DEGREE PROJECT IN THE FIELD OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY
AND THE MAIN FIELD OF STUDY
COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Alternating Control Flow Graph Reconstruction by Combining Constant Propagation and Strided Intervals with Directed Symbolic Execution

THOMAS PETERSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Alternating Control Flow Graph Reconstruction by Combining Constant Propagation and Strided Intervals with Directed Symbolic Execution

THOMAS PETERSON

Master in Computer Science
Date: 19 December 2019
Supervisor: Roberto Guanciale
Examiner: Mads Dam
School of Electrical Engineering and Computer Science
Swedish title: Alternerande kontrollflödesgrafsrekonstruktion genom att kombinera propagerande av konstanter och klivande intervaller med riktad symbolisk exekvering

Abstract

In this thesis we address the problem of control flow reconstruction in the presence of indirect jumps. We introduce an alternating approach which combines both over- and under-approximation to increase the precision of the reconstructed control flow automaton compared to pure over-approximation. More specifically, the abstract interpretation based tool, Jakstab, from earlier work by Kinder, is used for the over-approximation. Furthermore, directed symbolic execution is applied to under-approximate successors of instructions when these can not be over-approximated precisely. The results of our experiments show that our approach can improve the precision of the reconstructed CFA compared to only using Jakstab. However, they reveal that our approach consumes a large amount of memory since it requires extraction and temporary storage of a large number of possible paths to the unresolved locations. As such, its usability is limited to control flow automatas of limited complexity. Further, the results show that strided interval analysis suffers in performance when encountering particularly challenging loops in the control flow automaton.

Keywords
Control flow reconstruction, Static analysis, Binary, Symbolic execution, Control flow graph

Sammanfattning

I detta examensarbete studeras hur rekonstruktion av kontrollflöde kan utföras i närvaro av indirekta hoppinstruktioner. I examensarbetet introduceras ett alternerande tillvägagångssätt som kombinerar både över- och underapproximation för att öka precisionen av den rekonstruerade kontrollflödesautomaten jämfört med endast överapproximation. Mer specifikt används det abstrakta tolknings-baserade verktyget, Jakstab, tidigare utvecklat av Kinder, för överapproximation. Vidare nyttjas riktad symbolisk exekvering för att underapproximera efterträdare till instruktioner när dessa inte kunnat överapproximeras med hög precision. Resultaten av våra experiment visar att vårt tillvägagångssätt kan förbättra precisionen hos den rekonstruerade kontrollflödesautomaten jämfört med att endast använda Jakstab. Dock visar de att vårt tillvägagångssätt kan fordra en stor mängd minne då det kräver extraktion och tillfällig lagring av ett stort antal möjliga vandringar från programmets ingång till de olösta programpositionerna. Därmed är dess användbarhet limiterad till kontrollflödesautomater av begränsad complexitet. Vidare visar resultaten av våra experiment även att analyser baserade på klivande intervaller tappar i prestanda när de möter en vis typ av loopar i kontrollflödesautomaten.

Nyckelord
Kontrollflödesrekonstruktion, Statisk analys, Binär, Symbolisk exekvering, Kontrollflödesgraf

Acknowledgement

First and foremost, I would like to thank my supervisor Roberto Guanciale for his feedback and guiding throughout the research and writing of this thesis. Furthermore, I am also grateful for the members of the STEP group and the exchange students who made my stay at TCS more enjoyable. Finally, I want to thank my family for supporting me throughout my studies and the research conducted in this thesis.

Thank you!

Stockholm, December 2019

Thomas Peterson

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 2
- 1.2 Outline . . . 3

2 Theoretical Background **4**
- 2.1 Control Flow in the x86 Architecture . . . 5
- 2.2 Control Flow Graphs . . . 7
- 2.3 Static and Dynamic Analysis . . . 13
  - 2.3.1 Disassembly in Static Analysis . . . 14
  - 2.3.2 Challenges of Static Analysis Based Control Flow Reconstruction . . . 14
  - 2.3.3 Dynamic Control Flow Reconstruction . . . 16
- 2.4 Abstract Interpretation . . . 18
- 2.5 Data Flow Analysis . . . 20
- 2.6 Over-Approximation with Jakstab . . . 23
- 2.7 Abstract Domains . . . 26
  - 2.7.1 Location Analysis . . . 26
  - 2.7.2 Constant Propagation . . . 26
  - 2.7.3 Strided Intervals . . . 28
- 2.8 Related Work . . . 31

3 Methodology **36**
- 3.1 Limitations and Target Binaries . . . 37
- 3.2 Modifications to the CPA Algorithm . . . 38
- 3.3 Directed Symbolic Execution . . . 46
- 3.4 Motivating Examples for Using DSE . . . 49
- 3.5 Metrics for Evaluation . . . 55
- 3.6 Implementation . . . 58
- 3.7 Experiments . . . 60

4 Results **63**
4.1 Modifications to the Widening Operator of the Strided Interval Domain 64
4.2 Evaluation of Synthetic Binaries 65
4.3 Evaluation of GNU Coreutils 72
4.4 Conclusions 74
4.4.1 Limitations of Strided Intervals 74
4.4.2 Number of Possible Paths in a CFA 77

5 Discussion and Future Work **82**
5.1 Discussion 82
5.2 Future Work 84

Bibliography **87**

A Benchmarking **91**

Abbreviations

ACFR: Alternating Control Flow Reconstruction
BBG: Basic Block Graph
BSS: Basic Service Set
CFA: Control Flow Automaton
CFG: Control Flow Graph
CISC: Complex Instruction Set Computer
CNF: Conjunctive Normal Form
CPA: Configurable Program Analysis
DSE: Directed Symbolic Execution
IL: Intermediate Language
IP: Instruction Pointer
Jakstab: Java Tool**k**it for **Sta**tic Analysis of **B**inaries
PC: Program Counter
RAM: Random Access Memory
SAT: Boolean Satisfiability Problem
SIMD: Single Instruction, Multiple Data
SMT: Satisfiability Modulo Theories
Stdin: Standard Input
Stdlib: Standard Library
Stdout: Standard Output
Syscall: System Call

Glossary

Basic Block: A sequence of fall-through instructions.
BBG: A graph consisting of basic blocks and control flow transitions between these.
CFA: An automaton where nodes correspond to program locations and edges to instructions.
CFG: A BBG or a CFA.
Conditional Jump: A jump instruction which only performs a jump if a certain condition holds.
CPA: A framework which, given a binary, an initial state and a CFA, outputs a set of reachable abstract states.
Direct Jump: A jump instruction which jumps to an explicit address.
Double Word: 4 bytes.
DSE: Symbolic execution limited to a set of paths.
Fall-Through Instruction: An instruction whose only successor is the immediately following instruction.
Indirect Jump: A jump instruction which jumps to an address located in a register or a memory location.
IP: A register containing the address of the instruction that is currently being executed.
Jakstab: A static analysis platform for x86 binaries.
Jump Instruction: An instruction which is not a fall-through instruction.
Jump table: A data structure which sometimes results from switch cases and leads to indirect jumps.
PC: See IP.
Top Edge: An edge originating from a top location.
Top Location: A location whose successors could not be over-approximated without resorting to the $\top$ element.
Unconditional Jump: A jump instruction which results in a jump independently of the FLAGS register.
Word: 2 bytes.

List of Figures

2.1 Assembly code of the example BBG and CFA graphs. . . 8
2.2 The BBG of the "Hello World!" Program. 9
2.3 The generalized CFA of the "Hello World!" program where nodes encompass all register and memory content. 10
2.4 An example of soundness and accuracy in CFGs. 12
2.5 An example where random black-box fuzzing will have a low probability of exploring all possible branches. . . . 17
2.6 Example of a forward flow analysis worklist algorithm. . 21
2.7 cjmp.asm - An example containing a conditional jump which can be resolved without false positives using the constant propagation domain. 27
2.8 branch.asm - An example of when strided intervals can be more useful than constant propagation. 29

3.1 The original CPA+ Algorithm. 38
3.2 The CPA+ Algorithm of Jakstab. 40
3.3 The modified version of the Jakstab algorithm which leverages DSE to resolve unresolved instructions. 42
3.4 The second version of the ACFR algorithm which resolves all tops at each DSE invocation. 45
3.5 Pseudocode for the directed symbolic execution. 47
3.6 The algorithm which, given a set of paths, returns a set of paths which satisfy the condition that the address at index C is equal to the value of the IP register. 48
3.7 The algorithm which extracts successors of paths. 48
3.8 xyz.asm - An example of when two registers with unkown content are subtracted. 50
3.9 nestedCall.asm - An example of two nested function calls. 51

3.10 sequentialCallRet.asm - An example where one function is called more than once. 53
3.11 trueTop.asm - An example of a true top. 54
3.12 An overview of the implementation. 59
3.13 Hardware specs of the host on which the experiments were performed. 61

4.1 The relevant instructions of badCall.asm. 67
4.2 The relevant instructions of cjmpBitOp.asm. 69
4.3 A minimalistic example containing a loop which is problematic for the strided interval analysis. 75
4.4 The changes of the strided interval for the value of the *esp* register when the problematic loop is analysed. 75
4.5 Two CFAs with two loops each but structured differently. 77
4.6 An example of a small program with a large amount of possible paths. 79
4.7 A simplified illustration of the CFA for the small program with a large amount of possible paths. 81

List of Tables

4.1 Results of the synthetic evaluation for the first ACFR algorithm (Part 1). 66
4.2 Results of the synthetic evaluation for the first ACFR algorithm (Part 2). 68
4.3 Results of the synthetic evaluation for the second ACFR algorithm which differed from the results obtained with the first version. 70
4.4 Evaluation of the 5 smallest GNU coreutils binaries using the first ACFR algorithm. 73

A.1 Execution times for the synthetic evaluation with the first ACFR algorithm (Part 1). 92
A.2 Execution times for the synthetic evaluation with the first ACFR algorithm (Part 2). 93
A.3 Execution times for the synthetic evaluation with the second ACFR algorithm (Part 1). 94
A.4 Execution times for the synthetic evaluation with the second ACFR algorithm (Part 2). 95
A.5 Execution times for the evaluation of the 5 smallest GNU coreutils binaries using the first ACFR algorithm. 96
A.6 Execution times for the evaluation of the 5 smallest GNU coreutils binaries using the second ACFR algorithm. . . . 96

Chapter 1

Introduction

Software is normally not developed directly in machine code but indirectly in a higher-level programming language such as C. Before a program can be used, its code must be compiled into architecture-dependent machine code. However, the compilation process can vary depending on many factors such as, for example, the type of compiler and the optimization level. Thus, software written in a non-assembly language might have multiple binary representations, some of which might even be erroneous [1]. As such, when analysing software, it might be incorrect to expect properties which hold in the source code to hold in the compiled binary. A prime example of this is the malicious copies of installers for the iPhone application development framework, Xcode, that began circulating in 2015 [2].

These installers appeared to behave identically to the legitimate installers but would install a corrupted version of the framework. Subsequently, during compile time, the corrupted framework would inject malware into the application. Thus, development with the infected framework would result in infected applications. Consequently, source-code level analysis would not have been able to find anything wrong with the produced applications. As such, this example clearly motivates why binary-level analysis is paramount. In addition to corrupted compilation processes, another incentive to binary-level analysis is that source code often is unavailable as many commercial off-the-shelf software and third-party libraries are distributed only in binary form [1].

As binary code is more verbose than high-level code, it is infeasible to manually analyze large binaries. Consequently, many binary analysis platforms have been created to automate binary analyses [3][4][5][6]. Binary analysis has thus become a broad field with many different applications. It might, for example, be used for program verification [5], exploit generation [7] or detection of memory management errors [8]. Further, binary analysis can be useful even in situations where the source code is available. Such situations can, for example, occur when the compiler is not part of the trusted computing base and one would like to empirically establish that properties of the source code still holds in the corresponding binary.

1.1 Problem Statement

Control flow can be thought of as which of a binary's instructions can be executed in what order. This information is essential for many types of binary analysis. In other words, A fundamental obstacle when performing binary analysis is the lack of precise control flow information [9]. There are already many existing approaches to control flow reconstruction. Most of these can be categorized as either being based on static or dynamic analysis. When leveraging static techniques, the binary is studied without performing any concrete executions. Instead, the binary is analysed in an abstract manner. Dynamic techniques, on the other hand, concretely execute the binary with different inputs with the intention of triggering different control flow paths. As such, static analysis techniques often over-approximate control flow while dynamic techniques under-approximate control flow.

Static analysis often suffers from problems stemming from indirect jump instructions. These are jump instructions where the processor is instructed to jump to a value stored in a register or memory location rather than an explicit address encoded in the instruction. As it is expensive to keep track of all memory and register content for all possible states of a program, it is expensive to calculate the target location of such jumps. Consequently, many static analysis tools resort to approximations which might not always result in an optimal control flow reconstruction.

The main objective of this thesis is to study how the precision of control flow reconstruction can be improved in the context of alternating over/under-approximation approaches. Thus, the objective can be summarized as "How can the precision of CFA:s reconstructed using state-of-the-art algorithms be improved using alternating approaches?". To be more specific, it is studied how indirect jump resolution can be improved by using under-approximation when over-approximation can not deduce a finite set of possible successors to an instruction. The thesis is limited to 32-bit x86 binaries and targets binaries without additional source code information.

We investigate an approach based on alternating between over/under-approximation inspired from previous work [10]. Furthermore, the over-approximation approach uses constant propagation and strided intervals which are based on previous work by Kinder et al. [11]. Finally, the under-approximation is performed through the usage of directed symbolic execution (DSE) and an SMT solver.

1.2 Outline

The outline of the thesis is as follows. In chapter 2, a theoretical background is provided. This background serves as a foundation for understanding the remaining parts of the thesis. In chapter 3, the degree project methodology is presented and in chapter 4 the results are illustrated and described. Thereafter, in chapter 5, a summary of the thesis and discussions concerning possible future work, are provided.

Chapter 2

Theoretical Background

This chapter provides a theoretical background explaining the theory needed to understand the remaining parts of the thesis. The chapter starts with a brief introduction to control flow in the x86 architecture followed by an explanation of different types of control flow graphs. Thereafter, the advantages and disadvantages of static and dynamic analysis techniques are presented.

Afterwards, abstract interpretation is introduced and the fundamentals of data flow analysis are explained. Then, the theory behind the over-approximation tool Jakstab and the abstract domains that are used in this tool are described. Finally, we discuss related work and how they differ from what has been conducted in this thesis.

Throughout the thesis, a set of hand-made example binaries will be presented to illustrate the pros and cons of different approaches to control flow reconstruction. These binaries will be referred to as synthetic binaries. Furthermore, these binaries will be used for the synthetic evaluation of our approach presented in section 4.2.

2.1 Control Flow in the x86 Architecture

Assembly instructions can be categorized into three possible control flow transitions [12]. The first type is fall-through instructions. These instructions are instructions which, after being executed, simply increments the Instruction Pointer (IP) with the size of the instruction. Consequently, the execution continues with the instruction immediately after the fall-through instruction. Instructions which do not classify as fall-through instructions are denoted as jump instructions.

The last two types are direct and indirect jumps. Jumps are instructions which transfers the flow of execution by changing the IP register by a specified value. The difference between a direct and indirect jump is that a direct jump is a jump to an explicit address whereas an indirect jump is a jump to an address stored in a register or a location in RAM.

Furthermore, jumps can also be categorized into conditional and unconditional jumps. Conditional jumps are jump instructions that only perform a jump to a specified location if a certain condition holds. Otherwise, the IP is simply incremented by the instruction size to instruct the CPU to execute the next instruction. Conversely, unconditional jumps are jump instructions that always results in a jump to a specified location

The table below provides example instructions in x86, illustrating the difference between conditional and unconditional jump instructions as well as direct and indirect jump instructions. Generally, x86 instructions are represented using either Intel or AT&T syntax. In this thesis, x86 examples will be written using the Intel syntax.

	Direct	**Indirect**
Unconditional	jmp 0x00401078	jmp eax
Conditional	je 0x00401078	je eax

The instruction *jmp 0x00401078* is an example of a direct and unconditional jump instruction. It is unconditional since it uses the *jmp* instruction which instructs the CPU to jump to the specified address unconditionally. Furthermore, it is direct since the address *0x00401078*

is specified explicitly. Note that the address *0x00401078* is arbitrary chosen for the sake of illustration and does not have any special meaning.

The *jmp eax* instruction is an example of an indirect and unconditional jump instruction. It is indirect since the jump is performed to the address located in the register *eax*. The *je 0x00401078* and *je 0x00401078* instructions are examples of conditional jump instructions. These are instructions that execute a jump only if a specific condition is satisfied, in this case, if and only if the last pair of previously compared registers were equal.

Before a conditional jump is executed, the CPU normally executes an instruction to compare different register values. In x86, it is common to use the *cmp* instruction which takes two arguments. It subtracts the second argument from the first and sets a set of register flags according to the result of the subtraction.

Apart from explicit jump instructions, there are also *call* instructions which can alter the control flow of a program. Normally, *call* instructions are used to invoke a procedure. Initially, the program pushes its arguments to the stack (Or alternatively puts them in registers). Thereafter, the program executes the call instruction, which is a push of a return address to the stack and an unconditional jump to the start of the procedure. At the end of the procedure, there is a return instruction *ret* which pops the stack to obtain the return address that was pushed by the call instruction. Then, the CPU performs and unconditional jump to the obtained return address.

In this work, the theoretical framework of the over-approximation treats a *call* instruction as an unconditional *jmp* instruction which pushes the address of the subsequent instruction to the stack. Furthermore, a *ret* instruction is treated as an unconditional indirect jump whose target is the topmost value of the stack.

2.2 Control Flow Graphs

The control flow of a program is defined as what instructions are executed in what order. A Control Flow Graph (CFG) is a directed graph which illustrates the control flow of a program. This graph is usually represented by either a Basic Block Graph (BBG) or a Control Flow Automaton (CFA). Consequently, when it is not important whether the graph is represented by a BBG or a CFA, the notion CFG will be used.

A Basic Block Graph (BBG) is defined as a directed graph $G = (V, E)$ where V is a set of nodes representing blocks of instructions, called basic blocks, and E is a set of edges representing all the possible transitions between the basic blocks. Each basic block consists of a set of fall-through instructions and at most one jump instruction at the end the block.

As all fall-through instruction are contained in basic blocks, each edge in the BBG naturally corresponds to either a direct or an indirect jump. However, there is an exception when there is a jump whose targets lies inside a basic block. For this exception case, the targeted basic block has to be split in two, resulting in an edge between the fall-through instruction located immediately before the targeted instruction of the jump and the location of the targeted instruction.

A Control Flow Automaton (CFA), as originally defined by Henzinger et al. [13], is a graph where nodes represent states and edges correspond to statements representing control flow transitions between states. In our context, the CFA definition by Kinder [11] is used. More formally, let *Stmt* denote the set of statements and L the set of potential values of the IP. Then, a CFA for an intermediate language is defined as a tuple $< T, V, start, E >$, where $T \subseteq L$ is a set of program locations, V is the set of registers[1], *start* is an initial location such that $start \in T$ and E is an edge relation $E \subseteq T \times Stmt \times T$.

