
FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

**Nuutti Räihä**

# METHODS TO IMPROVE DEBUG FLOW FOR INTELLECTUAL PROPERTY PROTECTION

Master's Thesis
Degree Programme in Computer Science and Engineering
September 2022

**Räihä N. (2022) Methods to Improve Debug Flow for Intellectual Property Protection.** University of Oulu, Degree Programme in Computer Science and Engineering, 80 p.

# ABSTRACT

**Every company wants to protect their intellectual property and limit customer visibility of confidential information. A company may protect its proprietary information by different ways. This thesis will compare different methods that try to protect intellectual property while maintaining the software debugging capability.**

**Working with binary libraries without debug information makes customer support very difficult. When a company is developing a new product, time to market is important. Usually, the last months are very busy resolving urgent customer issues. Especially during this period, the slow process of debugging customer issues without debug information can cause delays and increase time to market.**

**The goal of this thesis is to compare methods that protects intellectual property by making reverse engineering more difficult. Study of the upcoming GNU Compiler Collection (GCC) features related to debug data formats, such as DWARF5, is also carried out while working with the thesis.**

**The approaches tried were split DWARF, injecting ELF files, stripping debug data, and code obfuscation. Also optimisation and their effect on disassembly was studied. The best solution was to compile the software with debug symbols and strip them to a separate file. This way the symbol data can be loaded separately into GDB. The symbol data layout and addresses are also always correct with the solution.**

**Keywords: compilers, debugging, reverse engineering, DWARF, obfuscation**

**Räihä N. (2022) Virheiden etsinnän työnkulun parantaminen immateriaaliomaisuudet huomioiden.** Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 80 s.

# TIIVISTELMÄ

**Yritykset haluavat suojella immateriaaliomaisuuksiaan ja rajoittaa asiakkaiden näkyvyyttä tietylle tasolle asti. Tämä lopputyö vertailee eri metodeja jotka koittavat suojata immateriaaliomaisuuksia, ilman että ohjelmiston virheidenkorjattavuus kärsii.**

**Binäärikirjastot ilman virheenkorjaustietoja vaikeuttavat asiakkaan tukemista. Uutta tuotetta kehitettäessä, markkinoille tuloaika on yritykselle tärkeää. Yleensä viimeiset kuukaudet ovat kiireisiä asiakkaan ongelmien tutkimuksien kanssa ja kyseiset ongelmat tulisi olla ratkaistuna mahdollisimman nopeasti.**

**Tämän lopputyön tavoitteena on vertailla mahdollisia metodeja, jotka suojaavat immateriaaliomaisuutta takaisinmallinnusta vastaan. Tarkoituksena on myös tutkia tulevia GNU kääntäjä-kokoelman (GCC:n) ominaisuuksia liittyen virheenkorjaustietoformaatteihin, kuten DWARF5.**

**Ongelman ratkaisuun koitettiin pilkottuja virheenkorjaustietoja, ELF-tiedoston injektointia, virheenkorjaustiedon riisumista ohjelmistosta ja koodin obfuskointia. Myös optimoinnin vaikutusta konekielestä takaisinmallinnettuun Assembly-muotoon tutkittiin. Paras ratkaisu oli kääntää ohjelmisto virheenkorjaustiedolla ja riisua ne omaan erilliseen tiedostoon. Näin ohjelmiston symbolitieto pystytään latamaan erikseen virheenjäljittemänä käytettyyn GNU Debuggeriin (GDB:hen). Näin symbolitietojen rakenne ja osoitteet ovat myös aina paikkansapitävät.**

**Avainsanat: kääntäjät, virheiden etsintä, takaisinmallinnus, DWARF, obfuskaatio**

# TABLE OF CONTENTS

# FOREWORD

I would like to thank my thesis supervisor Olli Silvén and second examiner Mehdi Safarpour. I would also like to express my gratitude to my technical supervisor Kari Suvanto and my manager Lari Manninen for all the guidance, time allocations, resource priority changes and the additional work caused by my thesis work.

I would also like to thank all the friends I made while studying. You entirely made this possible with all the support I have gotten from you – study wise as well as with all the time spent outside of studies, from get-togethers to relationships to cottage trips to long walks on the golf course.

Oulu, September 28th, 2022

Nuutti Räihä

# LIST OF ABBREVIATIONS AND SYMBOLS

| | |
|---|---|
| API | application programming interface |
| AST | abstract syntax tree |
| BIST | built-in-self-test |
| CFG | control flow graph |
| COFF | common object file format |
| DIE | debugging information entry |
| DL | dynamic loader |
| ELF | executable and linkable format |
| GCC | GNU compiler collection |
| GDB | GNU Debugger |
| IoT | internet of things |
| IP | intellectual property |
| IR | intermediate representation |
| ISA | instruction set architecture |
| LTO | link time optimisation |
| RTL | register-transfer language |
| SSA | static single assignment |
| SBST | software-based-self-testing |
| SoC | system on chip |
| R&D | research and development |

# 1. INTRODUCTION

Every company wants to protect their intellectual property and limit customer visibility of confidential information. The software industry is arguably one of the most important places where intellectual property (IP) is concentrated, as developing software requires significant amount of money, time and work.

Protecting intellectual property provides important benefits not only to the company that owns the IP, but also to its customers. Protecting intellectual property also protects information security. Breached proprietary information could expose customers to pirate products where their personal and sensitive data are at risk. Vulnerability in software or hardware could risk the customer's sensitive data too.

The number of embedded devices has increased tremendously. System-on-Chips (SoC) are commonly used in embedded systems and in the internet of things (IoT) [1]. In order to protect the outcomes of the research and development (R&D), copyrights and patents are used as legal protections [2].

Increased numbers of embedded devices mean increased amounts of data. Thus, enhanced information security is more important than ever before. Open source technologies can be considered secure as their code can be viewed by thousands of engineers daily. Features and improvements are added quickly, and any flaws are quickly spotted and fixed [3]. However, open source does not always provide feasible solutions.

SoCs are usually developed in-house from scratch and many features, if not all, need to be hand developed or heavily configured to get them working. As a result, embedded devices often include the hardware and the software in one package. If open-source software is used, it could be recognized from binaries in case the compiler is known. This could open attack vectors for known vulnerabilities, if unsecure versions are used.

As the number of embedded devices has increased and will increase in the near future, alongside protecting the software IP, protecting the hardware IP is also necessary. Built-in-self-test (BIST) can be implemented within the chip. With these tests, lower automated test equipment cost is achieved and time with testing is saved. However, BIST can have nontrivial area, performance, and design time overhead. [4, 5]

Software-based-self-testing (SBST) addresses problems caused by structural BIST. SBST methodology consists of memory self-testing, processor self-testing, global interconnect testing, and testing nonprogrammable cores. New products generally run new software. Even mature software usually includes new software because features and configurations are different. To provide fast and quality updates to the customer, good support is required from technology providers to application system designers. [5]

IoT has been, and still is, a milestone for the technology development in today's world and it is expanding. The internet alone has had a huge impact on education, communication, working, businesses, and a variety of other categories. Everyone has experienced this impact in the times of working and studying from home. The internet is arguably one of the most crucial technologies of the recent decades, and is continuing its fast growth. IoT represents the next evolution of the Internet. In 2003, there were 500 million devices connected to internet. In 2010, the number was 12.5 billion [6]. The extensive growth was due to an increase in smartphones and tablet computers.

Juniper Research [7] states there were 35 billion IoT connections in 2020, expected to rise to 83 billion by 2024.

Alongside the growth of smartphones, telecommunication systems have also evolved. Broadband cellular network standards, such as GSM and LTE, have been evolving and have increased the pervasiveness of mobile devices and mobile computing. With the arrival of newer technologies and standards, such as the fifth generation (5G) and upcoming sixth generation (6G) cellular standards, smartphone manufacturers compete to be first in the market with, for instance, 5G- or 6G-enabled products. As faster time-to-market is expected, shorter deadlines and higher pressure on product development teams are imposed. Engineers need to react quickly to customer requirements and issues, without sacrificing IP.

## 1.1. Intellectual Property

The software used in SoC devices can be divided into different categories, which are hardware-dependent software, middleware, and application software. Figure 1 shows the connections between the hardware and software. The lower we go in the stack, the more hardware dependent it gets. Hardware-dependent software represents application programming interface (API) for the hardware functionality of the SoC. These functionalities include hardware-dependent layers of the real time operating systems access to peripheral device drivers, system debug and maintenance functions, chip level configuration such as system registers, and often different kind of digital signal processing algorithms. [8]

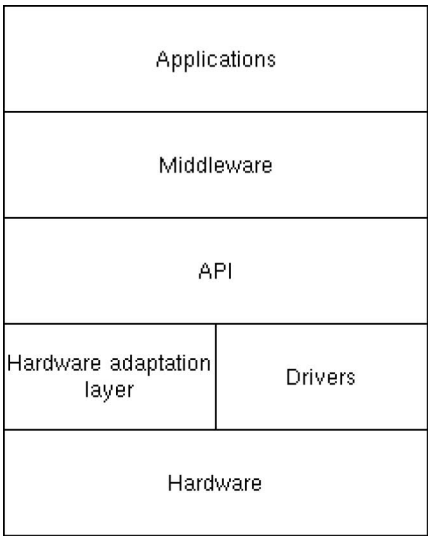


Figure 1. Hardware and software stack in SoC.

### *1.1.1. Protecting Intellectual Property*

Protecting the hardware layer is important because otherwise reverse engineering of hardware/software architectural partitioning and hardware functionality becomes

simpler. Some techniques to protect hardware IP are logic locking, IC camouflaging, and split manufacturing. [9].

As mentioned, middleware and application software does not really differ from application software, as the algorithms in this software do not necessarily rely on the hardware and they could be copied into different applications. Stripping debug symbols from the executable files is one rudimentary way to protect software IP. This way reverse engineering becomes significantly harder.

Not removing debug symbols is dangerous from application security point of view. Analysing code without debug symbols is difficult as debugger only provides limited support for this. Person analysing the code needs to guess or use other tools for different functions during the execution of the software. Other ways to protect software IP include obfuscation, encryption, and break point detection [10].

### *1.1.2. Reasoning Intellectual Property Protection*

Securing all the software is important in the SoCs. Vulnerabilities found with the reverse engineering can have serious breaches to the user's privacy. For example, there has been vulnerabilities where an attacker could potentially gain access to the audio data flow and eavesdrop on the mobile phone's user and vulnerabilities which permits root access to the user without unlocking the bootloader of the device. System on chip vulnerabilities enable also denial of service attacks or remote code execution attacks, for example, by crashing the baseband. This could lead to attacks to base transceiver stations. [11, 12, 13, 14, 15]

Security breaches can be caused by either insiders or outsiders. Both breaches can lead to leak of corporate secrets or confidential consumer information. Leaking confidential consumer information can be damaging to both the organization and the consumers. If crackers obtain personal data, organizations may face lawsuits and potentially even lose their competitive position. [16]

### *1.1.3. IP Abuse*

The definition of the IP abuse is broad. Some types of IP abuse are using IP lawsuits as a tool against competitors, using IP licensing agreements against new entrants to the market, and using contract law to expand the scope or term of intellectual property rights protection.

China has only recently set up their antitrust laws, whereas in the US and EU, the relationship between IP and antitrust laws has been developed over the past two decades. China is an important member of the World Trade Organization and its economy has grown rapidly; in 2008 it had become the second largest economy in the world after the US, when measured on a purchasing power parity basis. That's why any international IP commercialization, anti-monopoly or IP abuse prevention strategy cannot be ignored for a nation with such a big market. Hiding debug data might be a crucial way to protect against IP abuse, that could prevent the company from being first to market. [17]

## 1.2. Scope of the Thesis

IP protection is important for the companies. In the wrong hands it can be abused in order, for example, to weaken the company's time to market. When talking about SoCs that are made from scratch, protecting only software from reverse engineering is not enough. Hardware and software work together and when one is compromised, it affects the other also.

This thesis studies a way to improve the debugging flow while still protecting IP. The main objective is to protect the debug data generated in the compilation process when delivering binary releases to customers. Another objective is slowing down any potential reverse engineering, which could be achieved by code obfuscation. The motivation is to improve existing ways of working with customer issues, and to save precious time.

This thesis will first explain, in Chapter 2, the build process for embedded software and introduce the state of the art compilers. Chapter 3 discusses reverse engineering, focusing on static analysis. Some reverse engineering tools and protections against reverse engineering is discussed, as well as disassembly.

Chapter 4 explains what obfuscation is, what types and kinds of obfuscations there are and trade-offs the obfuscations cause. Chapter 5 explains debug data formats. The final parts of the thesis are about the implementations tested during the thesis work and discussion and conclusions about the results.

# 2. EMBEDDED SOFTWARE BUILD PROCESS

When an exception happens in an embedded system, the state of the system is captured in a memory dump. Often when debugging memory dumps no source code is available. Thus, engineers have to resort to machine code-level debugging. Machine code is directly stored in the ELF file so it cannot be hidden. If design patterns are employed in coding, disassembly results are easier to interpret. That is, good coding styles may make the product more vulnerable to reverse engineering. This is especially true if those doing reverse engineering are familiar with the used compilers and optimisations.

It is desirable that mapping machine code to source code not be straightforward for IP protection reasons. Luckily machine code and patterns are not always directly related to the source code. Source code undergoes multiple transformations and optimisations during the compilation process, and thus, a single expression in the source code does not always translate to the same machine code instruction or pattern.

One must understand the compilation process in order to understand what kind of data is created during the compilation, and how it can be used. For example, metadata files, so called .dat files, used in the debug process are generated during the compilation.

## 2.1. Build Process

Compiler can be divided into two parts: a front end and a back end. Front end breaks up the source program into pieces and imposes a grammatical structure on them, called abstract syntax tree (AST). The grammatical structure is used in creating an intermediate representation (IR) of the program. Optimisations done in the IR phase can be called the middle end. Back end of the compiler uses the symbol table and the IR to construct the target program. The main parts of the compiler can be seen in Figure 2. [18]

The compilation process can be further divided into different phases. These phases are lexical analysis, syntax analysis, semantic analysis, intermediate code generation, machine-independent code optimisation, code generation and machine-dependent code optimisation. Figure 3 illustrates the phases of a typical compiler.

The symbol table is used in all phases. As phases can be grouped together, not all compilers have the same phases as listed above. Optimisation phases are optional so one or both optimisation phases might be missing. [18]

### *2.1.1. Lexical Analysis*

The first phase of a compiler is lexical analysis. It is also known as scanning as it reads the source program's meaningful character sequences and groups them into lexemes. The lexical analyser produces a token from each lexeme. The form of the tokens can be seen in Equation (1).

$$\langle \text{token-type, attribute-value} \rangle \tag{1}$$

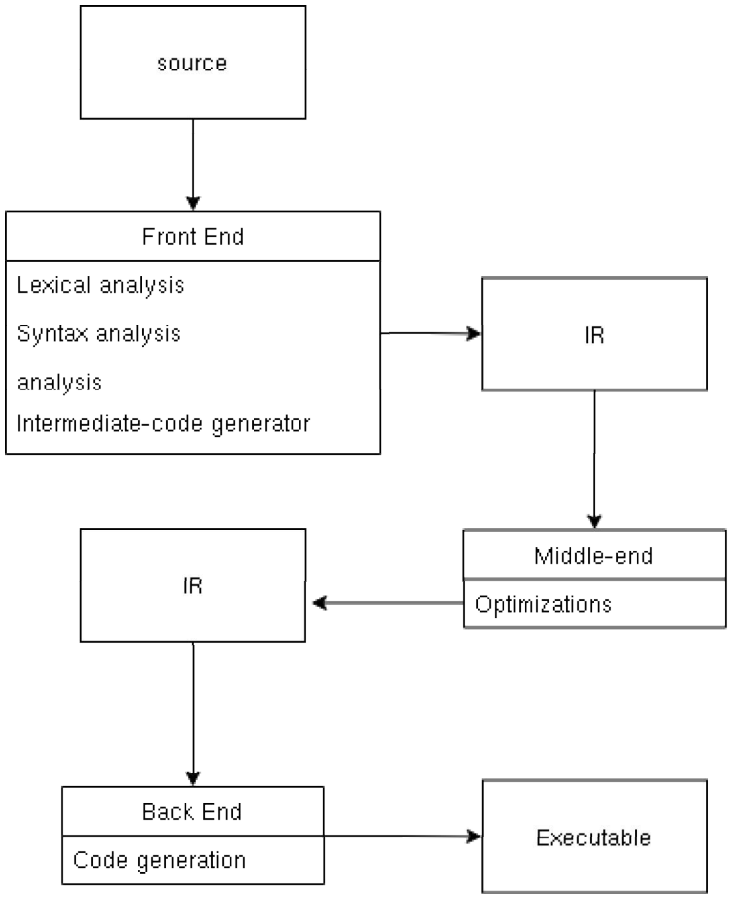


Figure 2. The main parts of the compiler.

