of the time it speeds up the development process, allowing the developers to take a closer look at specific parts of the code, examining it with more information than they would have by simply looking at it. Not only does it provide additional information, but it is useful for finding bugs, hence the name debugger.

1.1 Subject of research

The way most debuggers work as of today is by either running up to a breakpoint and stopping there, or by single stepping to the next line in the code. However, both options only move one way; forwards. If one were to run the program to the point of interest, but accidentally went one line too far, there would be no way for the non-reverse debugger to step one line back. In most cases, the whole process must be started over again. This is surprising in several ways:

- One would expect more from a tool that is powerful enough to access and change registers during run-time.
- It is tedious and time consuming to have to restart the whole process.
- All the information about what happened on the previous line is already there, because it already ran.
- Logically, looking at a real-life situation, it would make more sense to reverse a car than to circle around to try again. If you missed your exit down a one-way street you would rather back up than have to drive back around.

With all of this in mind, why is backwards, or reverse, debugging not already existing and implemented in all debuggers? The short answer is that it already exists, one being the Omniscient Debugger (Lewis 2003)[i]. However, it is not widely known nor implemented. During my research I have decided that to better understand how reverse debugging would be done, I would need a fundamental understanding of regular debuggers. To get this understanding I will be making my own basic debugger in C++. In this paper I will take a closer look at how debuggers work, making my own from scratch, ultimately researching the difficulties surrounding reverse debugging and why it is not widely implemented.

1.2 Thesis structure

This paper contains six main chapters, the first one being the introduction where the importance of debuggers is stressed and the topic question is defined, laying the groundwork for what to research.

1.2.1 Theory

The theory part of the paper is large and goes in-depth about how general debuggers work. Starting off with the basics: what a debugger does and its functionality, along with describing ptrace, an important tool for working with debuggers on Linux. The chapter continues with explaining how the functionality of single-step and breakpoints work, and their significance in making a debugger the powerful tool it is. Towards the end of the

chapter the paper goes in-depth about functionality and importance of debugging information and symbols.

1.2.2 Choice of technology and method

There are many choices to be made when it comes to which technology and methods to use when doing extensive research, and these choices are better explained in this chapter. A large part of it is dedicated to system architecture and choice of technology revolving my own creation, the lnaDB. The later part of the chapter is dedicated to the development process of making a debugger from scratch.

1.2.3 Results

The chapter starts with a section devoted to reverse debugging, the possibilities for how it can be done and the challenges that come with it. It aims to answer the question asked in the introduction; why reverse debugging is not widely implemented. After comes a section dedicated to the engineering process of the work, and lastly the administrative results of this paper.

1.2.4 Discussion

The previous chapter focused on the result of the research, the development and the work process. This chapter is about how the results came to be; why things worked out and why they did not, as well as alternative possibilities to the approaches.

1.2.5 Conclusion and future work

The chapter aims to answer the original question that was asked; why reverse debugging is not widely implemented. The conclusion is based on the results that were discovered and discussed through the thesis. The chapter ends with a section dedicated to work that can be done in the future based on the work and research in this thesis.

Chapter 2: Theory

A debugger is a process that either launches along with a program or is attached to it while running. The corresponding program is referred to as the *debuggee*, as it is the one being debugged. Primarily the debugger's job is to examine the call traces and variables in the debuggee. It has the ability to step through every line of code, *single-step*, or jump to a certain line using *breakpoints*. Advanced debuggers are able to execute and call functions in the debuggee, in addition to changing the code during runtime to see immediate changes.

2.1 ptrace

ptrace is a vital mechanism used in Linux. It allows for a parent process to observe and control the execution of a child process, enabling it to change its core image and registers (Padala 2002)[ii].

2.2 Single-step

Single-step is the process of stepping through a process one code line at a time. ptrace has its own call for this; PTRACE_SINGLESTEP. Note that this function of ptrace handles one line of assembly code at a time. That means it only does one simple instruction before stopping. In the case of debugging a process written in another language, for instance in C++, this single-step method would not actually step one line at a time. One line in the C++ code might translate to several instructions, meaning that in some cases it can take several single-steps to get through a single line of code.