[1] In essence, V is the set of free names. It should be noted that this set is architecture specific and thus does not vary between x86 binaries. The set is part of the definition since the free names are used in the set of statements.

```
1  SECTION .data
2  msg     db      'Hello World!', 0x0A
3
4  SECTION .text
5  start:
6          mov edi,10      ; Set the edi register to 10
7  compare:
8          cmp edi,0       ; Compare edi with 0
9          je quit         ; Jump to quit if edi=0
10 print:
11         dec edi         ; Decrement edi by 1
12         mov edx, 13     ; Set edx to the length of msg
13         mov ecx, msg    ; Set ecx to the address of msg
14         mov ebx, 1      ; Set ebx to stdout
15         mov eax, 4      ; Set eax to the opcode of write
16         int 0x80        ; Perform the syscall
17         jmp compare     ; Jump to compare
18 quit:
19         mov ebx, 0      ; Set ebx to 0
20         mov eax, 1      ; Set eax to the opcode of exit
21         int 0x80        ; Perform the syscall
```

Figure 2.1: *Assembly code of the example BBG and CFA graphs.*

When a CFA is depicted as a graph, the nodes represent program locations and edges correspond to statements in the relation E. In other words, a CFA is essentially a BBG where instructions labels the edges rather than the nodes. Consequently, CFAs have the same level of expressiveness as BBGs. However, unlike BBGs, CFAs can be generalized to have more expressiveness. More specifically, a generalized CFA is a CFA where nodes are groups of states based on some criterion.

The conventional CFA is thus a special case of the generalized CFA where the only condition imposed on the nodes is the value of the IP register. An advantage of generalized CFAs over BBGs and CFAs is that their nodes can naturally correspond to program states and the edges to state transformations. Consequently, as conventional BBGs and CFAs can only represent connections between instructions and program locations respectively, generalized CFAs can more accurately represent the control flow of binaries.

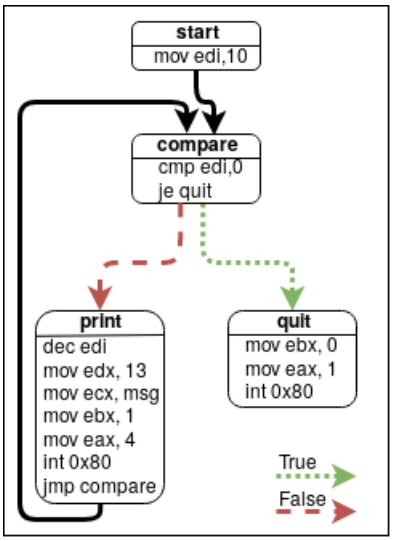


Figure 2.2: *The BBG of the "Hello World!" Program. True and false branch conditions are illustrated with dotted green and dashed red arrows respectively.*

To better understand this, consider the assembly code in Figure 2.1. The string "Hello World!" is stored in the data section of the binary and the text section contains the code for writing "Hello World!" to stdout ten times. The code contains the labels start, compare, print and quit as these can be used as jump targets for the jump instructions. Initially, the program sets the value of edi to 10 as this register is used to count how many times the string has to be printed to stdout.

The program then compares *edi* to 0 and jumps to *quit* if this holds. Otherwise, it decrements *edi* by 1 and fills the registers with values for performing a system call to write the string 'Hello World!' to stdout. The register *edx* is set to the length of the string, *ecx* to the address of the string, *ebx* to the file descriptor of stdout and *eax* to the opcode of the write instruction. After performing the write, the execution performs a jump back to the compare instruction to check if *edi* is still more than 0.

The BBG of the hello-world program is illustrated in Figure 2.2. There are multiple problems with this BBG. Firstly, from the BBG, it appears that a possible path could be *start*, *compare* and *quit*. However, this path is not possible in practice. Furthermore, it is also not possible to tell if

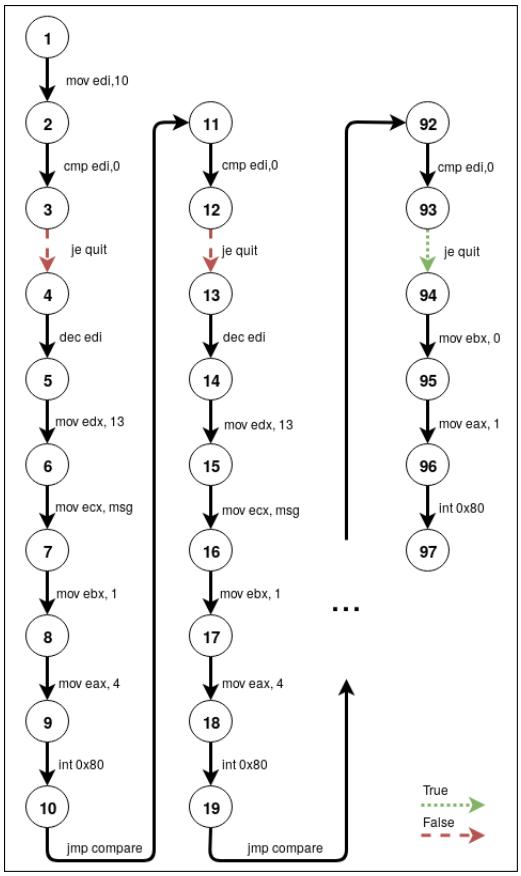


Figure 2.3: *The generalized CFA of the "Hello World!" program where nodes encompass all register and memory content. True and false branch conditions are illustrated with dotted green and dashed red arrows respectively.*

the program will ever terminate without studying the assembly code. This is because it appears that the program could execute *start* and then get stuck in an infinite loop between *compare* and *print*.

Assume that the criterion of the nodes of the generalized CFA is based on the complete register and memory content. Then, the generalized CFA of the hello-world program, depicted in Figure 2.3, only contains one possible path. This is because a node is no longer a group of instructions but rather a set of register values and a set representing memory content, thus accurately representing a concrete state.

Consequently, state 2, 11 and 92 are distinct as they differ in the *edi* value. More specifically, in this example, state 2, 11 and 92 correspond to $edi = 10$, $edi = 9$ and $edi = 0$ respectively. Note, that the conditional jump instruction *je quit* does not have to correspond to two possible edges as the condition can be fully determined by the state.

A problem with the usage of BBGs when representing control flow is when conducting control flow integrity enforcement [14]. When Control flow integrity enforcement is performed, the auditing software performing this enforcement uses information about what instructions can transfer control flow to which locations. It then audits executing programs to verify that all control flow transfers are performed according to these targets. If, however, a BBG is used, an attacker could enter a basic block from one edge and exit from an edge that should not be feasible in practice when the entry edge was used.

For example, if an attacker could manipulate the control flow of the hello world program so that it executed start, check and quit, this would be undetected by the control flow integrity enforcement. This problem is not solved by simply converting the BBG to a CFA representation of the control flow. Rather, it would be required to generate and use a generalized CFA as this representation could be able to distinguish between different concrete states at the same location.

Two desirable properties of CFGs are soundness and completeness, introduced by Shoshitaishvili et al. [7]. Soundness is defined as the extent to which the CFG contains the correct control flow transitions. Thus a CFG is said to be sound if and only if, it contains all the possible control flow transitions.

Similarly, completeness of a CFG is the extent to which the CFG only contains possible control flow transitions. Consequently, a CFG is said to be accurate if and only if, it only contains possible control flow transfers. In this work, completeness and the property of being complete will be referred to as accuracy and the property of being accurate respectively as we believe these terms are more intuitive.

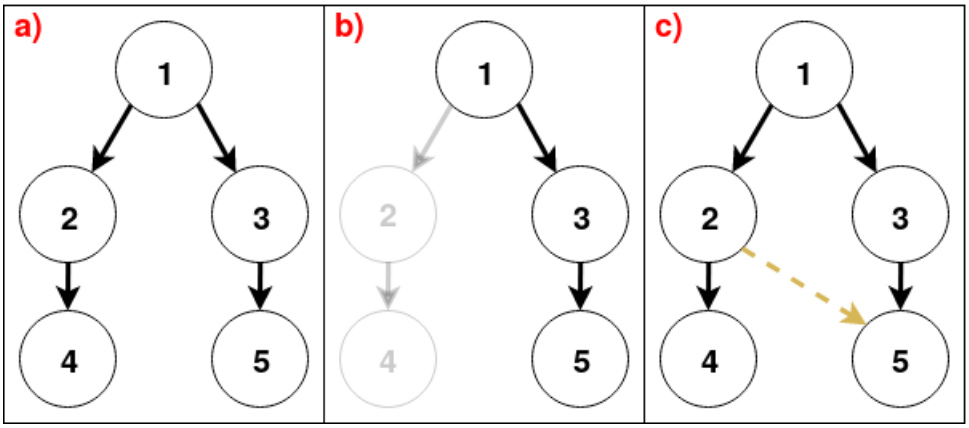


Figure 2.4: *An example of soundness and accuracy in CFGs. Graph a is the ideal CFG, graph b is an accurate approximation of the ideal CFG and graph c is a sound approximation of the ideal CFG. The transparent nodes and arrows denote unidentified parts of the graph. Dashed yellow arrows denote falsely identified edges.*

In Figure 2.4 the ideal CFG, an accurate under-approximation of the ideal CFG and a sound over-approximation of the CFG for some fictional binary program, is presented. Graph b is accurate since it contains all edges of the ideal graph. However, during reconstruction of graph b, the edge between node 1 and 2 was missed. Consequently, all nodes after node 2 were missed as well. There could possible exist more instructions after node 2 that could only be reached through the edge between node 1 and 2. Thus, missing one edge during the control flow graph reconstruction can lead to a cascading loss of precision.

Graph c is a sound over-approximation of the ideal CFG. The problem with graph c is that the edge between node 2 and 5 has been added. This might, at first glance, not seem very problematic. However, it can, for example, be a security risk during control flow integrity enforcement. Furthermore, if there are a large amount of falsely identified nodes or edges, the reconstruction time and memory usage can be significantly affected. In other words, over-approximation can consume significantly more time and memory resources than under-approximation since it has to explore more nodes and edges [10].

As there are currently no known polynomial time algorithms for reconstructing the optimal CFG of an arbitrary binary, several researchers have resorted to under and/or over-approximation approaches [1][7][9][12]. As purely over-approximating approaches, by definition, does not miss any edges of the ideal CFG, they are sound. Conversely, as under-approximation approaches never include superfluous edges, they are accurate. In this thesis, however, the primarily desire is to the maximize the precision.

The preciseness of a CFG can informally be thought of as the difference in the amount of edges between the subject CFG and the ideal CFG. Thus, an accurate CFG with 10 superfluous edges is more precise than an accurate CFG with 20 superfluous edges. A formal definition of preciseness of a CFG is provided in section 3.5.

2.3 Static and Dynamic Analysis

The goal of static and dynamic analysis techniques are to determine properties of a program [15]. Static analysis is an umbrella term for all methods of program analysis which are conducted by examining a program without subjecting it to any concrete executions. Conversely, dynamic analysis comprises all of the program analysis techniques which are based on concrete executions of the studied program. The disadvantage of static analysis techniques is that they are limited in precision because of the difficulty in statically resolving indirect branches [9]. On the other hand, dynamic analysis techniques suffer from poor coverage and scalability [9].

Binary analysis platforms which leverage static analysis techniques usually constitute of a hardware-independent representation, a mechanism to translate architecture dependent binaries to this representation as well as several architecture independent analyses that processes the hardware-independent representation [5]. The intermediate representation can be analyzed using abstract interpretation [11]. In this thesis, abstract interpretation is applied to over-approximate the control flow of the subject binary. For readers unfamiliar with abstract interpretation, it will be briefly introduced in section 2.4.

2.3.1 Disassembly in Static Analysis

A core issue related to the analysis of programs is the problem of distinguishing code from data [16]. As a consequence, static analysis techniques require some way to identify what bytes of a binary should be disassembled. These techniques are referred to as disassembly techniques. Disassembly techniques can, in general, be categorized as either being based on a linear sweep approach or a recursive traversal approach [17]. It should, however, be noted that none of these two approach are 100% precise [16]. Furthermore, there are also disassembly techniques which can not be categorized into these two categories, such as speculative disassembly [1] or on-demand disassembly.

Linear sweep approaches starts by disassembling the first byte at the start of the code section and then simply disassemble one byte after the other until an illegal instruction is encountered [16]. This makes linear sweep based disassemblers easy to thwart as an attacker can simply place a jump in front of some malformed instructions. This would make the control flow follow the jump instruction to pass the malformed code but could stop linear sweep disassemblers as they would be unable to disassemble the malformed instructions [16]. Some examples of linear disassembly tools are objdump, WinDbg and SoftICE [18].

Recursive traversal approaches starts at the beginning of the code section as linear sweep approaches. However, instead of treating branch instructions as fall-through edges, recursive traversal approaches proceed by following each branch instruction encountered in a depth-first or breadth-first manner. Some examples of popular tools that use recursive disassembly are IDA Pro, OllyDbg and PEBrowse Professional [18].

2.3.2 Challenges of Static Analysis Based Control Flow Reconstruction

Apart from the difficulties related to disassembly, static analysis also faces other challenges. One such challenge is the number of instructions in CISC architectures, such as x86, as these architectures normally have a very large amount of instructions and contains many special-

ized instructions for various specific tasks [19]. For example, there are over 300 SIMD instructions in x86 for performing fast vector operations on multiple bytes or words at once [20]. A static analysis which aims to over-approximate control flow can not simply ignore instructions but has to over-approximate all possible instructions on at least a coarse level.

In addition to the large amount of instructions, some architectures, such as x86, allows instructions of varying lengths. As such, instructions do not have to be aligned and overlapping instructions may thus exist. For example, if a 4 byte instruction exists at address x, a 3 byte instruction could possibly be hidden at address x+1. This is challenging for recursive and linear disassemblers but can be handled easier using on-demand disassembly as performed by Jakstab which is used to perform the over-approximation in this thesis.

Static analysis can be further complicated when analyzing malware as malware can use techniques that obfuscate the binary representation of a program [21]. As malware does not have to comply to calling conventions, they can abuse calls and returns to complicate attempts of reverse engineering. For example, a jump instruction such as *jmp eax* can be expressed as a push to the stack *push eax* followed by a return instruction *ret*. Additionally, malware sometimes leverages self-modification techniques where the instructions are modified on the fly [11]. As such, all instructions do not exist in the binary file but are synthesised during execution, making this part of the execution troublesome to analyze.

Finally, a last challenge is analyzing instructions which use data stored in registers or memory whose content is unknown. These type of instructions are normally memory updates using pointer dereferences or indirect jump instructions which was described in section 2.1 [11]. Memory updates to unknown locations are particularly bad as they imply that an over-approximation has to assume that any location in memory might have been modified. As such, all known memory content has to be discarded if the approximation is to be kept sound.

2.3.3 Dynamic Control Flow Reconstruction

An approach to under-approximating the control flow of a binary is to use dynamic analysis. This can be performed by executing the binary with some carefully generated input and adding the execution trace to the CFA. To craft this input, it is possible to use fuzzing techniques.

Fuzzing techniques are in general categorized as black-box, gray-box or white-box techniques [22]. Black-box techniques are techniques which treats the binary as a black box. Thus, black-box techniques only observes the input and output of the binary. Conversely, white-box techniques are techniques which exploit knowledge of the internals of the binary. Furthermore, Gray-box techniques are techniques which can not be categorized as purely white or black-box as they uses a mixture of the two techniques. As black box fuzzing typically exhibits low coverage, black-box techniques are typically not of interest in the context of control flow reconstruction [22].

A popular gray-box technique is mutation-based fuzzing. Mutation based fuzzers require an initial seed which is an initial well-formed input. The seed is given to the binary and some information from the execution is obtained. This information is then used to create new seeds by mutating existing ones. Common mutation techniques are bit-flipping, arithmetic mutation, block-based mutation and semantic mutation [22].

White-box techniques are usually some sort of guided fuzzing, mutation or concolic execution [22]. Guided fuzzing encompasses all techniques which leverage static or dynamic program analysis techniques for enhancing the effectiveness of the fuzzing process. Mutation includes techniques which modify the binary in a way that benefits the generated input seeds. For example, mutation can be used to remove a checksum check to ensure that all input pass the check [22].

During concolic execution, which is also known as dynamic symbolic execution, the program is initially executed with an initial concrete input. The input is then given to the binary in what is denoted as a concrete execution. During the concrete execution, symbolic expressions are recorded for the passed branches. Thereafter, an SMT solver

```
1  void check(int x):
2      if x == 25:
3          printf("x is 25\n");
4      else:
5          printf("x is not 25\n");
```

Figure 2.5: *An example where random black-box fuzzing will have a low probability of exploring all possible branches.*

can be given a variation of the obtained constraints to create a set of inputs that would take a specific path in the binary. This technique has the advantage of being more likely to generate inputs that improve coverage than random inputs [22]. On the other hand, a disadvantage of using white box techniques, in general, is that they are often slower than gray or black-box techniques [22].

Constraint-based fuzzing techniques, such as concolic execution, focus on solving constraints to create inputs that take different paths in the binary. Consequently, this types of approaches could potentially be a more attractive technique than mutation based fuzzing when reconstructing the control flow of a binary as they make it is easier to analyze narrow input ranges.

For example, consider the c code in Figure 2.5. This figure contains an if statement which is only taken if the value of the input x equals 25. Assume that x is supplied directly from standard input as the first 4 bytes. Then, for random black-box testing, assuming 32-bit integers, there would be a probability of $\frac{1}{2^{32}}$ that x would be 25 and the first print statement would be executed. However, if white-box fuzzing was used, this probability could be increased dramatically by examining the internals of the program.

In this thesis, DSE is applied to under-approximate successors of indirect jumps which could not be precisely over-approximated using abstract interpretation. The difference between DSE and conventional symbolic execution is that DSE guides the symbolic execution along a set of paths. For instance, in the example above, a path to the check function and if statement could be followed to obtain a symbolic state at the if statement. Then, the symbolic constraints for the next value

of the IP could be sent to an SMT solver which would return a set of solutions to the constrains corresponding to successor locations.

2.4 Abstract Interpretation

Abstract interpretation is a technique where concrete objects are represented as abstract objects. In general, there is a function, denoted by α, converting a concrete element into its corresponding abstract representation. This function is normally referred to as the abstraction or representation function. Conversely, there is a function, denoted by γ, converting an abstract object into its corresponding concrete object. This function is referred to as the concretization function

The reason for not carrying out analysis directly using concrete semantics is that this results in a state explosion as analysing a program in all of its possible execution environments is an undecidable problem[23]. Consequently, all non-trivial questions about the concrete semantics of a program are undecidable. However, if abstraction is used to only analyze properties which are of interest to the analysis, it can be possible to analyze the selected properties in finite time.

To be able to understand abstract interpretation, we introduce partially ordered sets[24] and Galois connections [25]. A partially ordered set (poset) consists of a set and a binary relation such that the binary relation, given two elements, indicates which of the elements should precede the other or that neither of them precedes the other. Additionally, the binary relation must be reflexive, anti-symmetric and transitive. The difference between partially and totally ordered sets is that the former allows elements in the set to be incomparable while the latter requires every pair of elements, belonging to the set, to be comparable.

A Galois connection is a binary relation between two partially ordered sets. There are two types of Galois connections: monotone and antitone. This thesis uses monotone Galois connections. Consequently, when a Galois connection is mentioned, it refers to a monotone Galois connection.