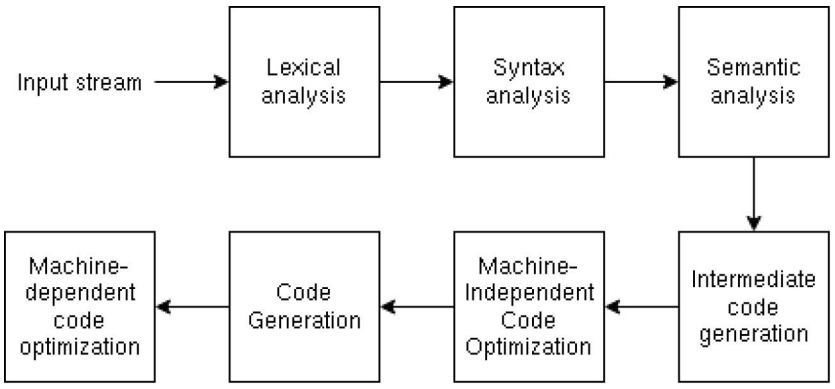


Figure 3. Phases of the compiler.

As the lexical analyser reads the source code, it can have other tasks besides identification of lexemes. The token-type is an abstract symbol that is used in syntax analysis. It is often return to the syntax analyser, along with a pointer to the lexeme. Information of the token is used in semantic analysis and code generation. For example, consider following assignment statement seen in Equation (2):

$$X = Y * 50 \tag{2}$$

Characters could be grouped into five following lexemes introduced in Equation (3):

$$X, =, Y, *, 50 \tag{3}$$

These five lexemes correspond to the tokens seen in Equation (4).

$$\langle \text{id, 1} \rangle \langle = \rangle \langle \text{id, 2} \rangle \langle * \rangle \langle \text{number, 3} \rangle \tag{4}$$

$X$ would be mapped to the token $\langle$id, 1$\rangle$, where id is an abstract symbol and 1 points to the symbol table. The symbol table entry for the identifier has information such as its name and type. The assignment symbol, $=$, is mapped to the next token. The second component can be omitted because the token does not need an attribute-value.

The second component 2 of the $Y$ lexeme's token points to the symbol table entry for $Y$. The $*$ lexeme is mapped to the token $\langle$*$\rangle$. Tokens are passed to the next phase, syntax analysis. [18]

### *2.1.2. Syntax Analysis*

Syntax analysis, also known as parsing, is the second phase of the compiler. The parser creates a tree-like structure from the tokens produced by the lexical analyser. This tree represents the grammatical structure of the token stream. At typical form is a syntax tree, where an interior node is an operation, and its children are the arguments of the operation. The syntax tree made from the tokens from Equation (4) can be seen in Figure 4. For example, GCC constructs an abstract syntax tree.

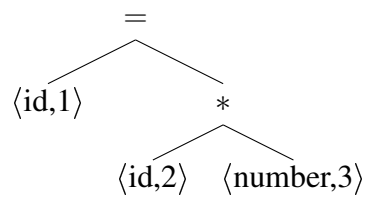


Figure 4. Syntax tree made from tokens.

The interior node * has $\langle$id,2$\rangle$ as its left child and $\langle$number,3$\rangle$ as its right child. The left child represents the identifier $Y$ and the right child represents the identifier for the integer 50. The node labelled $*$ tells us that we must first multiply the value of $Y$ by 50. The root node tells us that we must store the result of this multiplication into the location for the identifier $X$. Ordering of the operations is consistent with the usual arithmetic conventions. [18]

### *2.1.3. Semantic Analysis*

The syntax tree created in syntax analysis is used by the semantic analyser. It analyses the AST and checks the source program for semantic consistency with the language definition. The semantic analyser also collects type information for subsequent use in intermediate-code generation. One of the most important jobs of the semantic analyser is to check that each operator has legal operands. This is called type checking. For instance, an error would be raised if a floating-point number is used to index an array when the programming language requires an integer to be used.

Most languages support explicit and sometimes implicit type conversions. For example, sometimes binary operations can be performed on pairs of integers. In such cases, where an operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer to a floating-point number. [18]

### *2.1.4. Intermediate Code Generation*

After semantic analysis, compilers often generate a lower-level IR from the source program. This representation can be thought as a program for an abstract machine, and it has three important attributes: it is easy to produce, it is easy to translate to the target machine, and it is easy to do transformation or optimisations on the IR itself. [18]

One such IR is three-address code. This representation is composed of a sequence of assembly-like instructions. Each instruction has three operands, and each operand can act like a register. Three-address instructions have each at most one operator. For this reason, the instructions fix the order in which operations are executed. The compiler must generate temporary names to hold the values computed by three-addess instruction. [18]

### *2.1.5. Machine-Independent Code Optimisation*

Nowadays processor architectures have become more complex, as multicore and multithreaded machines have become universal. Thus, compilers perform more and more complex and important code optimisations. One of the problems in compiler design is to make sure that the optimisations are correct and improve the performance of the program, while keeping compilation times reasonable and the engineering effort manageable. Some of the noteworthy optimisations for computer architectures are parallelism and memory hierarchies. Parallelism can be in instruction level, where multiple operators are executed at the same time and in processor level, where multiple threads of the same application are run. Memory hierarchies enable us to build either very fast storage, or very large storage. [18]

In order to achieve better (faster, shorter, or less power consuming) code, the intermediate code needs to be optimised. A simple intermediate code generator combined with code optimisation produces reasonably way to generate good target code. Optimisations can be done at the compile time. The intermedaite code is often

divided into control flow graphs (CFG). If ASTs are high-level representations, CFGs are more detailed low-level representations.

A CFG consists of nodes which are basic blocks representing the statements of the code that are executed after each other without any jumps. The basic blocks are connected with directed edges that represent jumps in the control flow. CFGs can be used to eliminate unreachable or dead code. A CFG may have one or more exit nodes, depending on possible different return points in a function. [19]

For example, one optimisation could be replacing integer numbers with floating point number, thus possibly removing every function call that turns integer to floating point number. This optimisation would also eliminate the direct correspondences between the source code lines and the binaries, especially if the symbol tables will be stripped. This sort of optimisation can also lead to shorter sequences. Often most of the time is spent on these kinds of simple optimisations that improve the running time of the target program while keeping the compilation time relatively same. [18]

### *2.1.6. Code Generation*

IR of the source program is mapped into the target language with the code generator. In the case of target language being machine code, first the registers and memory locations are selected for each variable. Then the instructions are translated into machine instructions. One critical feature for code generator is to assign the registers to hold variables prudently. [18]

### *2.1.7. Symbol Table Management*

The symbol table is a data structure in the compiler. Symbol table contains information about source-program constructs. It is crucial for compiler, for example, to record variable names and to collect information about numerous attributes of each name. The collected attributes provide information about the storage allocated for its type, name and scope. Also, with procedure names, things like number and types of its arguments, the method of passing each argument and the type returned are collected. It is vital to design the data structure in a way that lets compiler to find the records quickly. [18]

### *2.1.8. Linkers*

In order to run even the simplest programs, they need to be linked. Linker combines independently assembled machine language programs and resolves all undefined labels into an executable file. In this manner future modifications do not necessitate recompilation of the whole program. Linker places the code and data modules symbolically in memory, determines the address of data and instruction labels and patches both, internal and external, references. From Figure 5 the toolchain from the perspective of the programmer can be seen. It shows how the source code ends up being executed.

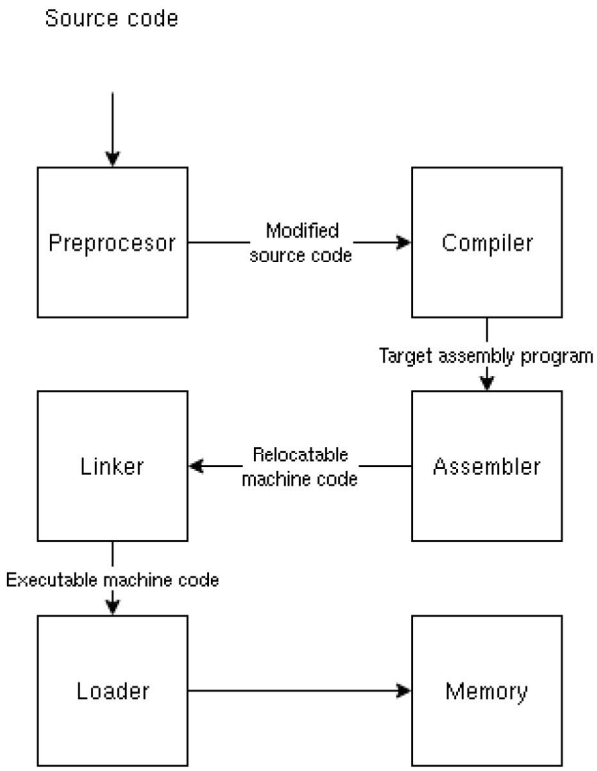


Figure 5. Toolchain from the programmer's point of view.

Symbol table and relocation information is used to resolve undefined labels which happens on branch and jump instructions and data addresses. When all external references are resolved, linker determines the memory locations where each module will occupy.

When module is placed in memory, all absolute references need to be relocated to reflects its true location. Output from linker is an executable file, and it is usually in the same format as an object file. The difference between the executable file and the object file is that there are no unresolved references in the executable file. However, files can be partially linked, as in library routines. Partially linked files will have some unresolved addresses and result in object files. [20]

### *2.1.9. Loaders*

Loader loads executable files to main memory. There are different kind of loaders that vary from simple to complex, and from small to large. In UNIX systems, loaders reads the executable file header and determines the size of different segments. Large enough address space for the text and the data is then created.

Instructions and data are copied into memory and any parameters for the main program is copied onto stack. Machine registers are initialised, and the stack pointer is set to the first free location. After all these, loader jumps to a start-up routine. The routine copies parameters into the argument registers and calls the main routine of the

program. After the main routine returns, system call exit is called in order to terminate the program. [21, 20]

### *2.1.10. Linking Loader*

Linking loaders are general loaders that can load several object files, relocate them and link them into an executable. Linking loader reads the names of all the object files which are loaded. Then it locates and opens the said files. Loader directive is read and the information read is used in calculating the total size of the program.

The loader locates large enough available memory area and reads next few loader directives from the beginning, and the information read is loaded to special symbol table. The process is then repeated for all the object files. Special symbol table is converted into global external symbol table, which is used later in the linking. [21]

After the beginning of the object files has been read, rest of the first object file is read and loaded. If necessary, instructions are relocated. All the loader directives are executed, and any required special relocations are handled as soon as they are read. This process is repeated for the rest of the object files. [21]

The main output of the loader is the loaded program, which is loaded in memory as one executable module. The second output of the loader is a listing file with error messages and a memory map. The last output is a single object file for the entire program. Besides the first output, all outputs are optional. [21]

### *2.1.11. Dynamic Loader*

Linking can be done in different times. The linking process can be though as composing independent virtual spaces into one composite virtual space. Leon Presser et al. [22] identified that linking at the late stages provide flexibility. The latest point of linking is at the run time and it is called dynamic linking. When libraries are loaded statically, they become part of the executable code. This results in updating every program when a new release of the library is available. Dynamically linked libraries try to solve this problem. [20]

Dynamic linking postpones the linking to the last possible moment. The linking happens when an instruction requiring linking is executed. This is slow way of working but it is more memory efficient as it does not require unnecessary routines. Dynamic linking is a good solution when a program has a lot of procedures but uses only a few of them.

Dynamic linking is implemented with loader routine called Dynamic Loader (DL). When a new procedure should be called, DL is called with a software interrupt, and the procedures is located, loaded and executed. After the procedure is executed, user program is restarted by the DL. DL manages the memory itself and it can release the memory used by the procedure and use it for other routines. [22]

#### *2.1.12. Summary of the linkers and loaders*

Linkers and loaders make it possible to edit the code without compiling everything again. It is good to understand that finding IP from the object files can result in person doing the reverse engineering to edit it and relink the module. This could compromise the security of the product.

Symbols are resolved at link time. This generates information like symbol name and memory address and these information can be reverse engineered. Dynamic loading can be used to overload functions, which can help in reverse engineering. One example of this is Linux's LD Preload functionality. Loaders can be tricked into loading a known dynamic linked library. Loaders can also receive new program code, which again, could be used to reverse engineer the software.

### 2.2. State of the Art Compilers

This section will focus on compilers for C language. Today, C compilers need to be optimizing compilers. Due to complex processors, C language is not as close to hardware as it was compared to i386 processors. Optimizing code is difficult and only two bigger compiler's could be seen as the state of the art – GCC and Clang. However, there are smaller non-optimising compilers like Tiny C Compiler.

#### *2.2.1. GNU Compiler Collection*

GNU Compiler Collection (GCC) is an open-source compiler produced by the GNU project. It was originally written as the compiler for the GNU operating system, but it includes front ends to C, C++ and many more. GCC can be considered as a standard compiler, as most *nix operating systems have adopted it. The intermediate representation of the GCC is called GIMPLE, and it supports multiple formats. [23]

#### *2.2.2. Clang*

Clang is a compiler for the C language family, developed by the Low Level Virtual Machine (LLVM) project. End-user features of the Clang are fast compiles, low memory use, expressive diagnostics and GCC compatibility. The motivation behind the Clang was the need for compiler that allows better diagnostics, better integration, and a nimble computer that is easy to develop and maintain. [24]

LLVM project is a collection of modular and reusable compiler and toolchain technologies. LLVM was originally a research project at the University of Illinois, with a goal of modern Static Single Assigment (SSA) based compilation strategy with both static and dynamic compilation support. LLVM provides it's own compiler back end, which thrives to be modular, making it easy to extend. Optimisations are implemented as modules, rather than statically built into the compiler. LLVM provides IR called LLVM IR, which is in SSA form. LLVM IR can be used as input to all LLVM tools and it can be edited manually with just a text editor. [25, 26]

### *2.2.3. GIMPLE vs LLVM IR*

GIMPLE is a three-address representation derived from GENERIC. GENERIC is a language-independent way of representing functions in trees. GCC turns source code into GENERIC and GENERIC gets converted to GIMPLE. GIMPLE is used for optimisations like inlining, constant propagation and tail call elimination. Then GIMPLE is converted into Static Single Assignment (SSA) GIMPLE. SSA GIMPLE makes sure all the values are consistent and performs optimisations. After the SSA GIMPLE phase is done, it is converted back to GIMPLE form, in order to generate register-transfer language (RTL) form. Back-end then generates assembly code from the RTL representation. Compilation process with the GCC can be seen from Figure 6. [27, 28]

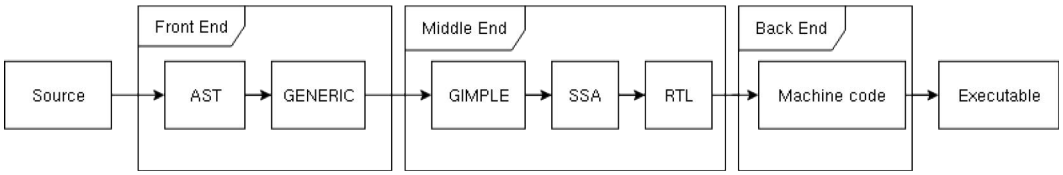


Figure 6. GCC compilation process.

LLVM IR is also three-address representation. It was originally created to use for multi-stage optimisation. LLVM IR aims to be universal IR and the LLVM compiler is designed to allow for optimisations at all stages of compilation, for example removing dead and redundant code, inlining functions, unrolling loops and deleting dead loops. LLVM IR is easier to process, transform and analyse compared to GIMPLE, as it is originally designed to be a standalone IR. [29, 26, 30]

As GIMPLE is tightly coupled with the compiler suite, it often is output from specific process and input to another specific process. LLVM IR is more versatile and modular, so the IR form can be used in more various ways. This does not automatically make LLVM IR harder to reverse engineer, but one could apply, for example, obfuscation to the IR and continue to the code generation. In other words, as LLVM IR is more modular, it can be easily processed, thus making it harder to reverse engineer. [26, 31]

## 2.3. Executable File Formats

The executable and linkable format (ELF) is a standard file format for object files. ELF is the standard binary file format for UNIX systems; however, it is a cross-platform standard. It should be mentioned that there are three mainstream executable file formats. Besides ELF files for UNIX systems, there are portable executable files for Windows systems and Mach-O for OS-X and iOS systems.

