FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit in Informatik

Control flow reconstruction from PowerPC binaries

Bogdan Mihaila

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit in Informatik

Control flow reconstruction from PowerPC binaries

Rekonstruktion des Kontrollflusses aus PowerPC Binärdateien

Author:	Bogdan Mihaila
Supervisor:	Prof. Helmut Seidl
Advisor:	Dipl.Inf. Michael Petter
Advisor:	Dipl.Inf. Andrea Flexeder
Date:	November 16, 2009

I assure the single handed composition of this diploma thesis
only supported by declared resources.

München, November 16, 2009 Bogdan Mihaila

Acknowledgments

Finishing this thesis would not have been possible without the help of many people, friends and colleagues alike. Foremost, I would like to thank my two advisors, Michael Petter and Andrea Flexeder for providing valuable advice during the development of the software and the written part of this thesis. Special thanks to my family who supported me throughout my whole studies and provided me with the necessary encouragement during this thesis.

Abstract

This thesis describes the issues faced when reconstructing control flow graphs (CFG) from PowerPC binaries. It explains the challenges posed by computed jumps that are sometimes only known at program runtime and how the techniques of static analyses handle these. The language constructs in the C programming language that lead to computed jumps are analyzed and this knowledge is used to reconstruct the control flow even for computed jumps. The steps that need to be performed to reconstruct switches and procedure calls are presented in detail, along with the static analyses to perform the task. Furthermore the problems and limitations of the approaches are stated and further analyses and improvements are proposed.

Contents

Outline of the Thesis

Part I: Introduction

This chapter presents the motivation for analyses on binaries.

Part II: CFG Reconstruction

CHAPTER 1: PROGRAM REPRESENTATION

This chapter presents the definitions and theory behind control flow graphs. It shows examples and describes why an interprocedural CFG is a suitable representation of the program.

CHAPTER 2: GOALS TO ACHIEVE

This chapter states the precise goals for the reconstruction process, in particular the problems with computed jumps and the need for an overapproximation of the possible control flow at runtime.

Part III: Analyses

CHAPTER 3: ANALYZED PROGRAM CLASS

This chapter describes the class of programs and the machine architecture is be used for the analyses in the next chapters.

CHAPTER 4: PRELIMINARY ICFG

This chapter describes how the interprocedural CFG is extracted from the binary. It shows the steps performed to build a preliminary ICFG that is analyzed and further refined by the analyses in the next chapters.

CHAPTER 5: SWITCHES

presents the analyses performed to reconstruct switch constructs. It explains how switches are compiled to assembler and what problems arise for the CFG reconstruction. It introduces the steps needed to reconstruct a switch and shows these by means of a static analysis.

CHAPTER 6: PROCEDURE CALLS

presents the analyses performed to verify procedure calls and resolve function pointers. The compilation of functions to procedures in assembler is explained and with it the mechanisms and structures that are needed during the program runtime for procedure calls. It then shows how to verify for violations to these procedure structures with a static analysis. In a second part the introduced analysis is used to infer function pointer calls.

CHAPTER 7: FUTURE WORK

Explains what can be done to improve and extend the work presented in the thesis.

Part IV: Results and related work

CHAPTER 8: COMPARISONS

Compares other existing tools and presents related work.

CHAPTER 9: CONCLUSION

A summary and conclusion of the thesis is given.

Part I

Introduction

Why analyses on binaries

High level programming languages—e.g. C, Java, Python, Fortran, Haskell—are a formal description of a program on an abstract level. They express concepts by using words and the combination of idioms and constructs that can vary in conciseness. Depending on the paradigms a language follows, it may be better suited for one field of application or problem domain than for another. In general high-level languages try to be closer to the human mind and human reasoning than to the machine. The expressiveness, paradigms and available constructs differ between the languages because of the historical background of each and the available knowledge and trends in computer languages of that time.
In contrast to high–level languages, assembler languages are very close to the machine. They directly represent the features of the machine architecture and are rather tedious to understand or process for humans. Since it is the only language that the machine understands any program must be translated to assembler from the higher-language representation, to be executed. Fortunately with the invention of the higher-level languages came the invention of compilers whose task is to perform this translation.

The result of the translation—the program binary—is the file that will be executed by the machine. And although it must be semantically equivalent to the program stated in the high-level language the translation process done by the compiler may introduce mistakes. This can happen as compilers are quite complex and apply code optimizations on various levels during the translation. Also each hardware platform provides a different set of instructions that the compiler needs to optimize for. As a consequence, if a program needs security critical analysis it is imperative [2] that the binary has to be analyzed as the compiler might introduce subtle artefacts that may change the program properties enough to invalidate the analysis results. This is especially true for worst-case execution time (WCET) analyses [12] where the timings of each machine instruction is of importance. Summarizing, the assembly code is what the machine executes and thus the only reliable program representation for analyses about the behvior of the program.

A second motivation for analyses on program binaries is that the source code may not be available. For proprietary code or malicious code it is only possible to analyze the binaries. The verification of proprietary code for security exploits or the verification of untrusted code (e.g. Google's sandboxing approach [36]) need to be performed on binaries.

Why the PowerPC architecture

Although the PowerPC architecture [13] on the desktop was abandoned by Apple it is still in wide spread use especially for embedded systems and high performance computing. It has proven quite useful since its introduction in 1992 and is used more and more in security critical applications, e.g. controllers in cars and aircrafts.
It has a clean design and is a RISC (reduced instruction set computer) architecture. This simplifies the work needed to cope with the amount of instructions provided by the processor and the existing corner cases and dependencies between the instructions. It was thus chosen as the platform to implement the analyses for control flow reconstruction in this thesis.

Part II

CFG Reconstruction

1 Program representation

There are various formats for executable binaries depending on the operating system and the supported machine architectures [21]. But all formats have in common that besides a header and various management information, the actual program to be executed is represented as a list of instructions that will be loaded into memory. A form that results from the sequential processing of instructions by the processor. Instructions are read from memory by the processor and executed sequentially.
Disassembling an executable translates the binary coded instructions to their textual representations (so called mnemonics) for humans to be able to read them. A process that is done by using a translation table that matches bit patterns with the corresponding mnemonics. Thus disassembling a binary results in a sequential list of instructions but instead of line numbers the instructions are identified by their address in memory. It is important that each instruction can be identified uniquely to be able to perform jumps and thus change the program flow.

Definition 1.1 (Program list)**.**
For a program p let $L_p = \{(A_1, I_1), \ldots, (A_n, I_n)\}$ with $n \in \mathbb{N}$ be a finite list of tuples of addresses A and instructions I that represent the program. Addresses are unique $A_i \neq A_j$ for $i \neq j$ but the same instructions can occur more often.

The possibility to jump between code not only enables a more compact representation for the program—think of loops—but a program can perform alternative computations depending on conditions that are set before or during its execution. It means that the same list of instructions can produce a different sequence of executed instructions for each program run. A single run of the program can therefore be seen as a trace through the instruction list.

Definition 1.2 (Program trace)**.**
A particular execution of the program p is a sequence of instructions through the program list L_p during this particular run expressed as a series of instruction addresses $T_p = \{A_1, \ldots, A_m\}$ with $m \in \mathbb{N}^\infty$. Note that the program may never end and thus the list of addresses in the trace can be infinite.

1.1 Control Flow Graphs

To analyze and reason about a program an abstract representation is needed. This representation should abstract from the characteristics of a high-level language and provide more information than a low-level assembly like language. One such representation is the control flow graph (CFG) (see figure 1.1), which is a directed graph that connects instructions with their predecessors and successors, thus capturing the flow of control. Because of jumps and branches and thus the existence of more than one successor and predecessor, the graph structure is a more appropriate model of a program than a simple linear list. Although equivalent, a sequential list—e.g. program source code and binaries—is conveying the program flow unnecessarily because the possible paths through the program are not immediately visible and are only revealed as the program is run. Whereas the graph explicitly shows the successor/predecessor relation and the possible paths. As a benefit of these direct connections it is easier to find dependencies of data and control flow and use this knowledge to analyze programs.

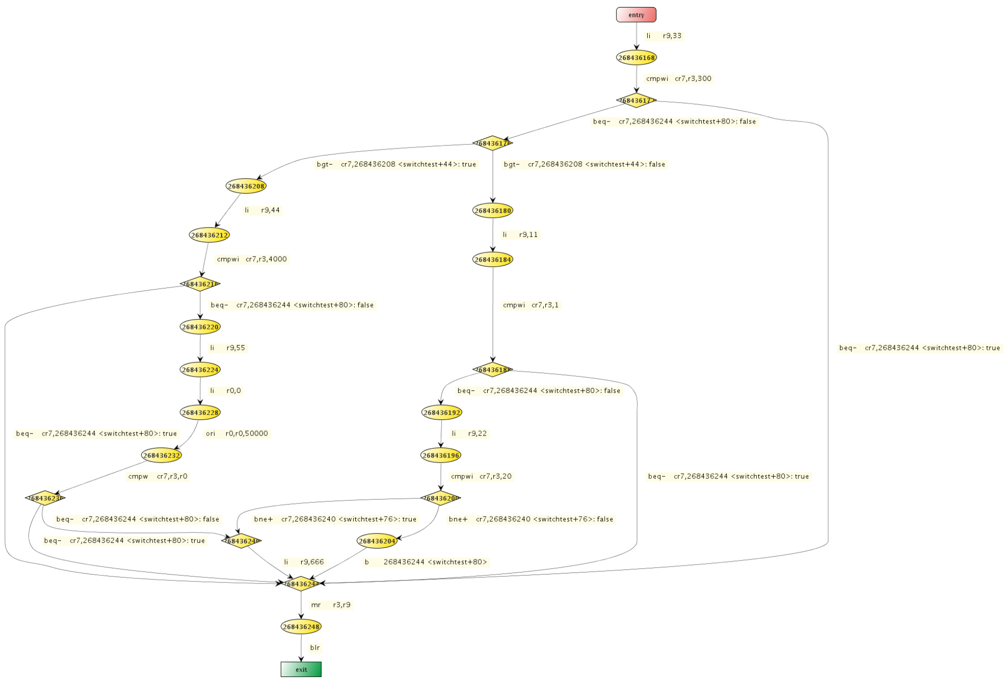
entry
li r9,33
268436168
cmpwi cr7,r3,300
beq- cr7,268436244 <switchtest+80>: false
bgt- cr7,268436208 <switchtest+44>: true
bgt- cr7,268436208 <switchtest+44>: false
268436208
li r9,44
268436212
cmpwi cr7,r3,4000
268436180
li r9,11
268436184
cmpwi cr7,r3,1
beq- cr7,268436244 <switchtest+80>: false
268436220
li r9,55
268436224
li r0,0
268436228
beq- cr7,268436244 <switchtest+80>: true
ori r0,r0,50000
268436232
cmpw cr7,r3,r0
beq- cr7,268436244 <switchtest+80>: true
beq- cr7,268436244 <switchtest+80>: false
268436192
li r9,22
268436196
cmpwi cr7,r3,20
beq- cr7,268436244 <switchtest+80>: true
beq- cr7,268436244 <switchtest+80>: false
bne+ cr7,268436240 <switchtest+76>: true
bne+ cr7,268436240 <switchtest+76>: false
beq- cr7,268436244 <switchtest+80>: true
268436204
li r9,666
b 268436244 <switchtest+80>
mr r3,r9
268436248
blr
exit

Figure 1.1: CFG example

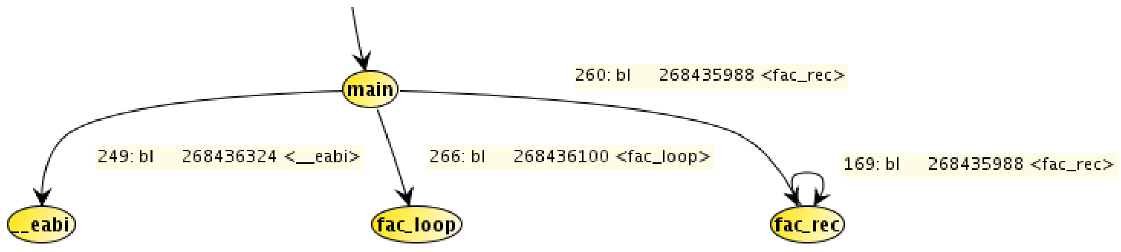
main
260: bl 268435988 <fac_rec>
249: bl 268436324 <__eabi>
266: bl 268436100 <fac_loop>
169: bl 268435988 <fac_rec>
__eabi
fac_loop
fac_rec

Figure 1.2: Call Graph example

1.1.1 Intraprocedural CFG

A program is a collection of different procedures, possibly calling each other. Each procedure f can be described by a finite intraprocedural control flow graph with these properties:

Definition 1.3 (CFG_f)**.**

- a set of nodes N, with one node n for each program point in the procedure,
- a set of edges $E \subseteq (N \times N)$ as transitions from one program point to another,
- a unique program point $entry_f \in N$ at which the procedure starts,
- a unique program point $exit_f \in N$ at which the procedure ends,
- and a mapping from edges to instructions and their addresses $label : E \rightarrow (A_i, I_i)$.

The unique entry point is given by the first instruction in a procedure[1] and the unique exit point can be achieved by connecting each instruction that leaves the function (i.e. return instructions) with a designated exit node in the CFG. The nodes represent the abstraction of a state in the program, whereas the edges represent the state transitions or the concrete instructions of the program. The program abstraction as a graph is similar to a list of instructions in that it is a container and the semantics are given by the instruction types that the edges are mapped to. Instruction semantics can range from low-level machine instructions to compounds of expressions or statements in higher level languages.

1.1.2 Call Graph

With procedures seen as single entities a program can be described by the call relations between them (see figure 1.2). Below are the properties of such a call graph for a program p:

Definition 1.4 (CG_p)**.**

- a set of nodes N, with one node n for each procedure in the program,
- a set of edges $E \subseteq (N \times N)$ for each call from one function to another,
- a unique function $entry_p \in N$ at which the program starts,
- a unique function $exit_p \in N$ at which the program ends,
- and a mapping from edges to call instructions and their addresses $label : E \rightarrow (A_i, I_i)$.

Again the unique entry is given by convention and is the one that is called first by the operating system when launching the program—e.g. the procedure `main`. The choice of a

[1] this is a simplification because there can be multiple entry points in low-level machine languages (an assembler jump instruction can jump to every address in the code) and languages that support coroutines

unique exit point may seem not so obvious because in theory there are many points where a program can terminate—intended by the programmer or not—still there is a special function to be called on exit to return control to the operating system. It should be noted that the call graph only contains edges for the calls but not for the returns as it is assumed that each call will eventually return to the caller and that just means following the edges backwards.