Typically, on start-up, the first thing the debuggee does is send a request to the OS kernel, asking it to allow its debugger to trace it (Bendersky 2011)[iii]. On Linux, this is done using PTRACE_TRACEME. This leads to the debuggee stopping after receiving any signal except SIGKILL, and the debugger is notified with a wait() call. Any calls to exec() by this process causes a SIGTRAP, allowing the debugger to take back control before execution. Once something interesting happens, wait() will return.

The request PTRACE_SINGLESTEP tells the OS to continue the debuggee, but to stop it after it executes the next instruction. This would typically continue in a loop, allowing a user to keep single-stepping.

2.3 Breakpoints

In larger projects, where there are many lines of code, it is no longer ideal to look at each line of code individually. The use of breakpoints allows the debugger to tell the debuggee to stop at a certain instruction and wait for further instructions, meaning the program will run as normal up to that point. From then on, the user might want to go back to single-stepping so they can inspect the code more closely.

2.3.1 Soft interrupt

A standard mechanism in operating systems are system calls. They allow easy access to underlying hardware and low-level services using a standard API. Soft interrupt is an instruction in the x86 instruction set. A process can invoke a system call by putting its arguments in registers and calling a soft interrupt (Linux 2005)[iv]. An interrupt is simply a signal sent to the OS, telling it that an event has occurred, resulting in changes when it comes to the sequence of instructions that the CPU executes. The soft interrupt causes

the CPU to switch from user mode into kernel mode. It is necessary because kernel mode has root access, allowing access to any memory space and resource on the system. User mode is restored after the kernel has satisfied whichever request that was made.

2.3.2 The trap in the breakpoint

A breakpoint is made by inserting a trap instruction, the byte 0xCC, at a given address in the memory of the process. For an instruction to simply be placed like that, it needs to replace whatever that already has this address. Therefore, it is necessary to save the original instruction at the given address before placing the trap instruction. When the debuggee hits the instruction on a given RIP, it will execute the trap and the RIP will increase by one, meaning it is ready to execute the next line upon continuing.

When it is time to continue after the breakpoint, it is vital that the breakpoint line is restored and executed properly before continuing. This is done by removing the trap instruction, replacing it with the original data that was temporarily saved elsewhere. Then the RIP must be subtracted by one, moving back to the previous line so it will in fact execute the correct instruction. When continuing, for instance by using PTRACE_CONT, it will execute the breakpoint line as if it never stopped.

In the case of wanting the breakpoint to remain at the given instruction, it is still necessary to remove it so the instruction on the line can execute the way it was intended. Commonly one would remove the breakpoint, single-step the line, then place the breakpoint back again before continuing.

2.4 Debugging information and symbols

Debugging information is a collection of information about a compiled binary program (IBM Knowledge Center n.d.)[v]. Typically, these collections map instructions from the compiled binary program to its corresponding item in the source code. Such information could be, among others, function or variable names and addresses, data types, local variable scopes, and line numbers. It describes the application in a way that is more understandable and easily accessible than the underlying assembly instructions.

2.4.1 Debugging flags

For a program to be subject to debugging, it must be compiled along with a debugging information flag. This flag is -g in gcc, and -gdwarf for DWARF, to name a few. This flag lets the compiler generate a symbol table alongside the machine code. The symbol table is a map of the debugging information.

2.4.2 Importance

The original location of where information is saved after compilation varies a lot. At times a variable may be in the stack, other times in the registers, depending on how the code was compiled, system settings and optimizations. Keeping track of where everything is located can be tricky, and that is what the symbol table is for. When trying to reach a certain function or variable, one would only need to look it up in the symbol table containing the debug information in order to find its memory address. It is therefore key when attempting to debug a program.

Compiled executable files will usually not contain debugging information. This is because the debug symbol table takes up extra space (Beekmans 2003)[vi], in addition to being a helpful tool for decompiling. The symbol tables are unique for each version of the

program and would have to be generated anew in the case of changes in the source code.

Chapter 3: Choice of technology and method

In order to make my own debugger from scratch, I aimed at initially making it as simple as possible. This includes using as few external libraries as possible, which in turn led me to use the built-in power tool in Linux; ptrace. The ability to work directly with registers, reading and changing them easily has proved invaluable.