There are three main types of object files, relocatable files, executable files, and shared object files. A relocatable file is used in linking with other object files in order to create an executable or a shared object file. A shared object file has two uses in linking. The linker may process it with other relocatable and shared object files to create another executable file. The dynamic linker can also combine it with an executable file (alongside other object code) to create a process image. [20, 32, 33, 34]

### *2.3.1. ELF Files*

ELF files primarily consists of binary representations of the program that is run on a processor. As object files participate in linking and execution, ELF files provide parallel views of the file's contents. Figure 7 shows the linking view and Figure 8 shows the execution view.

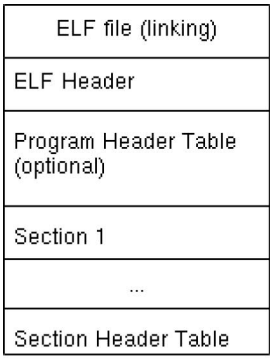


Figure 7. Structure of the elf file as seen by the linker.

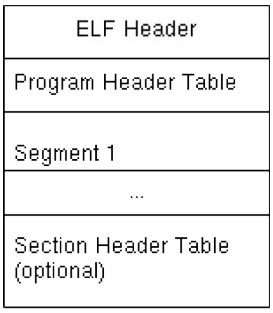


Figure 8. Structure of the elf file as seen during execution.

ELF files contain: the object file headers, the text segment, the static data segment, the relocation information, the symbol table, and the debugging information. The ELF header defines if 32-bit or 64-bit addresses are used. It also specifies the target ISA and operating system ABI, alongside other attributes. [32]

The program header table tells how to create a process image. It contains, for example, the offset of each section in the file image and the virtual address of the section in memory. The section header table contains information about the file's sections. If the ELF file is used to build a process image, a program header table is required. Relocatable ELF files do not need it. [32]

ELF files have special sections which hold program and control information. Sections that support debugging are, for example, .debug_info and .debug_line. Section .rodata contains initialised read-only data and .data contains initialised data. Some more sections are, for example, .bss (uninitialised data), .symtab (global symbol table, containing information to locate and relocate a program's symbolic definitions and symbolic references), .dynamic (all the information for dynamic linking) and .strtab (string table of symbols or sections). [32]

## 2.4. Summary

The build process is a complex process with various phases. The source code changes it's form multiple times during the process and in each form, there are changes, like optimisations, that differentiate the original source code from the executable. If the symbols are stripped from the resulting object file, reverse engineering comes considerably harder as there sometimes are little connections to the original code.

ELF files provide a parallel view of contents of the software. It is important to understand it's structure in order to find the IP inside it.

# 3. REVERSE ENGINEERING

Reverse engineering a compiled program is an effective and mature approach that can even be applied on complex software. It cannot be prevented but it can be made more difficult, slower and expensive. Reverse engineering can be used, for example, in supporting software maintenance and in finding vulnerabilities. It can also be used to violate the IP rights. It should be noted that not all reverse engineering is malicious.

Protections applied against the reverse engineering gives the attacker a second object to attack. This slows down the attacker as identifying and bypassing the protections also require reverse-engineering. However, there are lots of tools that help reverse engineers to bypass the protections. This chapter will have a brief assessment of common reverse engineering tools and techniques.

Reverse engineering was initially applied to the hardware. Nowadays, reverse engineering is used in a variety of fields, from software and databases all the way to the human DNA. M. Schwartz in the article "HOW-TO: Reverse-Engineering" describes reverse-engineering being like rebuilding a car engine or diagramming a sentence. According to Schwartz a lot can be learned by taking something apart and putting it back together and that is the concept behind reverse engineering. Breaking something down let us understand it, build a copy of it, or improve it. [35]

## 3.1. Why Reverse Engineering?

Reverse engineering is important in the computer hardware and software domain, as it is used as a learning tool. It can be used to maintain software and to make the software work more efficiently, and to uncover undocumented features. Reverse engineering can also be used to cope with the complexity, recovering lost information, detecting side effects and synthesizing higher abstractions. Alongside with replication of a product, reverse-engineering can be used to find vulnerabilities from the software. [36, 37]

Reverse engineering has been successfully applied in, for example, redocumenting programs and relational databases, identifying reusable assets, recovering architectures and design patterns, identifying clones, codesmells and aspects, reverse engineering binary code, renewing user interfaces, translating a program from one language to another, migrating and wrapping legacy code. [38, 39]

## 3.2. Reverse Engineering Methods

Reverse engineering methods can be divided into static and dynamic analysis. Static analysis does not include execution of the target program. The analysis is done with the code describing the program. The dynamic analysis is done by executing the program in a real or virtual processor. The results of the dynamic analysis depends on how the program is executed. Dynamic analysis can be used to collect data of the program. For example, snapshots of the state of the program and control flow graphs.

### *3.2.1. Static Analysis*

Often the obtained software is an executable file without any debugging information. Disassemblers can be used to obtain the representation of the target program in a form of source code. Static analysis methods are, for example, control flow analysis, data flow analysis and semantic analyser.

Branches, jumps and calls are analysed in the control flow analysis. This way, unreachable code or code with no effective exists can be identified. With control flow analysis it is also possible to determine a high level language loops structure if the target code uses multiple jumps and is hard to read. [40]

Data flow analysis considers the update of variables and registers in order to find reads from uninitialised memory or writes without any intervening reads. Semantic analyser can help people performing the reverse engineering to understand what the target code does. It analyses every execution path of the software and gives a symbolic equation for the values of the output of each path. This analysis can find paths that are not accessible, for example if clauses which can never be true. Target code with multiple infeasible paths can be rewritten without these paths.

## 3.3. Disassembly

As described in Chapter 2 software compilation process involves assembler translating assembly language into machine language. Disassembler translates machine language back to assembly language, often in human readable format. For this reason, disassemblers can be thought of as a reverse engineering tool. For example, GNU Debugger (GDB) includes a command to disassemble machine code [41].

When the source code is available, interpreting disassembly is easier and quicker than without source code. However, compile time optimisations can make the disassembly harder to understand. Without the source code and with compile time optimisations, understanding disassembly can be really hard. Disassembly is addressed more in detail in Chapter 6

## 3.4. Reverse Engineering Aids

Different software and hardware aids are used in the reverse engineering. Some of the software aids are debuggers, tracer, emulators, disassemblers, decompilers and hexadecimal editors. Hardware aids are, for example, developer boards with hardware debuggers. There are different kinds of reverse engineering frameworks and libraries, which include a set of reverse-engineering tools. To name a few, radare2 [42], LIEF [43], Ghidra [44] and IDA Pro [45].

### *3.4.1. Radare2*

Radare2 is a complete framework for reverse-engineering with a support of variety of executable formats across different processor architectures and operating systems.

Radare2 contains a set of tools used in the reverse engineering. For example, it includes hexadecimal editor and debugger and it implement a command line interface for moving around a file, analysing data, disassembling, binary patching, data comparison, searching, replacing and visualizing. Radare2 includes also tools for extracting information from different executable binaries, disassembler and assembler, hash tool, binary diffing, finding byte patterns in files, launcher for running programs within different environments, debugger and many more. Radare2 is usable in static analysis, dynamic analysis and software exploitation. [46]

### *3.4.2. Ghidra*

Ghidra is reverse engineering framework developed by the National Security Agency of the United States. Ghidra is a free alternative to IDA Pro and it features include disassembly, assembly, decompilation, graphing, scripting and many more features. [44]

## 3.5. Protecting Against Reverse Engineering

Perfect protection against reverse engineering is impossible. However, the protection techniques try to raise the cost of reverse engineering and time needed to perform the reverse engineering. When the protected software is attacked, attackers need to reverse engineer the protections alongside the targeted software. [47]

Debug symbols are often included in the object file [48]. This allows developers to debug their application during the development. However, the binary file can be used to gather information like variables, number of lines, function names and so on [49]. Companies often may want to protect this kind of information. Stack traces with the debug symbols can provide specific function addresses, which allows inserting breakpoints while executing the code, for example. This kind of information can be a dangerous from the software security point of view and make the reverse engineering process quicker.

# 4. OBFUSCATION

Code obfuscation is a mechanism to protect the code with a purpose of reducing human readability of the code. This can be achieved, for example, by renaming internal variables and function names. This sort of security by obscurity is often disdained in the security communities. However, obfuscation is cheaper than deobfuscation and obfuscated code increases the skill level of the attacker and even skilled attacker needs to give more effort compared to non-obfuscated code.

## 4.1. Performing Obfuscation

Obfuscation can be done at the source code level, intermediate level and binary level. In order to obfuscate at binary level, the obfuscator should support the platform of the binary. Obfuscation algorithms are also more complicated on the binary level, as data is lost in the compilation process.

Obfuscating source code requires obfuscator to have a support for the specific programming language used. However, obfuscating source code makes the developer's debugging work harder, as the obfuscated code needs to be either translated or reverse engineered back to the original.

As explained in the Chapter 2, compiler creates intermediate representation of the program that is independent of the programming language used. The compiler also translates the IR to the machine code, so obfuscations at the IR level do not depend on the used programming language, or the target platform. However, not all information is available at IR level and some target platform specific obfuscations cannot be implemented at IR level. [50]

Obfuscating can be combined with optimisations. One could even say that it is important to optimise the obfuscated code in order to have as minimal performance disadvantages as possible. Obfuscating LLVM IR has few rules. The rules can be seen in Table 1.

Table 1. Rules of mixing optimisations and obfuscations (Derived from [50])

| | |
|---|---|
| Rule 1 | Obfuscations must survive LLVM optimisations |
| Rule 2 | Performance is important: run classical LLVM optimisations first |
| Rule 3 | Then obfuscations are applied |
| Rule 4 | And a post optimisations pass is done |

The ISA of the executing hardware can be randomized, and software binary code can be encrypted. When the software binary code is encrypted, the file header usually contains information about the decryptor. Execution binary is encrypted, and it contains information how to decrypt instructions and execute them. Encryption can be used to prevent code analysis and to make disassembling more difficult.

Obfuscation tools, however, are often limited in functionality by design. Protections are pre-determined and injected code only includes only limited number of code patterns that differs from the patterns of the original software. This can make

deobfuscation easier and the process can be automated. One way to automate the deobfuscation is to identify fragments that are known before hand to be artificially complex replacements of simpler control flow constructs. Then the found snippets can be replaced with simpler variations. This way can be built on techniques like pattern matching, symbolic execution and abstract interpretation. Off-the-shelf extensible frameworks are available forattackers. Another way to fight the obfuscation is to assume that the protections are superfluous and complex with respect to the semantics of the protected software. Such a code will stand out from the original software in a detectable way. The protecting code can then be removed. [51, 52, 53]

## 4.2. Obfuscation Methods

Code obfuscations can be categorized into five types. These types are lexical obfuscations, data obfuscations, control obfuscations, preventive obfuscations and string encryptions. All the obfuscations are some sort of code transformations. [54]

### *4.2.1. Lexical Obfuscations*

Lexical obfuscations, also known as layout obfuscations, are one-way transformations. This includes renaming identifiers, changing formatting, removing comments and pruning. Pruning removes unused functionality. Overall these obfuscation changes the lexical structure of the target program and makes it more difficult for attacker to extract meaningful information from identifiers. [54]

### *4.2.2. Data Obfuscations*

Data obfuscations tries to prevent unauthorized access to sensitive materials. Data obfuscations can be split to three categories: storage and encoding obfuscations, aggregation obfuscations and ordering obfuscations. Storage and encoding obfuscations changes the representation and methods of usage of the variables. Aggregation obfuscations merges variables, modifies inheritance relations and splits or merges arrays. Ordering obfuscations reorder methods and arrays. [54]

Variable aliasing is a data obfuscation that changes how the variables are stored. For example, variable can be split into several variables, or variables can be merged into a single variable. For example four 8 bit variables could be merged into one 32 bit variable. If x, y, i and j are 8 bit variables, they can be merged into a 32bit variable by shifting and ORing. Example of splitting a variable into multiple variables would be the reverse of the previous example.

Another data obfuscations are refactoring classes and restructuring arrays. Object oriented code can be obfuscated by refactoring the classes in a way that changes the class inheritance. For example, method's of the class could be split into new classes. Array restructuring makes understanding the structure of the arrays more difficult. For example, array's indexes could be shuffled and another array could give us the correct order of the indexes.

### *4.2.3. Control Obfuscations*

Control obfuscations makes it harder to understand the control flow of the program. Control obfuscations have three groups: computation obfuscations, aggregation obfuscations and ordering obfuscations. Computation obfuscations changes the structure of the control flow, aggregation obfuscation merges and splits code fragments and ordering obfuscations reorder code block, loops and expressions. [54]

Opaque predicates will evaluate at compile time to known values. Opaque predicates can be used to create conditionals which complicates the control flow graph. Obfuscators could also create opaque predicates itself by introducing new variables with predecided relations.

Control flow flattening creates a proxy that is used in jumps between the basic blocks. This could reduce the control flow graph height to two levels. The proxy and all the basic blocks.

Function calls can be obfuscated with function pointer obfuscation. A pointer to each function could be stored in a global array. Each function call would need to be replaced by indexing the array correctly and calling the function pointer. This method could be used with previously mentioned array restructuring and opaque predicates.

Instruction substitution replaces standard binary operations by more complicated sequences of instructions which are functionally equivalent. This obfuscation is not very secure as it could be removed by re-optimising the generated code.

Bogus control flow modifies a function call graph. It adds basic blocks before the current basic block, which contains an opaque predicate and then makes unconditional jump to the original basic block. The original basic block is cloned and filled up with junk instructions.

Function transformations can be inline expansions. This makes finding all the callers of the function harder. Another example of function transformations is splitting the function to smaller parts in order to make harder to understand the context of the function.

### *4.2.4. Preventive Obfuscations and String Encryptions*

Preventive obfuscations try to make automatic deobfuscation techniques more difficult. With string encryption obfuscations, strings must be decrypted at run time. This means replacing string literals with calls to decrypting methods. [54]

## 4.3. Obfuscation Tools

Obfuscation tools are usually purchasable proprietary software. Open source obfuscators are used mainly for research purposes or as a start for further development. Otherwise the obfuscation results would not be safe as the tools often have predetermined protections. Some obfuscation tools are introduced below.

### *4.3.1. Obfuscator-LLVM*

Obfuscator-LLVM is a open source project by the information security group of the University of Applied Sciences and Arts Western Switzerland of Yverdon-Ies-Bains. It has also a commercial version, which has more advanced features. The open source version supports following obfuscations: instruction substitution, bogus control flow, control flow flattening and functions annotations. [31]

Instruction substitution replaces binary operators with functionally equivalent sequences of instructions that are more complicated. Bogus control flow adds a basic block with opaque predicates before the current basic block. The bogus block makes a conditional jump to the original block. Control flow flattening transfers the structure of the source code in a way that targets of any branches cannot be easily determined. [31]

### *4.3.2. DashO*

DashO is an enterprise application hardening and obfuscation tool developed by PreEmptive Solutions. It supports Java, Android and Kotlin applications. DashO obfuscates compiled code, injects code to automatically detect and respond to threats and it can inject calls to custom code. DashO provides renaming, control flow and string encryption obfuscations. [55]

### *4.3.3. Themida*

Themida is a binary level obfuscator developed by Oreans Technologies. Some of techniques supported by Themida are anti-debugging, anti-memory dump, anti-disassembly, API wrapping, virtual machine emulation in specific blocks of code, anti-tracing code insertion, and random internal data relocation. This makes Themida more than an obfuscator, as it provides techniques to protect the software from reverse engineering. [56]

## 4.4. Side Effects of Obfuscation

As the code protection is injected to the executable file, it can have unwanted side effects to the performance of the software. Obfuscation can make the software slower, increase the size of the binaries and make debugging by the developers harder.

