Investigating Decompilation Techniques Suitable for Recovering Source Code. How Much Useful Information can be Retrieved?

Version 1.01

J. R. Grant

research@jguk.org

A dissertation submitted in partial fulfilment of the requirements for the award of the degree of Master of Science in Computer Games Technology, University of Abertay Dundee, UK

Submitted 10th January 2003

Abstract

Keywords: decompilation, reverse engineering, source code recovery, video games.

Compilers have been in use since the 1950's. However, currently decompilers are still not widely available, even though there has been research.

In this research, a novel decompilation process has been designed to facilitate the recovery of useful information from programs. The question of how much useful information can be recovered is considered using the example of a video game.

Several approaches were considered for use in a decompilation process and the optimum methods were selected for inclusion. Input was translated from Assembler into intermediate-level source code. This intermediate-level source code was then analysed and high-level source code was constructed from the detailed information retrieved.

In this paper, it is demonstrated that the quality of the retrieved high-level

source code is equal in functionality to the original. In addition, the detailed information recovered would be extremely useful in documenting file formats or the architecture on which the video game runs.

Acknowledgements

This MSc was conducted under the guidance of L. Natanson of the University of Abertay Dundee, UK. Gratitude is also extended to C. Cifuentes, D. Evans, I. Guilfanov, A. Mycroft, S. Peyton-Jones, N. Ramsey, D. Ryan and G. Scotti for answering questions relating to their research.

Contents

1 Introduction

The first key aim of this paper is to establish if useful information can be retrieved from a program or library with no source code available. Useful information generated could be in the form of source code, data file formats or information about the target machine that the program runs on. The second key aim is an evaluation of different approaches suitable for use in a decompilation process. The final key aim is considering which approaches produce the optimum results for the requirements of a selected target application; then combining the most suitable methods into a decompilation process design.

Recovering source code requires decompilation of the program. Decompilation is effectively the inverse of the compilation process: namely, inputing a program, extracting the details of the programs execution and outputting source code. Detailed exploration of the program is required to discover the functionality within. Programs normally contain very little useful information left over from the original developers; therefore a certain degree of intelligence in the heuristic analysis is necessary to transform the program into a higher-level of abstraction. Once information has been disclosed from the internals of the program, high-level language source code of functions and data structures can be retrieved. The extent to which this information is valuable and usable is evaluated later in the paper.

1.1 Motivation

Developers are sometimes in the position of only having a program binary for one particular legacy computer system. (Cifuentes, 1997) noted that 30% of existing conventional software cannot be recompiled due to lost source code or the unavailability of the compiling tools and libraries used to create the software in the first place. In the case of video games there is no specific figure relating to this problem. However, due to the nature of the video game industry, video game systems are superseded frequently, which can mean that video game source code is required to be ported to new architectures. In some cases, programmers may have source code in a legacy language, such as BASIC, Pascal or Assembler; decompiling the program into a modern high-level language such as C would give the program a new life not originally considered. Video games were often written in Assembler in the 1980's and even up to the mid 1990's: the 1995 Commodore Amiga 1200 game Alien Breed 3D was written entirely in Assembler. In this case the original source code could be used to verify the results of the decompilation and provide comments and variable names useful to the maintainer. All of these are key uses where decompilation can be used to generate new source code for developers.

Some countries have created laws that state that decompilation of programs is illegal and breaks copyrights and the licensing agreement. Other countries have implemented copyright laws to allow decompilation and reverse engineering of information for interoperability between programs and their proprietary file formats. Some program licencees believe that under “fair use” rights, they are allowed to make full use of the product, including decompiling and reverse engineering proprietary file formats. Decompilation is sometimes essential if inter-operability is required as often the developer of the original program will not co-operate with the development of other products that are required to be compatible. Or, as in the case of Corel WordPerfect, development is ceased for one platform, and bugs can only be fixed by the licencees.

In this research only one source architecture will be tested due to time constraints. The MIPS architecture has been selected as it is common across graphics work stations (SGI), video game consoles (Sony PSOne, Nintendo N64 and Sony PS2).

Following this first chapter, Chapter 2 evaluates the requirements of a decompiler for particular use. Chapter 3 considers the most suitable approach for a decompilation process and details the stages. Chapter 4 is a case study of use of the methods considered. Chapter 5 is an evaluation of the results and analysis of the usefulness of the information retrieved. Chapter 6 is the conclusion, and also considers future developments.

2 Requirements

A decompiler is a very valuable tool for programmers to generate new source code from old programs. Currently there is no general decompiler on the software market which is beyond an α release. There are several key uses for a decompiler; each has different requirements. Three scenarios are evaluated against their specific requirements and summarised.

2.1 Video game decompilation

The first example chosen for consideration is a video games company that would like to port some legacy 1980's products, for re-release on a modern console of similar power to the Game Boy Advance (GBA). Essentially the decompilation is the means to facilitate the port from a legacy system; it is more complex than just recovering source code for a currently available architecture. In this case the input program would either be in a machine code binary executable format as used in ROM cartridges, or in a structured executable file format specific to the video game console architecture.

The requirements of this example project are considered to be the following:

- Porting with minimum time and cost.
- Minimum workforce input required.
- Excellent quality high-level language source code recovered.

Many 1980's video game consoles used 8bit CPU's in their consoles that are not in use frequently now. Each console has a unique architecture that affects the decompilation process. The other key components in a video game console include graphics and sound chips that many instructions in the video game will relate to. Time constraints for a project like this may have an impact on the process, results are required and costs are likely to be limited. The resulting source code and information is required to be complete and run as expected on the target video game console.

2.2 Date field verification

The example considered is a financial institution that requires verification that their computer software can deal with large date fields. This is similar to the Year 2000 date issue encountered several years ago.

A failure would be disastrous for the financial institution so the following key requirements are identified:

- Time and date functions recovered.
- Identification of data structures.
- Integration into their formal testing strategy.

The requirements of a financial institution are very different from a video games company. A financial institution requires complete certainty that solutions are found and that all eventualities are covered. Time is an issue, but only in an upper limit before the internal deadline. Minimising time to cut costs is not necessarily a key requirement, because complete certainty that the software is capable of dealing with date fields is required.

The computer system will most likely be a large computer server with several CPU's and many other components, which could cause difficulty during the process due to complexity. Although a financial institution's computer software would be far more complex than a video game, they only require proof that their systems are not going to fail. There is no planned release of regenerated source code on a new platform. The most important requirement is that data structures and functions relating to time and date use and storage need to be identified and focused on in the decompilation. Other areas of the program do not need to be decompiled. With this smaller amount of machine code to decompile, it is the data structures that are most important. Field types and sizes need to be checked against each function to make sure that the date is not stored in any abbreviated formats like “0000” for “1970” (1970 is when most modern internal computer clocks calculate from). If the systems requiring verification have been developed in a modular structure, it should be possible to identify a smaller amount of shared libraries and programs that require analysis. This would need to be integrated into a formal testing strategy to guarantee date field compliance.

2.3 Security verification

Verifying that a program is not a trojan, a virus or is just generally poorly designed, implemented and tested can be established by using a decompiler. Viewing the program flow, inputs, outputs and data flow, reveals whether the program is secure to use. It is desirable to decompile the whole program as illegitimate code is virtually always executed secretly by an unsuspected trigger action. Comprehending the whole of the generated source is the only way to be completely sure it is compliant and secure to use. Unfortunately, all too often bugs slip through internal testing departments, so decompilation is the only way to be sure of security, unless the company provided the original source (e.g. as in the case of Free Software covered by the GNU Public License).

In this example, the case of a small business user on a laptop computer is considered. The user may purchase software and download free software from the internet and wish to verify that it is safe to use.

2.4 Summary

Each project has different requirements (see Fig 1). In the case of video games and security verification, full program decompilation is essential. A date field verification project only requires specific information that can be generated from the selection of libraries and programs relating to the task. Each use considered runs on a different architecture; this means that the decompilation process would need to support each different architecture required.

	Video game	Date field verification	Security verification
Complete source	Yes	No	Yes
Main program flow	Yes	No	Yes
Key functions	Yes	Yes	Yes
Structures	Yes	Yes	Yes
Data file formats	Yes	Yes	Yes
Input and output	Yes	Partially	Yes

Fig 1: Comparison of decompilation requirements for the three scenarios.

For this research a video game decompilation process has been selected as the target application of this work. How to decompile video game programs is a complex issue which will be interesting to consider in this paper.

3.0 Design of a decompilation process for video games

Decompiling a program or library converts the functionality in the machine code binary version into a generic definition of the program at a much higher level of abstraction, without the particular quirks and idiosyncrasies that one particular CPU architecture might place upon a program.

The approach taken in this phase of the research affects the options available in other areas of the research. Implementing reverse engineering functions, that work directly on only one specific computer architecture, would mean there was limited scope for the comparison of algorithms across different architectures. Separating the process into key stages would allow the process to be applicable to other computer architectures that follow the general von Neuman computer architecture design. Taking a modular approach increases the development options available due to the flexibility. A front-end is required for each architecture supported and a back-end for each supported high-level language is required as well. The modular nature of the decompiler design will allow for further work to add other targets. Replacing certain functionality with an improved approach is also possible then.

A key feature in the von Neuman architecture is the dual use of memory, for both data and operations. This poses identification problems which are usually solved by their position in the machine code binary. Conventionally, different sections are used for different data in the structured executable file formats that are commonly used in video game consoles. The standard organisation of a program file stores each block of information in a different section as follows: `.text` contains machine code, `.data` contains pre-initialised variables and `.bss` contains uninitialised variables.

Instruction set designers may not follow these standard conventions and choose to use other names. In some cases on certain architectures such as the Hitachi SuperH (used in the Sega Dreamcast video game console) the constant data is mixed inline with the machine code. Heuristic techniques can be used for this detection. They need to be part of the machine eccentricity eliminations that are done by the front-end for the specific architecture. In some cases data can be positioned in between machine code instructions; this is common in indexed jump tables (`switch` in the C high-level language).

All architectures place constraints on what is possible in a program or library; in some cases this is memory aligned structures, or a direct mapping of boolean and integers to 32bit variables. Variable details are stored in a symbol table found in most programs and libraries. Sometimes there is also debugging information available which aids the decompilation process. This could include function names, arguments, types, local variable names and the scope a variable has, local or global. If the symbol table has information on type this will mean an incorrect assumption of a cardinal being an integer could not happen.

A compiler has several stages for the process of conversion from HLL to machine code. Modularising the decompilation process allows for one interface program, front-ends for each supported architecture and back-ends for each high-level language supported. The GNU Compiler Collection (GCC) makes use of one front-end and many target back-ends; this allows for support of many target architectures and many different source languages. (Cifuentes 1997) also implemented a several stage approach to decompile CISC programs. After considering the (Cifuentes 1997) approach and the GCC modular framework, the following three phases of the decompilation process are defined.

Phase one: Front-end - architecture dependent
Phase two: Universal decompilation machine (UDM)
Phase three: Back-end – high-level language target

3.1 Phase one: Front-end

This phase is specifically kept to a minimum, as it is architecture dependent. Any functionality beyond each architecture should be in phase two. To recover detailed information is the key target in this phase of the research. Analysis and further use of the relevant information will be used in later phases for further processing.