3.1 libdwarf

There is only so much one can get done before using outside libraries. It proved to be very difficult finding the memory address of a function programmatically. Therefore, I decided to use libdwarf; compiling the debuggee with the -gdwarf flag to enable DWARF debugging symbols, then accessing the symbols programmatically using libdwarf.

3.2 Architecture

The lnaDB consists of three classes. The classes are debugger, breakpoint and dwarf_function_information. The latter class is only used for storing the name and address of a function found in the debugging symbols, and it contains no member functions. I decided to make this class because it proved necessary to bundle the function name and address together, and to be able to pass them as arguments across functions in the debugger class.

The breakpoint class is used for storing data and the address of the breakpoint. I found it useful to have this information stored in an object, so that a breakpoint can exist regardless of it being enabled or disabled.

The debugger class is the largest one. It has no data members, but a lot of member functions. Some of them are run functions, functionality for single-stepping and running to a breakpoint, functions for enabling and disabling breakpoints, and continuing from a breakpoint.

The initial goals of the self-made debugger were to be able to single-step through the code, and to show the information located at each instruction. While this proved to work well with assembly files, there were difficulties when it came to debugging C++ files. While in assembly, one line is one instruction, there can be several instructions on one line of C++ code. Stepping through multiple lines of C++ code hoping to find the relevant one was tedious and led to the need for implementing breakpoints.

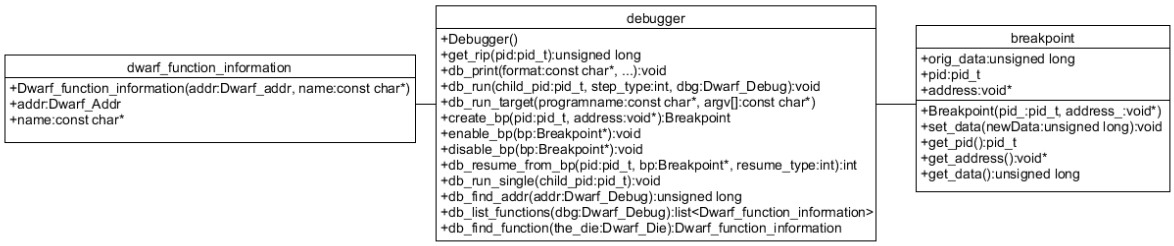

Figure 1.1 – Class diagram

3.3 Development process

At the start-up meeting with the university councillor, goals for the project were set. We determined that in order to learn more about reverse debugging, developing an own debugger from scratch would make a good fundamental understanding of how one would go about implementing a reverse debugger. The purpose of the process was learning about debugging development, while the end goal was a debugger where reverse debugging can be implemented. There were few specifications set at the time, the reason being that the development process is a learning process. The purpose was to learn what needed to be part of a debugger. Because of this, goals were set from meeting to meeting as I discovered which direction I wanted my debugger to go in, and what it would require to get there.

The development process can be described as iterative. On each meeting we set goals for next time. Goals could be "Learn what debugging symbols are" or "Implement single-stepping in your own debugger". Between meetings I would work on the goals, either succeeding and continuing with the next goal, or failing and learning why that was. At the next meeting we would go over what was done, discuss why certain things would work or not, and set the goals for next time, starting the process over again.

The interaction between the user and the debugger is solely through the console. UI was never a goal or a priority, so it was not developed. With more time a proper UI could be developed, but it did not seem necessary with the basic functionality the debugger has.

Chapter 4: Results

4.1 Scientific

The current known method for a debugger to step back in the code, is to have it somehow record every state change in the program being run (Lewis 2003). It cannot simply execute instructions in the opposite order of which they happened, as some instructions are irreversible; they destroy information. The way information is recorded and stored can vary a lot, and each come with their respective challenges.