Definition. ***Galois connection***
Let $(A, \sqsupseteq_A)$ *and* $(B, \sqsupseteq_B)$ *be two partially ordered sets. Furthermore, let* γ *and* α *be monotone functions such that* $\gamma : A \to B$ *and* $\alpha : B \to A$*. Then,* $(A, \sqsupseteq_A)$*,* $(B, \sqsupseteq_B)$*,* γ *and* α *form a Galois connection if for all* $a \in A$ *and* $b \in B$*:*

$$\gamma(a) \sqsupseteq_B b \text{ if and only if } a \sqsupseteq_A \alpha(b)$$

In abstract interpretation, a Galois connection is established between a concrete and an abstract domain. The concretization function γ and the abstraction function α used to map elements in the abstract domain with elements in the concrete domain and vice versa. Furthermore, the two posets in the definition correspond to the abstract and concrete domain. In abstract interpretation, these posets are typically bounded lattices.

A lattice is a poset where every pair of elements have a unique least upper bound and a unique greatest lower bound. The least upper bound is also known as the supremum or join. Similarly, the greatest lower bound is also known as the infimum or meet. Furthermore, if a poset satisfies the requirement that each pair of elements have a unique least upper bound it is a join-semilattice.

Similarly, if a poset satisfies the requirement that each pair of elements have a unique greatest lower bound it is a meet-semilattice. Thus, a lattice could also be defined as a poset which is both a join-semilattice and a meet-semilattice. A bounded lattice is a lattice which has a top element $\top$ and a bottom element $\bot$ such that, for each lattice element x, $\bot \leq x \leq \top$. Equivalently, a lattice is a bounded lattice if and only if every finite set of elements has a join and a meet. In this work, the posets of the abstract domains are join semi-lattices.

In abstract interpretation, states transitions are normally represented by a transfer function. The transfer function of the concrete semantics maps an element of the concrete domain to other elements of the concrete domain. Similarly, the transfer function of the abstract semantics maps an element of the abstract domain to other elements of the abstract domain.

In addition to a transfer function, an abstract domain normally includes a widening operator [26]. The widening operator combines two elements of the abstract domain into another element of the abstract

domain such that the resulting element is an upper bound to the first two elements. As such, a widening operator can be used to increase the performance of an abstract interpretation based analysis at the expense of accuracy.

In this thesis, we form a Galois connection between a concrete and abstract domain. Each element of the poset for the concrete domain corresponds to a set of concrete states, where a concrete state is an actual state of the CPU and its execution environment. More formally, each element of the poset for the concrete domain is an element of $\mathcal{P}(S)$ where S is the set of all possible concrete states and $\mathcal{P}$ is the power set of S. Furthermore, the binary relation for the concrete domain is set inclusion.

Each element in the poset of the abstract domain is an abstraction of a set of concrete states. The binary relation of the abstract domain indicates if one abstract domain element is subsumed by another. The semantics of this binary relation is dependent on what abstract domain is used. Additionally, the abstract transfer function is represented by the $\widehat{post}$ operator of the selected abstract domain. As such, different abstract domains has different abstract transfer functions. The abstract domains that were used in this thesis were based on constant propagation and strided intervals which will be explained in further details in section 2.7.

2.5 Data Flow Analysis

Data flow analysis is the problem of, for each program location, assigning a lattice element which over-approximates the possible concrete states at that program location [27]. Usually data flow analysis uses the program's BBG to determine to which parts of the program, values of variables might propagate. The most precise over-approximation can be determined by finding the least fixpoint. This can be performed by applying the transfer relation to abstract states and join the result with previous results at the corresponding program location. To determine whether the fixpoint has been reached or not, it is common to use a worklist which stores data flow facts [27].

```
1  in[B_s] := X_0
2  for each basic block B
3      out[B] := ⊥
4
5  worklist := {B: ∀ basic blocks B}
6  while(worklist ≠ ∅)
7      B := remove(worklist)
8      in[B] := ⊔ {out[B']: B' ∈ pred(B)}
9
10     if out[B] ≠ F_B(in[B])
11         worklist := worklist ∪ succ(B)
12         out[B] := F_B(in[B])
```

Figure 2.6: *Example of a forward flow analysis worklist algorithm.*

Techniques for data flow analysis can be categorized into two categories; forward flow and backward flow analysis. In forward flow analysis, each exit node of a basic block is expressed as a function of the entry node of the basic block. Conversely, in backward flow analysis, entry nodes are expressed as functions of exit nodes of the corresponding basic blocks. This function is referred to as the transfer function.

The semantics of the transfer function is defined by the semantics of the instructions inside the basic blocks. Forward analysis typically requires an initial entry state to be defined. This entry state is used as the starting point of the propagation of data flow facts and can, for example, contain the fact that no variables have any known values. Similarly, backward flow analysis requires an initial exit state to be specified. Examples of forward and backward flow analysis techniques are reaching definitions [28] and liveness analysis [29] respectively.

It is possible to conduct data flow analysis techniques on a different granularity than basic blocks. For example, they can be applied on the instruction level by using a relation between the entry node and exit node of each instruction. This is normally referred to as local analysis [30]. However, it is more common to work at the basic block level and then use local analysis inside basic blocks when instruction-level data flow information is desired [30]. In this thesis, we do not reconst-

ruct a BBG. Rather, we aim to reconstruct a precise CFA. As such, the transfer function of our abstract domain is defined by semantics at the instruction level.

An example of a worklist algorithm for a forward flow analysis is provided in Figure 2.6. This algorithm assumes that a BBG is available to determine the predecessors pred(B) and successors succ(B) of a basic block B. Furthermore, the information of the entry and exit of a basic block B is denoted by in[B] and out[B] respectively. Initially, the input information of the starting block B_s is set to a specific pre-defined value X_0. For each basic block, the exit information is then set to the empty lattice element $\perp$, signifying that no information is known. The worklist is then initialized with all basic blocks of the BBG and the while-loop is entered.

In the while loop, a random basic block B is removed from the worklist. The entry information of this block is then recalculated depending on the information of its predecessors using the join operator. Given a set of lattice element, the join operator creates a lattice element which is the least upper bound of the lattice element in the given set. Thus, the resulting element is an over-approximation of the possible data flow facts at the basic block B. Then, the algorithm checks if the new output of B, obtained by applying the transfer function F_B to the input information of B, is different from the previous output information of B. If this is the case, the output information of B is updated with the new generated information and all successors of B are re-added to the worklist as their input information is dependent on the output information of B.

Eventually, the join will not result in any new information for the in[B] of a basic block B. Consequently, the output of the basic block is unchanged and no blocks are added to the worklist. When this has happened to all basic blocks and the worklist is empty, a fixpoint has been reached. This means that for each basic block B, out[B] over-approximates the possible data flow facts outputted from B.

As data flow requires the BBG whose reconstruction is dependent of data flow facts, it is not trivial how to reconstruct one without the other. However, Kinder [11] devised an algorithm that can perform CFA reconstruction and deductions of data flow facts simultaneously. In this thesis, this algorithm is used for the over-approximation part of the alternating approach presented in chapter 3.

2.6 Over-Approximation with Jakstab

Jakstab [11] is an abstract interpretation based framework which combines control flow and data flow analysis to solve the chicken and egg problem of control flow reconstruction. It is highly configurable and thus allows for a wide range of analyses. Additionally, the disassembly is performed on-demand and can thus, contrary to conventional disassemblers, disassemble overlapping instructions.

To abstract the details of assembly language while still capturing its relevant branching behaviour, an intermediate language (IL) is leveraged. The author of the platform defines a set of basic and abstract statements denoted by *Stmt*. Assume that $e_1, e_2 \in Exp$ where *Exp* is the set of possible expressions. Then, the set of basic statements consists of b-bit memory assignments $m_b[e_1] := e_2$ to a location e_1, conditional indirect jumps $if\ e_1\ jmp\ e_2$ where the *IP* is set to e_2 if e_1 evaluates to true, assume statements *assume e*1 and a halt statement *halt*.

Furthermore, the abstract statements are statements which do not correspond to regular assembly instructions. Rather, they are used to abstract certain behavior of the execution environment [11]. Finally, as the intermediate language is free from side-effects, each assembly instruction of the binary is mapped to one or more basic and abstract statements. As such, each statement can be identified by a program location and an index.

The only data type of the IL are integers. Consequently, booleans are represented by 0 and 1 corresponding to false and true respectively. Furthermore, the finite set of program locations is denoted by $L \subseteq \mathbb{N}$. This set contains all addresses of the program. A program is denoted as $< \ell_0, P >$ where ℓ_0 is the starting location and P is a finite mapping

from program locations to statements and successors. More formally, P is defined as $P :: L \to (Stmt \times L)$. Furthermore, the statement $stmt$ at location $\ell \in L$ with successor ℓ' is denoted by $[stmt]^{\ell}_{\ell'}$.

Jakstab operates according to the guarded command language which assumes guard conditions to hold or not hold and bases the subsequent analysis on these assumption. As such, assume statements results from various types of jump instructions. For example, they can result from a *je* instruction which succeeds a *cmp* instruction which compares *eax* to the value 2. This would create two possible transitions, one where *eax* is equal to 2 and the jump is taken as well as one where *eax* is not equal to 2 and the jump is not taken. If the location of the *je* instruction is denoted by ℓ, the location of the immediately following instruction by ℓ' and the target of the jump by ℓ'', the two resulting statements are denoted by $[\textit{assume eax} \neq 2]^{\ell}_{\ell'}$ and $[\textit{assume eax} = 2]^{\ell}_{\ell''}$.

As Jakstab is based on abstract interpretation, it is configured by an abstract domain. The abstract domain is defined as $(\mathcal{A}, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{post}, \widehat{eval}, \gamma)$. The first six components $(\mathcal{A}, \bot, \top, \sqcap, \sqcup, \sqsubseteq)$ constitute a bounded lattice where $\mathcal{A}$ is the set of lattice elements. The remaining components of the lattice $\bot$, $\top$, $\sqcap$, $\sqcup$, $\sqsubseteq$ are the bottom element, top element, meet, join and partial order respectively. The next component is the abstract post operator $\widehat{post} : Stmt \to \mathcal{A} \to \mathcal{A}$ which maps a statement and an abstract state to a new abstract state. In other words, it over-approximates the concrete transfer function.

The subsequent component is the abstract eval operator $\widehat{eval} : Exp \to \mathcal{A} \to 2^{\mathbb{Z}}$ which maps an expression and an abstract state to a set of integers such that the set of integers over-approximates the evaluation of the expression in the corresponding states of the concrete domain. Finally, the last component is the concretization function $\gamma : \mathcal{A} \to 2^{State}$ which maps a lattice element to a set of concrete states. The least element of the abstract domain is mapped to the empty set.

An analysis of control flow should be able to determine the set of successors for indirect jumps based on the knowledge it acquires. In Jakstab, this is performed by the $resolve : L \to \mathcal{A} \to 2^{Edge}$ operator which, given a location and abstract state, computes a set of edges from this state. The semantics of the *resolve* operator is expressed us-

ing the $\widehat{post}$ and $\widehat{eval}$ operators of the abstract domain. Its definition is presented below. Note that the $\bot$ element represents that the analysis has not reached the location yet and thus that nothing is known about the abstract states at this location. Additionally, note that the third case evaluates the successor locations in the state $\widehat{post}[\![assume\,(e_1 \neq 0_1)]\!](a)\,)$ rather than a as this means that only states which fulfill the branch condition are included.

$$
\begin{aligned}
&resolve_\ell(a) := \\
&:= \left\{ \begin{array}{ll}
\emptyset & \text{if } a = \bot \text{ or } [stmt]^\ell_{\ell'} \text{ is } [\text{halt}]^\ell_{\ell'} \\
\{\,(\ell, stmt, \ell')\,\} & \text{if } a \neq \bot \text{ and } ([v := e]^\ell_{\ell'} \text{ or } [m[e_1] := e_2]^\ell_{\ell'}) \\
\{(\ell,\, assume(e_1 \neq 0_1 \wedge e_2 = \ell''), \ell'') \mid & \text{if } a \neq \bot \text{ and } [\text{if } e_1 \text{ jmp } e_2]^\ell_{\ell'} \\
\quad \ell'' \in \widehat{eval}[\![e_2]\!](\widehat{post}[\![assume\,(e_1 \neq 0_1)]\!](a))\,\} & \\
\cup\,\{\,(\ell,\, assume(e_1 = 0_1), \ell')\,\} &
\end{array} \right\}
\end{aligned}
$$

In some cases, the *resolve* operator might not be able to deduce one or more successors for a location $\ell \in L$. This can occur when the statement *stmt* is an indirect jump for which the target could not be over-approximated without resorting to the usage of the $\top$ element. When this happens, Jakstab warns the user that the analysis will be unsound and continues exploring the binary at other locations where the analysis has yet to reach a fixpoint. Additionally, the unresolved location ℓ is marked to have the successor $\top$, which signifies that the successor location ℓ' can be any location in the program. The $\top$ element is used as it is an upper bound for any possible lattice element in the location domain, which will briefly be introduced in section 2.7.1.

2.7 Abstract Domains

In this section the abstract domains used for the over-approximation are briefly and informally introduced. All of the abstract domains have a corresponding formal definition in Kinder's dissertation [11]. First, the location analysis domain is briefly described in section 2.7.1. Thereafter, the constant propagation domain is explained in section 2.7.2. Finally, the strided interval analysis is detailed in section 2.7.3. These sections will contain assembly examples to illustrate when they are useful. For these and other assembly code examples, each line of code will be paired with a comment to help readers unfamiliar with x86 assembly.

2.7.1 Location Analysis

The location analysis domain allows to omit location information from other abstract domains[2]. It contains a label of an intermediate level instruction, which is an address to a concrete instruction and an index as each concrete instruction can correspond to multiple intermediate level instructions. The location Analysis is never used on its own. Instead, it is always combined with another abstract domain.

2.7.2 Constant Propagation

The constant propagation domain consists of constant propagation and constant folding. Constant propagation signifies that constants are propagated through instructions so that register operands can be substituted with concrete values. Furthermore, constant folding signifies that constant expressions are calculated and substituted by the result of their evaluation.

For example, consider the synthetic program *cjmp.asm* depicted in Figure 2.7. The text section is the section containing the instructions of a binary. Its start is declared at line 1. Then, line 2 declares the *_start* label to be global, making it possible for the linker to know where the binary should start. The binary starts at line 5 by moving the constant value 0 into the register *eax*. Thereafter, through line 6 to 10, a series of registers operations are performed. These operations result in the

[2]For more information, see previous work by Kinder [11].

```
1  SECTION .text
2  global _start
3
4  _start:
5          mov eax, 0       ; eax = 0
6          add eax, 20      ; eax = eax + 20
7          mov edi, 12      ; edi = 12
8          sub eax, 30      ; eax = eax - 30
9          add edi, 10      ; edi = edi + 10
10         add eax, 10      ; eax = eax + 10
11         cmp eax, 0       ; Compare eax with 0
12         je exit          ; If ZF=1 jump to <exit>
13 unreachable:
14         jmp unreachable ; Jump infinitely
15 exit:
16         mov ebx, 0       ; Set exit code to 0
17         mov eax, 1       ; Set eax to sys_exit
18         int 0x80         ; Syscall
```

Figure 2.7: *cjmp.asm - An example containing a conditional jump which can be resolved without false positives using the constant propagation domain.*

values 0 and 22 being stored in the *eax* and *edi* register respectively. The *cmp* instruction at line 11 is then executed to set the flags according to the comparison between eax and 0. Since eax is 0, the zero flag *ZF* is set to one[3]. The instruction at line 12 performs a jump to *exit* if the *ZF* flag is set and thus results in a jump to *exit* where the program terminates.

Evaluation using the constant propagation domain starts with an empty state. The first instruction is then parsed into an instruction $eax = 0$ and the value of 0 is stored in the register eax. Then, the next instruction is parsed, resulting in the instruction $eax = eax + 30$. At this point, the value $eax = 0$ is propagated to this instruction, turning it into the instruction $eax = 0 + 20$ which is simplified to $eax = 20$ and stored in the new state. The next instruction, at line 7, is parsed and the state becomes *{eax=20,edi=12}*.

[3]The *ZF* flag is set when the subtraction performed in the *cmp* instruction results in zero. In essence, the *ZF* flag is set if the two compared values are equal.

The next instruction *eax = eax - 30* is loaded, *eax* is substituted with 30 and constant folding is performed to obtain the state *{eax=-10,edi=12}*. Thereafter, the same procedure repeats for the last two instructions *edi = edi + 10* and *eax = eax + 20* resulting in the state *{eax=0,edi=22}*. Now, the constant value of *eax* can be propagated to the succeeding compare instruction substituting *eax* in the expression and resulting in the instruction *cmp 0,0*. Consequently, the *ZF* flag can be determined to be one and the jump instruction at line 12 can be determined to only have the successor at line 16, eliminating a potential false positive edge in the CFA to the unreachable jump instruction.

While constant propagation has the benefit of being able to scale well to larger programs [10], it has the disadvantage of not being able to resolve successors of return instructions. The reason is that constant propagation does not assume any initial values for memory locations and registers and thus does not know the initial value of the stack pointer which is in turn required to capture the effect of the *call* instruction. Consequently, when the *ret* instruction is later executed, the address at the top of the stack is unknown. Another limitation of constant propagation is that it only allows each state to have one possible value per register and memory location. This limitation will be illustrated by the example in the next section where it will be shown when it can be more appropriate to use the strided interval domain rather than the constant propagation domain.

2.7.3 Strided Intervals

The elements of the strided interval domain consist of tuples with three elements. These three elements are integers for a minimum, maximum and a stride value. More formally, a strided interval is defined as:

Definition. ***Strided Interval***
A strided interval is an interval $s[l,u] = \{l + s * n \mid l \leq l + s * n \leq u, n \in \mathbb{N}_0\}$ *where* $s, l, u \in \mathbb{N}_0$ *and* $\mathbb{N}_0$ *is the set of natural numbers including the zero element.*

The stride of a strided interval can be thought of as the difference between two adjacent elements of the interval. For example, the interval $2[4,8]$ is equivalent to the set $\{4,6,8\}$ since it contains the lower bound 4 and values separated by a stride of 2 up to the upper bound 8.

```
1  SECTION .bss
2  buf      resb 4                ; Reserve 4 bytes of
3                                 ; memory
4  SECTION .text
5  global _start
6
7  increment:
8          add ebx, 1            ; ebx = ebx + 1
9          jmp continue          ; Jump to <continue>
10 _start:
11         mov  edx, 4           ; Length of buffer
12         mov  ecx, buf         ; Pointer to buffer
13         mov  ebx, 0           ; Standard input
14         mov  eax, 3           ; sys_read
15         int  0x80             ; Perform syscall
16
17         mov eax, dword [buf]  ; eax = ⊤
18         mov ebx,0             ; ebx = 0
19         cmp eax,1             ; Compare eax with 1
20         je increment          ; If eax = 1 jump
21                               ; to <increment>
22 continue:
23         add ebx, exit         ; ebx = ebx + <exit>
24         jmp ebx               ; Jump to ebx
25 exit:
26         nop                   ; No operation (1 byte)
27         mov ebx, 0            ; Set exit code to 0
28         mov eax, 1            ; sys_exit
29         int 0x80              ; Perform syscall
```

Figure 2.8: *branch.asm - An example of when strided intervals can be more useful than constant propagation.*

As a consequence, conventional intervals can be represented as strided intervals where the stride is equal to 1. Furthermore, a stride of 0 signifies a singleton set containing only the lower bound of the interval, for example, $0[2, 2] = 2$.