If all the variables and strings are obfuscated, you need to have a mapping file that translates all the random names. However, this increases the overhead as you need to preserve the mapping files of every build. It also requires more work that you need to use the mapping file to translate the obfuscated names.

Obfuscation should be done to release builds but not to debug builds. Debug builds need to be focused on the developer experience, as they are used for daily work. Obfuscation woudld preserve the two build system and it would result in big

differences between the release and debug builds. Code obfuscation can introduce new bugs only in the release builds.

# 5. DEBUGGING SOFTWARE

Programs can be debugged with print statements. Often this method is not the most efficient way of working but in some cases, printing can be the preferred method. There are also low-level debuggers that enable us to execute the debuggable program instruction by instruction, displaying registers and memory. With source-level debuggers we can set breakpoints, step through the program's source, print values, and call program functions while in the debugger. This way of working needs coordination between two different programs — compiler and debugger. [49]

During the compilation process, the compiler collects useful information about the program which is used later when debugging the program. However, as the compilation process often includes optimisations; it might turn out to be difficult to relate the optimised code to the original source code. For example, "peephole" [18] optimisations read the machine code and might rearrange and modify the code. The modifications can be, for example, taking a pattern of two to three instructions and turn it into a different pattern that is equivalent but more efficient.

A complete description of the executable program's relationship to the source is hard to achieve. The description needs enough detail to provide meaningful information to the programmer but it also needs to be concise enough so it can be interpreted quickly, and it will not take too much space. There are different debugging formats that aim to achieve this, including stabs, COFF, and DWARF. [49]

## 5.1. Debugging Formats

Debugging formats are the mechanism by which the compiler communicates debug information to the debugger. Debugging formats try to keep up with the optimisations done when building the software. DWARF will be studied in this thesis as it is the most recent and it is actively developed. It has major support and it has replaced most of the other debugging formats.

### *5.1.1. Stabs*

Stabs was created in the 1980s. It tackles the problem of the a.out [57] object file format not storing debugging information. Stabs solves this by encoding the information using special entries in the symbol table. Stabs was commonly used on Unix systems, but the DWARF format has replaced it. [58, 59]

### *5.1.2. COFF*

The Common Object File Format (COFF) is an object file format that replaced the previous format, a.out. COFF's elements describe the file's symbolic debugging information and file's sections. Debugging information includes symbolic string names for functions and variables, and line number information. COFF has now been replaced by ELF format in most systems. [60, 61]

### *5.1.3. DWARF*

DWARF is a debugging file format that is widely supported by compilers and debuggers. DWARF is widely used in the *nix systems, but DWARF is applicable to any operating system as well as to stand-alone environments. DWARF supports source-level debugging, and it is designed to be extensible to different programming languages. [62]

## 5.2. DWARF Debugging Format In detail

The DWARF Debugging Format is a representation of the relationship between an executable program and its original source in a format that can be efficiently processed by the debugger. In order to provide meaningful functionality, the debugger must reverse numerous transformations performed by the compiler. This means that all the data and states need to be converted to the terms that were originally used in the source program. DWARF aims to make this possible and easy. [49]

### *5.2.1. DWARF Structure*

The DWARF description of the program is a tree structure where each node can have children or siblings. Nodes can be types, variables or functions. In order to achieve a compact format, only the necessary information to describe the program is provided. The structure is extensible in a uniform fashion, so the debugger can recognize and ignore extensions even without understanding the meaning of them.

DWARF follows a block structured model, except for the topmost entry. DWARF descriptive entries are contained within a parent entry and may have child entities. If a node contains multiple entities, they are siblings as they are related to each other. DWARF is also designed to describe virtually any procedural programming language on any machine architecture, thus making it more versatile. DWARF is not dependent on the object file format but it is often used with the ELF object file format. [49]

DWARF abstracts out the basics of the programming language and provides a representation of them. This way it can support multiple programming languages, as languages often contain numbers of built-in data types, pointers, data structures and a way to create new data types. Base types are primary types which can be supported in hardware. Other data types are collections or compositions of the base types.

DWARF can be used with different processor architectures. It does not matter, for example, if the architecture is byte- or word-oriented. DWARF can be used with Von Neumann architectures, which have a single address spaces for code and data, with Harvard architectures, which have separate code and data address spaces, and potentially for different digital signal processing architectures. DWARF can also be used with common register-oriented architectures or with stack architectures. [48, 49]

### 5.2.2. Debugging Information Entry

The Debugging Information Entry (DIE) is the basic descriptive entity in the DWARF format. Each DIE has a tag and a list of attributes. A tag tells what a DIE describe and the attributes fill in the details and further describes the entity. A DIE is contained in, or owned by, a parent DIE and it may have sibling DIEs or child DIEs. Attributes may be, for example, constants, variables or references to another DIE. There are two types of DIEs, DIEs describing data, including the data types, and DIEs describing functions and other executable code. [49]

A DIE describing a named variable has several attributes, for example, the name of the variable, the file where it was declared, the line where it was declared, and a link to the base type. A DIE for a variable and the base type it refers could look like:

```
<1><57>: Abbrev Number: 3 (DW_TAG_base_type)
   <58> DW_AT_byte_size : 4
   <59> DW_AT_encoding : 5    (signed)
   <5a> DW_AT_name    : int

<2><90>: Abbrev Number: 6 (DW_TAG_variable)
   <91> DW_AT_name    : ret
   <95> DW_AT_decl_file : 1
   <96> DW_AT_decl_line : 5
   <97> DW_AT_decl_column : 6
   <98> DW_AT_type    : <0x57>
   <9c> DW_AT_location : 2 byte block: 91 64  (DW_OP_fbreg:
      -28)
```

### 5.2.3. Base Type

The DIE for the base type (DW_TAG_base_type) consists of the top four lines, and the DIE describing the named variable (DW_TAG_variable) is the remainder. We can see that DW_AT_type: <0x57> is a reference to the base type describing integers. A reference to a DIE is represented as the offset from the start of the compilation unit containing the DIE. References can be previously defined DIEs or DIEs that are defined later. Base types are also used in constructing other data types. For example, pointer to a character DIE would be defined as follows:

```
<1><65>: Abbrev Number: 4 (DW_TAG_pointer_type)
   <66> DW_AT_byte_size : 8
   <67> DW_AT_type    : <0x6b>

<1><6b>: Abbrev Number: 2 (DW_TAG_base_type)
   <6c> DW_AT_byte_size : 1
   <6d> DW_AT_encoding : 6    (signed char)
   <6e> DW_AT_name    : (indirect string, offset: 0x75): char

<2><9f>: Abbrev Number: 6 (DW_TAG_variable)
   <a0> DW_AT_name    : s
```

```
   <a2> DW_AT_decl_file : 1
   <a3> DW_AT_decl_line : 6
   <a4> DW_AT_decl_column : 8
   <a5> DW_AT_type    : <0x65>
   <a9> DW_AT_location : 2 byte block: 91 68  (DW_OP_fbreg:
      -24)
```

Variable $s$ points to a pointer type DIE which defines its size as eight bytes and in turn, refers to the character base type. DIEs also describe the const and volatile attributes, C++ reference types and C restrict types, amongst other types. For example, a C program with a command line argument const char **argv produces:

```
<2><a2>: Abbrev Number: 6 (DW_TAG_formal_parameter)
   <a3> DW_AT_name    : (indirect string, offset: 0xb7): argv
   <a7> DW_AT_decl_file : 1
   <a8> DW_AT_decl_line : 3
   <a9> DW_AT_decl_column : 33
   <aa> DW_AT_type    : <0xce>
   <ae> DW_AT_location : 2 byte block: 91 50  (DW_OP_fbreg:
      -48)

<1><ce>: Abbrev Number: 8 (DW_TAG_pointer_type)
   <cf> DW_AT_byte_size : 8
   <d0> DW_AT_type    : <0xd4>

<1><d4>: Abbrev Number: 8 (DW_TAG_pointer_type)
   <d5> DW_AT_byte_size : 8
   <d6> DW_AT_type    : <0x6c>

<1><6c>: Abbrev Number: 4 (DW_TAG_const_type)
   <6d> DW_AT_type    : <0x65>

<1><65>: Abbrev Number: 2 (DW_TAG_base_type)
   <66> DW_AT_byte_size : 1
   <67> DW_AT_encoding : 6    (signed char)
   <68> DW_AT_name    : (indirect string, offset: 0xb2): char
```

Here the DW_TAG_formal_parameter can be seen referring to another pointer type, which refers to const type which refers to the base type. For arrays, the DIE defines whether the data is stored in row major or column major form. The index for the array is represented by a subrange type that gives upper and lower bound dimensions. This way DWARF can support C-style arrays, which are zero-based, but also, for example, Pascal arrays, which may have any value for low and high bounds. [49]

### *5.2.4. Data Structures*

Programming languages usually have ways of grouping data together. There are structures, unions, variant records and interfaces. Different languages have their own terminology, but DWARF uses terminology based on C, C++ and Java. These ways to group data together are called struct, union, class and interface in DWARF. For

example, a class DIE is a parent DIE which contains each of the class members. Each class has a name and possibly other attributes. Class DIE descriptions do not differ very much from simple variable descriptions. For instance, C++ access specifiers, (“public”, “private”, “protected”) are described with accessibility attributes.

Structures, unions and interfaces are described in a rather similar way. Variables have a chunk of memory or register that can contain a value. The type of the variable describes what kind of values the variable can contain, and any possible restrictions on modifying the variable, such as the const keyword. Variables also have scopes. In C, variables declared in a function or in a block have function or block scope. Variables declared outside these are in global or file scope. This enables declaring variables of the same name across different files. [49]

### *5.2.5. Variables*

DWARF splits variables into constants, formal parameters and variables. Constants do not exist in C as “const” is just a keyword that states that the variable cannot be modified without an explicit cast. Formal parameters represent the values passed to a function. C allows declaring variables without defining them. When describing a variable declaration, the DIE provides a description of the variable without telling the debugger where the variable is. Often variables have a location attribute, which tells where it is stored. Usually the variable can be found by adding a fixed offset to a frame pointer. A variable can also be stored in a register. There are also other variables that may require more complicated computations to locate the data, for example, C++ class members. [49]

### *5.2.6. Functions*

Functions, which return values, and subroutines, which do not, are variations of the same thing in DWARF, as both are described with a subprogram DIE. A Subprogram DIE has a name, a source location triplet and an attribute that indicates whether the subprogram is outside the current compilation. A sub program DIE also has attributes that give the low and high memory addresses if the function is contiguous. If the function is not contained in contiguous set of memory addresses, a list of memory ranges is given instead.

The low program counter address is assumed to be the entry point for the function. The type attribute refers to the value that function returns. In C, void functions do not have this attribute. One common compiler optimisation is eliminating instructions that save the return address or frame pointer. DWARF handles this with two attributes. The address of the caller is stored in the return address attribute and the address of the stack frame for the function is computed with a frame base attribute. [49]

### *5.2.7. Subprogram*

A subprogram DIE owns DIEs that describe the subprogram. Any passed parameters are represented by variable DIEs. Variable DIEs have a parameter attribute. The cases where the parameters are optional or have default value are represented by attributes. The order for the DIEs of parameters is the same as the order of argument list for the function and there might be additional DIEs, for example, to define types used by the parameters. Function can define variables, and these variables can be global. These variables follow the parameter DIEs. If a programming language allows lexical blocks, they are described with lexical block DIEs. Lexical block DIEs can own variable DIEs or nested lexical block DIEs. [49]

### *5.2.8. Compilation Units*

Each source file is compiled independently when a program consists of multiple files. Every separately compiled source file is called a compilation unit by DWARF. DWARF data for a compilation unit starts with a Compilation Unit DIE. This DIE has information about the compilation, for instance, the directory and the name of the source file, the programming language used, and a string which identifies the producer of the DWARF data. Also, offsets into the DWARF data section are included to locate the line number and macro information.

When the compilation unit is contiguous in memory, there are also values for the low and high memory addresses for the unit. A list of the memory addresses that the code occupies is provided by the linker when the compilation unit is not contiguous. A compilation unit DIE is a parent of all the DIEs that describe the compilation unit. Usually, the first child DIEs describes data types, then global data, and then the functions that make up the source file. DIEs for variables and functions are in the same order as they appear in the source files. [49]

### *5.2.9. Debug Sections*

The generated debug data can become large, often larger than the executable code and data itself. DWARF reduces the size of the debugging data by creating a '.debug_str' section which contains most of the strings.

Duplicate strings can be removed and strings in the debug data becomes references to this section. Repeated declarations can be saved in separate compilation units in uniquely named sections. Afterwards, common data can be used to eliminate duplicate sections [48].

DWARF data can be generated for only type definitions, that are used as programs can reference many header files which contain many unused type definitions. Type definitions can be saved into '.debug_types' and thus, duplicates can be eliminated. In DWARF5 there are 15 different DWARF sections and 10 different DWARF split sections. All the debug sections start with the '.debug_' prefix. The DWARF sections and a description of each section's content can be seen in Table 2. Split object sections are listed in Table 3. [49]

Table 2. DWARF Sections

| Section Name | Content |
|---|---|
| .debug_abbrev | Contains abbreviation codes used in the .debug_info section |
| .debug_addr | Contains references to locations within the virtual address space of the program |
| .debug_aranges | Contains the lookup table for mapping addresses to compilation units |
| .debug_frame | Contains call frame information |
| .debug_info | The core DWARF data containing DIEs |
| .debug_line | Contains line number tables |
| .debug_line_str | Contains strings for file names used in combination with the .debug_line section |
| .debug_loclists | Contains location lists which are used in place of a location description whenever an object's location can change during its lifetime |
| .debug_macro | Macro descriptions |
| .debug_names | Contains the maintained name index for lookup by a name for each DIE that defined a named subprogram, label, variable, type, or namespace |
| .debug_rnglists | Contains range lists, which are used when the set of addresses for a debugging information entry cannot be described as single contiguous range. |
| .debug_str | Contains any strings referenced |
| .debug_str_offsets | Contains the string offsets table for the strings in the .debug_str section |
| .debug_str_sup | Establishes relationship between the supplementary object file and all the executable or shared objects files that reference entries in the supplementary object file. |

Table 3. DWARF Split Sections

| Section Name | Content |
|---|---|
| .debug_abbrev.dwo | Abbreviation table for the compilation unit and type units |
| .debug_info.dwo | Full compilation unit |
| .debug_line.dwo | Specialized line number table for the type units |
| .debug_loclists.dwo | Location lists |
| .debug_macro.dwo | Macro information |
| .debug_rnglists.dwo | Range lists |
| .debug_str.dwo | A string table |
| .debug_str_offsets.dwo | A String offsets table |

### *5.2.10. Split Object Files*

Split DWARF splits the debug information into two parts at compile time. One part remains in the object file and another part is written to a parallel .dwo file. The motivation is to reduce the total size of object files processed by the linker, in order to speed up link times and to reduce system memory requirements. The object files contain skeleton DWARF debug information and the .dwo files include the full DWARF debug information. There is a .dwo file for every source file so this method has some limitations with large projects. A program called dwp, which is a split DWARF packager, solves this problem by combining the .dwo files into a single .dwp file. [63]

### *5.2.11. DWARF Package Files*

While split DWARF object files allow compiling, linking and debugging a program more quickly with less link-time overhead, they are not optimally suited to saving the debug information for later debugging of a released program. A DWARF package file collects the debugging information from the object files, or from separate DWARF object files. This package has the .dwp extension mentioned earlier. A DWARF package file is an object file too, and it uses the same object file format as the corresponding application binary. The package file has only a file header, a section table, a number of DWARF debug information sections, and two index sections.

The index sections are called .debug_cu_index and debug_tu_index, cu meaning compilation unit and tu meaning type unit. The index files provide a quick way to locate debug information by compilation unit ID for compilation units and by type signature for type units. Both indexes use the same format and begin with a header, followed by a hash table of signatures, then a parallel table of indexes, a table of offsets, and then a table of sizes. [48]

# 6. IMPLEMENTATION

The goal of this thesis work was to compare methods that protects intellectual property by making reverse engineering more difficult. This chapter will list five possible methods to do so. Three of the methods delete the debug data and two of the methods modifies the code.

A customer program image is built without debug data, so examining the large structs containing system state or protocol data received from the network, is very difficult and time-consuming.