3.1.1 Input machine code binary

This section of phase one identifies the machine code architecture and format using the bfd (binary file descriptors) library. Extracting the sections for use in later stages and separating data from instructions is essential.

3.1.2 Identify library functions

Typically runtime initialisation functions are added to a program when it is compiled and linked. For example in `crt0.o` the function `__main()` sets up the stack and then passes the command line arguments to the `main()`

function. In cases such as `crt0.o` functions like `__main()` are architecture dependent. These should be identified to avoid including them in the high-level source code generated from this whole process.

Programs often link (statically or dynamically) to libraries such as libc (C library) and libm (maths library). Often information is available in the symbol table of the program or library. If this information is left from the original compilation process, it too can be used to identify functions present in libraries.

Programs that are statically linked require further analysis to aid identification. Generating signatures for the libraries used to compile a machine code binary allows for elimination of common functions that are trivial and included in standard libraries. (Cifuentes 1997) used a method of tracing the function call and mapping out the arguments and returned data:

```
var0 = strncpy(arg0, arg1, arg2);
```

where `var0` is the returned value and `arg0`, `arg1`, `arg2` are arguments. This information can be identified using the (Cifuentes 1997) approach to function call tracing.

Programs that are dynamically linked to the libraries, and thus do not contain the machine code of the library, pose less of an identification problem as the function call or system call name and number is known and can be inserted into the generated intermediate-level source code.

If a function such as `printf()` is identified in phase one of the decompilation process, the information should be passed to the following phase of the decompilation process.

3.1.3 Disassemble

This section of phase one covers disassembling the machine code program (binary, static or dynamically linked library) in the file format and architecture detected. The disassembler `objdump` from GNU was chosen for this stage because it is Free Software that can be modified to implement test approaches and see the results. Using `objdump` enables research to progress quickly as it is a finished product with a strong disassembler.

3.1.4 Intermediate code generation

There are several different approaches possible for this final stage in phase one of the decompilation process. Each instruction (opcode) needs to be abstracted away from low-level functionality and presented in a simplified representation of machine code. It is essential that all architecture dependencies are removed before moving on to the next phase. (Cifuentes 1996) created an intermediate language called Icode. Icode represented all operations possible, and mapped several different Assembler instructions onto one Icode instruction in a n:1 (many to one) relationship.

Creating my own syntax to represent the intermediate code was an option considered. Unfortunately, because type information is normally not in the release version of a program, it is difficult to be identified directly. Variables could all be declared as either integer or cardinal, then the second phase (Universal Decompilation Machine) could determine the appropriate type for each variable. Until this second phase a temporary type of int32 (integer 32bits) is the most suitable type (for MIPS); the following phase can choose a better type as necessary. We are only identifying it as being an integer variable that is 32bits in size, not as being signed or unsigned (in the same way that `int` in K&R C was not identified as signed or unsigned).

A key part of the intermediate code generation process is the removal of all the architecture specific optimisations and architecture oddities. For example instructions relating to the execution pipeline, the purpose of `nop` and `sync` is only to change the execution flow through the pipeline; beyond this low-level they are irrelevant and can be eliminated. During the elimination process instructions should be re-ordered to reflect the normalised control flow of the program.

The intermediate code generation process is not possible in only one pass because of the complexity (machine code is very different from high-level language source code, typically compilers require several passes to generate it). Architecture dependencies may only be visible after detailed forward and backwards analysis of a block of code. The requirements for the intermediate code output are as follows: load and use the symbol table and debugging information if available (used to describe function and variable names, types and arguments); eliminate architecture dependencies; define the arbitrary address that execution starts at; include in the intermediate-level source code a table of labels that are referenced by conditional and unconditional jumps (required by the following phase). Assembler calls that store the return address indicate a function at the called address; these can be passed on to the following phase.

This list of requirements is minimal. Maintaining the modular design of the decompilation process is essential if this defined approach will be suitable for use on multiple architectures (at a later date). Translation of the intermediate code takes place only if it is an architecture dependency such as delay slot execution (a decoded instruction after a jump is executed before the jump target instruction on MIPS), or assigning numbers to registers over two instructions due to architecture constraints relating to word size (commonly 32bit in MIPS). It is essential that the hardware architecture dependencies are removed so that the later phases can process abstract information to generate high-level source code.

All architectures have a specification and conventions that are used by the different compilers supporting the architecture. These conventions provide information that can be used to the decompiler's advantage. For example, on the MIPS architecture, registers 4-7 contain arguments passed to functions; registers 2-3 contain function return values. Using this information to name variables is better than leaving the third phase to deal with the naming; the modularity of the decompiler would be compromised if the high-level language generation layer is required to use low-level information. Typically, compilers and assemblers follow the specification for the architecture (to maintain compatibility with the OS software and libraries). If the program is an exception to this rule the conventions could be ignored or a different user-defined specification used instead. If register usage conventions are not adhered to all registers are effectively global variables. Alternatively, analysis of which blocks of code access specific variables could help identify which registers served a certain purpose.

The syntax used for the intermediate language would be more appropriate if the syntax was already familiar to the users. Using a pseudo C style of syntax with the keywords limited to type declarations, `if`, `goto` and function calls are the essential constructs required to represent low-level Assembler in pseudo C. Stack based systems often make use of the stack to pass arguments to functions (if bigger than register size), store variables and structures that are bigger than the register size. Following the conventions for MIPS means the stack usage could be illustrated with the addition of several new keywords.

Aho et al (1986) considered a simple representation of an abstract stack based system as used in the compilation process. In a similar vein additional stack manipulation keywords are considered (suitable for all modern computers that make use of a stack). Similar to the Aho el al (1986) approach for a compilation intermediate language, the following intermediate level language is defined. I have named this language intermediate-level C.   This language has been considered with decompilation constraints relating to varied stack usage in mind.

`spush x`	Push x onto the stack
`srpush l`	Push contents of data location `l`
`slpush l`	Push the address of data location `l`
`scopy`	Copy the top item of the stack
`spop x`	Pop into x off the top of the stack
`srpop l`	Pop contents into `l`
`slpop l`	Pop value (address) into `l`

Allowing for oddities left from removing optimisations, `spush` and `spop` could be used to add and remove items from the stack without returning a value. There is an issue of how to represent the number of bits that these keywords would move onto and off the stack. One solution would be to add the number of bytes onto the end of each instruction e.g. `scopy32.` Other keywords would be necessary to use intermediate-level C as the chosen intermediate-level source, a definition of the language as a whole and its syntax. This would be essential to ensure consistency and compatibility between the three different phases and optional extension modules implemented by different programmers.

This approach above is simpler than the (Peyton-Jones 1999) C-- approach which defines a whole intermediate-level language in great detail. C-- specifically is an abstraction of C and Assembler language combined. (Peyton-Jones 1999) stated “C-- should make it *possible* to implement high-level run-time services, but it should not actually *implement* any of them. Rather it should provide just enough 'hooks' to allow the front-end run-time system to implement them". C-- is ideally suitable for use in a decompilation process to facilitate the transfer of information from phase one to phase two.

Although C-- was designed with compilation in mind, C-- presents a very suitable intermediate-level language for use in a decompiler. The C-- specification includes keywords relating to low-level opcodes; these can simply not be used in the decompilation process considered in this research. The abstract C like format can then be analysed and translated into a directly usable format by phase two.   The intermediate language is required to be able to cope with many different high-level language features. Each high-level language implements run-time support, garbage collection and exception handling in different ways. Due to this, an intermediate language is required to be able to deal with all possible scenarios; this could be a problem at some point in the future.

Using C-- as an intermediate representation in a decompiler creates a problem associated with the front-end not being fully aware about the back-end and thus not being able to anticipate precisely how the machine code output will function. This is often a problem for memory management, debugging and exception handling. If extensions are added to facilitate program exceptions, Assembler, debugging information or complex details like breakpoints, the whole reason for using C-- is lost (i.e. to abstract the architecture differences).

C-- has a rich syntax very similar to the C family of languages. C-- allows for many different eventualities.

> A C-- program file is written in eight bit ASCII characters. It consists of a sequence of `data` layout directives, and/or procedure definitions, and/or `import` declarations, and/or `export` declarations, and/or `global` declarations, interleaved in any order.

Peyton-Jones et al (1999)

C-- supports the declaration of the `.data` section in a program. This section is a memory block accessed as a sequence of data of a specific type. Each data item can be initialised or uninitialised. The signed or unsigned type identification issue, that was not resolvable by the intermediate-level C even with keyword extensions, is avoided by C--. In C-- types are not detected, conventional registers are declared as `bits`*n* and floating point values are declared as `float`*n* (*n* represents the number of bits). Pointers, integers, cardinals and characters can all be represented by `bits`*n* (data type detected in phase two). Memory is considered an array of bytes, of which different types can be read from and written to. Global variables are defined in a block; typically global variables are stored in registers. In the case of a decompiler some global variables may in fact be stored in memory and loaded into a register when necessary (this poses detection problems in phase two). Stack and frame pointers can be isolated and treated as pointers for load/save instructions. Typically the frame pointer is used when the base of the stack cannot be identified at the time of compilation.

Function calls can be directly recovered on from MIPS Assembler source that uses the jump-and-link opcode which stores the return address. While the function call is now visible, all the other information necessary to construct a function, its arguments and return values is not. Therefore the function should be identified as being a function, but the arguments and return values should not be listed. To represent the function its number could be used e.g. `function0()`. Then the subsequent phase can determine the arguments and return values.

The formating of the intermediate-level source code is required to be readable, because for some purposes intermediate-level source might be all that is required. Intermediate-level source code could be directly assembled again.

3.1.5 Summary

The intermediate-level language "low-level C" considered by myself could be used in this process. However, after evaluating the functionality, C-- is a better component to use because it can represent the simple nature of Assembler in a more architecture independent way. The C-- features that relate to its low-level requirements are simply not implemented for the purposes of this research.

The C-- format is better than the low-level C language considered first because it is more flexible, with keywords for defining sections of the code, variables and data structures. C-- has also been standardised to a certain extent in the industry. It is for these reasons that it has been chosen as the best suited intermediate-level language format to use in the decompilation process considered.

Anything other than architecture abstraction is beyond the scope of phase one. The following phase targets other areas of the process.

3.2 Phase two: Universal decompilation machine

This phase receives the input intermediate-level source code from the previous front-end phase, processes the intermediate-level source code and generates information for the third phase to use. The intermediate-level source code is in an architecture independent format that is suitable for processing and further analysis. There are a range of approaches available for consideration for this phase of the decompilation process. They are evaluated in detail and then summarised.

3.2.1 Optimising input intermediate-level source code

This stage of the phase could have been implemented in phase one (architecture front-end). However, phase two is the best position for this functionality, because it can be applied to the intermediate-level source here. If this were implemented in phase one each different architecture would require the development of a specific version of this same functionality. (Cifuentes 1997) presented an extended register copy propagation method, implemented in their research decompiler. This method eliminated instructions and registers that were not essential (most likely added by the compiler), and inserted the stack and heap accesses into the resulting intermediate code output. This method is very useful in the recovery of high-level language constructs that break down to several Assembler instructions.