Strided intervals are useful to represent states at program locations where a register or memory location can contain more than one value. This will be illustrated through an example. Consider the assembly code of *branch.asm* depicted in Figure 2.8. In addition to the text section, the Basic Service Set (BSS) section is instructed to reserve four bytes

for the program at line 2. When executed, the program begins at the *_start* label and starts by setting up the registers to perform a system call using *sys_read*. At line 15, the CPU is instructed to perform the system call to read four bytes from stdin into the buffer buf which is located in the BSS section. This is done by performing a context-switch where the CPU is put in kernel mode where it has additional privileges and can perform the read into the buffer. The CPU then performs the read followed by a transition into user mode and a context switch resuming the execution of the program at the instruction after the syscall instruction.

At line 17 the content of *buf*, which contains the first 32 bits of the user's input, are copied into the 32-bit register *eax*. Thereafter, the *ebx* register is initialized to 0, as it will be used as a counter. Then, *eax* is compared with 1 to determine if control should be transferred to the *increment* label or directly to the *continue* label. In the first case, the CPU executes line 8 to increase *ebx* by 1 and then jumps to line 23. At line 23 the offset of the *exit* label is added to the *ebx* register. Afterwards, a jump to the content of *ebx* is performed. At this point, *ebx* is either equal to *exit* or *exit + 1*. Note, however, that the instruction located at *exit* is a *nop* instruction of size 1 byte. Consequently, the CPU will execute either the *nop* instruction and continue from line 27 or continue from line 27 directly. Thus, the semantics of the program are independent of the user input.

At the start of execution the interval analysis starts with a state only containing a strided interval with a stride of 0 for the stack pointer. After executing the syscall, each of the four registers which are used to pass parameters to the syscall, are represented by intervals of a stride of 0. Then, the *mov* instruction at line 17 is executed to load the content of the buffer into the *eax* register. However, as it is impossible to determine what user input is entered without actually executing the program, it is not possible to know what *buf* will contain after the syscall and thus not what will be loaded into the *eax* register at line 17. Consequently, the strided interval for *eax* will be removed from the state, i.e *{eax=0[3,3], ebx=0[0,0], ecx=0[buf,buf], edx=0[4,4]}* becomes *{ebx=0[0,0], ecx=0[buf,buf], edx=0[4,4]}*.

After loading the content of buf into *eax*, the *ebx* register is set to 0 and then the eax register is compared with 1. However, as there is currently no knowledge of eax in the state, it has to be assumed that all different flag values are possible after this statement. Consequently, when the *je increment* statement is reached, it has to be assumed that execution can continue on either line 8 or 23. Consequently, the state is split into one state where *ZF=1* and the execution continues at line 8 as well as one where *ZF=0* and the execution continues at line 23.

The analysis which continued on line 8 will increase the value of *ebx* by 1 by changing the strided interval *0[0,0]* to *0[1,1]*. Then, the jump instruction is taken to line 23. Now there are two states which can be merged at this location, resulting in a state containing a wider strided interval *ebx=1[0,1]]*. The *exit* label is then added to the interval of *ebx* and the indirect jump to the content of *ebx* is performed. Since, a strided interval can model the two possible values of ebx, it can be determined that *jmp ebx* results in a jump to either *nop* or *mov ebx, 1* at line 26 and 27 respectively. The state is then propagated through the last instructions to the system call *sys_exit* at line 29.

If constant propagation would have been used to analyse the binary, the two states at line 23 would not have been possible to merge. The reason is that they would have two different values for *ebx* and constant propagation only allows a state to hold at most one value per register and memory location.

2.8 Related Work

There have been many approaches to control flow graph reconstruction through the last couple of centuries. However, to our best knowledge, no one has succeeded in developing an algorithm that can construct the ideal CFG for an arbitrary binary. There are, however, many platforms that can create approximations of the ideal CFG. The approach described in this thesis differs from others in the sense that it gives up on properties such as soundness and accuracy to focus on precision. Below, other previous approaches to control flow reconstruction are described.

angr [7] is a state-of-the art binary analysis platform for exploit generation. It was created as part of the DARPA challenge and was used by the team Shellphish. It is heaviliy modularized and has borrowed the intermediate language VEX from Valgrind, a platform which focuses on finding memory leaks in programs. The angr project contains two CFG reconstruction algorithms named *CFGFast* and *CFGAccurate*. The *CFGFast* algorithm is a purely static analysis algorithm while *CFGAccurate* leverages dynamic analysis through its dynamic symbolic execution engine.

The *CFGAccurate* algorithm uses techniques such as forced execution, backwards slicing, symbolic execution and value set analysis to create an accurate representation of the CFG. The angr *CFGAccurate* algorithm is similar to our approach in the sense that it does not provide any soundness or accuracy guarantees and uses symbolic execution to under-approximate the control flow. However, a difference is that our approach represents the control flow by a CFA rather than a BBG and thus could be able to express control flow more precisely. Additionally, the approach presented in this thesis works with a different set of abstract domains.

Another approach, presented by Xu et al. [1], leverages forced execution to force the execution of specific control flow paths. Their algorithm approximates the control flow by manipulating concrete executions and distinguishes itself from many other approaches by working directly with machine code rather than using an intermediate representation of the binary. More specifically, the algorithm executes the program on a selected starting input and records snapshots of the program state at every jump instruction. Once the initial concrete execution finishes, the program state is reset to one of the earlier snapshots and resumed.

This process of restoring snapshots and resuming execution continues until every possible execution has been performed. A disadvantage with their approach is that the algorithm does not accurately under-approximate the control flow since it can force execution of infeasible edges. On the other hand, if the infeasible edges added through the forced execution of infeasible paths would be excluded, their algorithm would, in theory, reconstruct the optimal control flow. However, due to

several problems such as infinite loops, their algorithm can not be guaranteed to finish in finite time. Consequently, they adapt the algorithm to be more practical at the expense of the guarantee of exploring every possible CFG edge. Our approach presented in this thesis also becomes complicated when loops are introduced, as will be seen in section 4.4.2.

Another approach, introduced by De Sutter et al. [15], attempts to represent infeasible branches through hell nodes and hell edges. The paper defines a hell node as a node which represents all possible targets of an unresolved indirect jump. Further, they define hell edges to be edges connected to the hell node. They proceed with their analysis by noting that indirect jumps have five major sources: switch-statements implemented with a table lookup and an indirect jump, exception handling, goto statements to pointer assigned labels as well as the use of procedure pointers and dynamic method invocation in object oriented languages.

They are able to resolve all indirect jumps of switch cases by trying to match program slices of the part of the program calculating the jump target, with common instruction sequences for switches. For all other jumps they try to obtain the targets using constant propagation. Their work shows that matching known sequences of assembly instructions is a good way to resolve indirect branches stemming from switch statements.

An alternative to program slicing and constant propagation is to use abstract interpretation as in the CFG reconstruction algorithm developed by Bogdan Mihaila[12]. This algorithm soundly over-approximates the control flow in a top-down manner. Their analysis heavily focuses on resolving indirect jumps stemming from jump tables. They define an interval domain and a set domain using abstract interpretation.

These domains are then used to over-approximate the targets of the indirect jumps. A similarity between their work and ours is that they represents the CFG as a CFA rather than a BBG. Conversely, a difference is that they assume data structures in the binary to conform to source-code-level data structures, which is not a requirement for the approach presented in this thesis.

Among the static analysis approaches there are also approaches which do not rely on abstract domains as their primary analysis method. One such approach was presented by Reinbacher et al. [9]. In their work, they linearly disassemble the binary into instructions that are then grouped into basic blocks. Afterwards, they express the semantics of each basic block using propositional Boolean logic. Thereafter, each basic block is converted to its corresponding CNF form using Tseitin conversion. Based on these encodings, they implement forward and backward value-set analyses using existential quantification and incremental SAT solving.

The forward and backward traversal analyses can be applied to generate pre and post conditions for each binary block. Each pre and post condition contains a set of possible values for each register before and after the block respectively. The backward traversal is limited to k basic blocks, where k is a tunable parameter. Consequently, it is possible to tune to what extent previous blocks should be used when calculating pre and post conditions for a basic block. Furthermore, this offers a trade-off between speed and precision which can be useful in various applications which require less of one or the other.

As they use incremental SAT solving to over-approximate value sets formulated as pre and postconditions, an advantage of their approach is that the performance of the algorithm will benefit from performance increases of SAT-solvers in the future. Finally, the results of their empirical experiments indicate that the forward-backward analysis introduced can, not only be more precise, but also faster than pure forward analysis. The authors concluded that this is because the value sets propagated around the program tend to be smaller.

A state of the art platform for static analysis and control flow reconstruction is Jakstab [31], developed by Kinder [11]. This platform uses abstract interpretation to, under very general assumptions, simultaneously determine the control flow and data flow properties of a binary. This is conducted by using an intermediate language, on-demand disassembly, an extended version of the classic worklist algorithm and bounded address tracking. Bounded Address Tracking is a highly precise abstract domain that models registers and memory locations as both pointers and integer values.

In a later work, Kinder et al. extended Jakstab to work with alternating over/under-approximation [10]. In this work, they created an abstract domain representing the combination of their over- and under-approximation approaches. For the over-approximation they used the bounded address tracking domain of the Jakstab platform. Furthermore, for the under-approximation, they extended Jakstab to support replaying execution traces. The execution traces were recorded from executions of the binary with different input in qemu. The targets of branches which could not be approximated through over-approximation could then potentially be under-approximated with the traces obtained from the emulation in qemu.

In this thesis, the Jakstab tool is extended by using symbolic execution techniques for the under-approximation part of the alternating over/under-approximation approach. More specifically, directed symbolic execution is applied to determine successors of a set of unresolved locations. The successors of the instructions are obtained by following paths towards the unresolved locations, obtaining a symbolic expression for the addresses of the successors for each location. Solutions for these symbolic expressions are then obtained using an SMT solver. This approach of directing symbolic execution has the benefit of avoiding the path explosion problem of naive symbolic execution approaches.

Chapter 3

Methodology

Reconstructing the CFA of an arbitrary binary is generally considered to be hard [11]. As there appears to be a lack of earlier work performing alternating CFA reconstruction using DSE and abstract interpretation, this was perceived as an attractive approach to explore. In this chapter, our approach of alternating over/under-approximation is presented. Initially, the assumptions of the analysis are stated and examples of use cases are provided. Thereafter, the required modifications of the CPA algorithm are detailed and subsequently, the under-approximating directed symbolic execution is explained. Afterwards, examples are given of when DSE can be useful. Finally, the last sections provide details of the metrics used for the evaluation, an overview of the implementation as well as details of the experiments.

3.1 Limitations and Target Binaries

Since there are no restrictions on the target binaries except for the assumptions of Jakstab and Manticore, the presented approach can target all types of binaries that satisfy these assumptions. The following are the assumptions made by Jakstab:

- The global memory region, the stack, and the heap are assumed to not overlap.
- The stack pointer is initialized to a value of $(stack, 0)$
- No pointers escape their allocated memory bounds
- Values of arbitrary bit length can be stored at a single store address and no memory locations overlap

These assumptions are required for Jakstab to output a sound approximation of the control flow. However, they are not required for Jakstab to output a useful CFA. In other words, even if one or more assumptions are broken, the reconstructed CFA might still be not be too far from the ideal one.

As for Manticore, we do not fully understand the assumptions and limitations of the tool as the documentation lacks in this aspect. However, one of the limitations is that it only models some of the system calls and symbolic execution of binaries using other system calls will fail [32]. Additionally, it loads libraries from the host system, which could lead to different results on different hosts if the libraries differ [32].

As the final CFA will not necessarily be sound or accurate, it will not be of use for binary analysis which requires these properties. For example, it is not suitable for formal verification of binaries as this field requires sound over-approximations. Instead, it is of more use when the goal is to recreate a precise CFA at the cost of the soundness and accuracy properties. For example, more precise CFAs at the cost of soundness are useful in fields such as reverse engineering as reverse engineers' main desire often is a close to the ideal CFA.

3.2 Modifications to the CPA Algorithm

The Jakstab tool, which is used to over-approximate the control flow, builds upon the Configurable Program Analysis (CPA) framework devised by Beyer et al. [27]. The novelty of the CPA framework was that it was capable of combining model checking and program analysis. The framework was parameterized by one or more abstract domains, a merge operator merge and a termination check stop. The merge operator was used to determine if two abstract states should be merged or not. Furthermore, the termination check indicated if a state should be explored or not by checking if it is a subset of the states that have already been reached.

```
1  Input: A CPA with dynamic precision adjustment
2  D := (D, Π, ⇝,merge,stop,prec), an initial abstract state a_0 ∈ A with
3  precision π_0 ∈ Π and a CFA as a set of edges G.
4  Output: A set of reachable abstract states.
5  worklist := {(a_0, π_0)}
6  reached := {(a_0, π_0)}
7  while worklist ≠ ∅
8      pop (a, π) from worklist
9      (â, π̂) := prec(a, π, reached)
10     foreach a' with ∃g ∈ G. â ⇝^g (a', π̂)
11         // Combine with existing abstract states
12         foreach (a'', π'') ∈ reached
13             a_new := merge(a', a'', π̂)
14             if a_new ≠ a''
15                 worklist := (worklist ∪ (a_new, π̂)) \ {(a'', π'')}
16                 reached := (reached ∪ (a_new, π̂)) \ {(a'', π'')}
17         // Add new abstract state?
18         if ¬ stop(a', {a|(a, ·) ∈ reached}, π̂)
19             worklist := worklist ∪ (a', π̂)
20             reached := reached ∪ (a', π̂)
21 return {a|(a, ·) ∈ reached}
```

Figure 3.1: *The original CPA+ Algorithm.*

Later on, Beyer et al. extended their algorithm to also include dynamic precision adjustment and referred to the new version of the algorithm as the CPA+ algorithm [33]. The CPA+ algorithm, presented in Figure 3.1, is a variant of a standard worklist algorithm. The algorithm is parameterized by a configurable program analysis $\mathcal{D} = (D, \Pi, \rightsquigarrow$,merge,stop,prec) which consists of an abstract domain D, a precision domain Π, a transfer function $\rightsquigarrow$, a merge operator merge, a stop operator stop and a precision operator prec. These components influence the cost of the analysis and the precision of the resulting set of reached states.

Apart from the configurable program analysis, the CPA+ also requires a CFA as input. The output of the algorithm is, as for the original CPA, a set of reached abstract states. The algorithm works with a worklist and a set of reached abstract states initialized with only the first abstract state and its corresponding precision element. After the initialization, the algorithm contains a loop which loops until the worklist is empty, at which point, the set of reached abstract states is fully populated.

The loop starts by popping an abstract state and precision element of the worklist at line 8. It then uses the prec operator to refine the precision of the abstract state and precision element by using information available in the already reached abstract states. Thereafter, at line 10, the algorithm iterates over each successor of the current abstract state $\hat{a}$. Each successor a' of the abstract state corresponds to an edge g in the CFA G. Furthermore, the $\overset{g}{\rightsquigarrow}$ represents the transfer function which corresponds to the edge g.

For each successor a' of the current state, the analysis proceeds by iterating over each reached state. For each reached state, the algorithm merges this state with the successor state using the merge operator, creating the merged state a_{new}. The algorithm then checks if this merged state is equal to the successor state. If this is not the case, the merged state and its precision element is added to the worklist and the set of reached states.

Once the algorithm has checked if the successor state can be merged with each reached state, it proceeds to perform a termination check using the stop function. The stop function returns a boolean indicating

```
1  Input: A CPA 𝒟 := (D, Π, ⇝,merge,stop,prec), an initial abstract state
2  a_0 ∈ A with precision π_0 ∈ Π and a map P from locations to statements.
3  Output: A CFA of P and a set of reachable states.
4  worklist := {(a_0, π_0)}
5  reached := {(a_0, π_0)}
6  G := {}
7  while worklist ≠ ∅
8      pop (a, π) from worklist
9      (â, π̂) := prec(a, π, reached)
10     foreach g ∈ getTransformers(â,P)
11         foreach a' with â ⇝^g (a', π̂)
12             G := G ∪ g
13             // Combine with existing abstract states
14             foreach (a'', π'') ∈ reached
15                 a_new := merge(a', a'', π̂)
16                 if a_new ≠ a''
17                     worklist := (worklist ∪ (a_new, π̂)) \ {(a'', π'')}
18                     reached := (reached ∪ (a_new, π̂)) \ {(a'', π'')}
19             // Add new abstract state?
20             if ¬stop(a', {a|(a, ·) ∈ reached}, π̂)
21                 worklist := worklist ∪ (a', π̂)
22                 reached := reached ∪ (a', π̂)
23 return G, {a|(a, ·) ∈ reached}
```

Figure 3.2: *The CPA+ Algorithm of Jakstab*[1].

if the analysis should continue with the successor state a' or if the data flow facts of this state has already been included in the analysis. If the stop function determines that analysis should continue with the successor state, the successor state and its precision element are added to the worklist and the set of reached states. Otherwise, the algorithm ignores the successor state and the analysis continues with the next edge.

[1]In the PhD dissertation which introduced Jakstab [11], line 6 of the algorithm was originally "*reached*(a_0(*pc*)) := (a_0, π_0);". We believe that this is a mistake as the variable *reached* is treated as an unindexed set and have thus fixed this typo in the pseudocode above.

Both the original CPA algorithm and the CPA+ algorithm required a CFA to propagate data flow facts. The contribution by Kinder [11] was that the CFA would be rebuilt simultaneously to the propagation of the data flow facts. Thus, Kinder's contribution removes the requirement of already having a CFA for data flow analysis and hence provides a potential solution to the circular dependency between data flow analysis and CFA reconstruction.

The modified CPA+ algorithm used in Jakstab, which will be referred to as the Jakstab algorithm, is presented in Figure 3.2. The main difference between Jakstab's version of the CPA+ algorithm and the original CPA+ algorithm is that the Jakstab algorithm performs control flow reconstruction on the fly rather than assuming a CFA to be available at the start of the analysis. This is conducted by integrating the *resolve* operator as a call to a transformer factory which provides state transformers.

The transformer factory uses a statement map P which maps program locations to IL statements. At line 10, the Jakstab algorithm uses the getTransformer function which, given the statement map P and an abstract state a, returns a transformer which maps an abstract state to a set of succeeding abstract states with respect to the transfer relation. Each transformer is equivalent to an edge in the CFA which is added to the set of edges G, which represents the CFA, if it is not already a part of the graph.