4.1.1 Challenges

Determinism is the philosophy that the outcome of all events is already decided. From a software engineering perspective, there are multiple things that can change the outcome of a program from one execution to another without any changes in the original code, rendering the code non-deterministic. Factors that can affect the outcome, so-called non-deterministic factors, are among others: user input, data readings from files and multi-threading. These factors impact reverse debugging because they add an element of uncertainty. It was earlier stated that reverse debugging is possible because the code was already executed, so there is knowledge of what happened. But if the outcome changes from one execution to another, the statement no longer holds true. One cannot with certainty step back, because there is no knowing if the same execution would happen again. Non-deterministic factors cause programs to be complex and makes reverse debugging difficult.

Most possible solutions of reverse debugging contain some way of recording and saving instructions while the program runs. In some cases, the information is stored locally, other times externally; in a different process or on an entirely different hardware unit. Either way there are limitations about how much data can be stored. It is possible to record every single instruction a program does, but at the cost of a lot of memory space dedicated to saving information that might not be used at all. With choosing to save less of the data it comes at the cost of less precision. When reversing, there might be cases where the debugger does not have all the data it would require to showing what really happened. The matter of scalability is a choice between precision and space.

Running a complex program has its limitations when it comes to processing. When, as well as running the process, you run a process on top of that one attempting to debug it, more resources are demanded in terms of processing power and memory. A debugger capturing the traces and processing them, which is necessary for reversing, has a larger overhead than a simple one which does not record tracing. Minimizing overhead in an attempt to improve efficiency can prove challenging.

4.1.2 Possibilities

ODB has its own way of recording information in Java (Lewis 2003)[vii]. ODB records this information by obtaining the execution traces as they are loaded by the JVM, saving each instruction that JVM will execute. The data is then stored in the limited heap space of the JVM. There is no garbage collection due to references to objects being kept, despite no longer being in use. The references are kept because it is difficult to tell which objects will be necessary for reversing. To provide some garbage collection, a method called

lexical scoping is used. It allows for certain classes to be marked as trusted, meaning they are excluded from contributing to the recorded execution trace. This is an attempt to decrease the memory usage of the recorded information.

Chronicle has solved the issues of storing the execution traces in the heap by sending the data to an out-of-process disk-based database, as to not get limited by the available space (Google and Novell 2007)[viii]. In a similar manner, there is the case of embedded systems that have platforms with specialized hardware that is able to capture the trace data (Green 2019)[ix]. This does not incur any runtime overhead.

Replay-based debugging is the concept of recording non-deterministic operations like user-input when running, instead of recording every single instruction. Not needing to record every single instruction lowers the amount of runtime overhead by a lot. After recording, it will then replay the program the exact same way it was originally executed, using the data it saved from the non-deterministic operations, allowing the debugger to step wherever it would like without worrying about the complexity of the program, essentially making the replay deterministic. The way this type of debugger can allow partial reversing is by taking snapshots of the program during runtime, allowing it to save its state at a given time. This allows for jumping back and forth between specific places in the code. The navigation would however be slower than the previous alternatives as parts of the program would have to be replayed (Pothier 2011)[x].

4.1.3 Alternative

Instead of complete reverse debugging, lightweight options are more commonly used. An alternative is Eclipse's functionality called "drop to frame". It jumps back to the start of a function, resetting all the parameters and not executing finally-blocks (Eclipse n.d.)[xi].

4.2 Engineering

The purpose of developing an own debugger was to learn the fundamentals of debugging so that I could more easily understand the reverse debugger technology based on that. The goal was to create a simple debugger that has the potential of having a reverse debugger implemented. While I could not implement reverse debugging, I did implement traditional debugger functionality like single-stepping and setting, disabling and continuing from breakpoints.

The result of the development is a debugger that can trace another program, taking input from the user to decide how to continue. The user can make the debugger single-step, continue running, or set a breakpoint at a function given the function name. In the case of single-stepping and running to breakpoints, the debugger will execute the instruction and let the user know, asking how to continue from there. All the communication back and forth between the user and debugger happens in the console. UI was not a goal as I started the project, and there was no extra time to develop it.