Temporary variables that are only used as part of more complex instructions can often be combined and removed also.

> The optimisation pass that removes redundant intermediate branches, reduced the [control flow graph] by up to 50% in different programs tested by dcc.

(Cifuentes 1997)

3.2.2 Analysis of program control flow

(Cifuentes 1997) used a Control Flow Graph (CFG) technique to identify the different blocks of code and thus the flow of execution. A control flow graph is a structured representation of a program. Blocks of intermediate-level source code are identified using the criteria below, with six types of node representing the last instruction in the block:

1-way:	block that ends with an unconditional branch
2-way:	block that ends with a conditional branch
n-way:	block that ends with an unconditional branch on a register
call:	block that ends with a function call
return:	block that ends on a return
fall:	(fall through) block that is followed by a labelled block (i.e. there is transfer of control to the next instruction of this block)

With the exception of n-way nodes, generating the CFG is reasonably uncomplicated. (Cifuentes 1997) found that blocks that end in n-way nodes could be evaluated with extra processing. If a branch-on-register is detected, backwards slicing (traversing the instruction execution flow in reverse) can be performed to evaluate the instructions that depend on the register. If it is not resolvable, a forward walk of the sliced instructions often make it possible to calculate and resolve the size of the branch table and the offsets to search for. Occasionally it will not be possible to resolve; in this case it will be marked as having no exit point. (Cifuentes 1997) found that the RISC implementation of the n-way node resolver is far simpler to implement because the architecture is simpler than the conventional CISC equivalent. In a C-- based n-way resolver the original architecture is not relevant.

(Cifuentes 1997) use specific notation. The control transfer node is called the *header*, the first node in the following code block is called *follow*, and a latching node is called *loop*, this being the last node before the header.

`if(cond) then else`	2-way node neither out point is the *follow* node after the *header*.
`if(cond) then`	“false” out point is the *follow* node.
`if(cond) else`	“true” out point is the *follow* node.
`switch(cond)` *or* `case`	n-way node with unconditional out point.

3.2.3 Generating source without using intermediate goto's

A Java decompilation method was tested by (Proebsting 1997). Java bytecode represents a simple stack based system with 204 opcodes. It is this simplicity which enables Java to run on many platforms in a Java Virtual Machine (JVM). The JVM translates the Java bytecode into native machine code at run time. (Proebsting 1997) did not use any control flow graph transformations because they believe that this requires all high-level language constructs to be identified in advance, therefore unexpected low-level bytecode might not be reconstructible. Their approach generates legal Java source (no goto's), then processes this code to recover high-level language constructs by eliminating the unnecessary code (added while removing goto's). This unexpected low-level operation code problem could also occur in machine code. It is for this reason that this has been considered. In cases where the control flow graph in the first approach considered is not resolvable, this approach could be used for specific blocks of code.

Constraints and verifiability are built into each Java program (similar to symbols in conventional programs); this means the source generation process is much simpler than on a conventional machine code architecture. Variable types and function argument formats are stored in each Java bytecode class file. In the decompilation process considered in this paper, symbols provide a similar amount of information if they have not been removed from the program or library.

(Proebsting 1997) use an algorithm by (Ramshaw 1988) which translates each goto into a surrounding loop and a multi-level `break` to one depth out of the construct. This construct could be either a nested `switch`, `for`, `while`/`do` if the target language was C/C++ or Java. The algorithm determines the best position for these loops and inserts a `break` relative to where the original goto would have been in the intermediate-level source program.

(Proebsting 1997) extended the basic implementation of this technique to include the use of `continue` as well as `break`. This extended algorithm replaces each forward `goto` with a `break` and each backward `goto` with a `continue`. A perpetual loop, `while(1)` or `for(;;)`, is created that starts just before the target of each `continue`, and ends just before the target of each `break` statement. To make sure that each each loop will not iterate for eternity a `break` is placed before the end of each loop. In between each instruction and the next, there is an augmenting edge. (Ramshaw 1988) proved that if the augmented graph is reducible, it is possible to translate the program into an equivalent form without using goto.

The (Ramshaw 1988) method has a serious problem with inverse loops. If the control flow creates a jump directly into the loop there is no way to remove the goto. To avoid this problem (Proebsting 1997) reordered the reducible graph's instructions so that the augmented graph can be reduced.

The intermediate-level source is rewritten by following predefined rules to eliminate superfluous constructs and generate new source code. This evaluation is performed in a loop until no more translations can be performed. Loops that are not repeated are replaced with an `if`, providing all simple `break` statements can be removed and its last program location is never reached. The `continue` instruction can be removed if the program point after will be the same as before the `continue` statement. Similarly `break` may be removed if the following (unreachable) instruction is equivalent to the last program point in the loop.

(Proebsting 1997) missed an important translation. If any point after a `break` in a loop is not reachable (the `goto`'s have all been removed now) then those instructions can be removed up to the end of the loop. The `break` is also not necessary if it is the last statement in a loop as it has now been translated into an `if` statement.

This is an alternative to the (Cifuentes 1997) approach. It could be used if a block of intermediate-level source was unresolvable.

3.2.4 Removing goto statements

In some cases (see example in chapter 4 case study `goto-deps`) a block of code is not resolvable with either of the two advanced approaches considered so far. Leaving goto's in the final output of a decompiled program essentially makes a modular program an unstructured one. If possible all goto's should be removed to aid comprehension of the program.

(Erosa 1993) presented an approach that was intended for translating legacy source code that was not modular; however it presents a useful component for use in the whole decompilation process considered in this paper. The algorithm works by systematically applying goto-movement transformations followed by goto-elimination transformations.

The (Erosa 1993) implementation modifies the intermediate-level source to allow for it to proceed with goto elimination. A `goto l` is always represented as a conditional goto, if there is no condition it is represented as `if(true) goto l`. The approach removes all goto's but neglects to remove `break` and `continue` statements because (Erosa 1993) believe that they are only found as part of compound instructions; and thus will not pose such lack of structure problems (as goto's present). Use of `break` and `continue` statements can be useful for exiting switch and loop statements. For the most part I would not consider it structured programming, due to the flow of execution not being directly clear; there are cases when the use of `break` and `continue` is suitable.

The control flow of each block can be rearranged to remove the goto. True conditions are added if none are originally present.

```
 if(cond) goto l;                 if(!cond)
 stmt_1;                          {
 ...                               stmt_1;
l:                                 ...
 stmt_n;                    ->    }
                                 l:
                                  stmt_n;
```

Fig 2: Eliminate forward goto

A `goto` after the the label reference can be replaced by a `do while(cond)` loop. The first example was not suitable as a `do while(cond)` loop, this is because there was no backwards jump which is how `do while(cond)` loops are represented in Assembler on all architectures.

```
 stmt_1;                          stmt_1;
l:                                do
 stmt_2;                          {
 ...                               stmt_2;
 stmt_n;                     ->    ...
 if(cond) goto l;                  stmt_n;
                                  }
                                  while(cond);
```

Fig 3: Eliminate backwards branch.

(Erosa 1993) used a combination of these two translations (moving a goto to a suitable position and then eliminating it). The two elimination methods above are possible because the goto and the label are both in the same block of decompiled machine code; it would not be possible to directly remove goto's when labels are in different blocks of the program. The depth *n* of a goto is defined as the number of control block constructs (`{ }` blocks in C) that the goto is present inside; where depth 0 represents the root node base level of a function. To facilitate the removal of a directly related (one branch) goto/label from depth *n* requires either outward-movement transformations or inward-movement transformations so that the goto is at the same depth as the label statement. In the case that all goto's cannot be eliminated using translations, extra variables are inserted to store the condition of the goto which is later checked by an `if`.

There is an issue because interlinked goto statements cannot be resolved using this approach. An extension to the (Erosa 1993) technique by myself, to modify the order of the instructions and not add extra variables, corrects this small but important problem with their approach.

This approach is only a solution for one small problem that is part of the decompilation process; it is a useful component if there is a complex block of code that is not directly resolvable by the other two previous approaches.

(Erosa 1993) proved that it is always possible to eliminate a goto by creating extra variables. This also creates more overhead and thus more complicated source code.

3.2.5 Optimising retrieved information

The information now available at this point in phase three should be almost complete. However it is possible to optimise the information further; it is worth checking for the possibility of optimisation in all cases. Using C to illustrate this example, `while(stmt)` is the same as `for(;stmt;)` and as such can be translated. The initialisation statement for the loop can be searched for backwards from the beginning of the loop, looking for a variable that is referenced from within the loop. The update statement will most likely be the last instruction in the block and relate to the loop execution statement. Both of these can be added to the `for`.

```
init_stmt;                      for(init_stmt; stmt;)
while(stmt)                     {
{                                stmt_list;
 stmt_list;              ->      update_stmt;
 update_stmt;                   }
}
```

Fig 4: Include loop initialisation.

```
for(init_stmt;                  for(init_stmt; stmt;
stmt;)                   ->     update_stmt)
{                               {
 stmt_list;                      stmt_list;
 update_stmt;                   }
}
```

Fig 5: Include loop update.

Target branches that jump to a further unconditional jump can be replaced with a single branch. These branches are often the solution the compiler used to bypass a limitation in the distance a branch instruction can reach from the source of the branch. There are other reasons a branch to an unconditional jump could occur, poor compiler design and architecture support being common.

3.2.6 Structuring information for use in the final phase

All the information that is retrieved is required to be in a detailed format defining the characteristics of each statement and the part that it plays in the program as a whole. A loop which is a common construct in many high-level languages could be represented as a structure with the following fields.

```
struct
{
 char * comment;
 udm_stmt * init_stmt;
 udm_stmt * loop_cond;
 udm_stmt * update_stmt;
 udm_stmt * stmt_block;
 bool do_while_order;
} udm_loop;
```

Fig 6: Universal Decompilation Machine loop structure

This could accommodate `while`, `do while` and `for` loops. Using pointers allows the structures to be dynamically allocated. Each pointer can be modified or deleted when restructuring the intermediate-level source code from phase two. Similar to a linked list, NULL pointers are not traversed and the final pointer in any statement block is also NULL.

Structures are required to be un-optimised and architecture independent so that high-level language independent output can occur in phase three.

3.2.7 Summary

Several key approaches were considered in this phase. The (Cifuentes 1997) control flow analysis technique of dividing the intermediate-level source code into blocks determined by the last instruction, is the most suitable for use in the process as a whole. This is because it identifies information and control flow in far greater detail than both of the other approaches. Classifying the final instruction in each block of code as one of six node types, allows for high-level language identification of `if then`, `if then else`, function calls and n-way jumps (typically `switch` and `case` constructs).

(Proebsting 1997) used a novel approach to replace all goto's in the intermediate-level source from phase one with loops. This intermediate-level source code was then analysed to eliminate dead instructions, producing usable high-level information.