By using different abstract domains, the semantics of the *resolve* operator is altered. This does, in turn, affect the getTransformer function. Consequently, a switch from an abstract domain to another can potentially imply that different edges are returned by the transformer factory which, in turn, may result in a different CFA. For jump instructions, the transformer factory applies the *resolve* operator which then leverages the $\widehat{eval}$ operator to concretize the addresses of the succeeding instructions from the abstract states at the current location.

```
1  Input: A CPA 𝒟 := (D, Π, ⇝,merge,stop,prec), an initial abstract state
2  a_0 ∈ A with precision π_0 ∈ Π and a map P from locations to statements.
3  Output: A CFA of P and the set of reachable states.
4  worklist := {(a_0, π_0)}
5  reached := {(a_0, π_0)}
6  G := {}
7  unresolved := {}
8  DSEEdges := {}
9  while worklist ≠ ∅
10     pop (a, π) from worklist
11     (â, π̂) := prec(a, π, reached)
12     DSET := {(l_1, ·, ·)|(l_1, ·, ·) ∈ DSEEdges ∩ l_1 = L(â)}
13     transformers := getTransformers(â,P) ∪ DSET
14     if transformers = ∅
15         unresolved := unresolved ∪ (â, π̂)
16     foreach g ∈ transformers
17         foreach a' with â ⇝^g (a', π̂)
18             G := G ∪ g
19             //Attempt to merge with existing abstract states
20             foreach (a'', π'') ∈ reached
21                 a_new := merge(a', a'', π̂)
22                 if a_new ≠ a''
23                     worklist := (worklist ∪ (a_new, π̂)) \ {(a'', π'')}
24                     reached := (reached ∪ (a_new, π̂)) \ {(a'', π'')}
25             //Add new abstract state
26             if ¬stop(a', {a|(a, ·) ∈ reached}, π̂)
27                 worklist := worklist ∪ (a', π̂)
28                 reached := reached ∪ (a', π̂)
29     if worklist = ∅
30         paths := getPaths(G, unresolved)
31         DSEEdges := DSE(paths)
32         fixpoint := (G = G ∪ DSEEdges)
33         if fixpoint = false
34             for (a, π) ∈ unresolved such that (L(a), ·, ·) ∈ DSEEdges
35                 worklist := worklist ∪ (a, π)
36         unresolved := {}
37
38 return G, {a|(a, ·) ∈ reached}
```

Figure 3.3: *The modified version of the Jakstab algorithm which leverages DSE to resolve unresolved instructions.*

In this work, the Jakstab algorithm was modified to be able to include resolved successors of instructions marked as having unknown successors by Jakstab, through directed symbolic execution. Furthermore, as this work uses the constant propagation and strided interval domains which do not require the precision domain, the precision component of the CPA+ framework was set to be a singleton set containing only the *null* element throughout the analysis.

The first modified version of the Jakstab algorithm, which was adjusted to alternatingly include an under-approximation component, can be seen in Figure 3.3. In this pseudocode, which will be referred to as the first ACFR algorithm, the under approximation approach is assumed to be DSE. However, one could equivalently use another under-approximation approach such as fuzzing since the only thing changing would be the semantics of the DSE function and potnetially the naming of the variables .

In addition to the global variables of the Jakstab algorithm, the first ACFR algorithm requires an empty set of unresolved abstract states *unresolved* and a set of DSE edges $DSEEdges$. The *unresolved* set holds states which do not have any known successors and $DSEEdges$ contains edges which have been added by the under-approximation. The idea behind the loop that follows is to empty the worklist and ask DSE for successors when the worklist is empty but before checking the loop condition. If DSE finds new edges, their corresponding unresolved abstract states are put back into the worklist, thus ensuring that the loop condition holds for the next iteration.

The loop starts by popping an element from the worklist and adjusts its precision, as performed originally in the Jakstab algorithm. Then, the set $DSET$, which stands for DSE transformers, is populated to include all DSE edges which starts at the location of the precision adjusted abstract state $\hat{a}$. Subsequently, the $transformers$ variable is set to be equal to the union of both the DSE transformers $DSET$ and the transformers obtained through the getTransformers function.

At line 14, the if condition checks if any transformers were found for the location of the precision adjusted state. If this is not the case, the precision adjusted state is added to the set of unresolved states.

After the if condition, the two foreach loops at line 16 and 17, ensures that the algorithm loops over each successor state a' of $\hat{a}$. In the inner for loop, g is added to G if it is not already contained by G. Thereafter, between line 20 and 24, the algorithm proceeds with the merge processes similarly to the Jakstab algorithm. Furthermore, the termination check, stop, is used as in the Jakstab algorithm to check if the analysis should continue with the state a' or not.

After the loop over all transformers, an if statement was added. This if statement checks if the worklist is empty or not. If this is not the case, the algorithm will continue re-executing the code between line 10 and 28 until the worklist is empty. If the worklist is empty, we enter the if block which aims to obtain CFA edges through DSE. Initially, the getPaths function is called to obtain a set of paths *paths* from the CFA G and the set of unresolved abstract states *unresolved* which was populated during the analysis of the abstract states in the worklist at line 15. Note that the getPaths function can be any algorithm which finds a set of paths between a start node and a set of target nodes in a graph. In this work, it was set as a limited depth first search so that the analysis would be able to finish in finite time when binaries contained loops.

After obtaining the paths, the DSE function is called at line 31. This function takes a set of paths and sends them to the under-approximation tool, in this case the symbolic execution engine of Manticore. The DSE function returns a set of edges which are stored in the variable *DSEEdges*. Once the new DSE edges have been obtained, the boolean $fixpoint$ is used to represents if the analysis has reached a fixpoint or not. It is calculated by checking if the set *DSEEdges* contains any edges which are not yet present in the CFA G. If a fixpoint has not yet been reached, the worklist is populated with all unresolved abstract states which have a new edge starting at the location of the abstract state. Note that the notation $L(a)$ is used to represent the location of an abstract state a. When the worklist has been re-filled, the unresolved set is emptied to avoid re-evaluating previously unresolved abstract states in future DSE executions.

```
1  Input: A CPA 𝒟 := (D, Π, ⇝,merge,stop,prec), an initial abstract state
2  a_0 ∈ A with precision π_0 ∈ Π and a map P from locations to statements.
3  Output: A CFA of P and the set of reachable states.
4  worklist := {(a_0, π_0)}
5  reached := {(a_0, π_0)}
6  G := {}
7  unresolved := {}
8  tops := {}
9  DSEEdges := {}
10 while worklist ≠ ∅
11     pop (a, π) from worklist
12     (â, π̂) := prec(a, π, reached)
13     DSET := {(l_1, ·, ·)|(l_1, ·, ·) ∈ DSEEdges ∩ l_1 = L(â)}
14     transformers := getTransformers(â,P) ∪ DSET
15     if transformers = ∅
16         unresolved := unresolved ∪ (â, π̂)
17     foreach g ∈ transformers
18         foreach a' with â ⇝^g (a', π̂)
19             G := G ∪ g
20             //Attempt to merge with existing abstract states
21             foreach (a'', π'') ∈ reached
22                 a_new := merge(a', a'', π̂)
23                 if a_new ≠ a''
24                     worklist := (worklist ∪ (a_new, π̂)) \ {(a'', π'')}
25                     reached := (reached ∪ (a_new, π̂)) \ {(a'', π'')}
26             //Add new abstract state
27             if ¬stop(a', {a|(a, ·) ∈ reached}, π̂)
28                 worklist := worklist ∪ (a', π̂)
29                 reached := reached ∪ (a', π̂)
30     if worklist = ∅
31         tops := tops ∪ unresolved
32         paths := getPaths(G, tops)
33         DSEEdges := DSE(paths)
34         fixpoint := (G = G ∪ DSEEdges)
35         if fixpoint = false
36             for (a, π) ∈ tops such that (L(a), ·, ·) ∈ DSEEdges
37                 worklist := worklist ∪ (a, π)
38         unresolved := {}
39 return G, {a|(a, ·) ∈ reached}
```

Figure 3.4: *The second version of the ACFR algorithm which resolves all tops at each DSE invocation.*

By waiting until the worklist is completely empty before asking DSE for help, the amount of DSE edges that will be returned from the DSE function is maximized since the number of unresolved states at the time of the invocation is maximized. This is because it is more likely to have more tops to resolve the more abstract states that have already been analyzed.

The second version of the ACFR algorithm, hereby referred to as the second ACFR algorithm, is presented in Figure 3.4. The main difference between the second and first version is that the second version always re-resolves unresolved states. Consequently, it has the advantage of finding new successors of old tops when new edges have been added. Thus, this version should result in a higher or equal soundness compared to the previous version. However, as it has to symbolically execute paths to all tops that have ever been discovered at each DSE invocation, it could be slower.

Psuedocode wise, the only difference between the first and second version of the algorithm is that the second version uses an extra variable *tops*. The *tops* variable is a set containing all abstract states whose location is marked as having the successor $\top$. These type of locations will hereby be denoted as top locations. Furthermore, at line 31, the *tops* set is populated with the content of the *unresolved* set which, at each DSE invocation, contains the set of unresolved abstract states that resulted from the last time the worklist was emptied. As the *tops* set is sent to the getPaths function rather than the *unresolved* set, the obtained set of paths will be at least as large and lead to all tops rather than only the ones in the *unresolved* set.

3.3 Directed Symbolic Execution

The under-approximation of the CFA was performed through directed symbolic execution (DSE). More specifically, the Manticore platform was extended to act like a server. This server would listen to a user-specified port on localhost and wait for a request to arrive. Requests were required to contain a path to a binary and a set of paths defined by

```
1  Input: A program Pr, an initial state σ_init, a set of paths P
2  Output: A dictionary T mapping a path to a set of possible successors
3  of this path.
4
5  A[σ_init] := P
6  C[σ_init] := 1
7  worklist := {σ_init}
8  T := {}
9
10 while(worklist ≠ ∅)
11     pop σ_curr from worklist
12     T := T ∪ extract(P, σ_curr, σ_curr.IP, σ_curr.nextIP, C[σ_curr])
13     for σ_next ∈ fork(Pr, σ_curr)
14         C[σ_next] := C[σ_curr] + 1
15         A[σ_next] := update(A[σ_curr], C[σ_next], σ_next.IP)
16         if (A[σ_next] ≠ ∅)
17             worklist := worklist ∪ σ_next
```

Figure 3.5: *Pseudocode for the directed symbolic execution.*

a sequence of instruction addresses. The DSE server symbolically executed the given paths and used an SMT solver to calculate the possible successors for the last instruction of each path. We will now proceed to explain how the symbolic execution is directed along the supplied paths and how successors are extracted. Let C be a mapping from a state to an integer-valued constant. This constant is used to track the index of the instruction being executed in a certain state. Furthermore, let A be a mapping from a state to a list of paths that this state can be part of, which will be referred to as alive paths. The pseudocode of the directed symbolic execution is presented in Figure 3.5.

Initially, the alive paths for the initial state σ_{init} is set to all supplied paths. This is because the initial state corresponds to the first instruction in the binary which must be the first instruction in each path. Thereafter, the counter of the initial state is set to 1 as the initial state's instruction must be the instruction which is executed first when executing the binary. For other states, this counter will correspond to the number of symbolically executed instructions after the execution of the current state's instruction. Furthermore, the worklist is initially

```
1  Input: A set of alive paths P, a counter C and the current IP value IP.
2  Output: A set of paths denoted by P_new which is a subset of P satisfying
3  the condition that the addresses at index C of each path equals IP.
4
5  P_new := {}
6  foreach p ∈ P such that C <= length(p)
7      if p[C] := IP:
8          P_new := P_new ∪ {p}
```

Figure 3.6: *The algorithm which, given a set of paths, returns a set of paths which satisfy the condition that the address at index C is equal to the value of the IP register.*

```
1  Input: A set of paths P, a state σ, the current IP IP, a set of symbolic
2  constraint representing the next IP value nextIP and a
3  counter value C. Each path is a list of instruction addresses.
4  Output: A map T mapping a path to a set of possible successors
5  of the last instruction of this path.
6
7  T := {}
8  foreach path p in P such that length(p) = C∩last(p) = IP
9      foreach pcValue ∈ solve(σ, nextIP)
10         if T[p] = ≀
11             T[p] := {}
12         T[p] := T[p] ∪ {pcValue}
```

Figure 3.7: *The algorithm which extracts successors of paths.*

set to only contain the init state and the mapping T, which maps paths to successors, is initially empty. The while loop iterates until there are no more elements in the worklist. It starts by retrieving a state from the worklist at random. It then updates the alive paths for this state using the update algorithm presented in Figure 3.6. This algorithm filters the set of paths $A[\sigma_{curr}]$ to only contain paths that contain the address IP at index C. Note that a symbolic state contains symbolic expressions or concrete values for registers and memory locations. Thus, the IP value can be obtained from σ_{curr}. Since this state has already been reached, the IP field will contain a concrete value rather than a symbolic expression.

After updating the set of alive paths, the algorithm checks if this set is empty. If the set is empty, the state is simply dropped and the algorithm continues by iterating over the remaining states. Otherwise, the map T, containing successors of the paths, is updated using the extraction algorithm presented in Figure 3.7.

The extraction algorithm uses the set of desired paths, a state, the current IP value, a symbolic expression for the next IP value and the counter of the state to deduce the possible successors of the paths ending in this state. This is done by iterating over all paths which contains the state σ at index C and checking if this index matches the length of the path. If this is the case, the symbolic expression for the *nextIP* value is sent to the underlying SMT solver using the solve function. The SMT solver returns a set of solutions which are then put in the mapping of successors.

After the worklist algorithm has extracted the path successors and merged them into the map T, the algorithm performs a symbolic fork which generates succeeding symbolic states. Thereafter, the counter for these states are increased by 1 since the instruction of σ_{curr} has been executed. Furthermore, the set of alive paths is set to the set of alive path of σ_{curr} updated with the instruction of the new state. Thereafter, the new states are added to the worklist.

3.4 Motivating Examples for Using DSE

This section contains four examples which show how DSE can be useful when combined with an abstract interpretation based approach. The first example can be seen in Figure 3.8. In this binary, a buffer *buf* of size four bytes, is reserved in the BSS section. The execution of the binary starts at the *_start* label. Line 7 to 11 is used to read a double word from stdin into the buffer *buf* and copy this value into the register *eax*. However, as the user input can not possibly be known, the content of the *eax* register after line 11 is unknown. After loading the unknown value into the *eax* register, the content is copied from the *eax* register to the *ebx* register. Then, the two registers are subtracted and the result is stored in *eax*. Thereafter, the address of the *exit* label is added to *eax*. Then, a jump is performed to the content of *eax*. Since

```
1  SECTION .bss
2  buf      resb 4
3
4  SECTION .text
5  global _start
6  _start:
7          mov  edx, 4             ; Max length
8          mov  ecx, buf           ; Pointer to buffer
9          mov  ebx, 0             ; Standard input
10         mov  eax, 3             ; sys_read
11         int  0x80               ; Perform syscall
12         mov eax, dword [buf]    ; x = ⊤
13         mov ebx, eax            ; y = x
14         sub eax, ebx            ; z = x-y = 0
15         add eax, exit           ; z = <exit>
16         jmp eax                 ; Jump to z (<exit>)
17 exit:
18         mov ebx, 0              ; Set exit code to 0
19         mov eax, 1              ; sys_exit
20         int 0x80                ; Perform syscall
```

Figure 3.8: *xyz.asm - An example of when two registers with unkown content are subtracted.*

the two unknown registers with the same value were subtracted from each other at line 14, the content of *eax* should be 0 added with the location corresponding to the *exit* label. Thus, the indirect jump directs the execution to line 18 where the sys exit is performed with the exit status 0, signifying no errors.

The problem this binary causes for both the strided interval and constant propagation domains is that they can not properly model the values of the *eax* register after the execution of line 12. More specifically, after line 12 constant propagation will claim to not know anything about the content of *eax* since the content can not be modeled by one constant value. Thus, no information will be possible to propagate to determine the target of the indirect jump at line 16. Similarly, the strided interval will encounter problems as it will try to model the content of the *eax* register with an infinite interval. This infinite interval will then propagate through all instructions to the indirect jump, which will then be unresolved and thus marked as having $\top$ as its successor.

```
1  SECTION .text
2  global _start
3
4  _start:
5          call function1       ; Call <function1>
6          jmp exit             ; Jump to <exit>
7  function1:
8          call function2       ; Call <function2>
9          ret                  ; Return from <function1>
10 function2:
11         ret                  ; Return from <function2>
12 exit:
13         mov ebx, 0           ; Set exit code to 0
14         mov eax, 1           ; sys_exit
15         int 0x80             ; Perform syscall
```

Figure 3.9: *nestedCall.asm - An example of two nested function calls.*

If DSE is used to determine the successor of the indirect jump, it will start by marking the unknown content with a symbol, say s. Then, this symbol will be propagated to *ebx* such that *ebx* is represented by the same symbol. Then, when performing the subtraction, the *eax* register is set to contain the expression $s-s$ which can be solved for the constant value 0, but will not, as the concrete value of *eax* register is not required to determine the control flow yet.

The symbolic expression is then propagated to the indirect jump where the expression will be $s - s+$<*exit*> where <*exit*> is the location of the *exit* label. This symbolic expression will then be sent to the SMT solver which will solve the expression for a concrete value, namely the address of the *exit* label. Consequently, as DSE knows the value of the *eax* register to be <*exit*> at line 16, it can determine that the analysis should continue at line 18 after the indirect jump. The second example is depicted in Figure 3.9. This binary contains two nested function calls. The binary starts by calling the function *function1* at line 5. In *function1* the first instruction will call the second function *function2*. Then, the *ret* instruction at line 11 is executed to transfer the control flow back to line 9. The *ret* instruction of the first function is then executed to return to line 6. Finally, the direct jump to <*exit*> is performed and the binary exits with status 0.

While the control flow of this binary can be fully captured with strided interval analysis, this is not the case for constant propagation as constant propagation does not model the initial value of the *esp* register. Consequently, constant propagation can not possibly know what return address is popped of the stack when returning from a function. Thus, the *ret* instructions at line 9 and 11 can not be resolved by constant propagation.

However, if constant propagation is combined with DSE, the possible control transfers can be captured. First, constant propagation is executed until no new information can be acquired with this method. At this point, the control flow between line 5,8 and 11 have been discovered. The ACFR algorithm then performs the limited DFS from line 5 to obtain a path reaching line 11 through line 5 and 8. Note that this DFS does not require the fully reconstructed CFA but rather can be performed on the CFA that is being constructed.

After obtaining the path, the path is sent to the symbolic execution engine which performs the directed symbolic execution along the path and determines a possible successor to be line 9. The edge between line 11 and 9 is sent back to the ACFR algorithm which continues the analysis along this edge to discover that line 9 is also a location with unknown successors. The process repeats itself and the edge between line 9 and 6 is established.

The ACFR algorithm resumes the over-approximation which proceeds to find all edges until line 15. At this point, the worklist is empty and the path extraction finds no paths since there are no locations with unknown successors. Thus, no new edges are added to the CFA. Consequently, it is determined that a fixpoint has been reached and the CFA is written to disk. This example shows the necessity of alternating back and forth between the over-approximation an under-approximation rather than only executing the under-approximation once. Additionally, it illustrates how the CFA can be used while it is being constructed.

```
1  SECTION .text
2  global _start
3
4  _start:
5          call function           ; Call <function>
6          call function           ; Call <function>
7          jmp exit                ; Jump to <exit>
8  function:
9          ret                     ; Return from <function>
10 exit:
11         mov ebx, 0              ; Set exit code to 0
12         mov eax, 1              ; sys_exit
13         int 0x80                ; Perform syscall
```

Figure 3.10: *sequentialCallRet.asm - An example where one function is called more than once.*

The third example is presented in Figure 3.10. This binary contains two calls to the same function. The function <*function*> is first called at the beginning of the binary at line 5. Then the *ret* instruction at line 9 is executed to return to line 6. Line 6 contains a *call* instruction to the same function *function*. However, when the *ret* instruction is executed this time, it returns to line 7 instead of line 6. Thus, the *ret* instruction has two successors depending on the context. Finally, the jump to <*exit*> is performed and the binary performs the system exit.