4.3 Administrative

There were multiple deviations from the original plan when it comes to when things were expected to be finished. Sometimes I would encounter problems in the development section of my work, like issues with libdwarf not linking properly with my project. This is where one of the perks of working on a two-part project really shines. While I was stuck and in need of advice with libdwarf, I could work on the research part of the work, so time was not going to waste. Similarly, I got stuck on not understanding some aspects of breakpoints as I was researching reverse debugging. With some more work on the self-

made debugger I was able to clarify and better understand the concepts, allowing me to return to the research.

As I started the project, I expected the first half of my time to be dedicated to development, and the second half dedicated to research. Instead I worked on them in parallel, learning from one part and taking advantage of that knowledge, using it with the other part. Because I worked on them simultaneously, I did not complete the software in the time I thought it would take.

The goal of having meetings with the university councillor every two or three weeks did not go as intended. The reason being that at times I had not yet completed the goals we set at the previous meeting, and I was aware of what I needed to do and how to do it, so there was no need for more frequent meetings.

Illness caused some shifting in the number of hours worked some weeks compared to the number of hours expected. More about this can be seen in the schedule attachment.

Chapter 5: Discussion

5.1 Scientific

I expected complexity to be the reason that reverse debugging is not more widely implemented. It seems like a good way to improve development processes by speeding them up, so surely the reason not to be doing it would come down to the lack of technology. It was surprising to me to find that the reason is not the lack of technology, it is the lack of resources, or rather the lack of will to deploy those resources.

From the results about reverse debugging it is clear that when the references were published, reverse debugging seemed like a costly matter when it came to computer power and memory space. However, the world of computers is constantly evolving, and things that seemed resource demanding a few years ago is potentially not as bad as it seems with today's modern CPU-s. With more time and extensive research concrete numbers on how much slower a reverse debugger working on a modern CPU would be compared to a regular debugger. Perhaps the result of it will be that it is still very costly, but there is the possibility that it can be worth doing. Regardless of it being worth it at this time, the work could have importance for the future where our technology might be good enough to maintain the cost.

Research shows that debugging costs the global software industry a lot of money every year (Undo 2019)[xii]. Speeding up this process with reverse debugging has the potential of saving a lot of time, and potentially money. The economical advantage comes down to weighing the cost of memory space and computer power against the cost of time spent looking for bugs.

5.2 Engineering

There is more functionality that I would like for lnaDB to have, that was not made in time. The main functionality that I would like is printouts of values as they change. For instance, showing `a = 5` on one line, and after single-stepping to the next line it would show `a = 8`. Being able to witness the change in variables as one steps through the code is a vital part of debugging. The reason this was not implemented was due to the lack of time. After some work on the project it became clear to me that in order to implement that functionality, I would need to access the debugging symbols. Because of the need to find a function address dynamically, I was already learning to access the debugging symbols. This took a lot of time due to problems accessing libdwarf and learning how to use the library. The dynamic addresses functionality was the last functionality I was able to implement, unfortunately at the cost of printouts.

Other functionality that the lnaDB ideally would have is an improved single-step. The way single-step works at the moment is that it will step to the next instruction being done. If the debuggee is an assembly file, this is very straightforward. However, on a C++ file where there can be multiple instructions on one line, single-step actually takes multiple steps to get through one line of code. Ideally there would be functionality to better support stepping between actual lines instead of instructions. This would require making an own single-step function instead of using the existing functionality in ptrace. A

possibility is to extend the use of libdwarf to set breakpoints at a certain line in the code, as well as setting them at function names. A single-line-step functionality would then set a breakpoint on line `n + 1` and run to the next breakpoint.

5.3 Administrative

The choice of alternating between the two parts of this work, researching reverse debugging and creating an own debugger, turned out to be a good thing. It was not according to the plan, but it seemed like a more efficient approach after starting the project. Creating an own debugger was not strictly necessary to understand reverse debugging, but it did give me a fundamental understanding of how debuggers work and that was an advantage when researching. There are other ways of obtaining a similar understanding, like reading about how other people have made their debuggers.