1.1.3 Interprocedural CFG

To analyze the whole program i.e. functions and the call relations the above two structures will be combined into an interprocedural control flow graph. In addition to the simple CFG two new types of edges will be added that refine the simple definition of calls in the call graph which yields the following properties for the interprocedural control flow graph of a program:

Definition 1.5 ($ICFG_p$)**.**

- a set of intraprocedural control flow graphs, one CFG for each procedure in the program,
- a set of edges $E_{enter} \subseteq (N \times N)$ for each call from one procedure to another,
- and a set of edges $E_{exit} \subseteq (N \times N)$ for each return from one procedure to the caller.

1.2 Semantics of the CFG

To prove that an *ICFG* is a correct description of the assembly program it must be shown that the semantics that is described by the ICFG representation is equivalent to the semantics of a binary executable. As stated in definition 1.3 the CFG edges are mapped to a set of machine instructions, thus passing an edge has the same effect as executing an instruction in a binary. Passing an edge yields a trivial equivalence to execution.
The next step is to show that the connection relations in the CFG maintain the semantics of the execution of a program, viz. the sequence of instructions that can occurr during program execution is also possible to be produced by following the edges in the CFG. There are two types of instructions, the ones that don't alter the sequential control flow and the ones that do. The former are explained below:

Definition 1.6 (Fall-through edge)**.**
Because of the sequential way of execution each instruction (A_i, I_i) has an implicit successor, i.e. the next instruction (A_{i+1}, I_{i+1}) in the program list. As long as the instruction does not initiate a change of control (calls, jumps or branches) the implicit successor will be executed next. This type of instructions that don't change the sequential flow are called fall-through edges in the CFG representation.

As fall-through edges can be mapped directly to a linear CFG the latter type is more interesting. The instructions that have multiple alternatives as successors should be represented by multiple edges in the CFG, one for each successor.

Definition 1.7 (Jump edge).
A jump is an instruction (A_i, I_i) that states the address of the next instruction to be executed. The instruction executed after the jump can be any instruction (A_j, I_j) in the program.

There are two classes of jumps:
depending on how the target of the jump is known:

static Contains the jump target address in the instruction text as a constant value, thus the jump target is static and won't ever change during all runs of the program.

computed/dynamic/indirect The names express the different aspects of this jump type. Computed because it computes the target at runtime. Thus it can be different with each program execution (dynmamic), but does not necessarily have to be different. It is called an indirect jump because it uses a register as target operand and not a hard-coded constant.

and if the jump is always executed:

conditional The jump is executed only if a condition is met. Conditions are values of registers or flags. If the condition is not true, then the fall-through edge will be taken and the next instruction in the list will be executed.

unconditional The jump will always be executed.

A jump is always a combination of these two classes. A *branch* is another name for a conditional jump (in the CFG represented as a *true* and *false* edge). Calls can be any of the above but have additional properties that result in special instructions on some machine architectures, e.g. the *x86* architecture. There are jumps that can go far away from the actual position, so called far- or long-jumps. And calls are meant to return back at the next instruction (in similarity to a fall-through edge) after the code in the called function was executed. To achieve this there must be a jump back to the position after the address where the call was performed. Because a function can be called from different places, the jump back address cannot be hard-coded into a static jump, but must be saved before the call to a special place and then upon return this saved value is used for an indirect jump. These additional steps can be combined into a single instruction that is named a *call* (in the ICFG represented as an edge to the entry of the callee). Nonetheless it is important to point out that even on architectures that have a designated instruction for the call the same can be achieved by performing the above steps manually and using a plain jump (this will be of importance when trying to distinguish between a simple jump and a call).

When assuming that the target of a jump is known—i.e. it is a static jump or the value in the register of an indirect jump is known—the first two types of jumps will be treated equally. For each jump an edge will be inserted to the CFG to connect the instruction before the jump with the target instruction. In fact, because the instructions themselves are edges, the connections will be made between the nodes before the source and target instruction and the connecting edge will be the jump instruction itself. So for each jump target an edge has to be inserted to the CFG to reflect the possible path and if the jump is conditional then a fall-through edge must be inserted, too.

By inserting all the jump connections according to the semantics of the actual jump instructions the CFG will contain all the sequences of instructions as paths that the program can produce during execution. If the semantics of a program is seen as the entirety of all its possible executions then the CFG models all the possible program runs that can occur in reality and thus is semantically equivalent.
In analogy to a program trace 1.2 the same can be defined on an ICFG:

Definition 1.8 (ICFG trace).
For an ICFG of a program $ICFG_p$ let $P_{ICFG_p} = \{E_1, \ldots, E_n\}$ with $n \in \mathbb{N}^\infty$ be a possibly infinite list of edges on a path through the ICFG from $entry_p$ to $exit_p$. A trace in the ICFG T_{ICFG_p} is the sequence of addresses that is produced by applying the mapping $label : E \to (A_i, I_i)$ of edges to instructions on the path through a CFG: $T_{ICFG_p} = \{A_1, \ldots, A_n\}$

The above statement can thus be put more formally:

Theorem 1.9 (Trace equivalence).
The set of traces S^T_{bin} produced by all possible executions of the program binary is contained in the set of traces S^T_{icfg} that is produced by following the possible paths in an ICFG: $S^T_{bin} \subseteq S^T_{icfg}$

To prove this, let's see the possible cases:

1. The instruction produces only a fall-through ICFG edge. Passing an edge is a trivial equivalence to executing an instruction, thus has the same effect for both program and ICFG traces, of adding the instruction address to the trace.
2. The instruction produces a single jump edge to another node in the ICFG. As there is no alternative path it boils down to following this edge and is similar to a fall-through edge.
3. The instruction produces two alternative edges in the ICFG. The ICFG trace will be split into two different traces, one for taking the left path and one for taking the right one. As the program trace must take one of the branches it will be equivalent to one of the ICFG traces.
4. The instruction produces n alternative edges in the ICFG. This case is analogous to the one with two alternatives.

So by using the rules mentioned above to build the ICFG, for each possible trace that is produced by running the binary, there will exist an equivalent trace in the ICFG.

2 Goals to achieve

When analyzing programs for safety critical concerns (e.g. worst case execution times [32, 33]) or code verification the underlying ICFG must be as precise as possible. A slightly incorrect ICFG will produce meaningless results independently of a correct analysis. To achieve a precise ICFG any uncertainty or unpredictable features of control flow (e.g. computed jump targets) that are caused by the semantics of the underlying language must be resolved. Where an exact resolution is not possible, the ICFG should be approximated safely by inserting potential flow connections. They are potential because they may happen at runtime but not necessarily, thus being a superset of the jumps at runtime. The more of these potential connections can be categorized as infeasible the more precise the ICFG is. Sources of imprecision are jumps to targets that are computed at runtime and thus difficult to predict when analyzing the program statically.

2.1 Resolve computed jumps

Such jumps are produced by higher languages constructs like switches, gotos, function pointers (especially as virtual tables in object-oriented languages). They are computed jumps because the code part to be continued with, depends on input data that will be only resolved at runtime.
For example a switch jumps into any of its alternative cases depending on the value of the variable to be tested. Function pointers can be passed around and reassigned and called only much later. Especially object oriented languages use a mechanism similar to function pointers—the virtual methods table—to dispatch function calls depending on the type of the object at runtime. Although most languages provide only a static goto that declares its target statically in the source code, some allow to compute the target of a goto a runtime, e.g. the GCC language extension for C [34, p. 12] (see example in listing 2.1). Other languages provide constructs like continuations that are more powerful and flexible[1] when it is about changing the flow of control. Similar features can be implemented in C with the help of *setjmp/longjmp* or exist as language extensions for Java [29].
Because the above mentioned constructs cannot be resolved precisely in all cases a safe approximation has to be used when constructing the ICFG of a program.

[1]continuations are more powerful because they are first class citizens of the language and can be passed around to other functions. This also implies that continuations carry the context of their execution with them (saved values of the variables at that program point) whereas gotos are just plain jumps to an address in the code

```
void f(int i) {
  void *jmptable[3] = {&&label1, &&label2, &&label3};

  int index = (i + 5) % 3;
  void *target = jmptable[index];

  goto *target;

  label1:
    // do something ...
  label2:
    // do something else ...
  label3:
    // or do this ...
    return;
}
```

Listing 2.1: Targets of a goto can be computed at runtime when using the GCC

```
#include <stdio.h>

typedef int (*funptr)(void);

int main() {
  funptr p;
  int input;
  printf("Type the address that you want to jump to: ");
  scanf("%d", &input);
  p = (funptr) input;
  p();
}
```

Listing 2.2: The user can input a jump target address directly

2.2 Overapproximation of the CFG

Generally the path taken through a program depends on the input data that is coming from the user or other programs. On the machine-level this indirections are mostly expressed by a jump to different targets to continue execution at. This can be a static jump where the target is known and hard-coded in the program text or a jump whose target is computed at runtime if the program continuation depends on some input.

In some cases for jumps that depend on input data it is not possible to ever know beforehand where they will go, as the C example code in listing 2.2 demonstrates. To construct the ICFG for this example we need to know where the procedure call `p()` goes to and that we can only know when the program is run and the user types in a value for the address. Nevertheless each time the program is run the input may change, so the ICFG would change as well with each execution. Even worse, because the amount of different input values can be infinite

```
void f(int i) {
  if (i % 5 == 0)
    exit(0);
  else
    return;
}
```

Listing 2.3: In some cases the function will not return to its caller

there would exist infinitely many ICFGs, something that is infeasible to be examined.
What is needed is an ICFG that describes all this potential program runs. As example 2.2 above shows it is not always possible to know beforehand about all the possible runs, so an approximation is needed. The approximation should be chosen conservatively in that it should never miss a possible target but can contain a target that in a concrete program execution would never be jumped to. More precisely overapproximating the ICFG will still maintain theorem 1.9 that the paths through the ICFG are a superset of all the possible program traces.
Using this strategy for the reconstruction yields ICFGs that when used for analyses may produce false negatives but do not miss a path through in a concrete program run. So stating something about the program runs could be wrong if not all possible runs were taken into account.

Additional edges have to be inserted to show the alternative control flow paths. As seen in section 1.2 branches already perform an overapproximation in the ICFG because they model both possible control paths of the program. Which of the two alternatives is taken for a concrete program execution depends on program data and can change with each program run.
Similar alternative paths are opened up by computed jumps because the target can change on each program run. The idea is to split up an edge for a computed jump into multiple edges that connect to all the possible computed targets. The more precise the set of potential targets is, the closer the overapproximated ICFG is to all possible program executions.

Part III

Analyses

3 Analyzed program class

Analyzing machine code is pretty difficult because it does not exhibit the constructs and structures that high-level languages are built of. These constructs exist to allow for a better and simpler reasoning about the semantics of a program. Consequently humans are able to understand the code better and avoid mistakes while programming.
The analyses performed for the CFG reconstruction are working on assembler code but take advantage of the extra knowledge about where the code came from. By knowing the possible constructs in the source language and the requirements given by coding guidelines for a certain application class, it is easier to achieve precise results.
The high-level language used to build up and test the analyses in this thesis was the C language. Thus the handled constructs come from this imperative language and with it the few limitations it imposes to the programmer. The GNU Compiler Collection (GCC) serves as a cross-compiler to translate the examples to PowerPC binaries. Anyway it is only necessary that the code is generated by a compiler that adheres to the system ABI and calling conventions. No user annotations in the code that would provide hints for the analysis are needed, only plain C code. Debug informations are not necessary either, but the symbol table should be present to know about function boundaries and the function names. Inline assembly should be avoided as it could undermine the assumptions about the C language constructs. This also means that malicious code is excluded as it mostly uses assembly tricks to conceal the code semantics. Finally the binaries are expected to be fully linked to not have to deal with stubs for unknown functions.

For the switches reconstruction analysis the code should be compiled with an optimization level of 1 or higher. With this optimizations the compiler does not use the stack for the linear operations that the analysis is interested in, but performs them on registers. For the function calls analysis the optimization level does not matter because this analysis does keep track of the values on the stack.

The hardware and thus the assembler instructions are restricted to the 32-bit PowerPC architecture as described in [13]. The architecture has 32 general purpose registers, 32 floating point registers and several special purpose registers. All the registers are of global scope but depending on the ABI some registers $r_{14} - r_{30}$ are treated like local variables to the procedure [30]. Registers $r_3 - r_{10}$ are used for parameter passing and r_3, r_4 are also used for the return value. If the passed parameters do not fit in the parameter passing registers the caller allocates them on its stack frame, from where they can be accessed by the callee. Some registers have a special purpose by convention: r_0 is used as the accumulator or scratch register, where the intermediary results are saved; r_1 is the stack pointer register and r_{31} the frame pointer register, used to manage the local memory on the stack for each procedure call.
Calls are implemented as jumps to the procedure entry and calls that would violate this rule

are not allowed. The PowerPC does not have a designated instruction for a call but uses jumps that record the return address to a special register. By the calling convention [30] the callee has to save and restore all non-volatile registers, which is done in the prologue and epilogue of the procedure. Volatile registers $r_3 - r_{12}$ that the caller needs to preserve must be saved by the caller before the call. The link register (LR) that contains the return address is a special register that must also be saved by the caller before calling a function. An exception to this rule occurs when compiler perform tail-call optimizations [24] where the return will jump back to the caller's caller. Because this optimization also breaks the assumption that each function call installs a new stack frame it will be disabled for the analyzed code (see [34, p. 109]). Other than that no assumptions are made about the stack or heap management and neither about the recursion depth of function calls or loop bounds.

Certain data used during the analysis, i.e. jump tables, are expected to reside in the read-only section of the program. Thus it is known at compile time that the values there won't change at runtime. Fortunately ELF binaries [7] provide a section for read-only data (the operating system can share this data section among processes of the same binary [21]) and most compilers put the switch jump tables into this section. When the binary is loaded in memory for execution, the read-only property of this section is enforced by the operating system that can flag areas of memory as non-writable (see [17, chapter 6.6] and [4, page 50]).

For the analyses the instruction classes of interest are the ones performing linear operations on registers, loads and stores to memory, compares and jumps. The remaining instructions, like system calls and traps or exceptions, operations on special system registers and non-linear operations are only captured with the side effects they may have to the general purpose registers as described in the specification of each instruction (see [13]). This is modeled as a non-deterministic assignment where the affected registers are seen as containing every possible value.