In some cases, binary libraries of some modules are given to a customer alongside a partial source release. Then the customer will make their own changes to the code and build the software. When such a dump is debugged, there is no debug data in the ELF file as the binary libraries are stripped of the debug data. The software can be build internally with the debug data. Such 'debug ELFs' can be used instead.

Using such a debug ELF has its own problems. The debug ELF may have the wrong addresses for symbols. Also, the symbol layout may vary, as debug builds often include code that is not in the customer build. So overall, customer ELFs have correct symbol addresses but no type information for symbols. A debug ELF has type info but not all the type info is correct, and the symbol addresses are wrong. Currently standard practice is to find the address of a symbol using the customer ELF, then load the debug ELF and cast that address to the correct type. When there are multiple symbols you need to look at, this gets quite painful.

The first and rather obvious idea is to merge the type info into the customer ELF. Some other ideas were stripping the libraries from the debug data and adding them to the GDB when debugging. GDB also supports multiple symbol files, so symbols could be loaded from the ELF file that way. Debug fission was also considered, as well obfuscating the software during the compilation on IR level. The sections below describe each potential solution and it was tested.

## 6.1. Debug Fission

Large applications might have an extensive amount of debug data, which can cause problems. Debug fission is a response to these problems [64]. In debug fission, the debug information is split at compile time. Part is in the object file and another part is written to a parallel DWARF object file, which have a suffix of .dwo.

Debug fission is enabled with a '-gsplit-dwarf' switch in gcc versions 4.7 or greater. Fission must be used when only compiling, so the '-c' (compile source file to object code, do not invoke linker) switch is required when using '-gsplit-dwarf'. The .dwo files need to be linked with gold linker's '–gdb-index' switch at link time, or the user must manually set GDB's debugging directory to the location of the .dwo files with the GDB command 'set debug file directory *directories*' [65].

With debug fission, several benefits were expected, namely faster build time, and smaller build files. When build times were compared, link time decreased but post processing time increased more than link time decreased. So the total build time increased with debug fission. Debug fission also left some proprietary debug information in the ELF files, so this option did not meet the basic requirements of IP

protection. Split dwarf greatly reduced the size of the ELF files but when the .dwo files were packed into a .dwp file (DWARF objects packed), the total size was larger than without debug fission. It is worth noting that a rather old compiler version was used in these measurements. A more up-to-date version might provide better results.

## 6.2. Modifying ELF with Pyelftools

Another idea that seemed obvious was to modify the existing customer ELF, adding the symbol data of the debug ELF. This solution would not resolve any potentially wrong symbol layouts. Developing a program for this solution started with a python library called pyelftools, which parses and analyses ELF files and DWARF information [66]. The program would need to:

1. Take the user ELF, the debug ELF, the map file from the debug ELF and a list of compilation units as inputs.
2. Find the debug section for each compilation unit and record its length and offset from the map file.
3. Merge the debug sections ranges to the user ELF's section header.
4. Find call frame indexes and write them to the new ELF file
5. Find symbols and write the DWARF tags to corresponding location

Writing the DWARF tags turned out to be more complex than it initially appeared. Given a compilation unit, one can easily iterate through the DIEs with pyelftools. DIEs have DWARF tags as attributes. The DW_AT_name attribute tells us the name of the symbol, and we can use its value to get the symbol value from the .symtab section. If the DW_AT_name value can be found in the .symtab, then we can take the tags offset and write it to the new ELF file. However, sometimes the DW_AT_name attribute does not exist, and a symbol can only be found in other DIEs. For example, DW_AT_abstract_origin is used for inline instances of inline subprograms, or out-of-line instances of inline subprograms. The DWARF standard lists all the attributes in a table. It is almost six pages long, so there are lots of attributes.

In practice, finding the correct offsets for DWARF got too complicated for this approach to be suitable. For example, cached DIEs can have a .isra or a .constprop suffix added to the name. This needs to be taken into account when finding symbols. Range list offsets needs to be calculated if they are the same as the DW_AT_low_pc. [48]

## 6.3. Stripping debug info

The GNU Binary Utilities, or 'binutils', are a collection of tools for working with binary programs, object files, assembly source code and more. The main tools are 'ld', which is a linker, and a 'as', which is an assembler. Some of the other tools include addr2line, gold, nm, objcopy, objdump and readelf. [67]

Objcopy takes an object file as a parameter. If a second file name is given as a parameter, a copy of the input file will be created with a name of the second parameter. For example 'objcopy elf elf-copy' would copy an object file called 'elf' to a file called

‘elf-copy‘. Objcopy has many options that alter the behaviour of the command. For example, there are options called ’–strip-debug’ and ’–only-keep-debug’. If the ’–strip-debug’ option is passed to objcopy, it does not copy any debugging symbols or sections from the input file. The option ’–only-keep-debug’ does the exact opposite – it only copies debugging symbols and sections from the source file. [68]

Let us illustrate this with a small C program, seen in Appendices A, B and C. The file hello.c can be compiled into an object file with the debug data with the command ’gcc -g -c hello.c -o hello.o’. The main.c file can be compiled into an object file without debug data in a similar fashion with the command ‘gcc -c main.c -o main.o‘. These two object files can be linked together with the command ’gcc main.o hello.o -o main’. This will create an executable called main, which will run the program we just created.

As hello.o contains debug information, we can find the layout of the Greetings struct with GDB.

```
$ gdb -q -ex 'ptype g' -ex quit main
Reading symbols from main...
type = struct Greetings {
      STR hello;
      STR byeybye;
}
```

The debug data can be copied to a new object file with the command ’objcopy –only-keep-debug hello.o hello.debug’. This will create a new object file called hello.debug, which has only the debug symbols and sections from the hello.o. The debug information that was just copied can be removed from the original object file with the command ‘objcopy –strip-debug hello.o‘. Now if hello.o and main.o are linked again into an executable and we try to find the layout of the Greetings struct, no data is found.

```
$ gdb -q -ex 'ptype g' -ex quit main
Reading symbols from main...
type = <data variable, no debug info>
```

GDB supports multiple symbol files and the debug data can be added with a GDB command ‘add-symbol-file‘ [69].

```
gdb -q -ex 'add-symbol-file hello.debug' -ex 'ptype g' -ex
   quit main
Reading symbols from main...
add symbol table from file "hello.debug"
(y or n) y
Reading symbols from hello.debug...
type = struct Greetings {
      STR hello;
      STR byebye;
}
```

The stripped object file can be inspected more closely with the readelf program. Readelf displays information about ELF format object files. It has multiple options to control what information is being displayed. For example, with the ‘–file-header‘ option we can inspect the information in the ELF header at the start of the file. The

option ‘–sections‘ will display the information in the file’s section headers [70]. The stripped object file has the following content:

```
$ readelf --wide --sections hello.o
There are 13 section headers, starting at offset 0x3b8:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags Link Info Align
  [ 0]                   NULL            0000000000000000 00000000
       0000000000000000  0000000000000000  0    0    0
  [ 1] .text             PROGBITS        0000000000000000 00000040
       000000000000003d  0000000000000000  AX 0 0   1
  [ 2] .rela.text        RELA            0000000000000000 00000278
       0000000000000090  0000000000000018  I 10 1   8
  [ 3] .data             PROGBITS        0000000000000000 0000007d
       0000000000000000  0000000000000000  WA 0 0   1
  [ 4] .bss              NOBITS          0000000000000000 00000080
       0000000000000010  0000000000000000  WA 0 0   16
  [ 5] .rodata           PROGBITS        0000000000000000 00000080
       000000000000001a  0000000000000000  A 0  0    1
  [ 6] .comment          PROGBITS        0000000000000000 0000009a
       000000000000001c  0000000000000001  MS 0 0   1
  [ 7] .note.GNU-stack PROGBITS 0000000000000000 000000b6
       0000000000000000  0000000000000000  0    0    1
  [ 8] .eh_frame         PROGBITS        0000000000000000 000000b8
       0000000000000078  0000000000000000  A 0  0    8
  [ 9] .rela.eh_frame RELA               0000000000000000 00000308
       0000000000000048  0000000000000018  I 10 8   8
  [10] .symtab           SYMTAB          0000000000000000 00000130
       0000000000000120  0000000000000018 11   8    8
  [11] .strtab           STRTAB          0000000000000000 00000250
       0000000000000024  0000000000000000  0    0    1
  [12] .shstrtab         STRTAB          0000000000000000 00000350
       0000000000000061  0000000000000000  0    0    1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I
     (info),
  L (link order), O (extra OS processing required), G (group),
     T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

The output tells us that there are no debug sections in the file. If we compare to the original object file with all the debug data, we can see the debug sections that were stripped.

```
$ gcc -g -c hello.c -o debug-hello.o
$ readelf --sections debug-hello.o
There are 21 section headers, starting at offset 0x9c0:

Section Headers:
```

```
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags Link Info Align
  [ 0]                   NULL            0000000000000000 00000000
       0000000000000000  0000000000000000  0    0    0
  [ 1] .text             PROGBITS        0000000000000000 00000040
       000000000000003d  0000000000000000  AX   0    0    1
  [ 2] .rela.text        RELA            0000000000000000 00000598
       0000000000000090  0000000000000018   I   18   1    8
  [ 3] .data             PROGBITS        0000000000000000 0000007d
       0000000000000000  0000000000000000  WA   0    0    1
  [ 4] .bss              NOBITS          0000000000000000 00000080
       0000000000000010  0000000000000000  WA   0    0    16
  [ 5] .rodata           PROGBITS        0000000000000000 00000080
       000000000000001a  0000000000000000   A   0    0    1
  [ 6] .debug_info       PROGBITS        0000000000000000 000000a0
       00000000000000a1  0000000000000000   C   0    0    8
  [ 7] .rela.debug_info  RELA            0000000000000000 00000628
       0000000000000258  0000000000000018   I   18   6    8
  [ 8] .debug_abbrev     PROGBITS        0000000000000000 00000148
       0000000000000092  0000000000000000   C   0    0    8
  [ 9] .debug_aranges    PROGBITS        0000000000000000 000001e0
       000000000000002f  0000000000000000   C   0    0    8
  [10] .rela.debug_[...] RELA            0000000000000000 00000880
       0000000000000030  0000000000000018   I   18   9    8
  [11] .debug_line       PROGBITS        0000000000000000 0000020f
       000000000000005d  0000000000000000   0    0    1
  [12] .rela.debug_line  RELA            0000000000000000 000008b0
       0000000000000018  0000000000000018   I   18   11   8
  [13] .debug_str        PROGBITS        0000000000000000 00000270
       00000000000000ae  0000000000000001  MSC  0    0    8
  [14] .comment          PROGBITS        0000000000000000 0000031e
       000000000000001c  0000000000000001  MS   0    0    1
  [15] .note.GNU-stack   PROGBITS        0000000000000000 0000033a
       0000000000000000  0000000000000000   0    0    1
  [16] .eh_frame         PROGBITS        0000000000000000 00000340
       0000000000000078  0000000000000000   A   0    0    8
  [17] .rela.eh_frame    RELA            0000000000000000 000008c8
       0000000000000048  0000000000000018   I   18   16   8
  [18] .symtab           SYMTAB          0000000000000000 000003b8
       00000000000001b0  0000000000000018  19   14   8
  [19] .strtab           STRTAB          0000000000000000 00000568
       000000000000002c  0000000000000000   0    0    1
  [20] .shstrtab         STRTAB          0000000000000000 00000910
       00000000000000b0  0000000000000000   0    0    1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I
     (info),
  L (link order), O (extra OS processing required), G (group),
     T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
```