The (Erosa 1993) approach to goto elimination and translation works in all cases, but is only designed for removing goto's from already created high-level source code. Although this limitation is a problem, this approach can still be specifically used in unbeknown circumstances when a block of code is unresolvable. The extended approach suggested by myself to change the execution order of the instructions in preference to inserting extra variables solves the only failing in this approach. This extended approach shall be used as a third decompilation option in the considered process.

Phase two should proceed using the (Cifuentes 1997) control flow graph analysis technique. If a particular block of code proves unresolvable by this approach, it should be processed by the simpler (Proebsting 1997) approach. If after this there are still unresolved constructs or blocks of code, the (Erosa 1993) approach can be used to eliminate all goto's. Even with `do while` and `if` the extended (Erosa 1993) approach is still usable as a last option. It is very useful to have the option of a simpler approach, that works in a different way to solve the problem.

3.3 Phase three: Back-end high-level language generation

Although this high-level back-end is separate, the phase is closely connected to phase two because of the detailed information required to proceed with the generation of high-level language source code. In the same way that phase one was very architecture dependent, in this phase each back-end implementation is unique to each target high-level language. Although the back-end implementations may be similar or based on shared code, they are required to generate different outputs and as such cannot be implemented as one back-end for all high-level languages that are supported.

3.3.1 Optimise for target language

The detailed information from phase two may require restructuring for the selected target high-level language. Processing the information over several passes for each optimisation ensures that no optimisations will be missed. If a particular construct is not supported by the selected high-level language back-end, translation is required to remove it. For example common BASIC implementations did not support an n-way branch feature like `switch` in C; this would have to be translated into other construct(s). Translating the n-way branch into a sequence of `if`, `then` and `else if` statements would be one solution.

There might be other cases, for example where a `for` loop was not supported, a `for` is essentially a `while` loop (as illustrated in figure 3 and 4 in the previous section). The loop has an initialisation statement, a termination statement and an increment statement defined together. Thus by moving the initialisation out of the beginning of the loop and moving the increment to the last instruction before the un-conditional branch to the beginning of the loop, the `for` loop can be eliminated.

In the case of the target language being Object Oriented (OO), the functional constructs generated will be quite different from OO design. The information in phase one was heavily functional. Phase two is still essentially functional. To translate the recovered high-level constructs into good OO source code will be difficult. The output may be a single class as there is not very much information to help determine an OO design for program source code generated from a functional language (Assembler).

The Java language does not support pointers; this means that efficient approaches using pointer arithmetic during the execution of code have to be translated into an equivalent construct that Java does support. Function calls would also need to be modified so that variables and structures were passed and returned, rather than pointers being used. Java has other high-level features like automatic garbage collection which the generated high-level source code could take advantage of.

There may also be variables that are redundant and can be eliminated, for example temporary variables that were needed during the translation of the intermediate-level source code (phase two).

In some cases the code can be optimised in this phase. For example, in C, logical operators in condition expressions can be combined, `||`, && and `!` (or, and, not).

```
if(cond1)
 {
  if(cond2)
   {
    stmtlist;
   }
 }
```

Fig 7: Example instructions suitable for target language optimisation

When optimised it becomes:

```
if(cond1 && cond2) stmtlist;
```

This translation is possible because there are no `else` statements. If either statement had an `else` clause this combination would not be directly possible. Translation might allow optimisation, otherwise no optimisation is possible.

3.3.2 Generate target high-level source

Generating the new high-level source involves traversing the structured information retrieved; it has been translated/verified for the chosen target language by the previous section of the phase. Any functions that are identified as being available in the standard compiler library do not need to be output; however their `#include`'s, compilation and linker scripts do need to be. The actual outputting is a case of traversing the data structures and writing to the new source files following the target high-level language syntax.

3.4 Summary of the decompilation process

So far, various different approaches for each specific area of decompilation have been considered. The selected approaches have been chosen because they will optimally fulfil most of the key aims to retrieve valuable and useful information from a program or library. It is essential that a decompiler for video games retrieves source that is usable by the developer in their project.

Phase one generates the abstract intermediate code from the the disassembled sources. Phase two processes the intermediate code and generates higher level information from this. Phase three generates high level language source from the high level constructs retrieved in phase two.

Phase one: Front-end
Input machine code binary
Disassemble
Abstract intermediate code generation
Intermediate code output
Phase two: Universal decompilation machine (UDM)
CFG generation and analysis
Data Format analysis
Phase three: Back-end high-level language target
Optimise for target high-level language
Generate high-level source

Fig 8: Decompilation process

Decompilation is very suited to a modular design. Each component selected fits into one of the three phases. Each phase is independent; this enables the user to use the information before the end of the decompilation process if required. In this case the intermediate-level source code from phase one could be viewed while testing the decompiler. Also, the intermediate-level source code in C-- format would be directly usable in a C-- enabled compiler. Using C-- as an intermediate-level source language to port a program to another architecture is essentially using it as a low-level intermediate Assembler language. This might be desirable if it was not necessary to decompile a program completely.

The implementation of each phase may be connected by shared structures and functions. It would be possible to create phase one just as a “loader” for the UDM and phase three an “exporter”. If implemented this way there could be problems identifying bugs in the implementation of each phase, because of the lack of an intermediate representation of the information. The same is true for the translation into the target high-level language source code in phase three. This problem could be avoided by adding an option to print out the target information in the intermediate-level language C-- which would allow debugging and testing at the same time as allowing the three phases to be closely connected. Taking this approach would still mean that the generator and parser of the intermediate-level source code would still need to be present so that the whole decompilation process could have intermediate-level language source inserted directly into phase two. After the UDM has generated information about a program this data could be saved and re-used to generate source in phase three by loading the data file again.

The implementation of the phases does not need to maintain the rigid structure of the phases defined in the compilation process. It is wise to share source code from each of the phases where possible, providing that at the conceptual level the structure is not broken.

4 Case study

Video games often make use of many different techniques in their source code. Two example programs were selected to evaluate the process of information retrieval and generation of useful source in C from a MIPS program.

4.1 Recursion and iteration example

The program `int-subdivision` demonstrates a recursive function to solve the "total subdivisions possible of an integer" problem. Recursive approaches are used in several common video game algorithms (for example collision detection and AI). Iteration is used in many areas of video games in algorithms based on loops. For the case of four there are a total of five solutions.

```
1 1 1 1
2 1 1
2 2
3 1
4
```

Fig 9: Subdivisions possible for four.

All possible combinations are generated by the recursive function `sums()`. If the solution is unique, it is displayed by the iterative `print_sol()` function.

This is the implementation in C.

```
#include <stddef.h>
#include <stdio.h>

/* display buffer for solution */
int sol_buffer[32];

int sol_pos;

int sol_total;

int main(void)
{
  int answer;
  int max_num;

  max_num = 5;

  printf("integer partitions\n");
  for (answer = 1; answer <= max_num; answer++)
   {
     sol_pos = 0;
     sol_total = 0;
     sums(answer);
     printf("%d solutions for %d\n\n", sol_total, answer);
   }
  return 1;
```

Fig 10: `main()` function calls `sums()` to calculate all solutions up to five.

```
void sums(int total)
{
  int sums_loop;
  int check_loop;
  int valid;
  int previous;


  for (sums_loop = 1; sums_loop <= total; sums_loop++)
   {
     sol_buffer[sol_pos++] = sums_loop;
     sums(total - sums_loop);
     sol_pos--;
   }

/* display if new solution */
  if (total == 0)
   {
     valid = 1;
     previous = sol_buffer[0];

     for (check_loop = 0; check_loop < sol_pos; check_loop++)
      {
          if (sol_buffer[check_loop] > previous)
            valid = 0;
          previous = sol_buffer[check_loop];
      }

     if (valid)
       print_sol();
   }
}


void print_sol(void)
{
  int x;

  for (x = 0; x < sol_pos; x++)
    printf("%d ", sol_buffer[x]);
  printf("\n");
  sol_total++;
}
```

Fig 11: Original `int-subdivision` program source in C

To process the machine code binary in accordance with the decompilation phases, the `int-subdivisions` program was identified as a MIPS architecture ELF (Executable and Linkable Format) format binary. GNU objdump was used for disassembly. No library dependencies were found in the program (thus the program was statically linked) and all symbols had been removed. The phase one abstractions were applied, producing the following intermediate code.

```
#include “printf”
0x400040:
 stack_pointer -= 48;
 bits64[stack_pointer + 32] = return_address;
 bits64[stack_pointer + 16] = frame_pointer;
 frame_pointer = stack_pointer;
 var0 = 5;
 bits32[stack_pointer + 4] = var0;
 arg0 = 0x400000;
 arg0 += 0;
 printf();
 var0 = 1;
 bits32[frame_pointer = var0;

0x400078:
 bits32[frame_pointer] =  var0;
 bits32[frame_pointer + 4] =  var1;
 var0 = (var1 < var0);
 if (var0 == 0) goto 0x400094;
 goto 0x4000e0;

0x400094:
 asm_temp = 0x400000;;
 bits32[asm_temp + 1540] = 0;
 asm_temp = 0x400000;;
 bits32[asm_temp + 1536] = 0;
 arg0 = bits32[frame_pointer];

0x4000a8:
 0x400108();

 arg0 = 0x400000;
 arg0 += 24;
 arg1 = 0x400000;
 arg1 = bits32[arg1 + 1536];
 arg2 = bits32[frame_pointer];
 printf();
 bits32[frame_pointer] =  var0;
 var1 = var0 + 1;
 bits32[frame_pointer] = var1;
 goto 0x400078;
 var0 = 1;
 goto 0x4000ec;

0x4000ec:
 stack_pointer = frame_pointer;
 return_address = bits64[stack_pointer + 32];
 frame_pointer = bits64[stack_pointer + 16];
 stack_pointer += 48;
 goto return_address;

0x400108():
 stack_pointer -= 64;
 bits64[stack_pointer + 48] = return_address;
 bits64[stack_pointer + 32] = frame_pointer;
 frame_pointer = stack_pointer;
 bits32[frame_pointer] = arg0;
 var0 = 1;
 bits32[stack_pointer + 4] = var0;

0x400128:
 var0 = bits32[stack_pointer + 4];
 var1 = bits32[stack_pointer];
 var0 = (var1 < var0);
 if (var0 == 0) goto 0x400144;

 goto 0x4001b4;

0x400144:
 var0 = 0x400000;
 var0 += 1540;
 var1 = bits32[var0];
 arg0 = var1;
 arg1 = (arg0 < 2);
 arg0 = 0x400000;
 arg0 += 1792;
 arg1 += arg0;
 arg0 = bits32[frame_pointer + 4];
 bits32[arg1] = arg0;
 var1 ++;
 bits32[var0] = var1;
 bits32[frame_pointer] =  var0;
 bits32[frame_pointer + 4] =  var1;
 var0 -= var1;
 arg0 = var0;

 0x400108();
 var0 = 0x400000;
 var0 = bits32[var0 + 1540];
 var1 = var0 - 1;
 asm_temp = 0x400000;;
 bits32[asm_temp + 1540] = var1;
 var0 = bits32[stack_pointer + 4];
 var1 = var0 + 1;
 bits32[frame_pointer + 4] = var1;
 goto 0x400128;

0x4001b4:
 var0 = bits32[frame_pointer];
 if (var0 != 0) goto 0x400270;
 var0 = 1;
 bits32[frame_pointer + 12] = var0;
 var0 = 0x400000;
 var0 = bits32[var0 + 1792];
 bits32[frame_pointer + 16] = var0;
 bits32[frame_pointer + 8] = 0;

0x4001d8:
 var0 = bits32[frame_pointer + 8];
 var1 = 0x400000
 var1 = bits32[var1 + 1540];
 var0 = (var0 < var1);
 if (var0 != 0) goto 0x4001f8;
 goto 0x40025c;
 var0 = bits32[frame_pointer + 8];
 var1 = var0;
 var0 = (var1 < 2);
 var1 = 0x400000
 var1 += 1792;
 var0 += var1;
 var1 = bits32[var0];
 var0 = bits32[frame_pointer + 16];
 var1 = (var0 < var1);
 if (var1 == 0) goto 0x400228;
 bits32[frame_pointer + 12] = 0;

0x400228:
 var0 = bits32[frame_pointer + 8];
 var1 = var0;
 var0 = (var1 < 2);
 var1 = 0x400000
 var1 += 1792;
 var0 + = var1;
 var1 = bits32[var0];
 bits32[frame_pointer + 16] = var1;
 var0 = bits32[frame_pointer + 8];
 var1 = var0 + 1;
 bits32[frame_pointer + 8] = var1;
 goto 0x4001d8;
 var0 = bits32[frame_pointer + 12];
 if (var0 == 0) goto 0x400270;
 0x400288();

0x400270:
 stack_pointer = frame_pointer;
 return_address = bits64[stack_pointer + 48];
 frame_pointer = bits64[stack_pointer + 32];
 stack_pointer += 64;
 goto return_address;

0x400288():
 stack_pointer -= 48
 bits64[stack_pointer + 32] = return_address;
 bits64[stack_pointer + 16] = frame_pointer;
 frame_pointer = stack_pointer;
 bits32[frame_pointer] = 0;

0x4002a0:
 bits32[frame_pointer] =  var0;
 var1 = 0x400000
 var1 = bits32[var1 + 1540];
 var0 = (var0 < var1);
 if (var0 != 0) 0x4002c0;
 goto 0x400300;
```

```
0x4002c0:
 bits32[frame_pointer] =  var0;
 var1 = var0;
 var0 = (var1 < 2);
 var1 = 0x400000
 var1 += 1792;
 var0 + =var1;
 arg0 = 0x400000;
 arg0 += 48;
 arg1 = bits32[var0];
 printf();
 bits32[frame_pointer] =  var0;
 var1 = var0 + 1;
 bits32[frame_pointer] = var1;
 goto 0x4002a0;

0x400300:
 arg0 = 0x400000;
 arg0 += 56;
 printf();
 var0 = 0x400000;
 var0 = bits32[var0 + 1536];
 var1 = var0 + 1;
 asm_temp = 0x400000;;
 var1 = bits32[asm_temp + 1536];
 stack_pointer = frame_pointer;
 return_address = bits64[stack_pointer + 32];
 frame_pointer = bits64[stack_pointer + 16];
 stack_pointer += 48;
 goto return_address;
```