Although the PowerPC ISA is a RISC architecture it provides a wealth of instructions. For example it contains 9 different instructions for addition (*add, addc, adde, addi* ...), 30 for loading a value from memory and 27 for storing a value to memory. For the analysis these concrete instructions are summarized in a more abstract syntax using three-address notation (at most 3 operands are allowed in each instruction). An addition with immediate instruction `addi r0, r3, -234` is then written as $r_0 = r_3 - 234$ where r_0, r_3 are registers.
The word size on the machine is 4 bytes and both loads from memory and stores to memory can be performed on bytes, half-words, words (the suffix *b, h, w* of the instruction indicates the size—e.g. for the store instruction: *stb, sth, stw*). There are two possible ways to address memory:

based uses the value in a register as the base and a constant as offset—e.g instruction `lwz r0, 24(r3)` has the effect of assigning to the register r_0 the value from memory at the address given by the sum $r_3 + 24$. In the more abstract syntax this is expressed as $r_0 = Mem[r_3 + 24]$ with $Mem[x]$ denoting a memory access at address x.

indexed uses the value in a register as the base and the value in another register as the offset—e.g instruction `lwzx r0, r11, r9` has the effect of assigning to the register r_0 the value from memory at the address given by the sum $r_{11} + r_9$. From now on written as $r_0 = Mem[r_{11} + r_9]$.

Absolute addressing $r_0 = Mem[c]$ with $c \in \mathbb{Z}$ being a constant value, are performed in PowerPC by using one of the above methods and setting the values in the registers accordingly. For based addressing this means loading the base register with the value 0 and for indexed register the base register is loaded with 0 and the index register with the constant value. The architecture provides a shortcut for this absolute addressing case. When the base register is r_0 the memory unit will use the value 0 for the base instead of the value in the register. This way the additional instruction to load a register with the value 0 can be saved.

The machine and instruction model explained so far, is the basis for the analyses that will be described in detail in the next chapters. From the instruction classes mentioned above each analysis will handle only the subset that is needed to compute the precise results.

Assumptions about the code

If it is not possible to determine all the targets of a computed jump correctly, then the worst case is assumed, that the jump can go anywhere. Hence there will be jump edges inserted to all existing nodes in the program. Such a CFG is not very useful for analyses anymore and is something that should be avoided by using better analyses or making assumptions about the program's behavior, that narrow down the set of possible targets.

Memory protection

Even this worst case relies on the general assumption that the operating system has a mechanism for memory protection and is capable of separating program address spaces from each other. This in term creates the constraint that a program cannot call code from outside its address space without being killed because of the memory protection violation. In example listing 2.2 it helps to restrict the possible calls to all the addresses in the binary and not have to deal with the environment and possible side effects caused by that.

Language structures

Other assumptions can easily be made when knowing that the code is compiled from certain higher-level languages that carry some restrictions on the generated code. One such assumption is that a computed jump for a switch would never jump outside of the procedure itself because that is not possible to do in higher level languages with a switch construct.

Compilers

Nevertheless assumptions about the code's well formedness can be too optimistic when some of the libraries used in the program are written in low-level languages. Another problem can arise from aggressive optimizations performed by the compiler that may produce assembler code undermining the assumptions. Sometimes the assumptions go so far as to apply pattern matching for certain compilers to recognize known constructs from high-level languages [33, 16].

ABI

A weaker but still very useful assumption is to expect the code to comply to an ABI (application binary interface) imposed by the operating system [27] or by the processor architecture [30]. The good part is that all the compilers must obey the ABI of the platform they are producing code for. Hence assumptions on ABI compliance are indifferent of the compiler that was used and the language the code was written in. Additionally even hand crafted assembler code that wants to inter-operate with other programs and the operating system has to conform with the ABI.

For the code that is analyzed in this thesis the following assumptions are made: memory protection provided by the operating system, knowledge about the existing language constructs and ABI compliance of the code.

4 Preliminary ICFG

To reconstruct the ICFG a top-down approach is used (see [33] for a bottom-up vs.top-down approach comparison). In the first step a preliminary ICFG is build up from the binary and the second step uses further analyses to refine it. The preliminary ICFG is a set of ICFG fragments, that can be build up by resolving only static jumps and branches. It is a top-down approach because it uses an already existing but not necessarily complete ICFG to work on, instead of trying to construct the final ICFG in the first place when parsing the assembler instructions.

4.1 Extracting the preliminary ICFG

The disassembly text provided by the disassembler is parsed linearly to recognize PowerPC instructions by their opcode as specified in [13, 23]. At the same time the instructions are grouped together to procedure bodies according to the named procedure block they belong to in the text (see line 1 in example listing 4.1). This step is relying on information coming from the compiler—i.e. the symbol table. At this stage each instruction is seen as a standard fall-through edge between two points in the ICFG. The following post-processing steps handle all the jump and branch targets that can be revealed by parsing the instruction code, i.e. static jumps which contain the jump target address in the instruction code as operand.
Step one adds additional branch and local jump edges and removes incorrect fall-through edges. Unconditional jumps (see line 10) will be reconnected to their target and the fall-through edge removed. Conditional jumps (see line 7) need to replace the fall-through edge with an edge for the *false*-branch and add a new edge for the *true*-branch.
Step two inserts static function calls by looking at instructions known to be used as calls. On PowerPC calls are implemented with jump instructions that save the return address to the link register *LR* (see line 13). Thus static jumps that match this pattern are inserted as jump edges to the procedure entry they call. Calls that do not jump to the procedure entry are not allowed (see chapter 3). The return instruction on PowerPC is implemented as an indirect jump (see line 20) to the link register. For each jump that matches this pattern a jump edge to all the callers of the procedure is inserted. This is an overappoximation as it assumes that each call also returns to the caller. A later analysis (see chapter 6) will find potential violations to this assumption.

```
268436388 <readFile>:
268436388: stwu    r1,-40(r1)
268436392: mflr    r0
...
268436420: lwz     r0,24(r31)
268436424: cmpwi   cr7,r0,4
268436428: bgt-    cr7,268436444 <readFile+56>
268436432: li      r0,3
268436436: stw     r0,8(r31)
268436440: b       268436452 <readFile+64>
268436444: li      r0,4
268436448: stw     r0,8(r31)
268436452: bl      268436504 <getFileName>
268436456: mr      r9,r3
...
268436484: lwz     r0,4(r11)
268436488: mtlr    r0
268436492: lwz     r31,-4(r11)
268436496: mr      r1,r11
268436500: blr
```

Listing 4.1: Example PowerPC disassembly

4.2 Problems with the top-down approach

Relying on the information provided by the compiler can result in missing function boundaries or instructions wrongly grouped to a procedure as the information may be missing or the procedure are not clearly separated depending on the optimizations that are applied.
A sequential disassembler (e.g. *objdump* [14]) on the other side has the shortcomings that it does not know about the semantics of an instruction and thus cannot distinguish if a group of bytes is simple data or an instruction. These type of linear sweep disassemblers perform a pattern matching on the bytes and everything that does not fall into the known pattern for instructions is treated as data. An assumption that can produce more instructions than really exist in the program if a data word by coincidence matches the patterns. Recursive disassemblers (e.g. *IDA Pro* [16]) try to alleviate this problem by doing a recursive parsing. They follow static jumps and fall-through edges to discover code parts. Everything that is not reachable by applying this technique starting with the first instruction is data. Hence data cannot be misinterpreted as instructions but the opposite is true. Because computed jumps cannot be resolved at parsing time, code that is only reachable through such a jump will be treated as data. See [26] for an in depth comparison of the two approaches.

5 Switches

Switches are the high-level construct for the n-ary branch. Where the if-else construct allows only for two alternatives the switch is able to handle n possible values for a tested variable. But because of the way switches are implemented in imperative languages, they are restricted to use only integer variables and thus can only test for numeric values as the possible alternatives. As can be seen in example 5.1 a switch is useful wherever different code parts should be executed for different values of a variable. The variable is tested at runtime and if it is equal to one of the cases this code will be executed.
Break instructions in a switch case act as a shortcut exit. The code path jumps to the end of the switch. In absence of a break the next case will be executed (in example 5.3 case 2 will be executed after case 1 ended). This is the standard fall-through (see [11] for an interesting usage of the fall-through feature to perform loop unrolling). In terms of control flow it means that some case branches can be reached by fall-through and by a direct jump. The GCC provides an extension to the C language that allows to not only test for an exact value but also for a range of values (e.g. *1..5*). However this is only a nicer way to group together case values but can also be achieved by using the fall-through feature that switches provide.
Switches can also contain a default case (see example listing 5.4) that will be executed whenever there is no explicit case declared to handle a value of the variable. Thus the default case captures all the not declared case values. If a switch does not have a default case declared (as in listing 5.1) then the end of the switch or the instruction directly following the switch will be jumped to instead, whenever the tested value is not explicitly handled by one of the cases.

```
int day = getDayOfTheWeek();

switch(day) {
  case 1: printf("Monday"); break;
  case 2: printf("Tuesday"); break;
  case 3: printf("Wednesday"); break;
  ...
  case 7: printf("Sunday"); break;
}
```

Listing 5.1: Example usage for a switch

```
268437076: stwu    r1,-8(r1)
268437080: mflr    r0
268437084: stw     r0,12(r1)
// int day = getDayOfTheWeek();
268437088: bl      268436112 <getDayOfTheWeek>
// switch(day) {
268437092: cmplwi  cr7,r3,7
268437096: bgt-    cr7,268437264 <switch_weekDay+188>
268437100: lis     r9,4097
268437104: addi    r9,r9,-26452
268437108: rlwinm  r0,r3,2,0,29
268437112: lwzx    r0,r9,r0
268437116: add     r0,r0,r9
268437120: mtctr   r0
268437124: bctr
//  case 1: printf("Monday"); break;
268437128: lis     r3,4097
268437132: addi    r3,r3,-26420
268437136: crclr   4*cr1+eq
268437140: bl      268437768 <printf>
268437144: b       268437264 <switch_weekDay+188>
//  case 2: printf("Tuesday"); break;
268437148: lis     r3,4097
268437152: addi    r3,r3,-26412
268437156: crclr   4*cr1+eq
268437160: bl      268437768 <printf>
268437164: b       268437264 <switch_weekDay+188>
//  case 3: printf("Wednesday"); break;
268437168: lis     r3,4097
268437172: addi    r3,r3,-26404
268437176: crclr   4*cr1+eq
268437180: bl      268437768 <printf>
268437184: b       268437264 <switch_weekDay+188>
...
...
...
//  case 7: printf("Sunday"); break;
268437248: lis     r3,4097
268437252: addi    r3,r3,-26360
268437256: crclr   4*cr1+eq
268437260: bl      268437768 <printf>
268437264: lwz     r0,12(r1)
268437268: mtlr    r0
268437272: addi    r1,r1,8
268437276: blr
```

Listing 5.2: Disassembled instructions for switch example 5.1 (jump table)

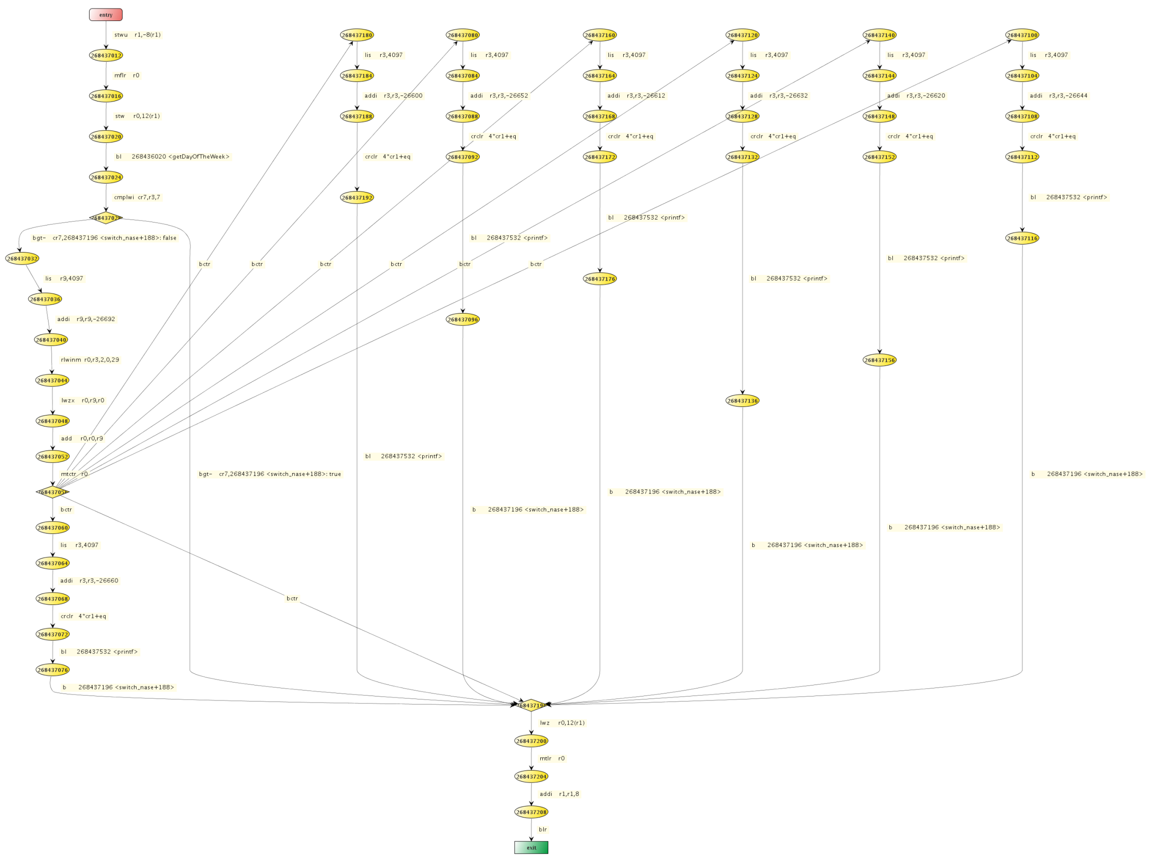

Figure 5.1: CFG for switch in example listing 5.1 (computed jump to cases)

```
switch() {
  case 1: ...
  case 2: ... break;
  case 3: ...
}
```

Listing 5.3: Example switch with fall-through case 1

5.1 Compilation of switches

Switches are compiled to assembler code either as a series of nested comparisons (analogous to an if-else cascade) or as a jump table containing absolute or relative jump targets for each case (compare the CFGs in figure 5.2 and figure 5.1). Sometimes these two components can even be mixed to a hybrid solution. Each compiler seems to have its own rules and heuristics for when to change the translation method but all of them depend on the amount of switch cases and the values of the cases. The following criteria are used to decide about the compilation strategy (see also [6]):

- Switches with less than 6 cases will always be compiled to if-else comparisons. This threshold arises from the trade-off between the space for the jump table plus the size of the code needed to compute the jump index and the code for the sequence of if-else queries. Also for less than three cases the GCC tries to avoid some of the comparisons by performing bitwise operations.
- For switches with more than 6 cases it depends on the values of the case labels. If the values of the cases are not too far apart and thus a table indexed by these values will be contiguous and compact, the switch is translated to a jump table of code addresses and instructions that compute the index into this table from the tested value (see figure 5.1). The threshold when the gaps between the values are becoming too big to be feasible depends again on hard-coded compiler logic. For example the GCC emits a dispatch table as long as the range for the values is not 10 times bigger than the amount of cases and thus the amount of comparisons that would be needed.
- Otherwise the compiler falls back to using a cascade of comparisons. The GCC performs here at least an optimization in that it sorts the cases in a balanced binary tree. The comparisons are then done on the branching of this tree and thus the linear search (worst-case comparisons needed $O(n)$) is replaced by a binary search (worst-case comparisons needed $O(\log n)$), resulting in a better runtime.
- In a hybrid approach some compilers may cluster value ranges together and use them for jump tables and handle the remaining values with if-else comparisons—note that the GCC does not use this technique.