This example is a good illustration of why the second ACFR algorithm can achieve a larger soundness than the first. The idea behind the second ACFR algorithm is that it should always resend all top locations every time a DSE invocation is performed. This should increase soundness at the expense of performance.

In the case when the first version is executed with constant propagation, the analysis only finds the edge between line 5 and 9 before symbolic execution has to be invoked. The symbolic execution returns the edge between line 9 and 6 which permits the analysis to continue. The constant propagation then finds the edges between line 6 and 9 before reaching the *ret* instruction. However, as the first ACFR algorithm only performs DSE when there are no known successors and the CFA is only location sensitive, the algorithm determines that the successor of the

```
1  SECTION .bss
2  buf      resb 4
3
4  SECTION .text
5  global _start
6
7  _start:
8          mov  edx, 4             ; Max length
9          mov  ecx, buf           ; Pointer to buffer
10         mov  ebx, 0             ; Standard input
11         mov  eax, 3             ; sys_read
12         int  80h                ; Perform syscall
13         mov eax, dword [buf]    ; eax = ⊤
14         jmp eax                 ; Jump to eax
```

Figure 3.11: *trueTop.asm - An example of a true top.*

instruction is line 5 and no request to DSE is necessary. Consequently, the first ACFR algorithm never finds the edge between line 9 and 7.

When the second ACFR algorithm is used with constant propagation, it starts at line 5 and also encounters a $\top$ successor at the *ret* instruction. Similarly to the first version, the second version extracts and sends the path to DSE which responds that line 6 is a successor to the *ret* instruction. Then, the ACFR algorithm continues the analysis until it has the same CFA as the first version ended at. At this point, the algorithm has no completely unresolved locations but has the location of the *ret* instruction as a top location. It thus searches for paths to send to DSE. Since the CFA now contains the edge between line 9 and 6, the search can find paths to the *ret* instruction using this edge.

Among other paths, the path through line 5,9,6 and 9 is sent to DSE. This path results in another edge than the previous path over line 5 and 9, namely the edge between line 9 and 7. This successor is sent back from DSE and the ACFR algorithm is resumed. The algorithm now explores line 7, 11, 12 and 13. At this point, the ideal CFA has been found and the analysis should thus be terminated. However, the algorithm needs to determine that a fixpoint has been reached and thus extracts all possible paths to the top location at line 9 and sends them to DSE. DSE answers with a set of edges where each of them starts at

the top location. However, as the algorithm notes that all these edges have already been added to the CFA, a fixpoint has been reached. Consequently, the algorithm terminates and the CFA is written to disk.

Note that the first ACFR algorithm can still find multiple successor of a top since the SMT solver searches for more than one solution to the constraints. The drawback is special cases such as the example above, where a successor of a top depends on another successor of the same top.

Finally, a last example is presented in Figure 3.11. At line 14, this binary contains a true top, an indirect jump which can have any possible successor. Line 8 to 12 are used to fill the buffer *buf* with an unknown value read from stdin. Then, the unknown value is copied into the *eax* register and a jump is performed to the address stored in *eax*.

Since this is a true top and abstract interpretation over-approximates successors, there is no abstract domain that could be used to not classify this instruction as a top. However, when paired with DSE, abstract interpretation can ask for the successors of the top. DSE would then return a set of edges which constitute an under-approximation of the control flow. Consequently, the analysis would be able to continue along these edges rather than stopping at the seemingly unresolvable location.

3.5 Metrics for Evaluation

The performance of the devised algorithm was evaluated by studying the soundness, accuracy and precision of its generated CFAs. These three metrics were measured relative to the ideal CFA which was created manually for the synthetic inputs and generated using the *CFGAccurate* algorithm of angr for non-synthetic inputs. All the metrics were defined only over edges as nodes can not have a degree of zero unless the CFA only contains a single node.

The first metric is accuracy. Accuracy is pragmatic for quantifying to what extent the generated graph contains superfluous edges with respect to the ideal graph. More formally, accuracy is defined as:

Definition. ***Accuracy***
The accuracy of two sets of edges E_I and E_G representing an ideal and generated graph respectively, is defined as:

$$\mathcal{A}_{E_I,E_G} = \frac{|E_I \cap E_G|}{|E_G|}$$

Furthermore, the second metric is soundness. Soundness is suitable for quantifying to what extent the edges of the ideal graph are included in the generated graph. More specifically, soundness is defined as:

Definition. ***Soundness***
The soundness of two sets of edges E_I and E_G representing an ideal and generated graph respectively, is defined as:

$$\mathcal{S}_{E_I,E_G} = \frac{|E_I \cap E_G|}{|E_I|}$$

Finally, precision combines the accuracy and soundness metrics and represents how far the generated graph is from the ideal graph, measuring both false positives and false negatives. The Precision of a generated graph with respect to an ideal graph, is defined as:

Definition. ***Precision***
Let E_I be the set of edges of the ideal CFA. Conversely, let E_G be the set of edges of the generated CFA. Then, the edge precision is defined as:

$$\mathcal{P}_{E_I,E_G} = \frac{\mathcal{A}_{E_I,E_G} + \mathcal{S}_{E_I,E_G}}{2} = \frac{|E_I \cap E_G|}{2|E_G|} + \frac{|E_I \cap E_G|}{2|E_I|}$$

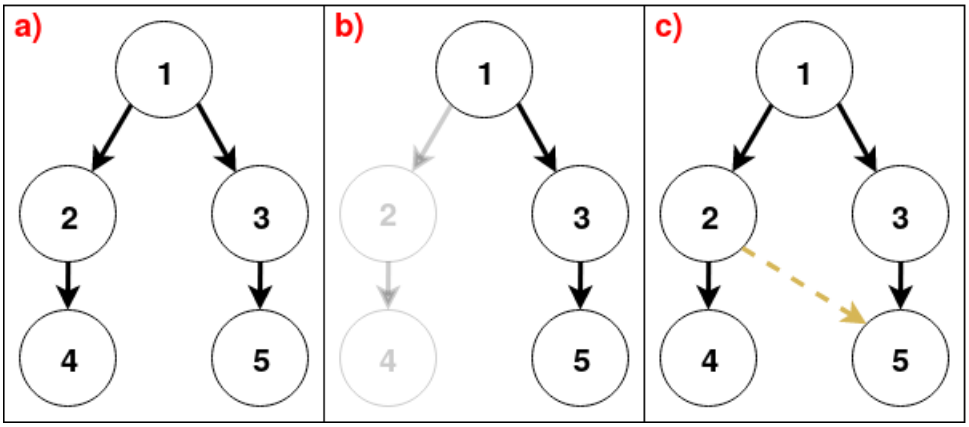
The division by 2 is used to normalize the expression to ensure that it is still a percentage. Defining metrics as percentages rather than absolute values has the benefit that a miss/addition of an edge/node will impact the precision relatively to the size of the CFA. For example, wrongly adding one false edge to a CFA with 10 edges will have a larger impact on the soundness than wrongly adding one false edge to a CFA of 10000 edges.

As Jakstab's algorithm can result in top nodes that may or may not be resolved by our under-approximation, the generated CFA might contain top nodes. A top node results in one edge for every possible instruction in the text section[2]. These type of edges will hereby be

[2]Note that the number of possible successors will be the same as the amount of bytes in the *.text* section as instructions in x86 are prefix-closed but can have varying length.

denoted as top edges. Since top nodes represent nodes which have unknown successors, it makes sense to compare the ideal CFA with both the generated CFA including top edges and the generated CFA excluding top edges.

As there are, to our best knowledge, currently no algorithms that can guarantee the creation of the ideal CFA, the generated CFA had to be evaluated relative to CFAs obtained through other state-of-the-art approaches such as angr's *CFGAccurate* algorithm, which was chosen to be used in this thesis[7][3]. It should be noted that *CFGAccurate* can not, as other state-of-the-art algorithms, guarantee the creation of the ideal CFA without errors. However, as it is considered a state-of-the-art algorithm, it was assumed to be reasonably precise.



Now, consider the three graphs from Figure 2.4 again, depicted above for convenience. For graph b, denoted by G_b, the number of edges in the generated graph and the number of intersecting edges is 2. Consequently, the accuracy, soundness and precision for graph b evaluates to the values below. Note that the accuracy evaluates to 100% and the graph is accurate since it does not contain any superfluous edges.

$$\mathcal{A}_{E_I,E_{G_b}} = \frac{|E_I \cap E_{G_b}|}{|E_{G_b}|} = \frac{2}{2} = 100\%$$
$$\mathcal{S}_{E_I,E_{G_b}} = \frac{|E_I \cap E_{G_b}|}{|E_I|} = \frac{2}{4} = 50\%$$
$$\mathcal{P}_{E_I,E_{G_b}} = \frac{\mathcal{A}_{E_I,E_{G_b}} + \mathcal{S}_{E_I,E_{G_b}}}{2} = \frac{1+0.5}{2} = 75\%$$

[3]The community behind later renamed the *CFGAccurate* algorithm to the *CFGEmulated* algorithm.

For graph c, which is denoted by G_c, the amount of edges in the generated graph is 5 and the amount of intersecting edges is 4. Thus, the accuracy, soundness and precision for graph c becomes as presented in the equations below. For this graph, the soundness evaluates to 100% and the graph is sound as it contains all edges of the ideal graph.

$$\mathcal{A}_{E_I,E_{G_c}} = \frac{|E_I \cap E_{G_c}|}{|E_{G_c}|} = \frac{4}{5} = 80\%$$
$$\mathcal{S}_{E_I,E_{G_c}} = \frac{|E_I \cap E_{G_c}|}{|E_I|} = \frac{4}{4} = 100\%$$
$$\mathcal{P}_{E_I,E_{G_c}} = \frac{\mathcal{A}_{E_I,E_{G_c}} + \mathcal{S}_{E_I,E_{G_c}}}{2} = \frac{0.8+1}{2} = 90\%$$

The calculations of the metrics presented above require the knowledge of which instructions a binary contains and the control flow transitions between these. For the synthetic binaries, the ideal CFA was manually reconstructed. For instructions containing top nodes, we assumed that no overlapping instructions existed and thus only added edges to the explicitly created instructions.

For the case study, we evaluated our algorithm on the five smallest GNU coreutils binaries. For these binaries, the ideal CFA was assumed to be the CFA obtained through the use of angr [7]. It should be noted that there are no guarantees that angr can find all instructions or that it will not accidentally announce data as instructions. However, determining the instructions of a program is as hard as determining the control flow of the program as they depend on each other. Consequently, this approach of approximating the ideal control flow with another state-of-the-art approach is not an uncommon evaluation method [1][10]. The evaluation proceeded as follows for each binary. First, the ACFR algorithm was executed on the binary to produce a generated CFA. Thereafter, angr was executed to generate the, presumably, ideal CFA. Finally, the accuracy, soundness and precision was calculated for the generated CFA with and without top edges with respect to the ideal CFA, where top edges were defined as edges originating from a program location whose successors are unknown.

3.6 Implementation

An illustration of the implementation is presented in Figure 3.12. Each node in the figure represent a module and each edge represents data transitioning between the components. The figure consists of one block

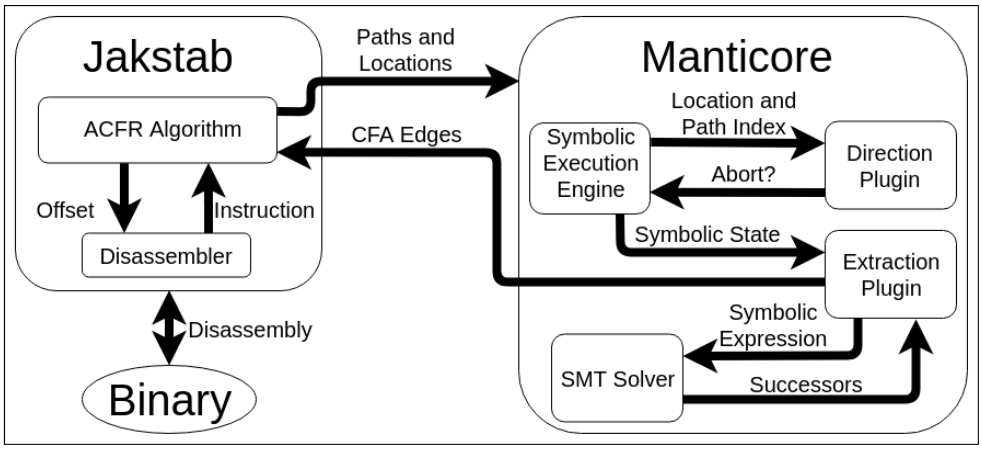


Figure 3.12: *An overview of the implementation.*

representing Jakstab and one block representing Manticore. Our contributions consist of the ACFR algorithm in Jakstab and the two plugin modules developed for Manticore. Both of the plugin modules communicate intensively with Manticore's Symbolic Execution Engine which also communicates with other components which were omitted for simplicity. Additionally, some of the arrows for the Symbolic Execution Engine node have been omitted as they are not relevant for understanding our implementation.

The process starts with the execution of the ACFR algorithm illustrated in the Jakstab box. The over-approximation conducts on demand disassembly to obtain instructions given an offset into the binary. The first offset is set to the program location that corresponds to the entrance of the binary and the subsequent offsets corresponds to the locations of subsequently obtained instructions.

When the over-approximation stagnates, a limited depth first search is performed towards the unresolved program locations to obtain all possible paths starting at the program entrance and ending in these locations. The unresolved locations and paths are then sent to Manticore where they are used in the Direction Plugin and Extraction Plugin. The reasons for using plugins for Manticore, rather than editing the source code directly, was to enable compatibility for future versions of the software with our approach. For Jakstab, plugins were not possible due to the required modifications to the Jakstab algorithm.

The Symbolic Execution Engine is started and informed about the plugins. Before each symbolic execution of an instruction, the Symbolic Execution Engine consults the Direction Plugin. This plugin receives the location of the instruction that is to be executed and the number of instructions that were executed to reach the current instruction. The plugin then responds to the Symbolic Execution Engine if the location should be analyzed or not. This decision is performed according to the pseudocode in Figure 3.6 presented in section 3.3.

After each analyzed symbolic state, the Symbolic Execution Engine sends the state to the Extraction Plugin which operates according to the pseudocode presented in Figure 3.7 introduced in section 3.3. The extractor plugin checks if the state corresponds to the last location of any of the paths and that it has executed the instructions according to the path. If this is the case for any path, the plugin sends the symbolic expression for the next IP value to an SMT solver which responds with a set of possible successors. The plugin use these to create a set of CFA edges which are not sent to Jakstab until the symbolic execution has completely terminated. Once the symbolic execution has completely terminated, the CFA edges are sent to Jakstab which adds them to the CFA and resumes the over-approximation

3.7 Experiments

Two sets of experiments were performed. The first set of experiments were executed on our own synthetic binaries. The second set of experiments were performed on the five smallest GNU coreutils binaries. Due to Manticore's preference for statically linked binaries over dynamically linked binaries, all programs were linked statically. Furthermore, during our experiments, only one core was used as to reduce the effect of nondeterministic task scheduling on the results.
A fundamental problem in research, which is a great obstacle for the scientific method, is the difficulty in reproducing other researchers' results. This ongoing methodological crisis was originally coined as the "replication crisis" [34]. As such, in this work, we aim to pay extra attention to provide the tools and information required for any scientist who desires to reproduce our results.

Architecture:	x86_64
CPU op-modes:	32-bit, 64-bit
CPU(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
CPU Model:	23
CPU Model Name:	Pentium(R) Dual-Core CPU E5300@ 2.60GHz
RAM:	4GB
L1d cache:	32KB
L1i cache:	32KB
L2 cache:	2048KB

Figure 3.13: *Hardware specs of the host on which the experiments were performed.*

The tool developed in this thesis is available online[4]. It was executed with python version 3.6.8, Manticore version 0.3.0 and java version 11.0.4. Jakstab was obtained from the original author online [31] and modified for our purposes. Furthermore, all input binaries were compiled into statically linked x86_32 executables. The scripts for compiling input binaries, generating results and extracting metrics are all available online[5]. As for the hardware, the relevant specs are provided in Figure 3.13. More information concerning reproducibility is available in the wiki of the project[6].

For the experiments, Manticore was configured to assume a symbolic input from stdin with a maximum of 256 characters. Additionally, Manticore was configured for at most 3 arguments of 20 characters each. The reason for limiting the size of these input sources was to reduce the amount of required computation. Finally, Manticore was used with the SMT solver Z3 as this was the default SMT solver of the tool.

[4] https://github.com/tpetersonkth/AlternatingControlFlowReconstruction

[5] https://github.com/tpetersonkth/AlternatingControlFlowReconstruction/tree/thesis/utilities

[6] https://github.com/tpetersonkth/AlternatingControlFlowReconstruction/wiki/Reproducibility

Originally, Manticore supported multithreaded symbolic execution. However, multithreading had to be disabled as a race condition bug was discovered[7]. As such, all symbolic execution was performed using only one thread. The time spent performing DSE can be expected to be reduced by a constant factor when Manticore is executed with multiple threads as multiple paths can be evaluated in parallel.

[7] https://github.com/trailofbits/Manticore/issues/1528

Chapter 4

Results

This chapter presents the results of our evaluations as well as the unplanned changes that had to be made to the strided interval domain of Jakstab. Initially, we present the changes that had to be made to the strided interval domain of Jakstab as it was essentially equivalent to the constant propagation domain due to an over-conservative widening operator. Thereafter, the results of the evaluations of the synthetic binaries and the GNU coreutils binaries are presented. Finally, a section is provided where the conclusions of the results are described.

In the tables presented in this chapter, different abbreviations are used for different analysis modes. More specifically, *c* and *i* are used to denote constant propagation and strided interval analysis respectively. Furthermore, *cD* and *iD* denotes constant propagation with DSE and strided interval analysis with DSE respectively. For simplicity, benchmarking was omitted from these tables. Instead, performance details for the different components can be found in appendix A.

4.1 Modifications to the Widening Operator of the Strided Interval Domain

The original strided interval domain used an over-conservative widening operator. More specifically, when performing widening, it would always default to top unless the strided intervals represented the same value. Thus the original strided interval domain was essentially equivalent to the constant propagation domain since strided interval domains could only exists if they had a stride of 0.