Fig 12: `int-subdivision` intermediate code from phase one

Architecture analysis of the program moves instructions in the delay slot to their normalised equivalent position before the branch or function call. Next irrelevant instructions are removed and library functions are identified. The function `printf` can be matched against a library signature and replaced with an include and function call.

Four other functions were identified because they use MIPS calling conventions. `0x400040()` is called from the header which means it is most likely `main()`. `0x400108()` is a function called by `main()`, this would be called `sums()` if symbols were present in the program or `function1()` if no symbols were present. `0x400288()` is called by `function1()`, it was called `print_sol()` in the original program; again we could use this name if symbols were found, otherwise it is named `function2()`.

The second phase is the UDM, blocks of code are classified and the control flow analysed. Constant strings of text can be directly inserted from the arg0 value before the call.

The final phase is to output the information in a meaningful format. In this case the imperative language C has been selected to allow comparisons with the original program.

```
#include <stddef.h>
#include <stdio.h>

/* global variables */
int frame_var0;
int frame_var1;
int frame_buffer0[32];

void sums(int total);
int main(void);
void function1(int stack_var0);
void function2(void);

int main(void)
{
  int stack_var0;
  int stack_var1;           /* loop variable */

  stack_var0 = 5;

  printf("integer partitions\n");

  for (stack_var1 = 1; stack_var1 <= stack_var0; stack_var1++)
   {
     frame_var0 = 0;
     frame_var1 = 0;
     function1(stack_var1);
     printf("%d solutions for %d\n\n", frame_var1, stack_var1);
   }
  return 1;
}


void function1(int stack_var0)
{
  int stack_var1;           /* loop */
  int stack_var2;
  int stack_var3;
  int stack_var4;           /* printf loop variable */


  for (stack_var1 = 1; stack_var1 <= stack_var0; stack_var1++)
   {
     frame_buffer0[frame_var0++] = stack_var1;
     function1(stack_var0 - stack_var1);
     frame_var0--;
   }

  if (stack_var0 == 0)
   {
     stack_var2 = 1;
     stack_var3 = frame_buffer0[0];

     for (stack_var4 = 0; stack_var4 < frame_var0; stack_var4++)
      {
          if (frame_buffer0[stack_var4] > stack_var3)
            stack_var2 = 0;
          stack_var3 = frame_buffer0[stack_var4];
        }

      if (stack_var2)
        function2();
    }
}

void function2(void)
{
  int stack_var0;            /* loop variable */

  for (stack_var0 = 0; stack_var0 < frame_var0; stack_var0++)
   {
     printf("%d ", frame_buffer0[stack_var0]);
   }
  printf("\n");
  frame_var1++;
}
```

Fig 13: int-subdivision source in C in accordance with phase three.

The `#include` for `printf` can be identified as `stdio.h`. With this addition, the program compiles and runs perfectly. This is a very good result, as this program demonstrates recursion and iteration. All constructs were recovered by following the (Cifuentes 1997) control flow graph technique. Then the `while` loops had initialisation and update statements identified and added to form a `for` loop. The source code recovered is slightly different from the original source code; some high-level constructs are in a different order. It is interesting that most of the functions look very similar to the original program, even though there were several architecture dependencies and optimisations to be eliminated in phases one and two.

The comments in the source code were added to assist readability. These comments could be machine generated, as the decompiler can observe that certain variables are loop counters and other common constructs.

4.2 Recursion and iteration results

Compiling the original and the recovered program allows a comparison of the two pairs of results.

Program version	Optimisations	Symbols	File size (bytes)
Original	None	Yes	11889
Original	Yes	None	3164
Recovered	None	Yes	11903
Recovered	Yes	None	3100

Fig 14: Comparison of results

The original program compiled with no optimisations and debugging information was 4 bytes smaller than the recovered version. This is expected because the recovered program was so similar to the original. However the optimised version of the recovered source code was 64 bytes smaller. This is interesting, because the program was translated and some type information was lost. For example the buffer array changed to an array of 32bit integers instead of bytes. It was not expected that an extra 96 bytes (extra 32 * 3) would make such a difference to the program. It appears that as common CPU's are designed for 32bit registers using arrays of bytes would have meant extra processing which caused the recovered optimised source code to be be smaller. The original program when running on the CPU would have had to mask off bytes and access them individually, requiring more instructions.

4.3 Interlinked goto statement problem

The following piece of code is not directly resolvable using either the (Cifuentes 1997) control flow graph approach or the (Proebsting 1997) goto elimination approach. Two goto statements are directly related; this means a different approach is required to resolve and remove the goto's.

```
090 <goto_cond1> addiu    $sp,$sp,-32
094 <goto_cond1+0x4> sd   $s8,16($sp)
098 <goto_cond1+0x8> move $s8,$sp
09c <goto_cond1+0xc> sw   $a0,0($s8)
0a0 <goto_cond1+0x10> sw  $a1,4($s8)
0a4 <goto_cond1+0x14> lw  $v0,0($s8)
0a8 <goto_cond1+0x18> beqz        $v0,0b8
<goto_cond1+0x28>
0ac <goto_cond1+0x1c> nop
0b0 <goto_cond1+0x20> b   0c8 <goto_cond1+0x38>
0b4 <goto_cond1+0x24> nop
0b8 <goto_cond1+0x28> nop
0bc <goto_cond1+0x2c> lw  $v0,0($s8)
0c0 <goto_cond1+0x30> addiu       $v1,$v0,1
0c4 <goto_cond1+0x34> sw  $v1,0($s8)
0c8 <goto_cond1+0x38> lw  $v0,4($s8)
0cc <goto_cond1+0x3c> addiu       $v1,$v0,1
0d0 <goto_cond1+0x40> sw  $v1,4($s8)
0d4 <goto_cond1+0x44> lw  $v0,4($s8)
0d8 <goto_cond1+0x48> slti        $v1,$v0,5
0dc <goto_cond1+0x4c> beqz        $v1,0ec
<goto_cond1+0x5c>
0e0 <goto_cond1+0x50> nop
0e4 <goto_cond1+0x54> b   0bc <goto_cond1+0x2c>
0e8 <goto_cond1+0x58> nop
0ec <goto_cond1+0x5c> move        $sp,$s8
0f0 <goto_cond1+0x60> ld  $s8,16($sp)
0f4 <goto_cond1+0x64> addiu       $sp,$sp,32
0f8 <goto_cond1+0x68> jr  $ra
```

Fig 15: Original Assembler source for the interlinked goto problem.

This example above can be simplified and represented in the intermediate-level source C--. The two statements (label1 and label2) increment the arguments, when cond2 is over 5 the function ends.

```
stack_pointer -= 32;
bits64[stack_pointer + 16] = frame_pointer;
frame_pointer = stack;
bits32[stack_pointer] = arg0;
bits32[stack_pointer + 4] = arg1;

var0 = stack;
if(var0 == 0) goto label1;
else goto label2;

label1:
var0 = stack;
var1 = var0 + 1;
stack = var1;

label2:
var0 = stack[4];
var1 = var0 + 1;
bits32[stack_pointer + 4] = var1;
var0 = stack[4];

if(!(v0 < 5) goto end;
goto label1;

end:
stack = frame_pointer;
frame_pointer = stack[16];
stack_pointer += 32;
goto return_address;
```

Fig 16: Intermediate source code for the interlinked goto problem.