5.2 Jump Tables

Of the two different types of code that will be produced by the compiler for switch constructs, the sequence of if-else comparisons is benign for CFG reconstruction. The comparisons are implemented with static branches (see figure 5.2) that lead to the code for each of the cases. Thus every case is reached on one of these paths. Because these branches are static jumps the CFG can be reconstructed by only parsing the instructions.
The second component used to translate switches—i.e. the jump tables (see figure 5.1)—requires more work. The difficulty is that the jump target address is only computed at runtime and it depends on the value of the variable that is tested, which is mostly only known at runtime. Hence it is necessary to find out what are all the possible paths that can

```
int error = readFile(fileHandle);

switch(error) {
  case  0: printf("File reading successful"); break;
  case -2: printf("File not found"); break;
  case -5: printf("File not accessible"); break;
  case -7: printf("I/O error"); break;
  default: printf("Some unknown error occurred"); break;
}
```

Listing 5.4: The default case catches all the other values

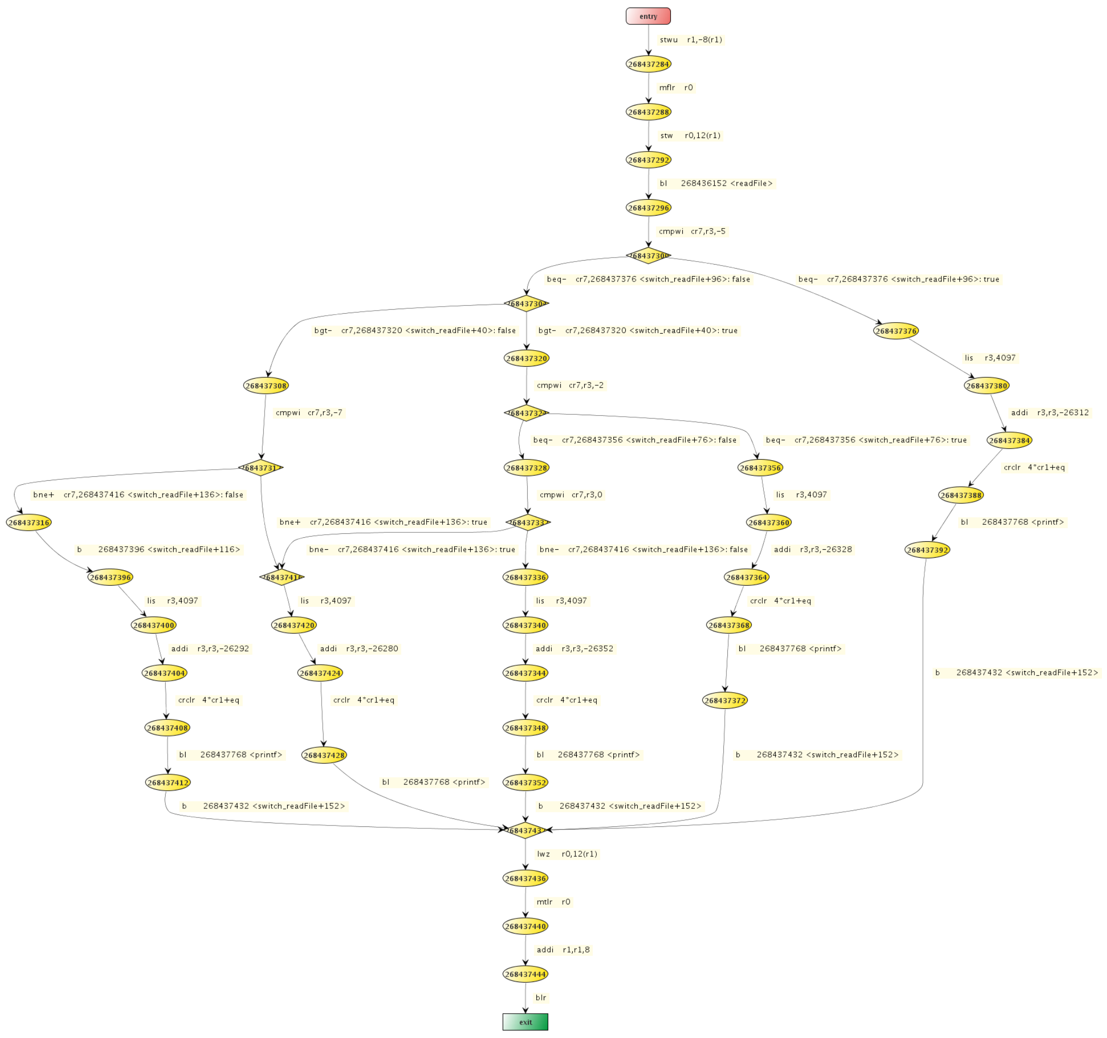

Figure 5.2: CFG for switch in example listing 5.4 (cascade of if-else tests)

```
268437280: stwu    r1,-8(r1)
268437284: mflr    r0
268437288: stw     r0,12(r1)
// int error = readFile(fileHandle);
268437292: bl      268436152 <readFile>
// switch(error) {
268437296: cmpwi   cr7,r3,-5
268437300: beq-    cr7,268437376 <switch_readFile+96>
268437304: bgt-    cr7,268437320 <switch_readFile+40>
268437308: cmpwi   cr7,r3,-7
268437312: bne+    cr7,268437416 <switch_readFile+136>
268437316: b       268437396 <switch_readFile+116>
268437320: cmpwi   cr7,r3,-2
268437324: beq-    cr7,268437356 <switch_readFile+76>
268437328: cmpwi   cr7,r3,0
268437332: bne-    cr7,268437416 <switch_readFile+136>
//  case  0: printf("File reading successful"); break;
268437336: lis     r3,4097
268437340: addi    r3,r3,-26352
268437344: crclr   4*cr1+eq
268437348: bl      268437768 <printf>
268437352: b       268437432 <switch_readFile+152>
//  case -2: printf("File not found"); break;
268437356: lis     r3,4097
268437360: addi    r3,r3,-26328
268437364: crclr   4*cr1+eq
268437368: bl      268437768 <printf>
268437372: b       268437432 <switch_readFile+152>
//  case -5: printf("File not accessible"); break;
268437376: lis     r3,4097
268437380: addi    r3,r3,-26312
268437384: crclr   4*cr1+eq
268437388: bl      268437768 <printf>
268437392: b       268437432 <switch_readFile+152>
//  case -7: printf("I/O error"); break;
268437396: lis     r3,4097
268437400: addi    r3,r3,-26292
268437404: crclr   4*cr1+eq
268437408: bl      268437768 <printf>
268437412: b       268437432 <switch_readFile+152>
//  default: printf("Some unknown error occurred"); break;
268437416: lis     r3,4097
268437420: addi    r3,r3,-26280
268437424: crclr   4*cr1+eq
268437428: bl      268437768 <printf>
268437432: lwz     r0,12(r1)
268437436: mtlr    r0
268437440: addi    r1,r1,8
268437444: blr
```

Listing 5.5: Disassembled instructions for switch example 5.4 (series of comparisons)

be taken as a result of the computations performed on the tested value, for every possible value that may occur.
Fortunately the switch construct itself handles all the possible values that may occur by providing the default case, which catches all the ones not handled explicitly. The compiler translates the catch-all feature by using two mechanisms. First it restricts the possible range of the tested variable by doing a comparison. The range is the interval containing all the numbers from the lowest case value to the highest one (the values for the cases are sorted by the compiler in ascending order). If the variable is not inside the range of the case values then the path to the code handling the default case is taken. And as a second measure the compiler inserts the address to the code for the default case in the jump table for all the case values that are not explicitly defined, viz. the case values that lie in the value's range but are not stated in the code. In example listing 5.4 this would be the numbers -1, -3, -4, and -6. Thus all the possible values for the tested variable will be handled correctly.
Now that the variable's value is in the cases' range interval it serves as an index for the jump table. The compiler always builds the table as big as the switch range, even if some values are not directly handled by case labels (for these missing cases it inserts jumps to the default case). By doing this the value of the tested variable can be used directly as an offset into the jump table. An additional step is performed before that though: the range bounds and tested variable need to be adjusted.

Adjusting the range bounds

To not waste space for indices in the jump table with entries below the lower bound of the range the value of the variable is adjusted by subtracting the lower bound from it. This also helps to save a comparison with the lower bound on architectures that have an instruction to compare unsigned numbers (e.g. the PowerPC *cmpl* instruction—compare logical) because the lower bound is then implicitly set to zero. So on PowerPC the interval describing the range of values for the switch is first translated to start with zero and then the switch value which was adjusted by subtraction is compared against the upper bound. If the value is lower or equal, then it is in range and can be used as index for the jump table, otherwise the default case is taken. This unsigned comparison works even if the tested value would be a signed number. As negative numbers are represented in two's complement when interpreting the value as unsigned, it results in a very big value (bigger than the highest available unsigned number to use in the comparison) and thus certainly not lower than the upper bound.
Analogously to adjusting an interval of positive numbers to start with zero by subtraction, an interval with negative values is adjusted by adding a number to it (corresponds to a subtraction with a negative number, which is in term the value of the lower bound).

Computation of the jump address

Next, after all the adjustments have been done the tested value can be used to compute an index into the jump table. The only thing to do is to multiply the value by the machine word size. This is the size of a cell in memory. On the PowerPC this means to multiply by 4 which is done by shifting the value two bits to the left.

After multiplication the index is added to the base address, which is the start address of the jump table in memory. So the index and base build up the final address for the memory cell that contains the target address value for the jump. This value can either be an absolute address or an offset relative to the beginning of the table. In the case of GCC (see listing 5.10) it is an offset and needs to be added to the start address of the jump table. Finally the whole process yields an absolute address to jump to, which jumps to the code part for the appropriate case in the switch. The address for the case $caseJumpAddress$ is calculated with this formula:

$$caseJumpAddress = Mem[testVariable * 4 + tableBase] + tableBase$$

where $testVariable$ contains the value that is used to decide which case has to be taken, and $tableBase$ denotes the start address for the jump table. The program's processing steps for the above formula are:

Algorithm 1 Steps to calculate the jump address for a switch

```
 1: $testVariable \leftarrow testVariable - lowerRangeBound$ // adjust values to start range at zero
 2: $upperRangeBound \leftarrow upperRangeBound - lowerRangeBound$

 3: **if** $testVariable \leq upperRangeBound$ **then**          // is the value in switch range
 4:     $tableBase \leftarrow$ table begin address in memory
 5:     $index \leftarrow testVariable * 4$
 6:     $caseJumpOffset \leftarrow$ memory content at address: $(tableBase + index)$
 7:     $caseJumpAddress \leftarrow tableBase + caseJumpOffset$
 8:     jump to switch case located at $caseJumpAddress$
 9: **else**
10:     jump to default case
11: **end if**
```

5.3 Analysis stages

When the switch is translated by the compiler to a series of comparisons, nothing needs to be done as the control flow is already precise but when a jump table is used, the address computation has to be reproduced by a static analysis. As already described in section 2.2 to reconstruct the overapproximated control flow for a switch, when the value of the tested variable is not known at compile time, the jump targets to all the cases have to be resolved.

The analysis is based on the principle of abstract interpretation [9, 10] to infer bounds for the accesses to the switch jump table that happen at runtime. In the *Table Indices* analysis stage an interval delimiting the possible values for the switch variable is computed. This interval is later used in the *Targets Set* analysis stage to read a set of offsets from a field in memory that starts with the location of the jump table. This set of offsets is then used to compute the absolute jump targets. Finally in the *Case Jumps* analysis stage, the set of jump targets is used to insert the edges that jump to all the inferred switch cases. These three stages are repeated as a unit, until the CFG stabilizes and there are no more edges to insert.

5.3.1 Inferring the jump table indices

In this *Table Indices* analysis the location of the jump table and the interval for the jump table indices has to be computed. Recalling the computation steps in algorithm 1 the location of the jump table can be obtained from the value of $tableBase$ and the size of the jump table can be identified from the range comparison in line 3 of algorithm 1. The value of $upperRangeBound$, also known at compile time, is exactly the table size. To be able to read useful values from the jump table one has to keep the size of the memory cell in mind (expressed in line 5 of algorithm 1). The reads must happen only at addresses that are a multiple of this memory cell size, which on most machines is equal to the machine's word size.
A way to handle these things is to use a strided interval (see [25, 2]) to represent the possible memory addresses. A strided interval is an interval extended by an additional value—the stride—that describes the distance between each member value in the interval. All the values in the interval are then multiples of this stride starting with the lower bound of the interval. In more formal terms a strided interval can be defined as:

Definition 5.1 (Strided Interval).
A strided interval $I = s\%[l, u]$ with the lower bound l and upper bound u and s the value of the stride contains all the numbers $I = \{i \mid i = l + s * n, n \in \mathbb{N}_0 \wedge i \leq u\}$.