```
l (large), p (processor specific)
```

The Option ‘–debug-dump‘ displays the contents of the DWARF debug sections in the file. When the ‘readelf‘ is run with that option on ‘hello.o‘, not much is displayed, not even base types, like int. When the same command is run for the object file with debug information, the output is over 7 times longer.

## 6.4. Optimisations

Let us illustrate this with a simple C program, seen in Appendix D. The program creates a simple linked list and uses bubblesort to sort it.

Source codes files are compiled into an object file with the ‘nanomips-elf-gcc‘ compiler, disassembly can be extracted with the ‘nanomips-elf-gdb‘. NanoMIPS Base ISA Technical Reference Manual can be found from the MIPS website [71].

The GDB’s disassembly output for all the functions can be seen in Appendices F, G, H, I and J.

The software is compiled with ‘-O3‘ optimisation and functions and data are placed in their own sections and section garbage collection is used. The disassembly with the source code reference can be seen in Appendix K. The source code information can be seen with the GDB command ‘disas /s <function>‘. When the source files cannot be found, only file name and line numbers are shown. This can be seen in Appendix L. When all the debug data is stripped, no references to the original source code cannot be seen. Such disassembly can be seen in Appendix M.

Downside of the ‘-O3‘ optimisation is the increase of the size. ‘-Os‘ is an optimisation level with the size in mind. It enables same optimisations than ‘-O2‘ except those that often increase the code size. Another optimisation is link time optimisation (LTO). LTO writes IR to the special ELF section. When the objects files are linked, function bodies are read from ELF and instantiated as if they had been part of the same translation unit.

Only the main functions disassembly is available as the software has seen some heavy optimisation. Functions have been inlined to the main function, and as data and functions are in their own sections with garbage collection enabled, references to inlined functions are deleted.

The amount of instructions and branches in the disassembled functions can be seen in Table 4. The optimised main function is 46 instructions long, with all the other functions inlined. The unoptimised disassembly had total of 126 instructions and 12 branches.

Table 4. Complexity of disassembled function

| Disassembly | Instructions | Branches |
|---|---|---|
| Unoptimised main | 24 | 4 |
| Unoptimised b_insert | 20 | 1 |
| Unoptimised b_sort | 36 | 7 |
| Unoptimised b_swap | 16 | 0 |
| Unoptimised init_arr | 31 | 0 |
| Optimised main | 46 | 7 |

### 6.5. Obfuscations

Few LLVM based obfuscation tools were tried. However, none of them were successfully integrated to the build environment and the toolchain. Existing results of the obfuscations results were search instead. Pascal Junod et al. had run some benchmark on OpenSSL with, and without, obfuscations [31]. These results will be compared.

LLVM obfuscator provides bogus control flow, control flow flattening and instruction substitution obfuscations. All these obfuscations adds code(?), so the library size is bound to be increased. The size of the libcrypto.a and libssl.a with and without different obfuscations can be seen in Table 5 and total size of the obfuscated libraries in percentages can be seen in the Table 6. The size is in bytes. The option for the column "Size with bogus control flow" is with the default compiler options: the pass is run three times, and a basic block is being obfuscated with a probability of 30%.

Table 5. Size of the OpenSSL libraries with and without obfuscations

| Library | libcrypto.a | libssl.a |
|---|---|---|
| No obfuscation | 3371184 | 592336 |
| Instruction substitution | 3382600 | 594128 |
| Bogus control flow | 3395424 | 593936 |
| Control flow flattening | 4388912 | 867456 |
| Control flow flattening and bogus control flow | 57046064 | 1191376 |
| All previously mentioned obfuscations | 21441280 | 5003640 |

Table 6. Size increase in the obfuscated OpenSSL libraries compared to unobfuscated libraries

| Library | libcrypto.a | libssl.a |
|---|---|---|
| Instruction substitution | 100.34% | 100.30% |
| Bogus control flow | 100.72% | 100.27% |
| Control flow flattening | 130.19% | 146.45% |
| Control flow flattening and bogus control flow | 169.20% | 201.13% |
| All previously mentioned obfuscations | 636.02% | 844.83% |

Besides the size of the libraries, execution speed can be measured. The results of the OpenSSL benchmarks on different algorithms, namely MD5, SHA512, RSA1024 and RSA4096 can be seen in Tables 7 and 8. The benchmark runs the algorithm routine with a different sized inputs. The results are processed bytes, in thousands, per second. Last row with all the obfuscation enabled has some additional compiler options enabled. Bogus control flow probability is 100% and the pass is run two times.

Table 9 and 10 shows the benchmarks for RSA algorithm with different moduli. Table 11 shows the performance difference between unobfuscated, instruction substitution, bogus control flow and control flow flattening obfuscated binaries. Percentages are the loss of performance compared to unobfuscated binary.

Table 7. OpenSSL benchmark for MD5 algorithm

| Obfuscations applied | 16 bytes | 64 bytes | 256 bytes | 1024 bytes | 8192 bytes |
|---|---|---|---|---|---|
| No obfuscations | 23552.89k | 78022.21k | 202613.08k | 337656.15k | 412469.93k |
| Instruction substitution | 20147.98k | 65221.23k | 182223.70k | 297061.38k | 373598.89k |
| bogus control flow | 21961.77k | 75055.73k | 191669.76k | 322189.99k | 405228.20k |
| control flow flattening | 4626.06k | 17899.95k | 63085.57k | 175522.13k | 357381.46k |
| control flow flattening and bogus control flow | 2426.18k | 8767.02k | 27689.22k | 60316.67k | 92009.81k |
| instruction substitution, control flow flattening and bogus control flow | 334.18k | 1335.83k | 5155.75k | 17650.01k | 59845.29k |

Table 8. OpenSSL benchmark for SHA512 algorithm

| Obfuscations applied | 16 bytes | 64 bytes | 256 bytes | 1024 bytes | 8192 bytes |
|---|---|---|---|---|---|
| No obfuscations | 16267.34k | 64235.24k | 112194.30k | 167185.07k | 190887.25k |
| Instruction substitution | 14304.69k | 56231.81k | 95766.36k | 140968.96k | 160869.03k |
| bogus control flow | 15192.15k | 58753.58k | 103328.60k | 152055.47k | 174394.03k |
| control flow flattening | 4058.90k | 16660.07k | 27805.70k | 40517.63k | 46282.07k |
| control flow flattening and bogus control flow | 1968.08k | 8033.96k | 13190.49k | 19611.65k | 22921.22k |
| instruction substitution, control flow flattening and bogus control flow | 289.59k | 1162.75k | 1818.71k | 2561.02k | 2894.51k |

Table 9. OpenSSL benchmark for RSA with moduli of 1024 bits

| Obfuscations applied | sign | verify | sign/s | verify/s |
|---|---|---|---|---|
| No obfuscations | 0.18ms | 0.01ms | 5493.4 | 70164.1 |
| Instruction substitution | 0.91ms | 0.04ms | 1098.5 | 24574.0 |
| bogus control flow | 0.93ms | 0.04ms | 1066.5 | 23674.0 |
| control flow flattening | 9.89ms | 0.42ms | 101.1 | 2377.3 |
| control flow flattening and bogus control flow | 17.41ms | 0.71ms | 57.4 | 1419.0 |
| instruction substitution, control flow flattening and bogus control flow | 111.89ms | 4.74ms | 8.9 | 210.8 |

Table 10. OpenSSL benchmark for RSA with moduli of 4096 bits

| Obfuscations applied | sign | verify | sign/s | verify/s |
|---|---|---|---|---|
| No obfuscations | 32.71ms | 0.44ms | 30.6 | 2272.5 |
| Instruction substitution | 34.36ms | 0.47ms | 29.1 | 2132.5 |
| bogus control flow | 35.46ms | 0.48ms | 28.2 | 2078.8 |
| control flow flattening | 328.71ms | 4.60ms | 3.0 | 217.3 |
| control flow flattening and bogus control flow | 521.00ms | 6.90ms | 1.9 | 145.0 |
| instruction substitution, control flow flattening and bogus control flow | 3710.00ms | 52.08ms | 0.3 | 19.2 |

Table 11. Performance loss compared to unobfuscated results in MD5 benchmark

| Obfuscations applied | 16 bytes | 64 bytes | 256 bytes | 1024 bytes | 8192 bytes |
|---|---|---|---|---|---|
| Instruction substitution | 14.45% | 16.41% | 10.06% | 12.02% | 9.42% |
| bogus control flow | 6.76% | 3.80% | 5.40% | 4.58% | 1.76% |
| control flow flattening | 80.36% | 77.06% | 68.86% | 48.02% | 13.36% |

# 7. DISCUSSION

As it turned out in Chapter 6, the solution involving binary tools worked well. The other solutions had some trade-offs that will be discussed in this chapter. Likewise, the changes required to apply the solution are discussed, together with the necessary the changes in workflow. Lastly some future work is considered.

## 7.1. Trialed solutions

The other solutions tried besides using binary tools were debug fission, ELF file injection, and obfuscation. All of these will be discussed in more detail in this section.

### *7.1.1. Debug Fission*

The main issue with debug fission was inadequate IP protection. As not all debug data was placed into the .dwo files, debug fission was not suitable as is. It would have been easy for the developers to adapt to, as almost all the changes would have been in the tools or the software build side. However, this would have caused some issues in the build environment.

Integrating debug fission into the build systems is not the most straightforward as the toolchain needs to be updated and it would have required a lot of background work and testing from the toolchain developers, not to mention the increased storage requirements and increased link times. As the debug fission left some proprietary information to the ELF files, it was not suitable solution. Even when combining debug fission with stripping the debug data from the object files, it would still be far from ideal solution as the storage requirements and linking times increased.

It is important to notice that the proof of concept did not use the most recent toolchain versions, so there could be improvements in the results with a newer toolchain version. Because the adaptation of a new toolchain version takes a toll, an older version was deemed adequate.

### *7.1.2. ELF file injection*

Pyelftools provided a good API to analyse and parse ELF files. Even though the solution seemed simple to develop initially, it was quickly discovered to be a complex. Maintaining the developed solution would have been a big increase to the workload, on top of the existing limited resources. Not to mention the problem of the different symbol layouts between the user and debug ELF files might be different in some cases.

This solution is good for keeping IP safe and does not affect any build systems. Users need to run the program themselves, or it can be integrated into an existing program that generates core files from the dumps. Parsing the ELF file took a long time, so it is not optimal in that way either, as developers might have multiple different issues every day and one issue might have multiple memory dumps. The slowness of the tool would increase the total working time substantially. ELF file injection is completely

possible to do, and a small-scale proof of concept was successfully completed, but more resources would need to be allocated to finish and maintain the solution.

### *7.1.3. Stripping debug info*

Stripping and glueing back the debug info was successful. The process is not heavy on the resources, adds IP protection to a certain degree, and it easy to enable the solution.

### *7.1.4. Optimisations*

Optimisations are somewhat a good way to protect the IP. Optimisations are trade off between the binary sizes and the amount of optimisations done. We can see from Table 4 that optimised disassembly is shorter. Even so, reverse engineering might be harder as there is less context. The example code was rather short, so optimisation from LTO were not seen. LTO makes the code more optimised and harder to reverse engineer.

### *7.1.5. Obfuscation*

Existing implementation were not working together with the build environment. Instead, existing results were studied. Library sizes started to increase even with a single obfuscations. When multiple obfuscations were used, library sizes easily doubled. Depending of the library, the size could increase over 600% or 700%. Considering the limitations of the embedded devices, these results are not acceptable.

The performance decrease was also seen after applying the obfuscations. RSA benchmarks showed a big difference in the performance even with the single obfuscations. The performance decrease depends on the algorithm and the input size. Smallest decrease overall was in the bogus control flow obfuscation. The loss of performance varies from 1.76% to 6.76%. Other obfuscations showed more performance loss even in the best case scenarios. These kinds of results are unacceptable for real-time operating system with time critical domains. Especially because obfuscation like instruction substitution is easily removed by re-optimising the generated code.

## 7.2. Results

This thesis work attempts to find the most suitable way to ease the burden of debugging customer issues without exposing sensitive IP. Some goals that had to be kept in mind while looking for a solution were:

1. Keeping intellectual property safe
2. Enabling easier debugging of customer issues
3. Finding an easy way for developers to use the solution
4. Finding a sensible solution that could be easily integrated into the build system

5. The size and the performance of the released binaries should not decrease

Table 12. IP Protection method comparison

| Method | Optimisation | LTO | Removing debug symbols | Obfuscation |
|---|---|---|---|---|
| Performance Gains | Yes | Yes | No | No |
| Interusability | Obfuscation can revert optimisations | Yes | Yes | Optimisations can revert obfuscations |
| Makes reverse engineering harder | Might be easier for experienced debugger | Yes | Yes | yes |
| Increase to size | Yes | No | No | Yes |
| Harder debugging | No | Yes, possibly | No | Yes |

IP protection is a hard requirement with no room for compromise. The solution had to balance the trade-offs between the other goals. After discussion, four options were evaluated: debug fission, injecting the ELF file, injecting the core file, and the usage of existing GNU bintools. Obfuscation tools were not successfully tested but existing results were studied. The existing results indicated unacceptable increase in the library sizes. Enabling obfuscations could result in the image file that cannot be flashed to the device. Besides the increase of the libraries, performance decreased a lot for a system where each clock cycle is considered important.

The solution using binary tools turned out to be a great success, even though it was initially deemed as unsuitable after earlier trials few years ago. The toolchain and its binary utilities have matured and now the implementations work better than before. This solution keeps the IP safe and there is no additional work for developers to enable the solution. The solution is also fast to implement in the build system and does not increase build time. However, more space is required to store the debug data. This can get quite large as there are plenty of products and their software can be built daily, even multiple times per day.

### 7.3. Workflow

In Figure 9 we can see the life cycle of the old debugging flow. It requires the whole build process to be done twice and requires launching two debugger instances, not to mention finding the address from the user load and using the second debugger instance to find the correct symbol layout.

The new debugging flow, seen in Figure 10 shows how much simpler the new way of working is. Engineers do not necessarily need to download all the debug files but instead they can download the debug file of their own modules. This way they do not need to load all the debug files into GDB, or load one big debug file into GDB. Engineers can even preserve old debugging files if they know there have not been any changes in the debug information. This makes the debugging flow even faster.

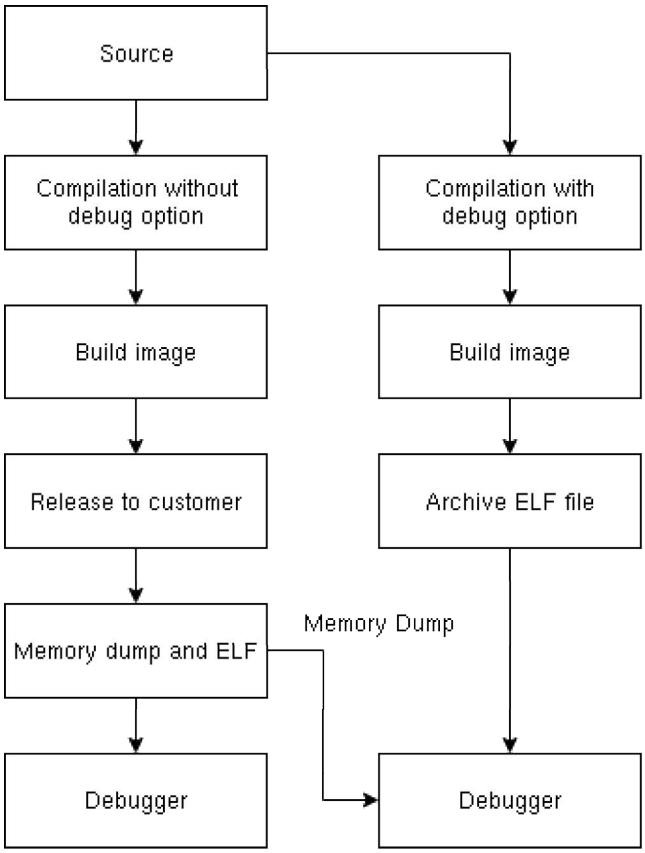


Figure 9. Old debugging flow.

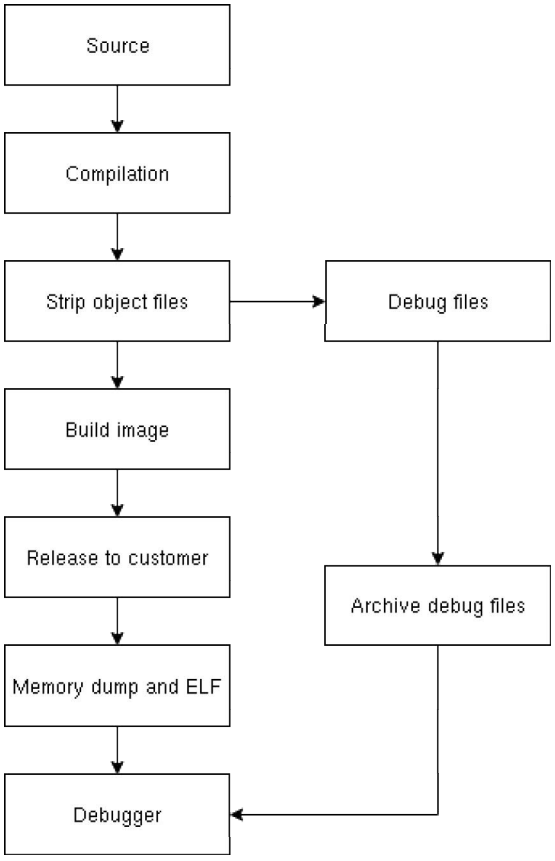


Figure 10. New debugging flow.

## 7.4. Required changes

Some tools require minor changes, but the big change needed is in the build system. Currently the majority of modules are not compiled with debug data so the build size will increase. In the end, this can add a lot to the space requirements.

The stripped debug data also needs to be stored somewhere, so the space requirement is not only at build time. Stripping the debug data is fortunately rather fast, so the only increase to the build time comes from the debug data. An additional benefit to this solution is that the debug data will be file specific. Most often developers only want to examine variables in their modules. Alternatively, developers could ask module owners what file they need to load in order to save space and not download all the debug files. Splitting debug data out of every object file enables developers to do all that and they do not have to load the debug data for hundreds of other modules.

## 7.5. Future Work

While working intensively with the DWARF format, some additional ways to solve the original issue were identified. Maybe the best solution in theory would be to generate the DWARF data with some sort of DWARF generator. That way every developer could create symbol files with the debug information they needed. Developers typically know when their data changes, so they would not need to regenerate the symbols for every issue. The input to the DWARF generator could be JSON or code files, such as file containing all the wanted structs and defines, or object files. To make the generator efficient, it would need to be an in-house implementation.

A python module called dwarfgen [72] was found but it was not extensively tested or worked with as it fell out of the scope of the thesis because of time constraints. It would not have been a straightforward solution, but it could been a good starting point. Another interesting library found was libdwarf [73]. This library provided a program called dwarfgen and more research could be conducted on it.

One solution considered late in the project was adding custom functionality to GDB. If one could tell GDB to ignore addresses from the symbol table, a debug ELF file could be used without any modifications. However, some symbol layout would be wrong in the debug ELF. Developers working with the debug ELFs usually only look at symbols from their own modules, and they know usually know when their modules have customer release specific code, which changes the symbol layout. Debugging ELF would not be ideal in some cases, but overall efficiency would be improved a lot. However, a ready solution for this was not found and time constraints prevented more familiarity with the GDB source code. Any custom work would also require maintenance work later.

Another future work item could be core file injection. Researching core files more could produce some new ideas for tackling the debugging issue. Also, when the toolchain version is internally updated to a more recent version, the debug fission ought to be retested.

At the very end of finishing the work part of the thesis, a cross platform library called LIEF [43] was found. LIEF can parse and manipulate executable file formats and its purpose is to provide a library that supports different operations on different executable

file formats. It provides an API for different languages and abstracts features from the different formats [74]. LIEF's blog [75] explains that there is an ELF builder integrated into LIEF. While the LIEF is not exactly in the scope of this thesis, as the LIEF's focus seem to be more reverse debugging than working with DWARF debug information, LIEF is still an actively developed library that could be researched more and followed actively in the case it later supports features that do fit this thesis' scope.

As the objcopy method seemed promising, one future item is to actually implement the required changes to the build system, to try the object debug data copying and stripping in practice with a full modem software build.

Obfuscation could be studied more and trialed in the actual embedded device. This way more suitable modules and scenarios for obfuscations could be found. It should be noted that this would require in developing the in-house obfuscator as the use case is rather specific. Company might not see this plausible, as the existing results indicate too much of unwanted side effects, such as increase in the library size and bugs caused by the obfuscation.

# 8. CONCLUSIONS

Protecting IP is important for companies providing proprietary software. Reverse engineering can happen by exploiting debug data. Therefore, companies are not willing to give debug data away even to trusted customers, due to worries over data leaks or malicious intentions from third parties. Even though IP needs to be kept safe, companies want to be the first to market, especially with flagship products. Customers are demanding and require fast response to the issues that developers are faced with. It is no good that the IP protection clashes with these two essential requirements, slowing down issue resolution.

While compilers and linkers have evolved, the principles to understand them are the same as in the past. Protecting IP is not a new requirement, and neither is the problem of how to hide debug data and still work with it later.

Initially the best sounding solutions were debug fission, injecting the ELF file with debug data, injecting the core file with debug data, and stripping the debug data and using it again later. Later, more ideas were developed, such as generating symbol files without addresses on demand or adding a functionality to GDB to ignore the addresses from an externally loaded symbol table. A great deal of future work remains in this area, but some solutions exist. The most suitable solution was stripping the debug data and adding it later to GDB, as it fulfills all the criteria described in Chapter 7. This solution does not compromise IP and it is easy and fast to integrate into the current workflow, it enables loading the object data only of wanted modules, and it requires little or no change to the internal tools. However, build size increases as the amount of debug data increases, and thus it also increases build time.

The results of the thesis were somewhat contradictory. On the one hand, binary tool utilities were researched again and found to be useful. The output of object file stripping is also more useful than expected as the developer can load the debug data per stripped module. On the other hand, changing the build system has its own bureaucracy and side effects. However, the relief in the developer's work is worth the extra development time. Then again, injecting the ELF file was deemed to be unsuccessful as it was slow, complex and hard to maintain.

# 9. REFERENCES


[1] System-on-a-chip - embedded artistry. https://embeddedartistry.com/fieldmanual-terms/system-on-a-chip/. Accessed 2022-07-17.

[2] Suh D., Hwang J. & Oh D. (2008) Do software intellectual property rights affect the performance of firms? case study of south korea. In: 2008 The Third International Conference on Software Engineering Advances, pp. 307–312.

[3] Murray D. (2020) Open source and security: why transparency now equals strength. Network Security 2020, pp. 17–19. URL: https://www.sciencedirect.com/science/article/pii/S1353485820300829.

[4] Olcoz K., Tirado F. & Mecha H. (1999) Unified data path allocation and bist intrusion. Integration 28, pp. 55–99. URL: https://www.sciencedirect.com/science/article/pii/S0167926099000127.

[5] Sosnowski J. (2006) Software-based self-testing of microprocessors. Journal of Systems Architecture 52, pp. 257–271. URL: https://www.sciencedirect.com/science/article/pii/S1383762105000780.

[6] Evans D. (2011) How the next evolution of the internet is changing everything. Tech. rep.

[7] Research J. (2020) Iot the internet of transformation 2020. Tech. rep.

[8] Martin G. (2002) How to choose semiconductor ip: embedded software. In: Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition, pp. 16–.

[9] Rajendran J.J.V. (2017) An overview of hardware intellectual property protection. In: 2017 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–4.

[10] ul Iman M. & Ishaq A.F.M. (2010) Anti-reversing as a tool to protect intellectual property. In: 2010 Second International Conference on Engineering System Management and Applications, pp. 1–5.

[11] Looking for vulnerabilities in mediatek audio dsp. https://research.checkpoint.com/2021/looking-for-vulnerabilities-in-mediatek-audio-dsp/. Accessed 2022-07-12.

[12] Flaw in qualcomm modems enables backdoor for hackers to record your phone calls. https://www.gsmarena.com/flaw_in_qualcomm_modems_allows_backdoor_for_hackers_to_record_your_phone_calls-news-49007.php/. Accessed 2022-07-12.

[13] Critical mediatek rootkit affecting millions of android devices. https://www.xda-developers.com/mediatek-su-rootkit-exploit/. Accessed 2022-07-12.

[14] Unisoc phone chip firmware vulnerable to remote crash. https://www.theregister.com/2022/06/03/uisoc-chip-flaw-check-point/. Accessed 2022-07-12.

[15] Cve-2022-20210. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-20210. Accessed 2022-07-12.

[16] Ou C., Zhang X., Angelopoulos S., Davison R. & Janse N. (2022) Security breaches and organization response strategy: Exploring consumers' threat and coping appraisals. International Journal of Information Management 65, p. 102498.

[17] Tian G.Y. (2009) Intellectual property (ip) protection versus ip abuses: The recent development of chinese ip abuse rules and recommendations for foreign technology-driven companies. Computer Law & Security Review 25, pp. 352–366.

[18] Aho A.V., Lam M.S., Sethi R. & Ullman J.D. (2006) Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley.

[19] Dŏ0e9vai R., Jŏ0e1sz J., Nagy C. & Ferenc R. (2013) Designing and implementing control flow graph for magic 4th generation language.

[20] Patterson D.A. & Hennessy J.L. (2013) Computer Organization and Design, Fifth Edition: The Hardware/Software Interface. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th ed.

[21] Salomon D. (1992) Assemblers and Loaders. Ellis Horwood.

[22] Presser L. & White J.R. (1972) Linkers and loaders. ACM Comput. Surv. 4, pp. 149–167.

[23] Gcc, the gnu compiler collection. https://gcc.gnu.org/. Accessed 2022-08-31.

[24] Clang: a c language family frontend for llvm. https://clang.llvm.org/. Accessed 2022-08-31.

[25] The llvm compiler infrastructure. https://llvm.org/. Accessed 2022-08-31.

[26] Llvm language reference manual. https://llvm.org/docs/LangRef.html. Accessed 2022-07-22.

[27] Gimple (gnu compiler collection (gcc) internals). https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html#GIMPLE. Accessed 2022-07-22.

[28] Gnu c compiler internals/gnu c compiler architecture. https://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture. Accessed 2022-07-22.

[29] Lattner C. & Adve V. (2004) Llvm: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004., pp. 75–86.

[30] Junod P., Rinaldini J., Wehrli J. & Michielin J. (2015), Obfuscator-llvm – software protection for the masses.

[31] Junod P., Rinaldini J., Wehrli J. & Michielin J. (2015) Obfuscator-LLVM – software protection for the masses. In: B. Wyseur (ed.) Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015, IEEE, pp. 3–9.

[32] TIS (1995) Tool interface standard (TIS) executable and linking format (ELF) specification Version 1.2. TIS committee.

[33] Overview of the mach-o executable format. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/Articles/MachOOverview.html. Accessed 2022-07-11.

[34] Pe format - win32 apps | microsoft docs. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format. Accessed 2022-07-11.

[35] How-to reverse-engineering. https://www.computerworld.com/article/2585652/reverse-engineering.html. Accessed 2022-07-17.

[36] Musker D.C., Protecting & exploiting intellectual property in electronics. https://web.archive.org/web/20110709001411/http://www.jenkins.eu/articles-general/reverse-engineering.asp. Accessed 2022-07-11.

[37] Kumar M., Reverse engineering and vulnerability analysis in cyber security. http://www.ijarcs.info/index.php/Ijarcs/article/download/3502/3456. Accessed 2022-07-11.

[38] Chikofsky E. & Cross J. (1990) Reverse engineering and design recovery: a taxonomy. IEEE Software 7, pp. 13–17.

[39] Canfora G. & Di Penta M. (2007) New frontiers of reverse engineering. pp. 326 – 341.

[40] Hamilton V. (1994) The use of static analysis tools to support reverse engineering. In: IEE Colloquium on Reverse Engineering for Software Based Systems, pp. 6/1–6/4.

[41] Machine code (debugging with gdn). https://sourceware.org/gdb/onlinedocs/gdb/Machine-Code.html. Accessed 2022-08-04.

[42] radareorg/radare2: Unix-like reverse engineering framework and commandline toolset. https://github.com/radareorg/radare2. Accessed 2022-07-17.

[43] Github - lief-project/lief: Lief. https://github.com/lief-project/LIEF. Accessed 2022-07-11.

[44] Nationalsecurityagency/ghidra: Ghidra is a software reverse engineering (sre) framework. https://github.com/NationalSecurityAgency/ghidra. Accessed 2022-07-17.

[45] Hex rays - state-of-the-art binary code analysis solutions. https://hex-rays.com/ida-pro/. Accessed 2022-07-17.

[46] The framework - the official radare2 book. https://book.rada.re/first_steps/overview.html. Accessed 2022-07-17.

[47] Basile C., Canavese D., Regano L., Falcarin P. & Sutter B.D. (2019) A meta-model for software protections and reverse engineering attacks. Journal of Systems and Software 150, pp. 3–21. URL: https://www.sciencedirect.com/science/article/pii/S0164121218302838.

[48] DWARF Debugging Information Format Committee (2017) DWARF Debugging Information Format Version 5. DWARF5.

[49] Eager M.J. (2007), Introduction to the dwarf debugging format.

[50] Challenges when building an llvm-based obfuscator. https://llvm.org/devmtg/2017-10/slides/Guelton-Challenges_when_building_an_LLVM_bitcode_Obfuscator.pdf. Accessed 2022-08-31.

[51] Collberg C. & Thomborson C. (2002) Watermarking, tamper-proofing, and obfuscation - tools for software protection. IEEE Transactions on Software Engineering 28, pp. 735–746.

[52] den Broeck J.V., Coppens B. & Sutter B.D. (2020) Flexible software protection. CoRR abs/2012.12603. URL: https://arxiv.org/abs/2012.12603.

[53] Christou G., Vasiliadis G., Papaefstathiou V., Papadogiannakis A. & Ioannidis S. (2020) On architectural support for instruction set randomization. ACM Trans. Archit. Code Optim. 17. URL: https://doi.org/10.1145/3419841.

[54] Park J., Kim H., Jeong Y., je Cho S., Han S. & Park M. (2015) Effects of code obfuscation on android app similarity analysis. J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl. 6, pp. 86–98.

[55] Preemptive protection dasho. https://www.preemptive.com/dasho/pro/userguide/en/index.html. Accessed 2022-08-31.

[56] Oreans technologies : Software security defined. https://www.oreans.com/Themida.php. Accessed 2022-09-14.

[57] The development of the c language*. http://www.bell-labs.com/usr/dmr/www/chist.html. Accessed 2022-08-04.

[58] Stabs. https://sourceware.org/gdb/current/onlinedocs/stabs.html#Overview. Accessed 2022-07-22.

[59] Rfr: 8065656: Use dwarf debug symbols for solaris. https://mail.openjdk.org/pipermail/build-dev/2014-November/013726.html. Accessed 2022-07-22.

[60] Common object file format. https://www.ti.com/lit/an/spraao8/spraao8.pdf?ts=1658481725958&ref_url=https%253A%252F%252Fwww.google.com%252F. Accessed 2022-07-22.

[61] What is the difference between elf and coff format? https://microchipsupport.force.com/s/article/What-is-the-difference-between-ELF-and-COFF-format. Accessed 2022-07-22.

[62] The dwarf debugging standard. https://dwarfstd.org/. Accessed 2022-07-28.

[63] RogueWaveSoftware (2017) Saving time and space with split dwarf. Tech. rep.

[64] Debugfission - gcc wiki. https://gcc.gnu.org/wiki/DebugFission. Accessed 2022-07-08.

[65] Separate debug files (debugging with gdb). https://sourceware.org/gdb/onlinedocs/gdb/Separate-Debug-Files.html. Accessed 2022-07-08.

[66] Github - eliben/pyelftools: Parsing elf and dwarf in python). https://github.com/eliben/pyelftools. Accessed 2022-07-11.

[67] Binutils - gnu project - free software foundation. https://www.gnu.org/software/binutils/. Accessed 2022-07-10.

[68] objcopy(1) — linux manual page. https://man7.org/linux/man-pages/man1/objcopy.1.html. Accessed 2022-07-10.

[69] Files (debugging with gdb). https://sourceware.org/gdb/onlinedocs/gdb/Files.html. Accessed 2022-07-10.

[70] readelf(1) — linux manual page). https://man7.org/linux/man-pages/man1/readelf.1.html. Accessed 2022-07-10.

[71] nanomips architecture. https://www.mips.com/products/architectures/nanomips/. Accessed 2022-08-05.

[72] dwarfgen - pypi). https://pypi.org/project/dwarfgen/. Accessed 2022-07-11.

[73] Da's dwarf page). https://www.prevanders.net/dwarf.html. Accessed 2022-07-11.

[74] Lief - library to instrument executable formats. https://lief-project.github.io/blog/2017-04-18-lief/. Accessed 2022-07-11.

[75] Lief - new elf builder. https://lief-project.github.io/blog/2022-01-23-new-elf-builder/. Accessed 2022-07-11.

# 10. APPENDICES

## A. Source Code of hello.h