In the intermediate-level source above there are four goto statements. They make up a complex control block. Below is a simplified version of this control flow problem in C.

```
void goto_cond1(int arg1, int arg2)
{
 if(cond1) goto L2;
 L1: stmt_1;
 L2: stmt_2;
 if(cond2) goto L1;
}
```

Fig 17: Interlinked C problem source code in C

(Erosa 1993) demonstrated forward goto replacement with an `if`, and backwards goto replacement with a `do while` loop. This leads to the following two options for elimination of one of the goto's.

```
{
 if(!cond1)
  {
   L1: stmt_1;
  }
 L2: stmt_2;
 if(cond2) goto L1;
}
```

Fig 18: Forward goto elimination

```
{
 if(cond1) goto L2;
 do
  {
   L1: stmt_1;
   L2: stmt_2;
  }
 while(cond2);
}
```

Fig 19: Backwards goto elimination

(Erosa 1993) follow this route to its conclusion. Their method inserts extra variables to the source to eliminate the goto's.

```
{
 goto_L2 = cond1;
 do
  {
   if(!goto_L2)
    {
     goto_L1=0;
     L2: stmt_1;
    }
    goto_L2=0;
    stmt_2;
    goto_L1=cond2;
  }
 while(goto_L1);
}
```

Fig 20: (Erosa 1993) solution to the interlinked goto problem.

While this source above works, it is difficult to follow and is convoluted compared to another possible solution below.   The following examples are all implemented in the `goto-deps` program in Appendix B.

It is possible to remove another variable by moving `cond2` into the `do while` construct, the other variable is unused elsewhere.   The `do while` loop is initialised and a `continue` statement (in this case the use of `continue` is ideal) is used to move to the next iteration and avoid running the instructions more than once in the initialisation run through the loop.

```
{
 goto_L2 = cond1;
 do
  {
   if(goto_L2)
    {
     goto_L2=false;
     stmt_2;
     continue;
    }
   stmt_1;
   stmt_2;
  }
 while(cond2);
}
```

Fig 21: Second solution example to the interlinked goto problem.

The statement `stmt_2` is duplicated for initialisation. Inside the `if(goto_L2)`, the `continue` moves the program counter (PC) to the `while(cond2);` instruction and then the loop can run through iteratively.

The extended approach by myself combines both the forward goto elimination and the backwards goto elimination method (Erosa 1993) demonstrated. This novel extension to the approach produces the following higher-level code.

```
if (!cond1) stmt_1;
do
 {
  stmt_2;
  if (cond2) stmt_1;
 } while (cond2);
```

Fig 22: Third solution example.

The order of the two statements is reversed, duplicating the `stmt_1` that will run when the `cond1` is false; before entering the `do while` loop. The extra `if (cond2)` statement ensures that `stmt_1` is not executed an extra time in the final iteration of the loop. The positive check on the loop exit is as in previous examples.

These three extra example solutions show that for each input intermediate code language program, there are many possible output translations, in a 1:m (one to many) relationship. All of the example programs run and demonstrate the same black box results (when demonstration `printf` is added).

5 Evaluation of information retrieved and its uses

One of the key targets set out at the beginning of this research was to discover how much useful information can be retrieved by decompiling a machine code binary program. The process of video game decompilation has been considered. Now that several key approaches and examples have been examined, an evaluation of the information retrieved can be considered.

There are different types of information that can be retrieved :-

- Source code
- Data structures and file formats
- Hardware architecture details
- Peripheral interface details

Video games are a real-time system, often running directly on the CPU with no OS functionality between the program and the machine architecture. This means that a proportion of the information retrieved will relate to the hardware architecture. This extra information that could dilute the amount of useful information retrieved should not be a problem, if the original libraries can be used to generate signatures and eliminate irrelevant source code from the resulting information derived from phase three. This signature generation process is required in advance of the decompilation.

Data files and file formats that are part of the video game may not be directly suitable for use without modification. In particular, colour palettes and audio formats have improved in quality significantly since the 1980's and changes to colour and audio files are likely to be necessary. Data files are typically in an optimised format for use in the original version of the video game. CPU's and memory are also no longer 8bit aligned. This might not be a problem if the new target is sufficiently superior in power to the original video game console architecture. In the case of the Game Boy Advance used to illustrate a possible target, the CPU is a 32bit Advanced Risc Machines (ARM) chip. All architectures have constraints, the porting programmer would need to be aware of issues relating to the original source architecture and the target architecture. It is often not possible to identify all eventualities in advance, so an experienced programmer should be managing the decompilation process.

In the example program `int-subdivisions` it was demonstrated that a function call included from a library can be replaced with a function call and an `#include`. However, if the functions identified are not present in the new target video game console libraries, the function identification is only useful for program comprehension. In this case, if the functions relate to a peripheral such as a control pad or a graphics chip, these functions would either have to be rewritten for the new architecture or replaced with other equivalent functions (present on the target architecture).

Recovering useful information about the source architecture could also be very useful. In the case of a company wishing to run existing video game titles on a new system, an emulation or API layer could be created to facilitate this. WineX has had some success implementing the DirectX API on GNU/Linux to allow MS-Windows games to run successfully on another OS. Bleem!, was a company that successfully decompiled the Sony PSOne internal BIOS and marketed an emulator product by the same name for MS-Windows; this allowed users to run their Sony PSOne video games on their home computers. Both of these products use the information retrieved differently and both make excellent use of the information.

Peripherals in some cases use proprietary protocols and software; this creates an inter-operability problem for other peripheral manufacturers. In the case of a video game, useful information about a peripheral such as a control pad would possibly allow someone to produce their own control pad that was compatible with the video game console. Also, as is often common now, socket adapters for other video game consoles and computers allow a multitude of different control pads to be connected. Each of these hardware devices uses a protocol that can be recovered by decompiling a piece of software that accesses the peripheral.

If it is not possible to completely decompile a program, it could be that it is the program that is the problem rather than the decompilation process.   The problem has been created to hinder the decompilation process. In Java programs it is becoming more common because the bytecode format is so similar to high-level language.

These issues may affect the quality of the information retrieved:

- Symbol table information.
- Obfuscation.
- Self modifying code.
- Encryption.

Symbol tables contain extra information about a program, its functions, structures and variables. If these have been removed, the recovered information might not be as useful as it could have been. In some cases the symbol table has been specifically replaced with incorrect data by the original developers which now hinders the recovery process. The work around solution is to disregard the symbols and work as though there were none present. Occasionally program headers are partially modified; the program will still run but key program code may be in a different section (hidden as the von Neuman architecture considers data and code memory the same), or even placed in the headers.

Obfuscation is the technique whereby the original program is modified so that its internal execution is convoluted, but its external black box execution appears identical to the original. In my opinion, no obfuscation scheme is impossible to resolve. I believe it has a lot in common with optimised source code, and as such, through a translation process the program can be simplified and thus comprehended. Never ending control flow and unresolvable blocks of code may just have to be commented as such, and left for a programmer to consider after the process has finished. The information and source code retrieved might not be as good as it could have been. Oddities such as loops that count backwards down to negative numbers, structures or functions that do not serve any purpose, are all signs that the program source was modified by a programmer or application to hide its real purpose.

Self modifying code is commonly used in secure programming, copy protection, worms and viruses. This poses a decompilation problem because the functionality changes. Unclear blocks of code can be commented as such allowing a programmer to check it.

If a block of machine code is encrypted, it will not be directly possible to decompile, because the method of encryption will have to be found. If this encryption method could be found, it would be possible to use this recovered information to write a small program to decrypt the block of machine code that was not decompilable.

A video games company using the decompiler would be able to use the source code generated by the decompiler as a near complete basis for their port of the video game. Optimising the source code for the new architecture might mean modifying the original data files and their respective file formats to accommodate the new features required.

A user could assist in the decompilation process by ensuring that the program or library has as much symbol and debugging information left in the file from compilation as possible. This can then be used to generate new source with correct function names and variables.

Often intermediate-level source code that initialises hardware writes values to addresses that are hardware mapped. While these will not be directly comprehensible to the decompiler, the intermediate code around the unknown instructions will. Often this gives clues as to the context of the unknown write to a memory address. If interrupts and other low level architecture calls are present these can only be represented in the high-level source code as Assembler. These blocks of Assembler code can be flagged for the attention of a programmer to check that all libraries have had function signatures created.

There are cases when the source code recovered might not be what was expected. For example, a program compiled for a CPU with no floating point capability would only ever use integer mathematics (for example Sony PSOne).

C-- does not implement run-time services such as garbage collection, this could be a problem if this decompilation process was used on a source architecture was a virtual machine rather than a low-level video game console architecture.

In some cases the recovered source code will not have the same function and variable names as the original program. However, as the functionality is clearly visible in the high-level source code, names can be modified and the source code reused by a video game developer, as required. Using library signatures was evaluated in order to replace statically and dynamically linked functions with their respective library function names. This removed duplicated source code and aided the overall comprehension of the recovered source code.

6 Conclusion

All three of the key aims were attained in this research. Useful information was retrieved to a high degree. Three approaches for retrieval of information were evaluated and the most suitable was selected. A decompilation process for video games was also designed and tested with examples. The process was targeted specifically at the requirements of porting a video game with only a copy of the binary file.

Through this investigation, a structured process was researched for decompiling video games. A modular process was designed with three phases.

For phase one, two different intermediate-level code languages were considered. C-- was selected because it supported all the functionality required and was similar to standard C. Identifying library functions in this phase, where possible, removed duplicate functions and aided the overall comprehension of the source code recovered. Phase one was the most suitable position in the decompilation process for the elimination of library functions because architecture dependencies are eliminated here.

Of the three algorithms considered for use in phase two of the decompilation process, one in particular faired best, the (Cifuentes 1997) Control Flow Graph technique. This retrieved the most detailed information through its identification of each block type and each block's position in the control flow of the whole program. The (Proebsting 1997) approach was also a very good solution to recover detailed information, but was based on a less comprehensive iterative technique. However, not all eventualities were supported in the translation phase of this approach. I therefore made an addition which improved on the original technique, supporting the elimination of unnecessary high-level constructs during the translation section. The final algorithm considered was the (Erosa 1993) approach to goto elimination. With an extension added by myself, this can resolve all blocks of intermediate-level source code. This method is useful as a last option if the two other algorithms cannot resolve a block of code. The approach introduces extra variables to simulate the original goto's control flow, so is not considered suitable for use as a first choice decompilation algorithm.

Phase three of the decompilation process translated the detailed information into the target language, translating high-level constructs where they are not present in the selected language.

By choosing the most suitable approach for each decompilation phase, the complex issue of extracting useful information from a machine code program has been tackled and very useful information can be retrieved. High-level source code can be generated and is of good quality, definitely suitable for reuse by a software development company.

I believe that if an individual or company recovers information from a program they should be allowed to publish this and use as they wish.

7 Future developments

The imperative language C was used in this paper. It would be interesting to evaluate the results from additional support of declarative languages, such as a Lisp module for use in phase three.

A further development in phase two would involve the decompiler receiving input not from an architecture front-end, but directly from a programming language front-end. This would mean that Assembler or legacy source code could be translated into C-- intermediate-level code, processed by phase two and output as new source in a selected target high-level language. This would expand the functionality of the decompiler into program translations as well.

It may be possible in the future for a more interactive decompilation process to be developed for use in particular complex situations. After reading the output data, a programmer could feedback into the decompilation process more identified information (variable, structure and function names). This would be used to improve the quality of the recovered source code after a further pass through the process. The resulting final recovered source code would then be closer to the original source code.

A complete implementation of a decompiler that is widely available to developers and researchers alike would be a great achievement. There are several public decompilation projects currently in various stages of completion; (Boomerang 2002) is currently one of the leading projects in this field.   I hope to see further research and development in this area in the future.