For $s = 1$ the strided interval becomes a common interval $[l, u] = 1\%[l, u]$. And for $l = u$ it becomes a single constant written as $l = 0\%[l, l]$.
With the above definition a jump table in PowerPC assembly can be described as:

$$I_{jumpTable} = 4\%[tableBase, tableBase + upperRangeBound]$$

On PowerPC all arithmetic operations need registers as operands. Values in memory must first be loaded into a register and the result will be written back. Hence the computations in algorithm 1 are performed on registers. Even the jump uses the value in a register. Thus to model all the computation steps it is necessary to keep track of the values in registers (as described in section 3 this analysis relies on compiler optimizations that remove unnecessary reads and writes to memory).
As the first step, the jump table indices have to be inferred. This is done using the domain D^I that maps registers at any program point to their corresponding values, represented by strided intervals:

Definition 5.2 (Interval Domain).
$\mathbb{D}^I : (R \rightarrow I_\perp)$

with

$$R = \{r_i \mid i \in [0, 31]\}$$

the set of all general purpose integer registers of the PowerPC and

$$I = \{(s\%[l, u]) \mid s \in \mathbb{N}_0, l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}$$

For the analysis to perform the fix-point computations, a complete lattice is needed. For $\mathbb{D}^I$ to be a complete lattice, a partial order $\sqsubseteq_{\mathbb{D}}$ between the elements of the domain must exist and a least $\perp$ and greatest $\top$ element are needed. Therefore the values for the strided intervals are extended by adding a least $\perp$ and a greatest $\top$ element where $\perp$ means that the register has not been initialized and $\top$ that the register contains every existing value, i.e. the interval $1\%[-\infty, +\infty]$. Two domain elements $D_a^I, D_b^I \in \mathbb{D}^I$ can be compared by performing a point-wise comparison of the strided intervals associated with each register:

$$D_a^I \sqsubseteq_{D^I} D_b^I \Leftrightarrow (D_a^I(r_i) \sqsubseteq D_b^I(r_i) \mid i \in [0, 31])$$

Here $D^I(r_i)$ denotes the strided interval for register r_i in domain $\mathbb{D}^I$.
The strided intervals are compared like common intervals with the additional constraints for the stride:

$$I_a \sqsubseteq_{\mathbb{D}^I} I_b \Leftrightarrow ((l_a \geq l_b) \wedge (u_a \leq u_b) \wedge (s_b | s_a) \wedge (l_a = l_b + n * s_b, n \in \mathbb{N}_0))$$

Informally $I_a \sqsubseteq I_b$ means that I_b contains all the values in I_a.
Additionally a least upper bound operator $\sqcup_{\mathbb{D}^I}$ is defined in analogy to the above as a point-wise application on the register values:

$$D_a^I \sqcup_{\mathbb{D}^I} D_b^I = \{D_a^I(r_i) \sqcup D_b^I(r_i) \mid i \in [0, 31]\}$$

where the merge of two strided intervals is the union of the intervals [25] with the stride value adjusted accordingly:

$$I_c = I_a \sqcup I_b = s\%[min(l_a, l_b), max(u_a, u_b)] \text{ with } s = \begin{cases} gcd(s_a, s_b) & \text{if } I_a \sqsubseteq I_c \wedge I_b \sqsubseteq I_c \\ 1 & \text{otherwise} \end{cases}$$

In the following the analysis is restricted to those instructions that can be abstracted by affine relations. The operations on the strided intervals are performed analogous to common intervals but they also adjust the stride value [25]. Let c be a constant $c \in \mathbb{N}_0$ and I_a a strided interval, then addition and multiplication with a constant can be expressed as follows:

$$I_a * c = (s * c)\%[l * c, u * c]$$
$$I_a + c = s\%[l + c, u + c]$$
$$I_a - c = s\%[l - c, u - c]$$

The PowerPC instructions alter the values of the registers and memory locations when executed. With abstract interpretation the same effects have to be performed on the domain $\mathbb{D}^I$ by using the abstract semantics of the instructions specified for the analysis domain, so called transfer functions. Not every instruction in the PowerPC instruction set is of interest for this analysis. The purpose of the analysis is to interpret the operations on registers, so only those instructions that alter a register value are of interest. Other instructions that have no effect on the value of a register (e.g. the write to memory instructions) will not change the register and are thus omitted here. The important class of instructions are the ones that perform arithmetic operations with registers, the ones that write to a register and the conditional branching upon a comparison. Using the domain defined in 5.2 for the analysis leads to the following constraint system:

$$\begin{array}{llcll}
C1_I: & D^I[entry_f] & \sqsupseteq & \top & f \in procedures \\
C2_I: & D^I[v] & \sqsupseteq & D^I[u] \oplus \{r_i \rightarrow \top\} & (u, r_i = ?, v) \\
C3_I: & D^I[v] & \sqsupseteq & D^I[u] \oplus \{r_i \rightarrow 0\%[c, c]\} & (u, r_i = c, v) \\
C4_I: & D^I[v] & \sqsupseteq & D^I[u] \oplus \{r_i \rightarrow D^I[u](r_j)\} & (u, r_i = r_j, v) \\
C5_I: & D^I[v] & \sqsupseteq & D^I[u] \oplus \{r_i \rightarrow [\![term]\!]^\# D^I[u]\} & (u, r_i = term, v) \\
C6_I: & D^I[v] & \sqsupseteq & D^I[u] \oplus \{r_i \rightarrow [\![r_i \diamond c]\!]^\# D^I[u]\} & (u, r_i \diamond c, v) \text{ with } \diamond \in \{>, \geq, <, \leq\} \\
C7_I: & D^I[v] & \sqsupseteq & D^I[u] \oplus \{r_i \rightarrow [\![r_i \diamond r_j]\!]^\# D^I[u]\} & (u, r_i \diamond r_j, v) \text{ with } \diamond \in \{>, \geq, <, \leq\}
\end{array}$$

Table 5.1: Constraint system D^I for the strided intervals domain $\mathbb{D}^I$

Because the analysis is intraprocedural, for all procedures f in the program at procedure start $entry_f$ the values for all the registers are unknown and thus can contain any value ($C1_I$). When loading a value from memory ($C2_I$) the register is set to the top element, possibly containing every value, as the values in memory are not kept track of in this analysis. The same effect of a non-deterministic assignment happens with instructions that cannot be modeled precisely, e.g. non-linear operations like bit masks. When considering assignments with a constant ($C3_I$) or another register ($C4_I$) the value in the register will be updated accordingly. Linear terms ($C5_I$) of the form $term \in \{r_i + c, r_i - c, r_i * c\}$ on the right-hand side are first evaluated according to the strided interval semantics (denoted by the semantic evaluation brackets $[\![\ldots]\!]^\#$) with regard to the current value $D^I[u]$ and then the resulting value is assigned to the register on the left-hand side. The term is evaluated by instantiating each term variable with the value it has in $D^I[u]$ and using the linear operations defined on strided intervals to combine these values. Branches that are guarded by a condition ($C6_I$, $C7_I$) are evaluated and the interval for the register on the left hand side is updated according to the guard if it would yield a more precise interval (see 5.1). Otherwise the value is left unchanged.

Example 5.3.
Let the *term* in $C5_I$ be a multiplication of the form $(u, r_0 = r_3 * 4, v)$ and $D^I[u](r_3) = 1\%[0, 5]$ the strided interval value of register r_3 at program point u. Then after passing the above edge, register r_0 would be assigned the value $4\%[0, 20]$ at program point v.
Now let's consider a conditional $C6_I$ edge $(u, r_3 \leq 7, v)$ with $D^I[u](r_3) = 2\%[4, 20]$. After passing the *true*-edge for this branch, the strided interval for register r_3 is set to $2\%[4, 6]$. On passing the *false*-edge for the above condition the value for this register will be $D^I[v](r_3) = 2\%[8, 20]$. Let's consider the same conditional edge but $D^I[u](r_3) = 0\%[4, 4]$. After passing the *true*-edge the value will be the same as it satisfies the guard and is more precise than the interval inferred from the guard. On passing the *false*-edge the value is $\bot$ as $D^I[v](r_3)$ does not satisfy the guard's condition. It means that this code path cannot be taken and thus the register is not initialized (see [35]).

The above transfer functions are monotonous functions. With the least upper bound operator $\sqcup_{D^I}$ performing a union of the intervals, the values in the domain D^I can only grow with every iteration of the fix-point computation. Termination of the iterative computation is achieved when the values at each program point reach a fixed point. And because of the monotonous transfer functions this will happen if there are no infinite ascending chains. These infinite chains can be avoided by using widening [10, 8]. Widening will allow updates to an interval only finitely many times before resulting in $\top$. Because the interval bounds for the jump table are obtained from guarded branches, with the comparison variable being a constant, the bounds are refined before being used. So widening will not destroy the precision of the results needed for the switches.

To recapitulate, the *Table Indices* analysis computes for every processor register a strided interval that approximates the values in a register that can occur at runtime. In particular it computes a strided interval for the switch variable that is used to decide where to jump.

5.3.2 Retrieval of jump targets

In the *Table Indices* analysis, strided intervals are inferred for every register. Now only the register values that are used for memory read accesses are of interest. In this *Targets Set* analysis the values read by these memory accesses represent jump offsets, which need to be added to a base address, producing absolute jump targets. These will then be used to jump to the switch cases. These two steps—memory access and addition with the base address—can be seen in algorithm 1 in lines 6 and 7.
Because the jump offsets retrieved from the jump table in memory point to arbitrary targets in the code, they cannot be described precisely by a strided interval, but must be treated as a set of distinct values. The analysis thus uses a domain that maps registers at any program point to a finite set of numbers:

Definition 5.4 (Sets Domain)**.**
$\mathbb{D}^S : (R \rightarrow S^\top)$

with R being the set of all PowerPC general purpose registers again and

$$S \in 2^{\mathbb{Z}} \wedge |S| \leq n \in \mathbb{N}$$

being a set of n integer numbers for a particular value n.
The least element is the empty set $\bot = \{\}$ and the greatest element is the set containing all numbers $\top = \{z \mid z \in \mathbb{Z}\}$. Note that $\top$ is not bounded in size by n and thus is always a superset of any other bounded set.
Let c be a constant $c \in \mathbb{Z}$ and S_a, S_b two sets of integer numbers, then the affine operations $\boxtimes \in \{+, -, *, /\}$ are defined as follows:

$$S \boxtimes c = \{s \boxtimes c \mid s \in S\}$$

$$S_c = S_a \boxtimes S_b = \begin{cases} \{s_a \boxtimes s_b \mid (s_a \in S_a) \wedge (s_b \in S_b)\} & \text{if } |S_c| \leq n \\ \top & \text{otherwise} \end{cases}$$

To form a complete lattice the ordering of two elements D^S_a and D^S_a is performed point-wise using the sets associated with the corresponding registers:

$$D^S_a \sqsubseteq_{\mathbb{D}^S} D^S_b \Leftrightarrow (D^S_a(r_i) \sqsubseteq D^S_b(r_i) \mid i \in [0,31])$$

where

$$S_a \sqsubseteq S_b \Leftrightarrow S_a \subseteq S_b$$

is defined by the common subset relation. In analogy the least upper bound operator is defined as the point-wise application to the sets associated with the corresponding registers:

$$D^S_a \sqcup_{\mathbb{D}^S} D^S_b = \{D^S_a(r_i) \sqcup D^S_b(r_i) \mid i \in [0,31]\}$$

where the merge of two sets S_a and S_b is their union:

$$S_c = S_a \sqcup S_b = \begin{cases} S_a \cup S_b & \text{if } |S_c| \leq n \\ \top & \text{otherwise} \end{cases}$$

For this analysis, again those PowerPC instructions that correspond to linear operations are of interest as the set of jump offsets needs to be transformed to absolute addresses (see line 7 of algorithm 1). Additionally, instructions that read from memory are taken into account because the addresses must first be read from the jump table, using the results from the *Table Indices* analysis. Any other operation on registers or memory will be ignored or will have the effect of destroying the values associated with a register. This leads to the following constraint system:

$C1_S$:	$D^S[entry_f]$	$\sqsupseteq$	$\top$	$f \in procedures$
$C2_S$:	$D^S[v]$	$\sqsupseteq$	$D^S[u] \oplus \{r_i \to \top\}$	$(u, r_i = ?, v)$
$C3_S$:	$D^S[v]$	$\sqsupseteq$	$D^S[u] \oplus \{r_i \to \{c\}\}$	$(u, r_i = c, v)$
$C4_S$:	$D^S[v]$	$\sqsupseteq$	$D^S[u] \oplus \{r_i \to D^S[u](r_j)\}$	$(u, r_i = r_j, v)$
$C5_S$:	$D^S[v]$	$\sqsupseteq$	$D^S[u] \oplus \{r_i \to [\![term]\!]^\# D^S[u]\}$	$(u, r_i = term, v)$
$C6_S$:	$D^S[v]$	$\sqsupseteq$	$D^S[u] \oplus \{r_i \to Mem[D^I[u](r_j) + c]\}$	$(u, r_i = Mem[r_j + c], v)$

Table 5.2: Constraint system D^S for the sets domain $\mathbb{D}^S$

The analysis is intraprocedural and thus the initial value at procedure start $entry_f$ for all the registers is set to unknown ($C1_S$). When the linear operation cannot be modeled precisely a non-deterministic assignment is assumed ($C2_S$) that sets the value of the register to unknown. Assignments ($C3_S$, $C4_S$) set the value of the register on the left-hand side, either to a constant or to the value in the register on the right-hand side. Linear terms ($C5_S$) using any combination of the operators defined for D^S, i.e. $term \in \{r_i \boxtimes c, r_i \boxtimes r_j\}$, are first evaluated according to the set semantics with regard to the current value $D^S[u]$ and the resulting value is assigned to the register on the left hand side. Finally if a value is read from memory ($C6_S$) by using based addressing with a register, the strided interval value I associated with that register will be added to the base $D^I[u](r_j) + c$ and will be used to resolve the target addresses to read from. The values read from these addresses will be assigned as a set to the register on the left hand side. Because this analysis does not keep track of writes to memory it can only handle reads from memory sections that are known to be read-only (the memory addresses for the read-only section are extracted from the binary, see section 3) and thus not changeable by write instructions.

Example 5.5.
Let *term* in $C5_S$ be of the form $r_3 + r_4$ where the values of r_3, r_4 at program point u are $D^S[u](r_3) = \{2, 3\}$ and $D^S[u](r_4) = \{1, 2\}$. Consider the assign edge $(u, r_0 = term, v)$; the resulting value for r_0 after the evaluation of the term is $D^S[v](r_0) = \{3, 4, 5\}$. Now let the size of the sets used in D^S be bounded by $n = 2$, then the above edge would produce $D^S[v](r_0) = \top$ as the evaluation of the addition yields more elements than the set is allowed to contain.
Considering the edge $(u, r_0 = Mem[r_0], v)$ with $D^I[u](r_0) = 4\%[1000, 1008]$ and the memory values at addresses: $Mem[1000] \rightarrow 6400$, $Mem[1004] \rightarrow 6120$ and $Mem[1008] \rightarrow 6670$ being part of the read only memory area. After passing the edge register r_0 will contain the set $\{6120, 6400, 6670\}$.