```
#include <stdio.h>
typedef char* STR;
typedef struct Greetings {
   STR hello;
   STR byeybe;
} Greetings;
STR say_hello(void);
STR say_bye(void);
void init_greetings(void);
```

## B. Source Code of hello.c

```
#include "hello.h"
Greetings g;
STR say_hello(void)
{
   return g.hello;
}
STR say_bye(void)
{
   return g.byebye;
}
void init_greetings(void)
{
   g.hello = "Hello, World!";
   g.byebye = "Bye, World!";
}
```

## C. Source code of main.c

```
#include "hello.h"
int main(void)
{
   init_greetings();
   printf("\%s\n", say_hello());
   printf("\%s\n", say_bye());
   return 0;
}
```

## D. Source code of sort.c

```
#include "sort.h"
#define SIZE 7
int arr[SIZE];
static void init_arr(void)
{
   arr[0] = 31;
   arr[1] = 8;
   arr[2] = 2;
   arr[3] = 11;
   arr[4] = 9;
   arr[5] = 90;
   arr[6] = 100;
}
void b_insert(struct Node **llref, int data)
{
   struct Node *ptr = (struct Node*)malloc(sizeof(struct
       Node));
   ptr->data = data;
   ptr->next - *llref;
   *llref = ptr;
}
void b_swap(struct Node *a, struct Node *b)
{
   int temp = a->data;
   a->data = b->data;
   b->data = temp;
}

   void b_sort(struct Node *ll)
{
   int swapped;
   struct Node *cur;
   struct Node *next;
   if (ll == NULL)
      return;
   do {
      swapped = 0;
      cur = ll;
      while (cur->next != next) {
         if (cur->data > cur->next->data) {
            b_swap(cur, cur->next);
            swapped = 1;
         }
         cur = cur->next;
      }
      next = cur;
   } while (swapped);
}
```

```
int main(void)
{
   struct Node *ll = NULL;
   init_arr();
   for (int i=0; i<SIZE; i++)
      b_insert(&ll, arr[i]);
   b_sort(ll);
   return 0;
}
```

## E. Source code of sort.h

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node *next;
};
void b_insert(struct Node **llref, int data);
void b_sort(struct Node *ll);
void b_swap(struct Node *a, struct Node *b);
static void init_arr(void);
```

## F. Unoptimised disassembly of the main function

```
Dump of assembler code for function main:
     0x0040026c <+0>: save 32,fp,ra
     0x0040026e <+2>: addiu fp,sp,-4064
     0x00400272 <+6>: sw   zero,8(sp)
     0x00400274 <+8>: balc 0x400174 <init_arr>
     0x00400276 <+10>: sw  zero,12(sp)
     0x00400278 <+12>: bc  0x400296 <main+42>
     0x0040027a <+14>: aluipc a3,0x20
     0x0040027e <+18>: lw  a2,12(sp)
     0x00400280 <+20>: sll a2,a2,2
     0x00400282 <+22>: ori a3,a3,0x874
     0x00400286 <+26>: addu a3,a2,a3
     0x00400288 <+28>: lw  a2,0(a3)
     0x0040028a <+30>: addiu a3,sp,8
     0x0040028c <+32>: movep a0,a1,a3,a2
     0x0040028e <+34>: balc 0x4001d0 <b_insert>
     0x00400290 <+36>: lw  a3,12(sp)
     0x00400292 <+38>: addiu a3,a3,1
     0x00400294 <+40>: sw  a3,12(sp)
     0x00400296 <+42>: lw  a3,12(sp)
     0x00400298 <+44>: bltic a3,7,0x40027a <main+14>
     0x0040029c <+48>: lw  a3,8(sp)
     0x0040029e <+50>: move.balc a0,a3,0x4001fa <b_sort>
     0x004002a2 <+54>: move a3,zero:   move  a0,a3
     0x004002a6 <+58>: restore.jrc 32,fp,ra
End of assembler dump.
```