References

Aho, A., Sethi, R., Ulman, D., 1986. *Compilers, Principals, Techniques and Tools.* Bell telephone laboratories, Inc., USA. [Printed]

Erosa, A. and Hendren, L, 1993. *Taming control flow: A structured approach to eliminating goto statements.* McGill University, Canada. [Printed]

Cifuentes, C., 1993. Dcc research decompiler. Queensland University of Technology, Australia. [Online] http://www.itee.uq.edu.au/~cristina/dcc.html [Accessed 2002-11-22]

Cifuentes, C., 1996. *Interprocedural data flow decompilation.* University of Tasmania, Australia [Printed]

Cifuentes, C. and Fraboulet, A. 1997. *Interprocedural data flow recovery of high-level language code from assembly*. The University of Queensland, Australia. [Printed]

Peyton-Jones, S., Ramsey, N. and Reig, R., 1999. *C--: a portable assembly language that supports garbage collection.* Microsoft Research Ltd, University of Virginia, USA and University of Glasgow, UK. [Printed]

Proebsting, T., Watterson, S. 1997. *Krakatoa: Decompilation in Java (Does bytecode reveal source?)*. Microsoft Research Ltd and University of Arizona, USA. [Printed]

Ramshaw, L., 1988. *Eliminating goto's while preserving program structure.* Journal of the Association for Computing Machinery, USA. [Printed]

Bibliography

GNU Project. 2002. *GNU Compiler Collection (GCC) Internals.* Free Software Foundation, USA. [Online] http://gcc.gnu.org/onlinedocs/gccint/ [Accessed 2002-10-04]

Kerninghan, B. and Richie, M., 1988. *The C Programming language* , Prentice Hall, USA. [Printed]

Peyton-Jones, S., Ramsey, N. and Reig, R., 1999. *The C-- reference manual.* [Online] http://cminusminus.org/ [Accessed 2002-10-01]

TransGaming Technologies 2002. *WineX, an implementation of DirectX API for GNU/Linux.* TransGaming Technologies, Canada. [Online] http://www.transgaming.com/technology.php [Accessed 2002-11-22]

Boomerang 2002. An attempt at a general, open source, re-targetable decompiler of binary files, SourceForge, USA. [Online] http://boomerang.sourceforge.net/ [Accessed 2003-01-02]

Appendix A: CD-ROM

The included CD-ROM contains :-

- Playstation 2 MIPS compiler and disassembler.
- Cygwin development environment for MS-Windows.
- Digital copy of this paper in PDF format.
- Original high-level source code of `int-subdivisions`.
- Generated intermediate-level source code of `int-subdivisions`.
- Recovered high-level source code of `int-subdivisions`.
- Original high-level source code of `goto-deps`.

Appendix B: Printed source code

- Original high-level source code of `int-subdivisions`.
- Generated intermediate-level source code of `int-subdivisions`.
- Recovered high-level source code of `int-subdivisions`.
- Original high-level source code of `goto-deps`.

```
/* recursive integer partition solver int-subdivisions
   gcc -Wall -o main main.c
*/

#define PC

#include <stddef.h>

#ifndef PC
#include <tamtypes.h>
#else
#include <stdio.h>
#endif


void sums(int total);
void print_sol(void);

#ifndef PC
int printf(char *format, ...);
#endif

/* display buffer for solution */
int sol_buffer[32];

int sol_pos;

int sol_total;

int main(void)
{
  int answer;
  int max_num;

  max_num = 5;

  printf("integer partitions\n");
  for (answer = 1; answer <= max_num; answer++)
   {
     sol_pos = 0;
     sol_total = 0;
     sums(answer);
     printf("%d solutions for %d\n\n", sol_total, answer);
   }
  return 1;
}


void sums(int total)
{
  int sums_loop;
  int check_loop;
  int valid;
  int previous;


  for (sums_loop = 1; sums_loop <= total; sums_loop++)
   {
     sol_buffer[sol_pos++] = sums_loop;
     sums(total - sums_loop);
     sol_pos--;
   }

/* display if new solution */
  if (total == 0)
   {
     valid = 1;
     previous = sol_buffer[0];

     for (check_loop = 0; check_loop < sol_pos; check_loop++)
      {
          if (sol_buffer[check_loop] > previous)
            valid = 0;
          previous = sol_buffer[check_loop];
      }

     if (valid)
       print_sol();
   }
}


void print_sol(void)
{
  int x;

  for (x = 0; x < sol_pos; x++)
    printf("%d ", sol_buffer[x]);
  printf("\n");
  sol_total++;
}



#ifndef PC
int printf(char *format, ...)
{
  return 1;
}
#endif
```

```
/*
  C-- recovered source of int-subdivisions
*/

Function symbols

0x400040:
main:

printf:
0x400340:

0x400288:
print_sol:

0x4004a8();
__main();



0x400040:
 stack_pointer -= 48;
 bits64[stack_pointer + 32] = return_address;
 bits64[stack_pointer + 16] = frame_pointer;
 frame_pointer = stack_pointer;
 0x4004a8();
 var0 = 5;
 bits32[stack_pointer + 4] = var0;
 arg0 = 0x400000;
 arg0 += 0;
 0x400340();
 var0 = 1;
 bits32[frame_pointer] = var0;

0x400078:
 bits32[frame_pointer] =  var0;
 var1 = bits32[frame_pointer + 4];
 var0 = (var0 <= var1);
 if (var0 == 0) goto 0x400094;
 goto 0x4000e0;

0x400094:
 asm_temp = 0x400000;;
 bits32[asm_temp + 1540] = 0;
 asm_temp = 0x400000;;
 bits32[asm_temp + 1536] = 0;
 arg0 = bits32[frame_pointer];

0x4000a8:
 0x400108();

 arg0 = 0x400000;
 arg0 += 24;
 arg1 = 0x400000;
 arg1 = bits32[arg1 + 1536];
 arg2 = bits32[frame_pointer];
 0x400340();
 bits32[frame_pointer] =  var0;
 var1 = var0 + 1;
 bits32[frame_pointer] = var1;
 goto 0x400078;
 var0 = 1;
 goto 0x4000ec;

0x4000ec:
 stack_pointer = frame_pointer;
 return_address = bits64[stack_pointer + 32];
 frame_pointer = bits64[stack_pointer + 16];
 stack_pointer += 48;
 goto return_address;

0x400108():
 stack_pointer -= 64;
 bits64[stack_pointer + 48] = return_address;
 bits64[stack_pointer + 32] = frame_pointer;
 frame_pointer = stack_pointer;
 bits32[frame_pointer] = arg0;
 var0 = 1;
 bits32[stack_pointer + 4] = var0;

0x400128:
 var0 = bits32[stack_pointer + 4];
 var1 = bits32[stack_pointer];
 var0 = (var1 <= var0);
 if (var0 == 0) goto 0x400144;

 goto 0x4001b4;

0x400144:
 var0 = 0x400000;
 var0 += 1540;
 var1 = bits32[var0];
 arg0 = var1;
 arg1 = (arg0 < 2);
 arg0 = 0x400000;
 arg0 += 1792;
 arg1 += arg0;
 arg0 = bits32[frame_pointer + 4];
 bits32[arg1] = arg0;
 var1 ++;
 bits32[var0] = var1;
 bits32[frame_pointer] =  var0;
 bits32[frame_pointer + 4] =  var1;
 var0 -= var1;
 arg0 = var0;

 0x400108();
 var0 = 0x400000;
 var0 = bits32[var0 + 1540];
 var1 = var0 - 1;
 asm_temp = 0x400000;;
 bits32[asm_temp + 1540] = var1;
 var0 = bits32[stack_pointer + 4];
 var1 = var0 + 1;
 bits32[frame_pointer + 4] = var1;
 goto 0x400128;

0x4001b4:
 var0 = bits32[frame_pointer];
 if (var0 != 0) goto 0x400270;
 var0 = 1;
 bits32[frame_pointer + 12] = var0;
 var0 = 0x400000;
 var0 = bits32[var0 + 1792];
 bits32[frame_pointer + 16] = var0;
 bits32[frame_pointer + 8] = 0;

0x4001d8:
 var0 = bits32[frame_pointer + 8];
 var1 = 0x400000
 var1 = bits32[var1 + 1540];
 var0 = (var0 < var1);
 if (var0 != 0) goto 0x4001f8;
 goto 0x40025c;
 var0 = bits32[frame_pointer + 8];
 var1 = var0;
 var0 = (var1 < 2);
 var1 = 0x400000
 var1 += 1792;
 var0 += var1;
 var1 = bits32[var0];
 var0 = bits32[frame_pointer +16];
 var1 = (var0 < var1);
 if (var1 == 0) goto 0x400228;
 bits32[frame_pointer + 12] = 0;

0x400228:
 var0 = bits32[frame_pointer + 8];
 var1 = var0;
 var0 = (var1 < 2);
 var1 = 0x400000
 var1 += 1792;
 var0 + = var1;
 var1 = bits32[var0];
 bits32[frame_pointer + 16] = var1;
 var0 = bits32[frame_pointer + 8];
 var1 = var0 + 1;
 bits32[frame_pointer + 8] = var1;
 goto 0x4001d8;
 var0 = bits32[frame_pointer + 12];
 if (var0 == 0) goto 0x400270;
 0x400288();

0x400270:
 stack_pointer = frame_pointer;
 return_address = bits64[stack_pointer + 48];
 frame_pointer = bits64[stack_pointer + 32];
 stack_pointer += 64;
 goto return_address;

0x400288():
 stack_pointer -= 48
 bits64[stack_pointer + 32] = return_address;
 bits64[stack_pointer + 16] = frame_pointer;
 frame_pointer = stack_pointer;
 bits32[frame_pointer] = 0;
```

```
0x4002a0:
 bits32[frame_pointer] =  var0;
 var1 = 0x400000
 var1 = bits32[var1 + 1540];
 var0 = (var0 < var1);
 if (var0 != 0) 0x4002c0;
 goto 0x400300;

0x4002c0:
 bits32[frame_pointer] =  var0;
 var1 = var0;
 var0 = (var1 < 2);
 var1 = 0x400000
 var1 += 1792;
 var0 + =var1;
 arg0 = 0x400000;
 arg0 += 48;
 arg1 = bits32[var0];
 0x400340();
 bits32[frame_pointer] =  var0;
 var1 = var0 + 1;
 bits32[frame_pointer] = var1;
 goto 0x4002a0;

0x400300:
 arg0 = 0x400000;
 arg0 += 56;
 0x400340();
 var0 = 0x400000;
 var0 = bits32[var0 + 1536];
 var1 = var0 + 1;
 asm_temp = 0x400000;;
 var1 = bits32[asm_temp + 1536];
 stack_pointer = frame_pointer;
 return_address = bits64[stack_pointer + 32];
 frame_pointer = bits64[stack_pointer + 16];
 stack_pointer += 48;
 goto return_address;




printf:
0x400340():
 addiu    $sp,$sp,-160
0x400344 bits64[stack_pointer + 16] = frame_pointer;
0x400348 frame_pointer = stack_pointer;
0x40034c bits64[stack_pointer + 104] = arg1;
0x400350 bits64[stack_pointer + 112] = arg2;
0x400354 sd         $a3,120($s8)
0x400358 sd         $t0,128($s8)
0x40035c sd         $t1,136($s8)
0x400360 sd         $t2,144($s8)
0x400364 sd         $t3,152($s8)
0x400368 swc1       $f12,88($s8)
0x40036c swc1       $f14,92($s8)
0x400370 swc1       $f16,96($s8)
0x400374 swc1       $f18,100($s8)
0x400378 bits32[frame_pointer] = arg0;
```

```
0x40037c var0 = 1;
0x400380 b 0x400388
0x400384 nop
0x400388 stack_pointer = frame_pointer;
0x40038c frame_pointer = bits64[stack_pointer + 16];
0x400390 addiu      $sp,$sp,160
0x400394 goto return_address;
```

```
/*
   Recovered source code for int-subdivision
*/

#include <stddef.h>
#include <stdio.h>

/* global variables */
int frame_var0;
int frame_var1;
int frame_buffer0[32];

void sums(int total);
int main(void);
void function1(int stack_var0);
void function2(void);

int main(void)
{
  int stack_var0;
  int stack_var1;              /* loop variable */

  stack_var0 = 5;

  printf("integer partitions\n");

  for (stack_var1 = 1; stack_var1 <= stack_var0; stack_var1++)
   {
     frame_var0 = 0;
     frame_var1 = 0;
     function1(stack_var1);
     printf("%d solutions for %d\n\n", frame_var1, stack_var1);
   }
  return 1;
}


void function1(int stack_var0)
{
  int stack_var1;              /* loop */
  int stack_var2;
  int stack_var3;
  int stack_var4;              /* printf loop variable */


  for (stack_var1 = 1; stack_var1 <= stack_var0; stack_var1++)
   {
     frame_buffer0[frame_var0++] = stack_var1;
     function1(stack_var0 - stack_var1);
     frame_var0--;
   }

  if (stack_var0 == 0)
   {
     stack_var2 = 1;
     stack_var3 = frame_buffer0[0];

     for (stack_var4 = 0; stack_var4 < frame_var0; stack_var4++)
      {
          if (frame_buffer0[stack_var4] > stack_var3)
            stack_var2 = 0;
          stack_var3 = frame_buffer0[stack_var4];
      }

     if (stack_var2)
       function2();
   }
}


void function2(void)
{
  int stack_var0;              /* loop variable */

  for (stack_var0 = 0; stack_var0 < frame_var0; stack_var0++)
   {
     printf("%d ", frame_buffer0[stack_var0]);
   }
  printf("\n");
  frame_var1++;
}
```

```
/*
   illustrate interlinked goto problem
   goto-deps example program
*/

#include <tamtypes.h>
#include <stddef.h>
#include <stdio.h>
#include <kernel.h>
#include <sifrpc.h>


//#define DEBUG

void goto_cond1(int arg1, int arg2);
void goto_cond2(int arg1, int arg2);
void goto_cond3(int arg1, int arg2);
void goto_cond4(int arg1, int arg2);
void goto_cond5(int arg1, int arg2);

int main(void)
{
 goto_cond1(0, 0);
 goto_cond2(0, 0);
 goto_cond3(0, 0);
 goto_cond4(0, 0);
 goto_cond5(0, 0);

 return 1;
}


void goto_cond1(int arg1, int arg2)
{
#ifdef DEBUG
 printf("goto test1\n");
 printf("start arg1: %d arg2: %d\n", arg1, arg2);
#endif
 if(arg1) goto L2;
 L1: arg1++;
 L2: arg2++;
#ifdef DEBUG
 printf("current arg1: %d arg2: %d\n", arg1, arg2);
#endif
 if(arg2<5) goto L1;
}


void goto_cond2(int arg1, int arg2)
{
#ifdef DEBUG
 printf("\ngoto test2\n");
 printf("start arg1: %d arg2: %d\n", arg1, arg2);
#endif
 if(!arg1)
  {
   L1: arg1++;
  }
 arg2++; // L2
```

```
#ifdef DEBUG
 printf("current arg1: %d arg2: %d\n", arg1, arg2);
#endif
 if(arg2<5) goto L1;
}


void goto_cond3(int arg1, int arg2)
{
#ifdef DEBUG
 printf("\ngoto test3\n");
 printf("start arg1: %d arg2: %d\n", arg1, arg2);
#endif
 if(arg1) goto L2;
 do
  {
   arg1++; // L1
   L2: arg2++;
#ifdef DEBUG
   printf("current arg1: %d arg2: %d\n", arg1, arg2);
#endif
  }
 while(arg2<5);
}


void goto_cond4(int arg1, int arg2)
{
 int goto_L2;
#ifdef DEBUG
 printf("start arg1: %d arg2: %d\n", arg1, arg2);
 printf("\ngoto test4\n");
#endif
 goto_L2 = arg1;
 do
  {
   if(goto_L2)
    {
     goto_L2=0;
     arg2++; // L2
     continue;
    }
   arg1++; // L1
   arg2++; // L2
#ifdef DEBUG
   printf("current arg1: %d arg2: %d\n", arg1, arg2);
#endif
  }
 while(arg2<5);
}



void goto_cond5(int arg1, int arg2)
{
#ifdef DEBUG
 printf("\ngoto test5\n");
 printf("start arg1: %d arg2: %d\n", arg1, arg2);
#endif
```

```
 if(!arg1) arg1++; // L1
 do
  {
   arg2++; // L2
#ifdef DEBUG
   printf("current arg1: %d arg2: %d\n", arg1, arg2); //here as loop
is wrapped
#endif
   if(arg2<5) arg1++; // L1
  }
 while(arg2<5);
}
```