Here again the transfer functions are monotonous as there are only linear operations and the combination of two values is handled by the $\sqcup_{\mathbb{D}^S}$ operator that can only add values to the set of values associated with a register. Also the assignments to a register caused by a read from memory will always result in the same values, because the addresses have been computed in the *Table Indices* analysis and the memory section accessed is read only. Hence the reads will always happen at the same address and the values there cannot change. With the size of the sets being bounded by a finite number n the updates must come to an end and the fix-point iteration terminates.

To recapitulate, this part of the analysis loads all the jump target offsets belonging to a switch, from the associated jump table. Then with these offsets it computes the final jump targets to each case of the switch construct.

5.3.3 Adding the jump edges

With the jump targets computed by the *Targets Set* analysis, this *Case Jumps* analysis can change the CFG by inserting the jump edges to each of the resolved targets.
The changes in the CFG have consequences for the analysis because the inserted edges add new paths through the CFG. First, the new paths could have build up a connection to parts of the CFG that were not reachable before. These formerly unreachable parts need to be analyzed to perform the same reconstruction steps for any additional unresolved switches. Thus iteratively reconstructing the CFG with each new discovered paths.
As a second consequence the parts of the CFG that have already been analyzed, may now be also reachable through these additional new paths. Each path to a program point is also a set of constraints for that program point as defined by the constraint systems of the analysis (see table 5.1 and table 5.1). Thus new edges leading to a program point also add constraints for that program point making it necessary to reevaluate the analysis at these points.
For both aspects of the new paths the analysis has to be repeated to take the updated constraints caused by the changed CFG structure into account. Because this analysis part can only add edges to the CFG, the changed CFG also contains the older one and with it the

new constraints are a superset of the former constraints. A new run of the analysis would then not invalidate the previous results.
The *Table Indices* analysis computes an overapproximation of the interval for the possible

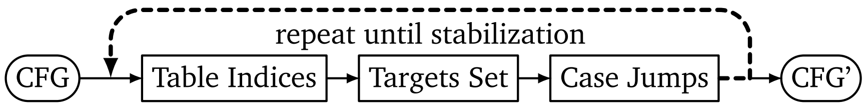

Figure 5.3: The CFG reconstruction process

values of a register, in particular the register containing the index into the jump table. Using these results the *Targets Set* analysis computes the potential jump targets, also maintaining the overapproximation property and the *Case Jumps* analysis inserts edges to these computed jump targets. Each stage of the whole analysis thus maintains the premise of computing overapproximated results (even when the results are not precise, see section 5.3.4). So putting theses analyses together in the order how each one depends on the results of the previous one (see figure 5.3) yields an analysis that reconstructs switch constructs by using overapproximation of the control flow.
The repetition is performed as long as the CFG changes and thus new constraints are introduced. As soon as the CFG has reached a point of saturation, i.e. no new edges must be inserted, the analysis ends. The termination of the repeated analysis execution is guaranteed because new edges can only be inserted for switch cases and the amount of cases in the program is finite, as the program code itself is finite.

Example 5.6.

```
switch(par1) {
  case 1:
      switch(par2) {
         case 1: ... break;
         ...
         case 5: ... break;
         default: ...
      }
      break;
  case 2: ... break;
  ...
  case 5: ... break;
  default: ...
}
```

Listing 5.6: A nested switch

```
for (i = 0; i < 3; ++i) {
  switch(i) {
    case 0: ... break;
    case 1: ... break;
    case 2: ... break;
    case 3: ... break;
    case 4: ... break;
    case 5: ... break;
    default: ...
  }
}
```

Listing 5.7: A switch in a loop

Consider example listing 5.6 where a switch is nested in a case block of another switch. The value of the variable tested in the outer switch `par1` is unknown, so the analysis would infer on the first run the jump edges to all the cases. Only after inserting these jump edges, the inner switch can be reached and analyzed, as it is part of the code of a case block. Assuming that the value for `par2` is unknown, in the second run the inner switch would

be reconstructed by inserting edges to all the cases that belong to this switch. A third run would not perform any changes to the CFG as there are no unresolved switches left.
Next, consider example listing 5.7 where a switch is used inside a loop. On the first run of the analysis the exact value for the loop counter $i = 0$ (represented as the strided interval $0\%[0,0]$) is used (see section 5.3.1) to insert a jump edge to the code `case 0: ... break;` thus adding a path through the loop, that connects the loop entry with the loop end (closing the loop path in the CFG). Note that before the insertion of the jump edge, the analysis does not propagate any values to the end of the loop body as the only existing path is the path to the default case and that is not taken because the value of the switch variable does not satisfy the guard condition for the default case. On the second run there now exists a path through the loop that propagates updates of the loop counter, causing the *Table Indices* analysis to iterate until a strided interval for the possible values of the loop counter has been found, here $1\%[0,2]$. Thus, on the second run the edges to the cases 1 and 2 will be inserted. And a third run would not insert any new edges, as the jump to the cases is bound by the range of the loop counter variable i. So cases 3, 4 and 5 will never be jumped to.

5.3.4 Handling the worst-case

As described in section 2.2 it may not be possible to precisely resolve all computed jumps. In such cases where the values for the computed jump are not known (e.g. the value in the jump register is $\top$) the analysis must assume the worst-case and insert edges that jump to every possible point in the program, to yield safe results. This is too imprecise and meaningless for the CFG reconstruction. To lessen such worst cases, assumptions about the code (see section 3) are used to restrict the set of possible jump targets.

Overapproximating the jump table

If it is not possible to compute the indices and thus the size of the jump table in the *Table Indices* analysis (see section 5.3.1) but the base address and thus the location of the jump table is known, the analysis can read till the end of the read-only section and interpret all the data there as possible jump targets. Because the compiler puts the jump tables of all the switches in the program and also several other data into the read-only section, this still leads to coarsely overapproximated jumps, but to a lot less than with the plain "jump to everywhere" strategy.

Allowing only local jumps

Another assumption is that the compiler uses a procedure local jump for the switches. This limits the potential targets to the current procedure. On the PowerPC architecture the jumps to other functions need to set the *LR* (link register) to save the value the callee has to return to. This helps to classify jumps as either local jumps or calls. Some compilers that perform tail-call optimizations [24] do not adhere to this rule. When a function is called as the last instruction in the function body there is no need to return back to this point as it would

```
int procedure(int value) {
  int retval = 1234;

  switch(value) {
    case -5: retval = -55; break;
    case -4: retval = -44; break;
    case -3: retval = -33; break;
    case  1: retval =  11; break;
    case  2: retval =  22; break;
    default: retval = 666;
  }

  return retval;
}
```

Listing 5.8: C source code for switch example

be anyway followed by another return. So compilers that perform this optimization use plain jumps if possible in such cases and it is not so easy anymore to classify a jump as a local jump. As mentioned in section 3 for the analyzed programs tail-call optimizations were disabled, so in the occurrence of an unresolved jump that does not set the *LR* the analysis would insert edges to all the program points in the current procedure.

Combining the assumptions

Nevertheless when both assumptions from above hold (an overapproximated jump table and local jumps only) they can be combined to restrict the jumps to target addresses retrieved from the read-only data section that point to the current procedure.

5.4 Implementation issues

For the implementation of the above analysis some peculiarities need to be taken into account. Some of them arising from the characteristics of the PowerPC architecture.

5.4.1 Conditional Branches

In transfer functions $C6_I, C7_I$ (see table 5.1) the conditional branching is used to infer bounds for the strided interval that is associated with a register. On assembler level conditional branches are implemented in two steps. First a comparison is performed (see line 2) that sets the flags in a special register *CR* according to the result (the flags are *lower, greater, equal*). Then a conditional instruction examines the flags in this condition register (see line 3) to decide which branch to take. The benefit of this approach is that there is only one comparison needed and then arbitrary many subsequent conditional instructions can

```
268436252: addi    r0,r3,5
268436256: cmplwi  cr7,r0,7
268436260: bgt-    cr7,268436332 <procedure+80>
268436264: lis     r9,4096
268436268: addi    r9,r9,7580
268436272: rlwinm  r0,r0,2,0,29
268436276: lwzx    r0,r9,r0
268436280: add     r0,r0,r9
268436284: mtctr   r0
268436288: bctr
268436292: li      r3,-44     // case -4: retval = -44;
268436296: blr                // break;
268436300: li      r3,-55     // case -5: retval = -55;
268436304: blr                // break;
268436308: li      r3,-33     // case -3: retval = -33;
268436312: blr                // break;
268436316: li      r3,11      // case  1: retval =  11;
268436320: blr                // break;
268436324: li      r3,22      // case  2: retval =  22;
268436328: blr                // break;
268436332: li      r3,666     // default: retval = 666;
```

Listing 5.9: Disassembled instructions for switch example 5.8

```
268443036: .long -6736
268443040: .long -6744
268443044: .long -6728
268443048: .long -6704
268443052: .long -6704
268443056: .long -6704
268443060: .long -6720
268443064: .long -6712
```

Listing 5.10: Disassembled jump table for switch example 5.8

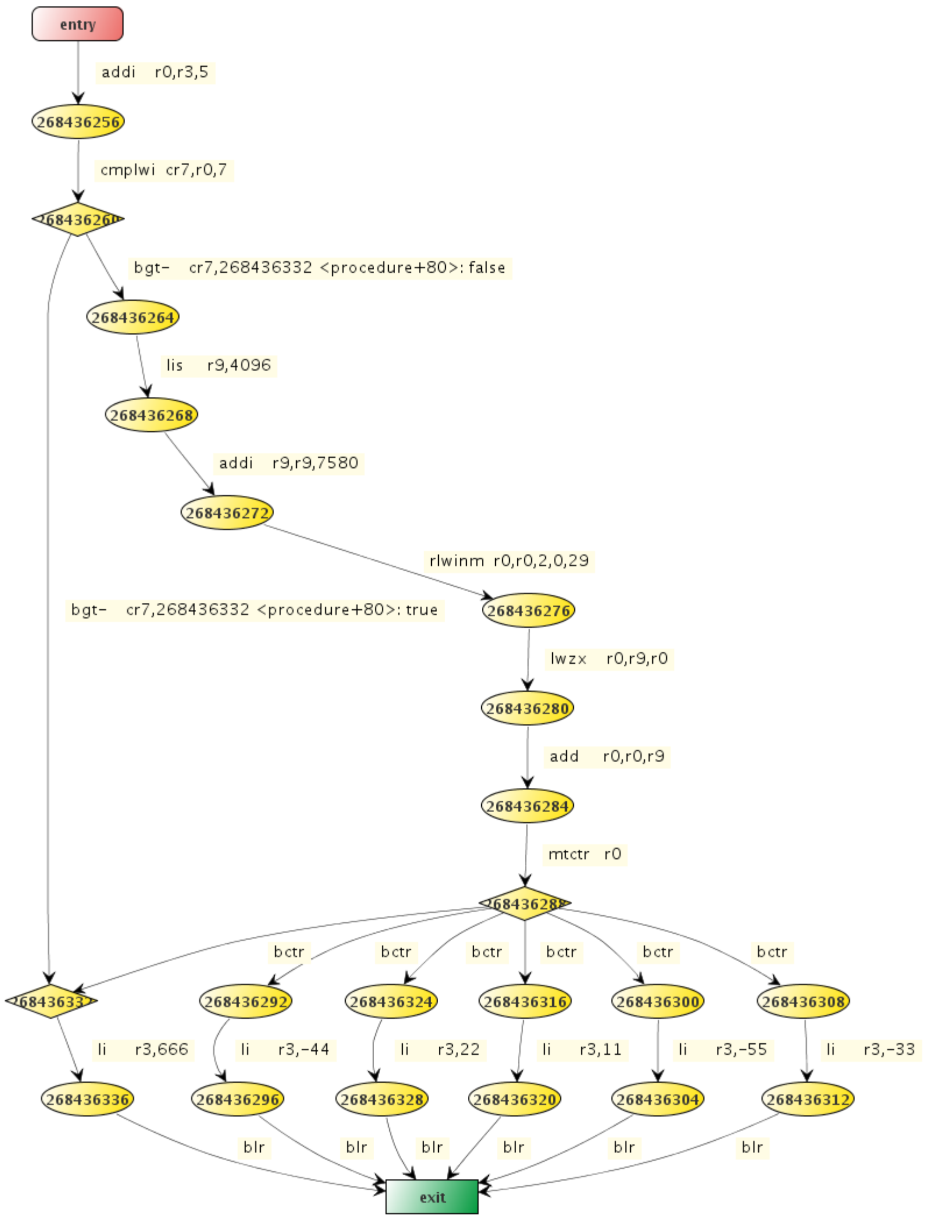

Figure 5.4: CFG for switch in example listing 5.8

use the result. No need to carry out the test again on each branching. For the analysis it introduces some problems. Because the conditional branching is split up in two instructions they must get related to each other. At the branch instruction in line 3 only the condition register is read, but what is important for the analysis is the comparison that changed this register and the operands involved in the comparison. In the above example 5.9 the analysis infers an interval for the possible values in a register at the branching instruction—i.e. if the branch is the true or false edge. Therefore it needs to know the register that was used in the comparison and the number it was compared to. This kind of information must be extracted from the instruction that set the condition register last. In the example 5.9 the comparison instruction is immediately above the branch but that is not always the case (see figure 5.2). As mentioned above the result of a comparison can be examined again by later branch instructions, thus the comparison and the branch can be separated by arbitrary many other instructions. Especially because the *CR* has 8 fields with the same set of condition flags, it must be kept track which comparison was last to change which field. This is done by using a pre-analysis that associates each field in the condition register with the instruction that changed it last at every program point. Thus at each branching point it is known what comparison instruction needs to be examined.

5.4.2 Maximum amount of switch cases

The sets used for the domain D^S in section 5.3.2 are bounded by a particular number n. This is crucial for termination of the analysis but also helps keeping the memory usage and runtime of the analysis manageable. It is important though, that this limit is high enough to be able to save the biggest possibly occurring amount of cases in a switch for the analyzed program. If not, the reconstruction will fail as it cannot memorize all the potential jump targets.
The jump targets for a switch construct are bounded by the amount of instructions in the program, as the code for the switch cases is part of the program code. Any switch in the program cannot contain more cases than there are instructions in the program. Thus choosing the program size—given by the amount of instructions—as the maximum size for the sets cannot lead to a loss of precision with the value set analysis. In practice however one can choose a lower value, e.g. $n = 1000$, to be able to handle all common programs.