## G. Unoptimised disassembly of the b_insert function

```
Dump of assembler code for function b_insert:
   0x004001d0 <+0>: save 48,fp,ra
   0x004001d2 <+2>: addiu fp,sp,-4048
   0x004001d6 <+6>: sw   a0,12(sp)
   0x004001d8 <+8>: sw   a1,8(sp)
   0x004001da <+10>: li  a0,8
   0x004001dc <+12>: balc 0x4002a8 <malloc>
   0x004001de <+14>: move a3,a0
   0x004001e0 <+16>: sw  a3,28(sp)
   0x004001e2 <+18>: lw  a3,28(sp)
   0x004001e4 <+20>: lw  a2,8(sp)
   0x004001e6 <+22>: sw  a2,0(a3)
   0x004001e8 <+24>: lw  a3,12(sp)
   0x004001ea <+26>: lw  a2,0(a3)
   0x004001ec <+28>: lw  a3,28(sp)
   0x004001ee <+30>: sw  a2,4(a3)
   0x004001f0 <+32>: lw  a3,12(sp)
   0x004001f2 <+34>: lw  a2,28(sp)
   0x004001f4 <+36>: sw  a2,0(a3)
   0x004001f6 <+38>: nop
   0x004001f8 <+40>: restore.jrc 48,fp,ra
End of assembler dump.
```

## H. Unoptimised disassembly of the b_sort function

```
Dump of assembler code for function b_sort:
   0x004001fa <+0>: save 48,fp,ra
   0x004001fc <+2>: addiu fp,sp,-4048
   0x00400200 <+6>: sw   a0,12(sp)
   0x00400202 <+8>: sw   zero,20(sp)
   0x00400204 <+10>: lw  a3,12(sp)
   0x00400206 <+12>: beqzc a3,0x400246 <b_sort+76>
   0x00400208 <+14>: sw  zero,28(sp)
   0x0040020a <+16>: lw  a3,12(sp)
   0x0040020c <+18>: sw  a3,24(sp)
   0x0040020e <+20>: bc  0x400232 <b_sort+56>
   0x00400210 <+22>: lw  a3,24(sp)
   0x00400212 <+24>: lw  a3,0(a3)
   0x00400214 <+26>: lw  a2,24(sp)
   0x00400216 <+28>: lw  a2,4(a2)
   0x00400218 <+30>: lw  a2,0(a2)
   0x0040021a <+32>: bgec a2,a3,0x40022c <b_sort+50>
   0x0040021e <+36>: lw  a3,24(sp)
   0x00400220 <+38>: lw  a3,4(a3)
   0x00400222 <+40>: lw  a0,24(sp)
   0x00400224 <+42>: move.balc a1,a3,0x40024a <b_swap>
   0x00400228 <+46>: li  a3,1
   0x0040022a <+48>: sw  a3,28(sp)
   0x0040022c <+50>: lw  a3,24(sp)
   0x0040022e <+52>: lw  a3,4(a3)
   0x00400230 <+54>: sw  a3,24(sp)
   0x00400232 <+56>: lw  a3,24(sp)
   0x00400234 <+58>: lw  a2,4(a3)
   0x00400236 <+60>: lw  a3,20(sp)
   0x00400238 <+62>: bnec a3,a2,0x400210 <b_sort+22>
   0x0040023c <+66>: lw  a3,24(sp)
   0x0040023e <+68>: sw  a3,20(sp)
   0x00400240 <+70>: lw  a3,28(sp)
   0x00400242 <+72>: bnezc a3,0x400208 <b_sort+14>
   0x00400244 <+74>: bc  0x400248 <b_sort+78>
   0x00400246 <+76>: nop
   0x00400248 <+78>: restore.jrc 48,fp,ra
End of assembler dump.
```

## I. Unoptimised disassembly of the b_swap function

```
Dump of assembler code for function b_swap:
   0x0040024a <+0>: save 48,fp,ra
   0x0040024c <+2>: addiu fp,sp,-4048
   0x00400250 <+6>: sw   a0,12(sp)
   0x00400252 <+8>: sw   a1,8(sp)
   0x00400254 <+10>: lw  a3,12(sp)
   0x00400256 <+12>: lw  a3,0(a3)
   0x00400258 <+14>: sw  a3,28(sp)
   0x0040025a <+16>: lw  a3,8(sp)
   0x0040025c <+18>: lw  a2,0(a3)
   0x0040025e <+20>: lw  a3,12(sp)
   0x00400260 <+22>: sw  a2,0(a3)
   0x00400262 <+24>: lw  a3,8(sp)
   0x00400264 <+26>: lw  a2,28(sp)
   0x00400266 <+28>: sw  a2,0(a3)
   0x00400268 <+30>: nop
   0x0040026a <+32>: restore.jrc 48,fp,ra
End of assembler dump.
```

## J. Unoptimised disassembly of the init_arr function

```
Dump of assembler code for function init_arr:
   0x00400174 <+0>: save 16,fp,ra
   0x00400176 <+2>: addiu fp,sp,-4080
   0x0040017a <+6>: aluipc a3,0x20
   0x0040017e <+10>: li  a2,31
   0x00400180 <+12>: sw  a2,2164(a3)
   0x00400184 <+16>: aluipc a3,0x20
   0x00400188 <+20>: ori a3,a3,0x874
   0x0040018c <+24>: li  a2,8
   0x0040018e <+26>: sw  a2,4(a3)
   0x00400190 <+28>: aluipc a3,0x20
   0x00400194 <+32>: ori a3,a3,0x874
   0x00400198 <+36>: li  a2,2
   0x0040019a <+38>: sw  a2,8(a3)
   0x0040019c <+40>: aluipc a3,0x20
   0x004001a0 <+44>: ori a3,a3,0x874
   0x004001a4 <+48>: li  a2,11
   0x004001a6 <+50>: sw  a2,12(a3)
   0x004001a8 <+52>: aluipc a3,0x20
   0x004001ac <+56>: ori a3,a3,0x874
   0x004001b0 <+60>: li  a2,9
   0x004001b2 <+62>: sw  a2,16(a3)
   0x004001b4 <+64>: aluipc a3,0x20
   0x004001b8 <+68>: ori a3,a3,0x874
   0x004001bc <+72>: li  a2,90
   0x004001be <+74>: sw  a2,20(a3)
   0x004001c0 <+76>: aluipc a3,0x20
   0x004001c4 <+80>: ori a3,a3,0x874
   0x004001c8 <+84>: li  a2,100
   0x004001ca <+86>: sw  a2,24(a3)
   0x004001cc <+88>: nop
   0x004001ce <+90>: restore.jrc 16,fp,ra
End of assembler dump.
```

## K. Optimised disassembly of the main function with source code

```
Dump of assembler code for function main:
sort.c:
61    {
   0x004000e2 <+0>: save 32,ra,s0-s3

9         arr[0] = 31;
   0x004000e4 <+2>: aluipc a2,0x20
   0x004000e8 <+6>: li   a1,31

10        arr[1] = 8;
   0x004000ea <+8>: lapc a3,0x420874 <arr>

9         arr[0] = 31;
   0x004000ee <+12>: sw  a1,2164(a2)

10        arr[1] = 8;
   0x004000f2 <+16>: li  a2,8
   0x004000f4 <+18>: move s0,a3
   0x004000f6 <+20>: sw  a2,4(a3)

11        arr[2] = 2;
   0x004000f8 <+22>: li  a2,2
   0x004000fa <+24>: sw  a2,8(a3)

12        arr[3] = 11;
   0x004000fc <+26>: li  a2,11
   0x004000fe <+28>: sw  a2,12(a3)

13        arr[4] = 9;
   0x00400100 <+30>: li  a2,9
   0x00400102 <+32>: sw  a2,16(a3)

14        arr[5] = 90;
   0x00400104 <+34>: li  a2,90
   0x00400106 <+36>: sw  a2,20(a3)

15        arr[6] = 100;
   0x00400108 <+38>: li  a2,100
   0x0040010a <+40>: addiu s3,a3,28

62        int i;
63        struct Node *ll = NULL;
   0x0040010c <+42>: move s1,zero

15        arr[6] = 100;
   0x0040010e <+44>: sw  a2,24(a3)
   0x00400110 <+46>: bc  0x400114 <main+50>
```

```
  0x00400112 <+48>: move s1,a0

67          b_insert(&ll, arr[i]);
  0x00400114 <+50>: lw  s2,0(s0)

22       struct Node *ptr1 = (struct Node*)malloc(sizeof
   (struct Node));
  0x00400116 <+52>: li  a0,8
  0x00400118 <+54>: balc 0x4001e0 <malloc>
  0x0040011a <+56>: addiu s0,s0,4

23       ptr1->data = data;
  0x0040011c <+58>: sw  s2,0(a0)

24       ptr1->next = *start_ref;
  0x0040011e <+60>: sw  s1,4(a0)

64       init_arr();
65       /* Insert elements of the array to linked list
   */
66       for (i = 0; i< SIZE; i++)
  0x00400120 <+62>: bnec s3,s0,0x400112 <main+48>
  0x00400124 <+66>: move s0,zero

44                swapped = 1;
  0x00400126 <+68>: movep a2,a3,a0,s1

39          swapped = 0;
  0x00400128 <+70>: move a5,zero

40          cur = start;
41          while (cur->next != next) {
  0x0040012a <+72>: beqc s0,a3,0x400144 <main+98>

42             if (cur->data > cur->next->data) {
  0x0040012c <+74>: lw  a1,0(a2)
  0x0040012e <+76>: lw  a4,0(a3)
  0x00400130 <+78>: bgec a4,a1,0x40013c <main+90>

50    }
51
52    void b_swap(struct Node *a, struct Node *b)
53    {
54       int temp = a->data;
55       a->data = b->data;
  0x00400134 <+82>: sw  a4,0(a2)

44                swapped = 1;
  0x00400136 <+84>: li  a5,1

56       b->data = temp;
```

```
  0x0040013a <+88>: sw  a1,0(a3)

44                swapped = 1;
  0x0040013c <+90>: move a2,a3
  0x0040013e <+92>: lw  a3,4(a3)

41          while (cur->next != next) {
  0x00400140 <+94>: bnec a3,s0,0x40012c <main+74>
  0x00400144 <+98>: move s0,a2

45             }
46             cur = cur->next;
47          }
48          next = cur;
49       } while (swapped);
  0x00400146 <+100>: bnezc a5,0x400126 <main+68>

68       b_sort(ll);
69       return 0;
70    }
  0x0040014a <+104>: move a0,zero
  0x0040014c <+106>: restore.jrc 32,ra,s0-s3
End of assembler dump.
```

## L. Optimised disassembly of the main function without source code

```
   Dump of assembler code for function main:
sort.c:
  0x004000e2 <+0>: save 32,ra,s0-s3

9     in sort.c
  0x004000e4 <+2>: aluipc a2,0x20
  0x004000e8 <+6>: li   a1,31

10    in sort.c
  0x004000ea <+8>: lapc a3,0x420874 <arr>

9     in sort.c
  0x004000ee <+12>: sw  a1,2164(a2)

10    in sort.c
  0x004000f2 <+16>: li  a2,8
  0x004000f4 <+18>: move s0,a3
  0x004000f6 <+20>: sw  a2,4(a3)

11    in sort.c
  0x004000f8 <+22>: li  a2,2
  0x004000fa <+24>: sw  a2,8(a3)

12    in sort.c
  0x004000fc <+26>: li  a2,11
  0x004000fe <+28>: sw  a2,12(a3)

13    in sort.c
  0x00400100 <+30>: li  a2,9
  0x00400102 <+32>: sw  a2,16(a3)

14    in sort.c
  0x00400104 <+34>: li  a2,90
  0x00400106 <+36>: sw  a2,20(a3)

15    in sort.c
  0x00400108 <+38>: li  a2,100
  0x0040010a <+40>: addiu s3,a3,28

62    in sort.c
63    in sort.c
  0x0040010c <+42>: move s1,zero

15    in sort.c
  0x0040010e <+44>: sw  a2,24(a3)
  0x00400110 <+46>: bc  0x400114 <main+50>
  0x00400112 <+48>: move s1,a0
```

```
67    in sort.c
  0x00400114 <+50>: lw  s2,0(s0)

22    in sort.c
  0x00400116 <+52>: li  a0,8
  0x00400118 <+54>: balc 0x4001e0 <malloc>
  0x0040011a <+56>: addiu s0,s0,4

23    in sort.c
  0x0040011c <+58>: sw  s2,0(a0)

24    in sort.c
  0x0040011e <+60>: sw  s1,4(a0)

64    in sort.c
65    in sort.c
66    in sort.c
  0x00400120 <+62>: bnec s3,s0,0x400112 <main+48>
  0x00400124 <+66>: move s0,zero

44    in sort.c
  0x00400126 <+68>: movep a2,a3,a0,s1

39    in sort.c
  0x00400128 <+70>: move a5,zero


40    in sort.c
41    in sort.c
  0x0040012a <+72>: beqc s0,a3,0x400144 <main+98>

42    in sort.c
  0x0040012c <+74>: lw  a1,0(a2)
  0x0040012e <+76>: lw  a4,0(a3)
  0x00400130 <+78>: bgec a4,a1,0x40013c <main+90>

50    in sort.c
51    in sort.c
52    in sort.c
53    in sort.c
54    in sort.c
55    in sort.c
  0x00400134 <+82>: sw  a4,0(a2)

44    in sort.c
  0x00400136 <+84>: li  a5,1

56    in sort.c
  0x0040013a <+88>: sw  a1,0(a3)
```

```
44    in sort.c
  0x0040013c <+90>: move a2,a3
  0x0040013e <+92>: lw  a3,4(a3)

41    in sort.c
  0x00400140 <+94>: bnec a3,s0,0x40012c <main+74>
  0x00400144 <+98>: move s0,a2

45    in sort.c
46    in sort.c
47    in sort.c
48    in sort.c
49    in sort.c
  0x00400146 <+100>: bnezc a5,0x400126 <main+68>

68    in sort.c
69    in sort.c
70    in sort.c
  0x0040014a <+104>: move a0,zero
  0x0040014c <+106>: restore.jrc 32,ra,s0-s3
```

## M. Optimised disassembly of the main function without debug info

```
Dump of assembler code for function main:
   0x004000e2 <+0>: save 32,ra,s0-s3
   0x004000e4 <+2>: aluipc a2,0x20
   0x004000e8 <+6>: li   a1,31
   0x004000ea <+8>: lapc a3,0x420874 <arr>
   0x004000ee <+12>: sw  a1,2164(a2)
   0x004000f2 <+16>: li  a2,8
   0x004000f4 <+18>: move s0,a3
   0x004000f6 <+20>: sw  a2,4(a3)
   0x004000f8 <+22>: li  a2,2
   0x004000fa <+24>: sw  a2,8(a3)
   0x004000fc <+26>: li  a2,11
   0x004000fe <+28>: sw  a2,12(a3)
   0x00400100 <+30>: li  a2,9
   0x00400102 <+32>: sw  a2,16(a3)
   0x00400104 <+34>: li  a2,90
   0x00400106 <+36>: sw  a2,20(a3)
   0x00400108 <+38>: li  a2,100
   0x0040010a <+40>: addiu s3,a3,28
   0x0040010c <+42>: move s1,zero
   0x0040010e <+44>: sw  a2,24(a3)
   0x00400110 <+46>: bc  0x400114 <main+50>
   0x00400112 <+48>: move s1,a0
   0x00400114 <+50>: lw  s2,0(s0)
   0x00400116 <+52>: li  a0,8
   0x00400118 <+54>: balc 0x4001e0 <malloc>
   0x0040011a <+56>: addiu s0,s0,4
   0x0040011c <+58>: sw  s2,0(a0)
   0x0040011e <+60>: sw  s1,4(a0)
   0x00400120 <+62>: bnec s3,s0,0x400112 <main+48>
   0x00400124 <+66>: move s0,zero
   0x00400126 <+68>: movep a2,a3,a0,s1
   0x00400128 <+70>: move a5,zero
   0x0040012a <+72>: beqc s0,a3,0x400144 <main+98>
   0x0040012c <+74>: lw  a1,0(a2)
   0x0040012e <+76>: lw  a4,0(a3)
   0x00400130 <+78>: bgec a4,a1,0x40013c <main+90>
   0x00400134 <+82>: sw  a4,0(a2)
   0x00400136 <+84>: li  a5,1
   0x0040013a <+88>: sw  a1,0(a3)
   0x0040013c <+90>: move a2,a3
   0x0040013e <+92>: lw  a3,4(a3)
   0x00400140 <+94>: bnec a3,s0,0x40012c <main+74>
   0x00400144 <+98>: move s0,a2
   0x00400146 <+100>: bnezc a5,0x400126 <main+68>
   0x0040014a <+104>: move a0,zero
   0x0040014c <+106>: restore.jrc 32,ra,s0-s3
```

```
End of assembler dump.
```