To be able to use intervals of other strides than 0, the widening operator was modified to not over-approximate to $\top$ whenever two intervals did not have the same constant value. Rather, the widening operator was modified to combine the two strided intervals by creating a strided interval which did not include $\top$ unless necessary. Let $I_1 = s_1[l_1, u_1]$ and $I_2 = s_2[l_2, u_2]$ be two strided intervals. Furthermore, let $I_w = s_w[l_w, u_w]$ be the interval resulting from widening I_1 towards I_2. Then, l_w and u_w was defined as below. Note that l_w is the minimum of the left bounds and r_w is the maximum of the upper bounds.

$$l_w = l_1 \text{ if } l_1 < l_2, \text{ otherwise } l_2$$
$$r_w = r_1 \text{ if } r_1 > r_2, \text{ otherwise } r_2$$

For the stride s_w, however, things were more complicated as it needs to ensure that I_w includes all elements of I_1 and I_2. This can essentially be broken down into two requirements. Firstly, s_w must divide both s_1 and s_2. Secondly, s_w must be set such that the lower bounds l_1 and l_2 are aligned. In essence, the second requirement can be phrased as $s_w \mid |l_1 - l_2|$. Now, let $gcd(x, y)$ be a function that returns the greatest common divisor of two integers x and y if $x \neq 0$ and $y \neq 0$. Otherwise, if one of the two parameters is zero, $gcd(x, y)$ returns the other parameter. The stride of the widened interval can now be defined as follows:

$$s_w = \begin{cases} |l_1 - l_2| & \text{if } s_1 = 0 \text{ and } s_2 = 0 \\ gcd(gcd(s_1, s_2), |l_1 - l_2|) & \text{otherwise} \end{cases}$$

By these definitions, it should be noted that the widening operator is symmetrical. In essence, widening I_1 towards I_2 or I_2 towards I_1 yields the same strided interval.

4.2 Evaluation of Synthetic Binaries

In this section the results of the synthetic evaluations are presented. All of the synthetic binaries are available online[1]. Thus, this section only describes the parts of the binaries that are of relevance for understanding our conclusions. Table 4.1 and 4.2 present the results of executing the first ACFR algorithm on all synthetic binaries with a timeout of 2 hours and DFS depth of 200 instructions. The depth was set to 200 instructions as there might be loops in the binaries and thus, even if a binary only contains a small amount of instructions, there might be longer paths.

The first column of the tables denotes the name of the binary. Then, the second is the analysis mode that was used. Here, *c* and *i* signifies constant propagation and strided intervals respectively. Further, a *D* signifies that the analysis was executed with DSE. The third column is the number of identified instructions. Thereafter, the next three columns are the accuracy, soundness and precision of the generated CFA, measured in percent, with respect to the ideal CFA.

Thereafter, **uTops** and **Tops** denote the number of unresolved tops and total amount of Tops. Note that an unresolved top is defined as a top where no successor could be determined. As such, a top which have multiple successors where only one of these could be determined, is not considered as an unresolved top. The top stats are followed by the accuracy, soundness and precision of the generated graph, without top edges and expressed as percentages, with respect to the ideal graph. Note that the ideal graph, the generated graph and the top edge free version of the generated graph are denoted by E_I, E_G and E_{TEF} respectively.

[1] `https://github.com/tpetersonkth/AlternatingControlFlowReconstruction/tree/thesis/input/synthetic/bin`

Binary	Mode	Instructions	%			uTops	Tops	%		
			$\mathcal{A}_{E_I,E_G}$	$\mathcal{S}_{E_I,E_G}$	$\mathcal{P}_{E_I,E_G}$			$\mathcal{A}_{E_I,E_{TEF}}$	$\mathcal{S}_{E_I,E_{TEF}}$	$\mathcal{P}_{E_I,E_{TEF}}$
badCall	c	4	14.81	66.67	40.74	1	1	100.0	50.00	75.00
badCall	cD	7	20.00	100.0	60.00	0	1	83.33	83.33	83.33
badCall	i	7	85.71	100.0	92.86	0	0	85.71	100.0	92.86
badCall	iD	7	85.71	100.0	92.86	0	0	85.71	100.0	92.86
branch	c	17	20.45	100.0	60.23	1	1	94.12	88.89	91.50
branch	cD	17	20.45	100.0	60.23	0	1	94.12	88.89	91.50
branch	i	17	94.74	100.0	97.37	0	0	94.74	100.0	97.37
branch	iD	17	94.74	100.0	97.37	0	0	94.74	100.0	97.37
callRet	c	2	8.33	40.00	24.17	1	1	100.0	20.00	60.00
callRet	cD	6	17.86	100.0	58.93	0	1	80.00	80.00	80.00
callRet	i	6	83.33	100.0	91.67	0	0	83.33	100.0	91.67
callRet	iD	6	83.33	100.0	91.67	0	0	83.33	100.0	91.67
cjmp	c	11	90.91	100.0	95.45	0	0	90.91	100.0	95.45
cjmp	cD	11	90.91	100.0	95.45	0	0	90.91	100.0	95.45
cjmp	i	11	90.91	100.0	95.45	0	0	90.91	100.0	95.45
cjmp	iD	11	90.91	100.0	95.45	0	0	90.91	100.0	95.45
cjmpBitOp	c	7	85.71	100.0	92.86	0	0	85.71	100.0	92.86
cjmpBitOp	cD	7	85.71	100.0	92.86	0	0	85.71	100.0	92.86
cjmpBitOp	i	8	75.00	100.0	87.50	0	0	75.00	100.0	87.50
cjmpBitOp	iD	8	75.00	100.0	87.50	0	0	75.00	100.0	87.50
conditionalTop	c	11	33.33	100.0	66.67	1	1	100.0	57.89	78.95
conditionalTop	cD	11	33.33	100.0	66.67	0	1	100.0	57.89	78.95
conditionalTop	i	11	33.33	100.0	66.67	1	1	100.0	57.89	78.95
conditionalTop	iD	11	33.33	100.0	66.67	0	1	100.0	57.89	78.95
doubleCallRet	c	17	9.47	81.82	45.65	2	2	100.0	72.73	86.36
doubleCallRet	cD	22	11.28	100.0	55.64	0	2	95.24	90.91	93.07
doubleCallRet	i	22	95.65	100.0	97.83	0	0	95.65	100.0	97.83
doubleCallRet	iD	22	95.65	100.0	97.83	0	0	95.65	100.0	97.83
espAssignment	c	9	16.33	100.0	58.16	1	1	87.50	87.50	87.50
espAssignment	cD	9	16.33	100.0	58.16	1	1	87.50	87.50	87.50
espAssignment	i	9	16.33	100.0	58.16	1	1	87.50	87.50	87.50
espAssignment	iD	9	16.33	100.0	58.16	1	1	87.50	87.50	87.50
indMemJmp	c	6	83.33	100.0	91.67	0	0	83.33	100.0	91.67
indMemJmp	cD	6	83.33	100.0	91.67	0	0	83.33	100.0	91.67
indMemJmp	i	6	83.33	100.0	91.67	0	0	83.33	100.0	91.67
indMemJmp	iD	6	83.33	100.0	91.67	0	0	83.33	100.0	91.67
infiniteLoop	c	10	66.67	100.0	83.33	0	0	66.67	100.0	83.33
infiniteLoop	cD	10	66.67	100.0	83.33	0	0	66.67	100.0	83.33
infiniteLoop $\triangle_T$	i	10	66.67	100.0	83.33	0	0	66.67	100.0	83.33
infiniteLoop $\triangle_T$	iD	10	66.67	100.0	83.33	0	0	66.67	100.0	83.33
int	c	9	88.89	100.0	94.44	0	0	88.89	100.0	94.44
int	cD	9	88.89	100.0	94.44	0	0	88.89	100.0	94.44
int	i	9	88.89	100.0	94.44	0	0	88.89	100.0	94.44
int	iD	9	88.89	100.0	94.44	0	0	88.89	100.0	94.44
kindConditionalTop	c	8	26.83	100.0	63.41	1	1	100.0	63.64	81.82
kindConditionalTop	cD	8	26.83	100.0	63.41	0	1	100.0	63.64	81.82
kindConditionalTop	i	8	26.83	100.0	63.41	1	1	100.0	63.64	81.82
kindConditionalTop	iD	8	26.83	100.0	63.41	0	1	100.0	63.64	81.82

Table 4.1: *Results of the synthetic evaluation for the first ACFR algorithm (Part 1). Analyses which had to be interrupted as they did not finish within 2 hours are marked with $\triangle_T$.*

The relevant instructions of the first binary, *badCall*, are presented in Figure 4.1 below. The binary contains push to the stack with a return address and a jump to a location with a *ret* instruction. Thus, a call is performed without using the *call* instruction. As can be seen in the table, the constant propagation could only resolve the instructions up to the *ret* instructions since it does not assume an initial *esp* value. However, with the help of DSE, constant propagation was able reconstruct a sound CFA. Note, however, that the top edge free version of

the generated graph did not include the edge from the resolved top to its successor and thus could not achieve 100% soundness. On the other hand, the top edge free version had a higher precision since the reconstructed CFA had a higher accuracy. Finally, the interval domain did not encounter any tops and thus did not need to use DSE.

```
1  function:
2          ret                 ; Return to <exit>
3  _start:
4          mov eax, exit       ; Set eax to <exit>
5          push eax            ; Push <exit>
6          jmp function        ; Perform the artificial call
7  exit:
8          ...
```

Figure 4.1: *The relevant instructions of badCall.asm.*

The *branch* binary corresponds to the assembly code presented in section 2.7.3. For this binary, the constant propagation based analyses generated a CFA which was sound but whose top edge free version was unsound since it excluded all edges from top nodes. Furthermore, the constant propagation analysis had the same accuracy, precision and soundness independently of if it took help from DSE even though a top was resolved when DSE was used.

DSE contributed with one extra top edge which, however, did not increase the precision since Jakstab can not know if the new set of edges is an over-approximation of the true edges from the top location and thus still must include one edge from the top location to each other program location. Furthermore, for the strided interval analysis, there were no tops and thus no difference between when DSE was used and not used, Additionally, the precision was fairly high compared to the precision achieved for other binaries. The reason that a CFA with a precision of 100% was not reconstructed was that one edge was falsely identified as being part of the CFA.

The third binary is *callRet* which is a binary containing a simple function call and return statement. Similarly to the *badCall* binary, constant propagation could not resolve the *ret* instruction since it does not assume an initial *esp* value. Thereafter, the subsequent binary corres-

Binary	Mode	Instructions	%			uTops	Tops	%		
			$\mathcal{A}_{E_I,E_G}$	$\mathcal{S}_{E_I,E_G}$	$\mathcal{P}_{E_I,E_G}$			$\mathcal{A}_{E_I,E_{TEF}}$	$\mathcal{S}_{E_I,E_{TEF}}$	$\mathcal{P}_{E_I,E_{TEF}}$
longRegisterEquals	c	14	15.38	100.0	57.69	1	1	66.67	100.0	83.33
longRegisterEquals	cD	14	15.38	100.0	57.69	1	1	66.67	100.0	83.33
longRegisterEquals	i	14	15.38	100.0	57.69	1	1	66.67	100.0	83.33
longRegisterEquals	iD	14	15.38	100.0	57.69	1	1	66.67	100.0	83.33
memCallRet	c	4	10.53	57.14	33.83	1	1	100.0	42.86	71.43
memCallRet	cD	8	16.67	100.0	58.33	0	1	85.71	85.71	85.71
memCallRet	i	8	87.50	100.0	93.75	0	0	87.50	100.0	93.75
memCallRet	iD	8	87.50	100.0	93.75	0	0	87.50	100.0	93.75
minimalistic	c	5	10.71	30.00	20.36	1	1	100.0	20.00	60.00
minimalistic	cD	11	19.35	60.00	39.68	0	1	100.0	50.00	75.00
minimalistic	i	11	19.35	60.00	39.68	1	1	100.0	50.00	75.00
minimalistic $\triangle_T$	iD	15	13.68	80.00	46.84	1	2	100.0	65.00	82.50
nestedCallRet	c	3	9.68	42.86	26.27	1	1	100.0	28.57	64.29
nestedCallRet	cD	8	10.94	100.0	55.47	0	2	83.33	71.43	77.38
nestedCallRet	i	8	87.50	100.0	93.75	0	0	87.50	100.0	93.75
nestedCallRet	iD	8	87.50	100.0	93.75	0	0	87.50	100.0	93.75
registerEquals	c	8	15.15	100.0	57.58	1	1	62.50	100.0	81.25
registerEquals	cD	8	15.15	100.0	57.58	1	1	62.50	100.0	81.25
registerEquals	i	8	15.15	100.0	57.58	1	1	62.50	100.0	81.25
registerEquals	iD	8	15.15	100.0	57.58	1	1	62.50	100.0	81.25
sequentialCallRet	c	2	10.34	42.86	26.60	1	1	100.0	14.29	57.14
sequentialCallRet	cD	3	13.33	57.14	35.24	0	1	100.0	28.57	64.29
sequentialCallRet	i	7	87.50	100.0	93.75	0	0	87.50	100.0	93.75
sequentialCallRet	iD	7	87.50	100.0	93.75	0	0	87.50	100.0	93.75
stackBO	c	13	37.23	81.40	59.31	1	1	100.0	27.91	63.95
stackBO	cD	13	37.23	81.40	59.31	0	1	100.0	27.91	63.95
stackBO	i	22	95.45	48.84	72.15	0	0	95.45	48.84	72.15
stackBO	iD	22	95.45	48.84	72.15	0	0	95.45	48.84	72.15
trueTop	c	7	34.29	100.0	67.14	1	1	100.0	50.00	75.00
trueTop	cD	7	34.29	100.0	67.14	0	1	100.0	50.00	75.00
trueTop	i	7	34.29	100.0	67.14	1	1	100.0	50.00	75.00
trueTop	iD	7	34.29	100.0	67.14	0	1	100.0	50.00	75.00
twoVars	c	7	18.37	75.00	46.68	1	1	100.0	58.33	79.17
twoVars	cD	11	22.64	100.0	61.32	0	1	90.91	83.33	87.12
twoVars	i	7	18.37	75.00	46.68	1	1	100.0	58.33	79.17
twoVars	iD	11	22.64	100.0	61.32	0	1	90.91	83.33	87.12
xyz	c	10	16.95	83.33	50.14	1	1	100.0	75.00	87.50
xyz	cD	13	19.35	100.0	59.68	0	1	91.67	91.67	91.67
xyz	i	10	16.95	83.33	50.14	1	1	100.0	75.00	87.50
xyz	iD	13	19.35	100.0	59.68	0	1	91.67	91.67	91.67

Table 4.2: *Results of the synthetic evaluation for the first ACFR algorithm (Part 2). Analyses which had to be interrupted as they did not finish within 2 hours are marked with $\triangle_T$.*

ponds to the example illustrating the capabilities of constant propagation in section 2.7.2. For this binary, there were no tops and all modes resulted in the same CFA. Then, the analysis of the *cjmpBitOp* binary suggests that constant propagation can be more accurate than strided intervals. The relevant instructions of this binary are presented in Figure 4.2 below. The binary contains a *je* instruction which performs a jump if the two operands of a previous *cmp* instruction were equal. As the *cmp* compares two equal values, the *je* instruction should always result in a jump. However, the first operand, *eax*, was shifted before the comparison. This was problematic for the interval domain which over-approximated the new value of the register while the constant propagation could handle the shifting without over-approximation.

```
1 mov eax, 0x7fffffff ; Set eax to 0x7fffffff
2 shr eax, 31         ; Shift eax right by 31 bits
3 cmp eax, 0          ; Compare eax with 0
4 je exit             ; Jump if eax was 0
```

Figure 4.2: *The relevant instructions of cjmpBitOp.asm.*

The next binary, *conditionalTop*, did not exhibit any significant differences between the analysis modes. Furthermore, the *doubleCallRet* binary did not show anything particular except that constant propagation needed DSE to handle *ret* instructions as had already been determined. Thereafter, the *espAssignment*, *indMemJump*, *infiniteLoop*, *int*, *kindConditionalTop* and *longRegisterEquals* binaries did not show any variation in the quality of the reconstructed CFAs between the analysis modes. However, during the analysis of *infiniteLoop*, the interval timed out after 2 hours. The reason for this timeout was a loop in the reconstructed CFA which had an unequal number of *call* and *ret* instructions. As a consequence, the strided interval of the *esp* register was decreased by 4 byte for each iteration of the loop. Thus, the analysis could not reach a fixpoint. This limitation of the strided interval domain will be explained in further detail in section 4.4.

The subsequent binaries; *memCallRet*, *minimalistic*, *nestedCallRet* and *registerEquals*, did not exhibit anything out of the ordinary concerning precision apart from constant propagation's dependence on DSE. However, the analysis of *minimalistic* timed out at 2 hours in mode *iD*. The reason was the same as for the *infiniteLoop* binary presented earlier. Note, however, that the timeout only occurred when the strided interval domain was combined with DSE. From the log files of our experiments, we could determine that the problem was not explicitly caused by DSE. Rather, DSE contributed with an edge that permitted the analysis to continue and incorrectly identify a cycle with an unequal amount of *call* and *ret* instructions at a later point.

For the next two binaries, *sequentialCallRet* and *stackBO*, it can be observed that the strided interval domain based analyses had a higher precision than the constant propagation based analyses both with and without DSE. The former binary corresponds to the third example in

Binary	Mode	Instructions	%			uTops	Tops	%		
			$\mathcal{A}_{E_I,E_G}$	$\mathcal{S}_{E_I,E_G}$	$\mathcal{P}_{E_I,E_G}$			$\mathcal{A}_{E_I,E_{TEF}}$	$\mathcal{S}_{E_I,E_{TEF}}$	$\mathcal{P}_{E_I,E_{TEF}}$
minimalistic	c	5	10.71	30.00	20.36	1	1	100.0	20.00	60.00
minimalistic	cD	35	10.53	100.0	55.26	0	3	47.06	80.00	63.53
minimalistic	i	11	19.35	60.00	39.68	1	1	100.0	50.00	75.00
minimalistic $\triangle_T$	iD	15	13.68	80.00	46.84	1	2	100.0	65.00	82.50
sequentialCallRet	c	2	10.34	42.86	26.60	1	1	100.0	14.29	57.14
sequentialCallRet	cD	7	20.59	100.0	60.29	0	1	83.33	71.43	77.38
sequentialCallRet	i	7	87.50	100.0	93.75	0	0	87.50	100.0	93.75
sequentialCallRet	iD	7	87.50	100.0	93.75	0	0	87.50	100.0	93.75

Table 4.3: *Results of the synthetic evaluation for the second ACFR algorithm which differed from the results obtained with the first version. Analyses which had to be interrupted as they did not finish within 2 hours are marked with* $\triangle_T$.

section 3.4 and contains two calls to one function in sequence. For constant propagation with DSE and the first ACFR algorithm, the successors of the *ret* instruction are only requested once. As such the analysis with the *cD* mode could not achieve 100% soundness for the generated graph like the strided interval domain.

For the latter binary, *stackBo*, the constant propagation based analyses had higher soundness than those based on strided intervals. This binary contains a function vulnerable to a stack buffer overflow which can overflow the return pointer, located on the stack, with an arbitrary value. By comparing the soundness of the original version of the generated graph and the top edge free version for the constant propagation based analyses, it can clearly be seen that the top edges contributed significantly to the soundness of the graph.

The reason was that the only top was located at the *ret* instruction in the vulnerable function and thus this top was actually a true top since the return pointer could be arbitrarily modified. Furthermore, as the buffer overflow violates the assumptions of Jakstab, the interval domain based analyses claimed that no tops were discovered.

The next binary, *trueTop*, which corresponds to the fourth example in section 3.4, exhibited no differences between the CFAs over the different analysis modes. Finally, the last two binaries, *twoVars* and *xyz* (Which corresponds to the first example in section 3.4), exhibited an increase of precision when using DSE for both of the abstract domains. This shows that there are cases which can not be evaluated with any of the two abstract domains but can be handled with DSE.