6 Procedure calls

```
  268436668: stwu    r1,-56(r1)
2 268436672: mflr    r0
  268436676: stw     r0,60(r1)
4 268436680: stw     r31,52(r1)
  268436684: mr      r31,r1
6 268436688: stw     r3,40(r31)
  268436692: stw     r4,44(r31)
8 ...
```

Listing 6.1: Prologue of a procedure

```
  ...
2 ...
  268436760: lwz     r11,0(r1)
4 268436764: lwz     r0,4(r11)
  268436768: mtlr    r0
6 268436772: lwz     r31,-4(r11)
  268436776: mr      r1,r11
8 268436780: blr
```

Listing 6.2: Epilogue of a procedure

Procedures are functional entities in high-level languages that group the code for a certain task. One benefit of procedures is that the variables in a procedure have local scope, thus increasing the modularity of the code. In assembler the local variables for procedures are implemented by using the stack. Each procedure call installs an own stack frame where it can save registers used for local variables and restore them on procedure exit. For the management of the stack frame two special registers are used: the stack r_1 and the frame r_{31} pointer register. The frame pointer value must not be changed throughout the execution of a procedure, whereas the stack pointer can change if the stack needs to grow dynamically.
On procedure entry (see listing 6.1) the prologue saves the values of the old stack pointer (line 1), the old frame pointer (line 4) and the link register *LR* (lines 2, 3) on the stack. The stack and frame pointer values are changed when installing the stack frame for the current procedure (lines 1, 5) and have to be restored when leaving the procedure which is done in the epilogue (see listing 6.2 lines 1, 5 and 1). The *LR* is used to save the return address when calling another procedure so the callee can return by using the `blr` (*branch to LR*) instruction. Hence the LR is only altered if the procedure would call another procedure. Nevertheless in absence of tail-call optimizations it is always saved and restored (restoration in lines 4, 5 of the epilogue 6.2).

6.1 Link register restoration

According to the calling conventions a call to a procedure should also return to the caller, but with code that contains bugs or malicious code the *LR* can be altered (buffer overflows use this technique [1]) before the return instruction, thus jumping to a different location instead to return back to the caller. For the reconstruction of the CFG this means that the assumption of a return to the caller (see chapter 4) could be violated.

By analyzing the code and comparing the value in the *LR* at procedure entry with the one at procedure exit it is possible to issue a warning in such cases. To assure that the values have not been changed it is necessary to know about accesses to the memory location where the *LR* was saved and from where it is restored in the epilogue.

To keep track of the values in memory the addresses used in a procedure to read and write to memory have to be inferred by the analysis. The *Memory Locations* analysis models the local memory—i.e. the stack—and the global memory by identifying the locations that are read and written to in a procedure. As already mentioned, arithmetic operations are performed only on registers in PowerPC. To change a value in memory it has to be loaded first into a register and after the operations the result has to be written back. Because this is necessary for every operation, even for the computation of memory addresses, the analysis needs to reproduce linear operations performed on registers. Especially because the addressing of memory is done using registers (see chapter 3).
In the following the memory will be modeled by using a domain that maps registers and memory locations to memory addresses. Because memory addresses can be global or local, i.e. relative to the current stack pointer value, the addresses are described as register based constants—a tuple of special registers and constants.

Definition 6.1 (Memory Domain).
$\mathbb{D}^M : ((R \cup M \cup \{lr\}) \to (M \cup \{ret\})_{\perp}^{\top})$ with $M = (R^\star \times \mathbb{Z})$ and $R^\star = \{sp, zero\}$

Where R is again the set of all PowerPC general purpose registers and M is the set of all memory locations, described as a tuple of base registers and address values. $R^\star$ is a set of artificial registers where sp denotes the stack pointer and register $zero$ always contains the value 0. The first artificial register is needed because the exact stack value is not known but is set at procedure entry and accesses relative to this address have to be recognized as a local access. The latter artificial register is introduced for a uniform representation of global or absolute memory addresses as addresses relative to a base register.
The set of general purpose registers R is extended to include the link register lr and the value ret is a marker for the initial value of the link register at procedure entry. Here again, a special value is used because the real value is different with each call of the procedure.

Two elements of the domain are compared by performing a point-wise comparison with the values of the elements, i.e. the values of each register and the values at each memory address. Two elements of M are equal if and only if the registers are equal and the constants' values are equal as well. Two additional elements are introduced for the least $\perp$ and the greatest $\top$ element.
Arithmetical operations on memory addresses are defined as follows ($c_1, c_2 \in \mathbb{Z}$ are arbitrary constants):

$(zero, c_1) + (zero, c_2) = (zero, c_1 + c_2)$
$(zero, c_1) + (sp, c_2) = (sp, c_2) + (zero, c_1) = (sp, c_1 + c_2)$
$(sp, c_1) + (sp, c_2) = \top$
$(zero, c_1) - (zero, c_2) = (zero, c_1 - c_2)$
$(zero, c_1) - (sp, c_2) = \top$
$(sp, c_1) - (zero, c_2) = \top$
$(sp, c_1) - (sp, c_2) = (zero, c_1 - c_2)$

$(zero, c_1) \cdot (zero, c_2) = (zero, c_1 \cdot c_2)$
$(zero, c_1) \cdot (sp, c_2) = \top$
$(sp, c_1) \cdot (sp, c_2) = \top$
$(zero, c_1) + ret = \top$
$(sp, c_1) + ret = \top$
$(zero, c_1) \cdot ret = \top$
$(sp, c_1) \cdot ret = \top$

The purpose of this analysis is identifying memory locations, so the instructions that need to be interpreted with abstract semantics lead to the constraint system below:

$C1_M$:	$D^M[entry_f]$	$\sqsupseteq$	$\top \oplus \{r_1 \to (sp, 0), lr \to ret\}$	$f \in procedures$
$C2_M$:	$D^M[v]$	$\sqsupseteq$	$D^M[u] \oplus \{r_i \to \top\}$	$(u, r_i = ?, v)$
$C3_M$:	$D^M[v]$	$\sqsupseteq$	$D^M[u] \oplus \{r_i \to (zero, c)\}$	$(u, r_i = c, v)$
$C4_M$:	$D^M[v]$	$\sqsupseteq$	$D^M[u] \oplus \{r_i \to D^M[u](r_j)\}$	$(u, r_i = r_j, v)$
$C5_M$:	$D^M[v]$	$\sqsupseteq$	$D^M[u] \oplus \{r_i \to [\![term]\!]^\# D^M[u]\}$	$(u, r_i = term, v)$
$C6_M$:	$D^M[v]$	$\sqsupseteq$	$D^M[u] \oplus \{addr = D^M[u](r_j) + c$ $\begin{cases} r_i \to \top & \text{if } addr \in \{ret, \bot, \top\} \\ r_i \to D^M[u](addr) & \text{otherwise} \end{cases}\}$	$(u, r_i = Mem[r_j + c], v)$
$C7_M$:	$D^M[v]$	$\sqsupseteq$	$D^M[u] \oplus \{addr = D^M[u](r_j) + D^M[u](r_k)$ $\begin{cases} r_i \to \top & \text{if } addr \in \{ret, \bot, \top\} \\ r_i \to D^M[u](addr) & \text{otherwise} \end{cases}\}$	$(u, r_i = Mem[r_j + r_k], v)$
$C8_M$:	$D^M[v]$	$\sqsupseteq$	$D^M[u] \oplus \{addr = D^M[u](r_i) + c$ $\begin{cases} \{a \to \top \mid a \in M\} & \text{if } addr \in \{ret, \bot, \top\} \\ D^M[u](addr) \to D^M[u](r_j) & \text{otherwise} \end{cases}\}$	$(u, Mem[r_i + c] = r_j, v)$
$C9_M$:	$D^M[v]$	$\sqsupseteq$	$D^M[u] \oplus \{addr = D^M[u](r_i) + D^M[u](r_j)$ $\begin{cases} \{a \to \top \mid a \in M\} & \text{if } addr \in \{ret, \bot, \top\} \\ D^M[u](addr) \to D^M[u](r_k) & \text{otherwise} \end{cases}\}$	$(u, Mem[r_i + r_j] = r_k, v)$
$C10_M$:	$D^M[v]$	$\sqsupseteq$	$D^M[u] \oplus \{lr \to D^M[u](r_i)\}$	$(u, lr = r_i, v)$
$C11_M$:	$D^M[v]$	$\sqsupseteq$	$D^M[u] \oplus \{vol \to \top, ((zero, c) \mid c \in \mathbb{Z}) \to \top\}$ with $vol \in \{(r_i \mid i \in [3, 12]) \cup \{lr\}\}$	$(u, call, v)$

Table 6.1: Constraint system D^M for the memory domain $\mathbb{D}^M$

This analysis is intraprocedural, so for all procedures f in the program at procedure start $entry_f$ the values in memory and the registers are set to unknown $\top$. Additionally the register r_1 is marked as the stack register ($C1_M$) because by convention this register is used as the stack pointer (see section 3 and [30]). By setting r_1 to the artificial stack pointer, accesses to memory that use other registers for the address computation can still be recognized as stack accesses. Also the link register lr is initialized with the marker value ret. Non-deterministic assignments ($C2_M$) are assumed for operations that cannot be modeled precisely in the analysis, e.g. bitwise operations. Assignments to a register ($C3_M, C4_M$) with a constant or a register set the value of register according to the right-hand side. Linear terms ($C5_M$) of the form $term \in \{r_i + c, r_i - c, r_i * c, r_i + r_j\}$ are instantiated with the value of every register at the program point and then evaluated according to the operations defined above on memory addresses. A based load from memory ($C6_M$) assigns the value at the memory address specified by the value in the address register r_j added to the base c to the register on the left-hand side r_i. If the evaluated address for the load is not known or the special value *ret* is used the value of the assigned register is destroyed. The process is analogous if the memory address is computed by with indexed addressing ($C7_M$). A store to memory ($C8_M$) sets the value at the memory address specified by the address register and the base. If the address is not known the store destroys the whole memory as it could write to any address. The process is analogous if the memory address is computed with indexed addressing ($C9_M$). Assignments from registers to the link register ($C10_M$) are treated as with other registers and set the value of the *lr*. On procedure calls ($C11_M$) some assumptions are made. The procedure complies to the ABI and does restore the non-volatile registers, whereas the values of volatile registers are set to $\top$. It is also assumed that the called procedure does not write to local memory of the caller, but no assumptions are made about the global memory. The value of the link register is modeled by a non-deterministic assignment as it is overwritten on the procedure call.

On procedure exit if the value in the link register is not *ret* then the restoration of the link register did not succeed and the procedure may not return to the caller. This case issues a warning.

Termination of the fix-point computation is achieved because the transfer functions above are monotonous and the domain is a flat lattice. The lattice elements are ordered as follows: $\bot \sqsubseteq_{\mathbb{D}^M} D_x^M \sqsubseteq_{\mathbb{D}^M} \top$ with $D_x^M \in \mathbb{D}^M \wedge D_x^M \neq \bot \wedge D_x^M \neq \top$. As all elements of the domain that are not $\bot$ and not $\top$ are either incomparable or equal, there exist only finitely many updates to an element—exactly 2.

6.2 Function pointers

```
typedef int (*funptr)(int p1, int p2);

int plus(int a, int b) {
  return a + b;
}

int add(int par1, int par2) {
  funptr fun = plus;
  return fun(par1, par2);
}
```

Listing 6.3: Function pointer example

```
...
268436224: lis     r9,4096
268436228: addi    r0,r9,532
268436232: stw     r0,12(r31)
268436236: lwz     r0,12(r31)
268436240: mtctr   r0
268436244: lwz     r3,24(r31)
268436248: lwz     r4,28(r31)
268436252: bctrl
...
```

Listing 6.4: Dissambled instructions for function pointer example 6.3

Function pointers in C are used to be able to change the called function at runtime by using variables that point to functions (see listing 6.3). In assembler, function pointers are addresses to the code of a procedure. They are implemented as indirect jumps to the value in a register—the counter register *CTR* (see listing 6.4). Because the *Memory Locations* analysis is mapping registers to addresses, the results can be used to resolve the values for

function pointers. As the value of the stack pointer is unknown, only the absolute addresses computed by the analysis are usable for indirect jumps. However the code address of a function is known after linking, so function pointers are never translated to jumps relative to the stack pointer value, especially as the stack pointer value can be different on each call of the procedure. Thus indirect jumps relative to the stack pointer do not occur.
The *Memory Locations* analysis has to be extended to keep track of the value in the *CTR* that is used to perform the jump. The domain is extended as follows:
Definition 6.2 (Function Pointers Domain)**.**
$\mathbb{D}^P : ((R \cup M \cup \{lr, ctr\}) \rightarrow (M \cup \{ret\})^{\top}_{\bot})$ with $M = (R^\star \times \mathbb{Z})$ and $R^\star = \{sp, zero\}$

and the following constraint is added to the constraint system in table 6.1:

$$C1_P: \quad D^P[v] \quad \sqsupseteq \quad D^P[u] \oplus \{ctr \rightarrow D^P[u](r_i)\} \quad (u, ctr = r_i, v)$$

Table 6.2: Constraint system D^P for the function pointer domain $\mathbb{D}^P$ as extension to table 6.1

A further analysis *Function Pointer Jumps* is added, that inserts the jump edge to the resolved addresses. As this analysis changes the CFG it has to maintain the overapproximation property of the ICFG. As the *Memory Locations* analysis does perform a constant propagation [35] of addresses, the jump target can either be a precise address or unknown $\top$. In the first case the analysis resolved the procedure that is called for every possible program execution and thus the ICFG is precise. In the latter case the address value is unknown and it must be assumed that every point in the program can be jumped to. This worst-case assumption can be improved by knowing that a call can only jump to the start of a procedure (as established in chapter 3) and thus only jumps to all the procedure entries in the program will be added. With this assumption the inserted jumps do not change the constraints of the *Memory Locations* analysis for any program point as the analysis is intraprocedural and at procedure entry the values for the registers and memory are always set to an initial value.

6.3 Issues

In the C programming language data can be allocated on the stack via `alloca()` function. This is reflected in assembler by changing the value of the stack pointer to reserve space for the data. As the stack pointer is changed to a value that is unknown at compile time it is not possible to model the stack memory allocated by a call to `alloca()`. Thus the *Memory Locations* analysis would not produce any precise results if dynamic stack allocation is used.

7 Future work

7.1 Switches

Memory locations

The *Switches* analysis presented above does not model the memory, so it is limited to compute constraints about the values in registers. Without compiler optimizations the GCC saves the results of each computation on the stack and loads it again from memory before it is used. For the *Switches* analysis these steps—although redundand and removed by higher optimization levels—destroy on each load the value associated with the register that is loaded, as the analysis assumes a non-deterministic assignment on loads from memory (see section 5.3.1). To be independent of compiler optimizations the *Switches* analysis must be extended to use the results computed by the *Memory Locations* analysis and thus be able to keep track of the values on the stack.