A schedule is of high importance when working alone on a project like this, because there is no one else to motivate and push you to do your work. When working in groups one might feel compelled to work out of obligation for the others, while when working alone one is only responsible for oneself. Working alone has its perks; one is the freedom to work whenever and wherever. In my case the freedom allowed me to work from home and not have to spend time and money going to the university or some office to work at. However, it has also led to some sporadic work hours. Some days I did not do any work at all, while other days I worked all day. Overall the workload has evened out, and I feel content with the amount of work I put in. Towards the end of the project a lot of extra time was put in, in order to finalize the project and be able to publish a version of it that I am happy with.

Typically, I mostly enjoy front-end development, actively visualizing the result of my work. To go from that to working on a low-level project like this has been a challenge. It has broadened my knowledge of computer programming, and I think working with something different than what I usually do has been a good thing. Getting out of my comfort zone in order to have a wider field of experience has been rewarding.

During my work there has been little to no ethical reflection revolving the work. To me it has not seemed like this technology could be morally challenged in any way. I am unable to see any negative outcomes of neither my work process nor the technology of reverse debugging. If anything, reverse debugging could be a good thing as it would speed up development processes, allowing programmers to work faster and in turn get more done.

Chapter 6: Conclusion and future work

Reverse debugging is possible, but it comes with its challenges. It has a huge potential for speeding up software development processes, so if done correctly, using reverse debugging can be an advantage. The main challenges when it comes to reversing are efficiency, scalability and complexity.

What remains to be answered is why it is not widely implemented. Looking at the challenges described in 4.1.1, they tend to go hand in hand. For instance, if one decides to reduce overhead, it could result in a less precise debugging experience. Alternatively, if one decides to record everything, it would improve the precision, but at the cost of a lot of memory space and computer power.

For reverse debugging to be viable it would take someone being willing to invest the time to set up the debugging environment properly for their system, along with dedicating the necessary memory and power. In some cases, it would not be worth all that work, and it is limited by access to a lot of resources. The technology is there, but because of the high requirements it is not widely used.

For the future it would be interesting to look at the concrete numbers for how much memory and power is needed to do reverse debugging and compare them to what modern CPU-s are capable of handling. Potentially one could attempt to find the threshold for when it is worth doing.

When it comes to lnaDB there is a lot of functionality that can be added, but as it stands it is a simple, functional debugger, capable of both single-stepping and running to breakpoints.

References

[i] Lewis, Bill. 2003. "Debugging Backwards in Time."
https://arxiv.org/abs/cs/0310016
last visited 16/5/2019

[ii] Padala, Pradeep. 2002. "Playing with ptrace, Part I"
https://www.linuxjournal.com/article/6100?page=0,1
last visited 17/5/2019

[iii] Bendersky, Eli. 2011. "How debuggers work: Part 1 – Basics"
https://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1
last visited 13/5/2019

[iv] The Linux Information Project. 2005. "int 0x80 Definition"
http://www.linfo.org/int_0x80.html
last visited 17/5/2019

[v] IBM Knowledge Center. n.d. "Debug information"
https://www.ibm.com/support/knowledgecenter/en/SSB23S_1.1.0.15/gtpd3/d3cinfo.html
last visited 17/5/2019

[vi] Beekmans, Gerard. 2003. "About debugging symbols"
http://www.iitk.ac.in/LDP/LDP/lfs/5.0/html/chapter06/aboutdebug.html
last visited 17/5/2019

[viii] Google, and Novell. 2007. "chronicle-recorder"
https://code.google.com/archive/p/chronicle-recorder
last visited 17/5/2019

[ix] Green Hills Software. 2019. "TimeMachine Debugging Suite"
https://www.ghs.com/products/timemachine.html
last visited 13/5/2019

[x] Pothier, Guillaume. 2011. "Towards practical omiscient debugging"
http://www.tesis.uchile.cl/tesis/uchile/2011/cf-pothier_g/pdfAmont/cf-pothier_g.pdf
last visited 13/5/2019

[xi] Eclipse. n.d. *help.eclipse.* "Drop to frame"
https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fviews%2Fdebug%2Fref-droptoframe.htm
last visited 17/5/2019

[xii] *Undo. 2019.* "What is Reverse Debugging, and why do we need it?"
https://undo.io/resources/reverse-debugging-whitepaper/
last visited 17/5/2019

NTNU
Kunnskap for en bedre verden