Appendix C

- MSc Project proposal

MSc Project Proposal IC550

Investigating Different Implementations of
Source Code Recovery Techniques.
How much Valuable Information can be Retrieved?

Jonathan Grant
0013499

1. Overview of Project Aims

Over the last 5 months I have read many Journals, Internet articles and other sources of information. The aims are developed from the bibliography and chosen as the best direction for the project to take. The angle chosen is considered the most applicable, incorporating research, journal articles, programming and critical appraisal at the end.

An automated program that could half the workload of a programmer porting source code is the dream of many development managers. Now using modern analytical techniques and a logical analysis of assembly language code it is possible to implement this in a program and recover information.

The research will focus on specific aims to answer after developing test programs:

- How far could such a tool take a compiled executable and 'restore' the source?
- How much of the algorithm recovered would be the same as the original?
- How much of the information would be useful?

Other smaller topics that are optional extras in the same field:

- Investigating Obfuscation programming techniques and how this affects source code recovery.
- Compressed data blocks could have a decompressing routine in the executable. If the algorithm routines could be found this could be fed back into the program to restore the rest of the data segment C code.

There are certain key papers referenced in the Bibliography section later have theoretically researched this topic. Using this preliminary research the aims have been chosen not to repeat other work; thus to further the research in this topic as a whole and develop new techniques and methods to utilise in the programming / implementation section.

There are several situations where this research would be the key to saving substantial time.

- Recovery of lost program source code.
- Translation of a program from an obsolete language into usable source.
- Algorithm recovery.

Using intelligent techniques that can recognise functions, structures, variables and optimisations to generate usable source again. Take advantage of symbols in the executable to give function and variable names.

Developing a program working assembler back to pseudo C with if's and goto's will be the first stage. The major part of the work will be devoted to recognising higher level abstractions from the executable. The output should be Functions, Algorithms, Structures and an overview profile of the way the program works. This will be meaningful output and can be abstracted straight away and incorporated again into further software development projects.

2. Reasons for selecting this subject for the project

Drawing on my previous skills helped in the process of choosing an applicable topic for the MSc project. With extensive experience of MIPS assembler on the Playstation and other architectures I chose to build on this solid grounding. Also being a keen Linux evangelist meant I am very skilled UNIX and Linux (First used Linux in 1996, using the 1.x kernel).

Developing on this stable OS using the powerful GNU tools mean I have a solid base with the support of 15 years of tried and tested GNU software (full source code available). Compatible with over 30 different CPUs and architectures the program has plenty of scope for further work.

There is a large potential to achieve with various key issues to discus, technical, moral and also contentious legal issues. It is also very challenging to work on a new and different area of research and development.

3. Project objectives and major tasks

This is the overview schedule, each is considered a milestone: -

a) Generate program shell
b) Pseudo C output (if/goto)
c) Integrate 'for' and 'while' loop recognition
d) Add support for the symbol table.
e) Function recognition
f) Testing
g) Investigating how much information can be recovered by testing with simple algorithms and complicated algorithms.

4. Description of how each task will be accomplished

The first iteration should be generated as pseudo C, from that point features can be added so that the measurement of success will not be too hard to achieve.

Assembler contains very basic operations; it will be identifying these smaller groups of op codes that represent higher level language constructors that will be one of the keys to developing source code retrieval techniques.

```
lui r1, 0x29B
ori r1, 0x1A
```

This is the actual implementation of the op code 'li r1, 0x29B001A'. 'li' is a pseudo MIPS specific op code; similar pseudo op codes exist on many systems. Recognising simple sequences of op codes that do specific things is the key to understanding more complex functionality.

5. Specific resources required

Key success will be the methods/algorithms used to implement the actual program. This would be a large project but with the availability of the free base package 'binutils' from http://www.gnu.org/ released under the GPL license it shall save a lot of time, meaning work can instantly start and avoid wasting time writing 'general' program functionality. The project can be developed to take advantage of the functionality already available. For example executable format loaders are implemented. This would mean that the research would be specific and no time would be wasted. With this good tested framework it would enable work to start straight away.

6. Project Schedule / Overview Project Plan

As I am on an Internship Placement for 12 Months in Tokyo the Project Plan can not be completed until I know the situation I will have for part time study while in Japan. This is the reason it is better to leave this section until I know all the information to complete it at the start of the MSc

7. Key References and initial Bibliography

Brand van den, M., Klint, P. and Verhoef, C. 1996. *Reverse Engineering and System Renovation – An Annotated Bibliography*. Programming Research Group, University of Amsterdam. [Printed]

Cifuentes, C., Simon, D. and Fraboulet, A. 1998. *Assembly to High-Level Language Translation.* University of Queensland & Sun Microsystems. [Printed]

Collberg, C., Thomberson, C. and Low, D. 1998. *Breaking Abstractions and Unstructuring Data Structures.* University of Auckland, New Zealand. [Printed]

Cifuentes, C. and Fitzgerald, A. 1999. *Is Reverse Engineering Always Legal?* IT Pro [Printed]

Cifuentes, C. 2000. *The Impact of Copyright on the development of Cutting Edge Binary Reverse Engineering Technology.* University of Queensland & Software Engineering Australia. [Printed]

Schwartz, K. 2001. *Reverse Engineering: Necessary Function Or Illegal Activity?* Planet IT [Online]
http://www.planetit.com/techcenters/docs/security/news/PIT20010123S0001
[Accessed 22 May 2001]