Jump table in code section

Some compilers are putting the jump tables in the code section itself (see [33]). The switch analysis only uses the read-only data section of the executable to look for jump tables. It could be extended to also handle addresses in the code section when resolving the jump table.

Jump table anywhere

As mentioned in section 5.3.2 the memory area where the jump offsets are retrieved from, i.e. the location of the jump table, is expected to reside in the read-only section of the program. Thus it is known at compile time that the values there won't change at runtime. If the jump table would reside in a memory section that can be written to, the analysis has to keep track of all the writes to these memory locations for all execution paths of the program to know what values are at the specified jump table address.

7.2 Procedure Calls

Stack corruption

Analogous to the test of correct restoration of the *LR*, the analysis could verify if the stack and frame pointer are restored correctly in the epilogue of the procedure call (see listing 6.2).

Other domains for the values

At the moment the *Memory Locations* analysis maps memory locations and registers to a domain of constants. It should be extended to handle intervals and sets for the values domain, to yield more precise results. Then function pointers that are assigned to the same variable on different paths through the CFG (e.g. branches, loops and switches) could be resolved and the overapproximated set of jumps to all the inferred function targets would be inserted.

Interprocedural

The *Memory Locations* should be extended to an interprocedural analysis, thus being able to perform constant propagation for the parameters of a procedure. This would improve the resolution of function pointers as they are mostly passed as parameters to other procedures in the C language.

7.3 Related analyses

In the *Memory Locations* analysis, the assumption is made that procedure calls do not write to the stack frame of the caller. This may not always be true. When parameters in C are passed by reference (e.g. arrays) or are pointers to local memory the procedure will write directly to this memory locations. These variables are then escaping the local stack [12]. A further analysis is needed that finds out what side-effects with regard to memory, a procedure call exhibits. These ideas were investigated further in the scope of the SuReal [31, 28] project.

Part IV

Results and Related Work

8 Comparisons

As control flow graphs are a crucial component for code analyses, the reconstruction process has been studied in some previous works. For example in [5] the reconstruction of control flow from binaries has been performed with the intention to understand the code, going as far as trying to decompile the source code. Other motivations are to reconstruct a precise CFG as the basis for further analyses, e.g. for worst-case execution times [33]. The techniques used range from pattern matching and known compiler idioms to code slicing and abstract interpretation. Next, the most common tools performing CFG reconstruction will be presented.

UQBT - Retargetable binary translator

UBQT is a tool developed within the decompilation research of Cifuentes [5, 6]. Switches are resolved by using backward slicing and forward expression substitution for indexed jumps. The instruction slice is then matched against three compiler idioms for switch constructs (called “switch normal forms”). It achieves very good results for the switch reconstruction. Nevertheless, the slicing approach does not handle memory accesses or procedure calls.

IDA Pro

IDA Pro is one of the widely used comercial tools fro disassembly. It performs recursive disassembly [16] and pattern matching for compilers but has limited implementations of static analyses [20]. It was extended within CodeSurfer [15] that adds new analyses [3, 2] to refine the CFGs provided by IDA Pro. The latter uses strided intervals to infer values about the memory and registers.

aiT

Absint offers a collection of tools to perform program analyses. It has frontends for various binaries and some publications deal with the issues of CFG reconstruction [33, 32, 18]. It uses recursive disassembly and a bottom-up approach to reconstruction. It can resolve switches with the help of pattern matching for known compilers. The methods described in [33] rely on static jumps and mention static analyses as an improvement.

Jakstab

The tool was designed to reconstruct the control flow iteratively by using abstract interpretation [19, 20]. In particular it uses an own disassembler frontend to iteratively disassemble code that was revealed by the analyses. The important features are symbolic execution and interprocedural constant propagation. On function calls it assumes no side-effects.
A couple of tests with the examples used in this thesis showed that it is capable to infer function pointers but does not reconstruct switches with unknown values for the switch variable. It is only capable to infer switch cases if the value can be resolved with constant propagation.

It remains for future work to make in depth comparisons of the implemented analyses in this thesis with the tools mentioned above.

9 Conclusion

This thesis introduced ways to improve the reconstruction of CFGs from PowerPC binaries by employing static program analysis. Several constructs of the C programming language and the difficulties they present to CFG reconstruction were discussed. For each of these problems a solution was presented and explained on the basis of the algorithms used for the static analyses. Possible limits of the approaches were mentioned and several improvements and extensions to the analyses were proposed.

The main challenge of reconstructing the interprocedural CFG (ICFG) from binaries is the resolution of computed jumps. As the name implies, the targets of such jumps are computed at runtime of the program, thus making it difficult to determine the targets before running the program. These jumps are generated by compilers during the translation of various high-level language constructs. Incoporating prior knowledge about the limitations and semantics of such high-level constructs allows for resolving these jumps without running the program, by means of static program analysis.
The main focus of the thesis was to analyze code produced from switch constructs and procedure calls. It was shown how switch constructs are translated to assembly code and how the computed jumps that are used to implement a switch can be resolved even without knowing the exact value of the switch variable. Furthermore it was proven that the reconstruction algorithm for switches produces safe approximations for the control flow of the program at runtime. Procedure calls on the assembler level were analyzed to assure they do not violate the properties of functions in high-level languages. Additionally, computed jumps caused by function pointers were resolved wherever possible.

Most of these concepts can be applied to other platforms than the PowerPC and are general enough to handle other languages than C. In addition the described algorithms do not depend on code idioms produced by a certain compiler but abstract the techniques used by compilers in the translation of presented high-level constructs. The achieved results can be used in disassembly or decompilation tools but especially to provide a basis for various program analysis tools that need to cope with binaries.

All the presented analyses were implemented and tested on code examples within the VoTUM framework. Other existing tools of similar purpose have been discussed briefly. Benchmarks with common source code packages remain for future work.

The techniques presented in this thesis show promising results but are merely a starting point for a series of further static analyses, refining the reconstruction of control flow from disassembled binaries.

Appendix

A Toolchain

A.1 Analyses framework

The analyses in this thesis were implemented and tested using the *VoTUM* framework and the GCC cross-compiler. The example files and the framework can be downloaded from [22]. The framework was developed prior to this thesis as a visualization tool for static analyses, mainly targeting analyses for a simple intermediate representation of the language C. For this thesis it was extended to parse PowerPC binaries with the help of *objdump* and perform analyses on PowerPC disassembly code.

A.2 Cross-compiler

To compile the examples for the PowerPC platform, a cross-compilation build of the GCC was used. The cross-compiler can be build with this shell script:

```
#!/bin/sh

# see http://gcc.gnu.org/install/specific.html for more examples
# on host and target specifics
# host will be autodected by configure scripts
#HOST="i686-pc-linux-gnu"
TARGET="powerpc-eabisim"

# a parallel build with e.g. MAKEOPTS="-j 2" seems not to work
MAKEOPTS=""

abort() {
  echo
  echo "**************************************************************"
  echo error: $@
  echo "**************************************************************"
  echo
  exit 1
}

success() {
  echo
  echo "***************************************************************"
  echo success: $@
  echo "***************************************************************"
  echo
  echo "\a"   # ring a bell
  sleep 5s
}

# gcc should not be build in the source dir
# see bug http://gcc.gnu.org/ml/gcc-bugs/2007-06/msg00234.html
prepare_clean_build() {
  rm -rf $BUILDDIR
  mkdir $BUILDDIR
  cd $BUILDDIR
}

CROSSDIR=$HOME/ppc
BUILDDIR=$CROSSDIR/build
SOURCEDIR=$CROSSDIR/sources
DOWNLOADS=$CROSSDIR/downloads
INSTDIR=$CROSSDIR/cross-gcc

export PATH="$INSTDIR/bin:$PATH"

test -d $CROSSDIR || mkdir $CROSSDIR
test -d $DOWNLOADS || mkdir $DOWNLOADS
test -d $SOURCEDIR || mkdir $SOURCEDIR
test -d $INSTDIR && rm -rf $INSTDIR; mkdir $INSTDIR


# 1. install binutils
BINUTILS=binutils-2.19.1

cd $DOWNLOADS
test -f  $BINUTILS.tar.gz  || wget http://ftp.gnu.org/gnu/binutils/$BINUTILS.tar.gz
cd $SOURCEDIR
test -d  $BINUTILS || tar -xzf $DOWNLOADS/$BINUTILS.tar.gz

prepare_clean_build

$SOURCEDIR/$BINUTILS/configure --prefix=$INSTDIR --target=$TARGET \
  --disable-nls --disable-werror
make $MAKEOPTS all install || abort "building of $BINUTILS failed"

success "$BINUTILS successfully installed to $INSTDIR"


# 2. install gcc
# gcc version 4 and above need the gmp and mpfr development libraries to be installed
GCC=gcc-4.3.2

cd $DOWNLOADS
test -f $GCC.tar.bz2 || wget http://mirrors.kernel.org/gnu/gcc/$GCC/$GCC.tar.bz2
cd $SOURCEDIR
test -d $GCC || tar -xjf $DOWNLOADS/$GCC.tar.bz2

prepare_clean_build

$SOURCEDIR/$GCC/configure --prefix=$INSTDIR --target=$TARGET --disable-nls --disable-libssp \
  --enable-languages="c" --without-headers  --with-newlib
make $MAKEOPTS all install || abort "building of $GCC failed"
```

```
success "$GCC successfully installed to $INSTDIR"


# 3. install newlib
NEWLIB=newlib-1.17.0

cd $DOWNLOADS
test -f $NEWLIB.tar.gz || wget ftp://sources.redhat.com/pub/newlib/$NEWLIB.tar.gz
cd $SOURCEDIR
test -d $NEWLIB || tar -xzf $DOWNLOADS/$NEWLIB.tar.gz

prepare_clean_build

$SOURCEDIR/$NEWLIB/configure --prefix=$INSTDIR --target=$TARGET --disable-nls
make $MAKEOPTS all install || abort "building of $NEWLIB failed"

success "$NEWLIB successfully installed to $INSTDIR"


echo "To use the cross compiler, do:"
echo "$INSTDIR/bin/$TARGET-gcc -msim sourcefile.c"
echo "where -msim chooses the simulator as the machine type for"
echo  "PPC (default when using powerpc-eabisim as target)"
echo "and can be replaced by one of the following:"
echo "-msim, -mads, -myellowknife, -mmvme"
echo "See the GCC manual section for machine type options"
echo "for PowerPC for more information."
```

Listing A.1: Shell script to build the GCC cross-compiler

List of Figures

Listings

List of Tables

Bibliography

[1] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996. 63

[2] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. Wysinwyx: What you see is not what you execute. Technical report, University of Wisconsin, 2005. 19, 51, 73

[3] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *In Comp. Construct*, pages 5–23, 2004. 73

[4] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, third edition, 2005. 36

[5] Christina Cifuentes and John K. Gough. Decompilation of binary programs. *Software Practice and Experience*, 25(7):811–829, July 1995. 73

[6] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40:171–188, 2001. 46, 73

[7] TIS Committee. Executable and Linking Format (ELF) Specification. http://refspecs.freestandards.org/elf/elf.pdf, 1995. retrieved at 18.10.2009. 36

[8] Agostino Cortesi. Widening operators for abstract interpretation. In *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 31–40, Washington, DC, USA, 2008. IEEE Computer Society. 54

[9] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming*, pages 106–130, April 1976. 50

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM. 50, 54

[11] Tom Duff. Duff's Device. http://en.wikipedia.org/w/index.php?title=Duff%27s_device&oldid=322067014, 1983. retrieved at 29.10.2009. 43

[12] Andrea Flexeder, Michael Petter, and Helmut Seidl. Analysis of executables for WCET concerns. Technical report, Technische Universität München, 2009. 19, 70

[13] Freescale. Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture. http://www.freescale.com/files/product/doc/MPCFPE32B.pdf, 2005. retrieved at 18.10.2009. 19, 35, 36, 41

[14] GNU-Project. GNU Binutils. http://www.gnu.org/software/binutils/, 2009. retrieved at 29.10.2009. 42

[15] Grammatech. Codesurfer. http://www.grammatech.com/products/codesurfer, 2009. retrieved at 18.10.2009. 73

[16] Hex-Rays. IDA Pro Disassembler. http://www.hex-rays.com/idapro, 2009. retrieved at 18.10.2009. 39, 42, 73

[17] Randall Hyde. *The Art of Assembly Language*. No Starch Press, 2003. 36

[18] Daniel Kästner and Stephan Wilhelm. Generic control flow reconstruction from assembly code. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 46–55, New York, NY, USA, 2002. ACM. 73

[19] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 423–427, Berlin, Heidelberg, 2008. Springer-Verlag. 74

[20] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *VMCAI '09: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 214–228, Berlin, Heidelberg, 2009. Springer-Verlag. 73, 74

[21] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 23, 36

[22] Bogdan Mihaila. VoTUM - a framework for static analyses. http://www2.in.tum.de/projects/votum, 2009. retrieved at 29.10.2009. 79

[23] Motorolla. RISC central processing unit reference manual. http://www.freescale.com/files/microcontrollers/doc/ref_manual/RCPURM.pdf, 1999. retrieved at 18.10.2009. 41

[24] Mark Probst. Proper tail recursion in C. Master's thesis, Technische Universität Wien, 2001. 36, 58

[25] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 100–111, New York, NY, USA, 2006. ACM. 51, 52

[26] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 45, Washington, DC, USA, 2002. IEEE Computer Society. 42

[27] SCO. System V Application Binary Interface. http://www.sco.com/developers/devspecs/gabi41.pdf, 1997. retrieved at 18.10.2009. 40

[28] Helmut Seidl, Andrea Flexeder, and Michael Petter. Side-Effect Analysis of Assembly Code. Technical report, Technische Universität München, 2009. 70

[29] Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. Lazy continuations for Java virtual machines. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 143–152, New York, NY, USA, 2009. ACM. 29

[30] Steve Zucker (SUN) and Kari Karhi (IBM). PowerPC Application Binary Interface. `http://refspecs.freestandards.org/elf/elfspec_ppc.pdf`, 1995. retrieved at 18.10.2009. 35, 36, 40, 65

[31] SuReal. Sicherheitsgarantien unter Realzeitanforderungen. `http://www.sureal-projekt.org/index.html`, 2008. retrieved at 29.10.2009. 70

[32] Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju-do, South Korea, December 2000. 29, 73

[33] Henrik Theiling. *Control Flow Graphs For Real-Time Systems Analysis*. Ph. D. Thesis, Universität des Saarlandes, Saarbrücken, Germany, 2002. 29, 39, 41, 69, 73

[34] William von Hagen. *The Definitive Guide to GCC, Second Edition*. Apress, 2 edition, August 2006. 29, 36

[35] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991. 53, 67

[36] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society. 19