The results of the analysis of synthetic binaries using the second ACFR algorithm can be observed in Table 4.3. This table only includes the two binaries which had a different precision when executed with the second version compared to the first version of the algorithm. The results of the second ACFR algorithm is similar to the first. In fact, the only time the analyses did not reconstruct the same CFA as the first version was for the *sequentialCallRet* and *minimalistic*. As the former binary was the example of why the second algorithm was better than the first, illustrated in section 3.4, this should not appear as a surprise. The second binary was a minimalistic c program compiled statically without stdlib. For this binary, the soundness of the CFA resulting from the *cD* mode increased from 60% to 100% for the CFA containing top edges.

Furthermore, for the top edge free version of the CFA, the soundness increased from 50% to 80% in this mode. However, due to a large number of false positives, the accuracy decreased by 8.82 and 52.94 percentage points for the top edge including and top edge free version of the CFA respectively. This resulted in the somewhat strange result that the usage of the second ACFR algorithm created a CFA which was more precise when including top edges but less precise when excluding top edges. From the precision uncertainties, it can be seen that the second ACFR algorithm does not guarantee an increase in precision. Rather, it guarantees the soundness to be equal or higher than the soundness resulting from the usage of the first version of the algorithm.

Another somewhat surprising aspect concerning the analysis of this binary is that the constant propagation with DSE identified 35 instructions while only 19 instructions actually existed in the binary. The *minimalistic* binary was compiled without the c standard library and with the main method set as the entry point of the binary. As such, the binary ended with a *ret* instruction which should not have any well-defined successors. However, when the over-approximation reached this instruction, it asked DSE for a possible successor which was then provided. The over-approximation then progressed by disassembling instructions at the new location, continuing to ask for successors of any encountered tops. This proceeded until the analysis reached a sequence of bytes which did not correspond to an instruction.

Note that it could possibly be argued that the manually reconstructed CFA was wrong since it is actually possible to execute the last *ret* instruction. Furthermore, it could then be argued that the preciseness of the reconstructed CFA using the *cD* mode would be higher than the other modes. However, we chose to limit out definition of the ideal control flow to instructions with well-defined behaviour.

4.3 Evaluation of GNU Coreutils

Due to a limited amount of computational resources, the evaluation was limited to the 5 smallest GNU coreutils binaries. The results of the evalutions using the first ACFR algorithm are presented in Table 4.4. The results of the evaluation using the second ACFR algorithm did not differ in accuracy, soundness or precision for any of the binaries and are thus not presented.

As could have been expected, the constant propagation finished quickly but was not useful on its own, only managing to identify the first 10 instruction of each binary. However, with the help of DSE, constant propagation was able to increase the soundness of the reconstructed CFA. On the other hand, during the extraction of possible paths for DSE, the analysis had an out of memory exception. This occured because of the large amount of possible paths due to loops in the CFA. For example, for a loop which has to be traversed, there is at least one path for each iteration of the loop. Moreover, if there are multiple loops, the number of paths can be significantly larger, as will be explained in section 4.4.2.

As can be seen in the table, for each binary, analyses based on the strided interval did not finish within 2 hours. After 2 hours, these analyses had to be interrupted and the graph available at this point was used for the calculations of the metrics. It should be noted that the over-approximation stalled and thus DSE was thus never used. The reason for not finding a fixpoint could be non-zero sums of *call* and *ret* instructions, as described in section 4.4.1. Additionally, another thing to note is that the soundness was higher when using strided intervals as opposed to constant propagation with DSE.

Binary	Mode	Instructions	%			uTops	Tops	%		
			$\mathcal{A}_{E_I,E_G}$	$\mathcal{S}_{E_I,E_G}$	$\mathcal{P}_{E_I,E_G}$			$\mathcal{A}_{E_I,E_{TEF}}$	$\mathcal{S}_{E_I,E_{TEF}}$	$\mathcal{P}_{E_I,E_{TEF}}$
echo	c	10	0.00211	0.01302	0.00757	1	1	100.0	0.01302	50.00651
echo $\triangle_M$	cD	59	0.00328	0.08103	0.04216	0	4	100.0	0.08103	50.04052
echo $\triangle_T$	i	1253	0.03355	1.86372	0.94863	9	9	91.47727	1.86372	46.6705
echo $\triangle_T$	iD	1346	0.03607	2.00408	1.02008	9	9	92.02658	2.00408	47.01533
false	c	10	0.00212	0.01315	0.00763	1	1	100.0	0.01315	50.00657
false $\triangle_M$	cD	59	0.00329	0.0818	0.04255	0	4	100.0	0.0818	50.0409
false $\triangle_T$	i	1253	0.03365	1.88147	0.95756	9	9	91.47727	1.88147	46.67937
false $\triangle_T$	iD	1346	0.03619	2.02317	1.02968	9	9	92.02658	2.02317	47.02487
make-prime-list	c	10	0.00221	0.01407	0.00814	1	1	100.0	0.01407	50.00703
make-prime-list $\triangle_M$	cD	59	0.00343	0.08755	0.04549	0	4	100.0	0.08755	50.04377
make-prime-list $\triangle_T$	i	1346	0.03771	2.16518	1.10144	9	9	92.02658	2.16518	47.09588
make-prime-list $\triangle_T$	iD	1346	0.03771	2.16518	1.10144	9	9	92.02658	2.16518	47.09588
printenv	c	10	0.0021	0.0128	0.00745	1	1	100.0	0.0128	50.0064
printenv $\triangle_M$	cD	59	0.00326	0.07967	0.04147	0	4	100.0	0.07967	50.03984
printenv $\triangle_T$	i	1253	0.03332	1.83251	0.93292	9	9	91.47727	1.83251	46.65489
printenv $\triangle_T$	iD	1346	0.03583	1.97052	1.00317	9	9	92.02658	1.97052	46.99855
true	c	10	0.00212	0.01315	0.00763	1	1	100.0	0.01315	50.00657
true $\triangle_M$	cD	59	0.00329	0.0818	0.04255	0	4	100.0	0.0818	50.0409
true $\triangle_T$	i	1253	0.03365	1.88147	0.95756	9	9	91.47727	1.88147	46.67937
true $\triangle_T$	iD	1346	0.03619	2.02317	1.02968	9	9	92.02658	2.02317	47.02487

Table 4.4: *Evaluation of the 5 smallest GNU coreutils binaries using the first ACFR algorithm. Analyses which had to be interrupted after 2 hours and analyses that terminated because of insufficient memory are marked with $\triangle_T$ and $\triangle_M$ respectively.*

The analysis sometimes resulted in a different CFA when executing the strided interval domain with and without DSE. Since DSE was not used in *iD* mode, we believe that this could be a result of nondeterministic behaviour in Jakstab. For example, the worklist of the CPA algorithm was implemented with a set, which is a data structure that does not guarantee any particular order of its stored elements. As such, elements might have been retrieved from the worklist in a different order, leading to a different sequence of widenings and thus potentially more or less widenings being performed. As such, the analysis could require a different amount of time when executed multiple times in the same environment.

As a final note, the reader is invited to notice that, for all binaries, the top edge free version of the CFA of the constant propagation analysis had a higher precision than the strided interval based analyses. The reason is that pure constant propagation results in very few edges which leads to having less false positives and thus to a very high accuracy. Hence, only looking at the precision metric to determine if a CFA is useful or not can be misleading. Rather, it is preferable to consider both the accuracy and the soundness when making this decision.

4.4 Conclusions

In this section we elaborate on our conclusions concerning the reasons for the timeout when using the strided interval domain and why the out of memory exceptions was a result of the large amount of paths in a CFA with loops. In the first subsection, the limitations our modifications to the widening operator imposed on the strided interval domain are described. Furthermore, an example binary which can not be analyzed within a reasonable amount of time using the strided interval based analyses, is explained. In the second subsection, an intuition to why CFAs with loops enable a large number of possible paths to the unresolved locations, is provided.

4.4.1 Limitations of Strided Intervals

The problems concerning the termination of the strided interval based analyses was due to the widening operator only widening as much as necessary and the over-approximation resulting in infeasible cycles in the CFA. More specifically, the infeasible cycles contained a different amount of *call* and *ret* instructions. As such, a strided interval representing the stack pointer at a location in one of these cycles would not reach a fixpoint. This was the reason why the analysis of the *infiniteLoop* and *minimalistic* binaries never finished.

To understand how these loops occur, consider the simplified version of the *minimalistic* binary presented in figure 4.3. This binary contains two functions: a and b. Initially, the analysis can determine that there are two edges from the location corresponding to line 9[2]. The first edge goes from line 9 to line 6 and the second from line 9 to line 12. For a CFA where nodes are only locations, as in this work, control flow can propagate through the edges independently of earlier instructions. In other words, the analysis is path insensitive.

[2]To find the two edges and trigger the illustrated problem for this binary in practice, the strided interval analysis has to be executed with DSE. Otherwise, the location corresponding to line 9 is marked as a top location.

```
1  SECTION .text
2  global _start
3
4  _start:
5          call a          ; Push next IP and go to line 9
6          call b          ; Push next IP and go to line 11
7          jmp exit        ; Jump to exit
8  a:
9          ret             ; Return to line 7 or 12
10 b:
11         call a          ; Push next IP and go to line 9
12         ret             ; Return to line 7
13 exit:
14         mov ebx, 0      ; Set exit code to 0
15         mov eax, 1      ; Set eax to sys_exit
16         int 0x80        ; Syscall
```

Figure 4.3: *A minimalistic example containing a loop which is problematic for the strided interval analysis.*

Line	Instruction	Strided Interval
5	call a	0[x,x]
9	ret	0[x-4,x-4]
6	call b	0[x,x]
11	call a	0[x-4,x-4]
9	ret	4[x-8,x-4]
6	call b	4[x-4,x]
11	call a	4[x-8,x-4]
9	ret	4[x-12,x-4]
6	call b	4[x-8,x]
11	call a	4[x-12,x-4]
...		

Figure 4.4: *The changes of the strided interval for the value of the esp register when the problematic loop is analysed. The value x is the initial value of the esp register.*

The analysis can falsely identify a path which is not possible in practice. This path is the sequence of program locations corresponding to the path through lines 5, 9, 6, 11, 9, 6 and so on. This example thus contains a loop in the CFA that is not possible in practice but enabled by the path insensitivity of the analyses. In a concrete execution, control flow can only be propagated through the edge between line 9 and 6 if the last *call* instruction was the *call* instruction at line 5.

To see why this path creates a problem, an execution trace of the program will be explained. Let x be the initial stack value, then the strided interval of the *esp* register for the first nine instructions can be observed in figure 4.4. Note that the strided intervals represent the possible values of the *esp* register before executing the instruction at the line number. Furthermore, bear in mind that the studied loop poses a problem to the strided interval analysis since it contains a different amount of *call* and *ret* instructions.

Initially, the strided interval for the *esp* register is $0[x, x]$ as the initial value is x. Then, the call instruction at line 5 pushes the 32-bit return address to the stack. Hence, the *esp* register is decremented by 4 byte. Then, the first instruction of the loop is executed at line 9. As this is a *ret* instruction, the stack is popped for the return address and the *esp* register is increased by 4 byte, changing the strided interval back to $0[x, x]$. Subsequently, the *call* instruction at line 6 is executed, decrements the interval by 4 byte.

Thereafter, the last instruction of the loop is executed. This instruction, located at line 11, is a *call* instruction and thus decrements the strided interval by 4 byte. Then, the *ret* instruction at line 9 is reached once again. Note, however, that this time, the strided interval will be different than the last time the instruction at line 9 was executed. The merge operator in the ACFR algorithm returns the strided interval $4[x - 8, x - 4]$ as the over-approximation of possible values of the *esp* register at this location is the merge of all reached states at the location. Since the merge of all reached states changed with this new state, the new state is determined to contribute with new information and the analysis continues with this state. The same happens for the next two instructions at line 6 and 11.

The *ret* instruction at line 9 is now reached for a third time. The two previous strided intervals for the *esp* register at this location are merged to obtain the strided interval $4[x-12, x-4]$. Again, the ACFR algorithm checks if the new interval $4[x-12, x-4]$ contains new information by checking if it differs from the strided interval obtained by merging all reached states at this location. Since $4[x-12, x-4] \neq 4[x-8, x-4]$ the analysis continues.

This pattern repeats itself over and over again until the smallest possible 32-bit integer is reached. As such, the analysis might continue for an unreasonable amount of time. Especially if the loops are large. Additionally, the singleton representation of the *esp* value at all locations in the loop is lost. As such, further locations which can only be reached through these locations will not be able to resolve *ret* instructions.

4.4.2 Number of Possible Paths in a CFA

In section 4.3 we saw that the constant propagation experienced an out of memory error. This happened during the path extraction from the CFA. The reason was the large amount of possible paths a CFA with multiple loops enables.

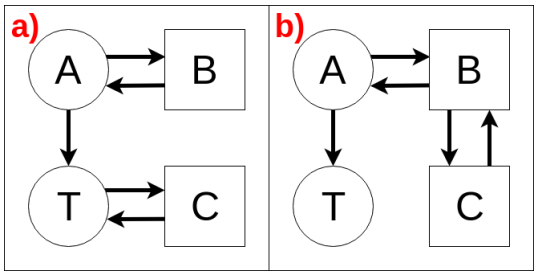


Figure 4.5: *Two CFAs with two loops each but structured differently. Circles represent one location corresponding to a single instruction while squares represent subgraphs whose locations correspond to fall-through instructions.*

Figure 4.5 illustrates a CFA which contains two loops in serie and a CFA which contains two nested loops. The circles represent locations of single instructions while squares represent subgraphs only containing locations of fall-through instructions. In Figure 4.5.a, the first loop is between A and B while the second loop is between T and C where

the T node represents a top location. Let the length of the two loops, including the edges between the nodes in the illustration, be L_{loop}. Additionally, let the maximum allowed length of the paths be denoted by L_{lim}.

If no loops are used, there is only one path along the edge from A to T. Otherwise, if only the loop between A and B is used, the number of possible paths is approximately L_{lim}/L_{loop} since each iteration of the loop results in a new path. Since L_{loop} is a constant, it is possible to see that the number of paths is linear in L_{lim} for a CFA with only one loop. Furthermore, if both loops are allowed, for each number of iterations in the AB loop I_{AB}, there are $I_{TC}(I_{AB}) = (L_{lim} - I_{AB} * L_{loop} - 1)/L_{loop}$ iterations in the TC loop. Thus, the total number of paths using both loops is:

$$\sum_{i=0}^{L_{lim}/L_{loop}} I_{TC}(i) = \sum_{i=0}^{L_{lim}/L_{loop}} (L_{lim} - i * L_{loop} - 1)/L_{loop} = \sum_{i=0}^{L_{lim}/L_{loop}} (L_{lim}/L_{loop} - i - 1/L_{loop})$$

Since L_{loop} is a constant, the order of magnitude is proportional to:

$$\sum_{i=0}^{L_{lim}} (L_{lim} - C) \text{ where } C \text{ is a constant}$$

Since this scales proportionally to L_{lim}^2, the number of paths in a CFA, where two loops are in serie, is quadratic in the maximum length of the paths L_{lim}.

With the same reasoning, it can be seen that the the amount of paths scale similarly, albeit with one important difference, for nested loops. Consider the CFA in Figure 4.5.b. In this CFA, there are two nested loops. Similarly to the CFA with the loops in series, without using any loops, there is only one path between A and T. Furthermore, if only the first loop is used, there are around L_{lim}/L_{loop} possible paths since there is one possible path for each iteration of the loop. Moreover, if both loops are used and I_{AB} denotes the number of iterations in the first loop, the number of possible iterations in the second loop is $I_{BC} = (L_{lim} - I_{AB} * L_{loop} - 1)/L_{loop}$.

In the first CFA, the number of iterations in the first loop was already determined when the second loop was reached. For the second CFA, however, this is not necessarily the case. For example, a path might iterate 1 time in the AB loop, 2 times in the BC loop followed by another iteration in the AB loop. Hence, the number of paths is not

```
1  SECTION .data
2  w1      db      'Too '
3  w2      db      'many '
4  w3      db      'paths', 0x0A
5  SECTION .text
6  global _start
7
8  _start:
9          push w1          ; Push address of w1
10         push 4           ; Push length of w1
11         call print       ; Print 'Too '
12         push w2          ; Push address of w2
13         push 5           ; Push length of w2
14         call print       ; Print 'many '
15         push w3          ; Push address of w3
16         push 6           ; Push length of w3
17         call print       ; Print 'paths\n'
18         jmp quit         ; Jump to quit
19 print:
20         mov edx, [esp+4]; Length
21         mov ecx, [esp+8]; Address of msg
22         mov ebx, 1       ; Set ebx to stdout
23         mov eax, 4       ; Set eax to the opcode of write
24         int 0x80         ; Perform the syscall
25         ret
26 quit:
27         mov ebx, 0       ; Set ebx to 0
28         mov eax, 1       ; Set eax to the opcode of exit
29         int 0x80         ; Perform the syscall
```

Figure 4.6: *An example of a small program with a large amount of possible paths.*

only dependent on the number of iterations in the loops but also on the permutations of these. As such, for CFAs with nested loops, the number of paths scales exponentially in L_{lim}. For CFAs corresponding to larger programs, the structure can be more complicated and thus these CFAs can have a much larger amount of permutations between loop iterations. Consequently, they are likely to have a large amount of paths.

For example, consider the small program presented in Figure 4.6. This program contains a function *print* which outputs a string to stdout. The *print* function is called three times from three different locations. As such, the *ret* instruction at line 25 has three successors. These successors are line 12, 15 and 18. However, since each node in the CFA only corresponds to a location, it is possible to identify paths which enter through a specific *call* instruction but exits through an edge from the location of the *ret* instruction which is not possible if that specific *call* instruction was the last executed *call* instruction. For example, a path in the CFA could use the *call* instruction at line 14 to enter the *print* function and then use the edge from line 25 to line 12 to arrive back at line 12. While this path is not possible in practice, it is possible in the CFA since states at the same locations are merged.

To illustrate how this creates a large number of paths, consider Figure 4.7 which is a simplification of the CFA for the program in Figure 4.6. This simplfied CFA represents locations for groups of fall through instructions by a square for ease of illustration. The $\_start_0$ corresponds to line 9 to 11, $\_start_1$ corresponds to line 12 to 14, $\_start_2$ corresponds to line 15 to 17 and $\_start_3$ corresponds to line 18. Furthermore, *print* and *quit* represent line 20 to 25 and line 27 to 29 respectively. Finally, assume that the unresolved location is the location of the *ret* instruction of the *print* function.

As can be seen in the CFA, it is possible for a path to loop multiple times in either of the two loops. In fact, the paths may even constitute a mixture of the two loops in any particular order and with any number of iterations as long as the length of the path is lower than the limit on the amount of instructions. Executing a DFS limited to a depth of 100 instructions on this CFA, with the *ret* instruction marked as an unresolved location, yields 2047 unique paths. Furthermore, increasing the depth by 10 and 20 instructions increases the number of unique paths to 4095 and 8191 respectively.

Moreover, by studying the graphs generated by angr for the five smallest GNU coreutils binaries, it is possible to see that the number of instructions in these types of binaries are in the tens of thousands. As such, our earlier experiments would probably require a larger depth than 500 instructions for our approach to perform better than the state

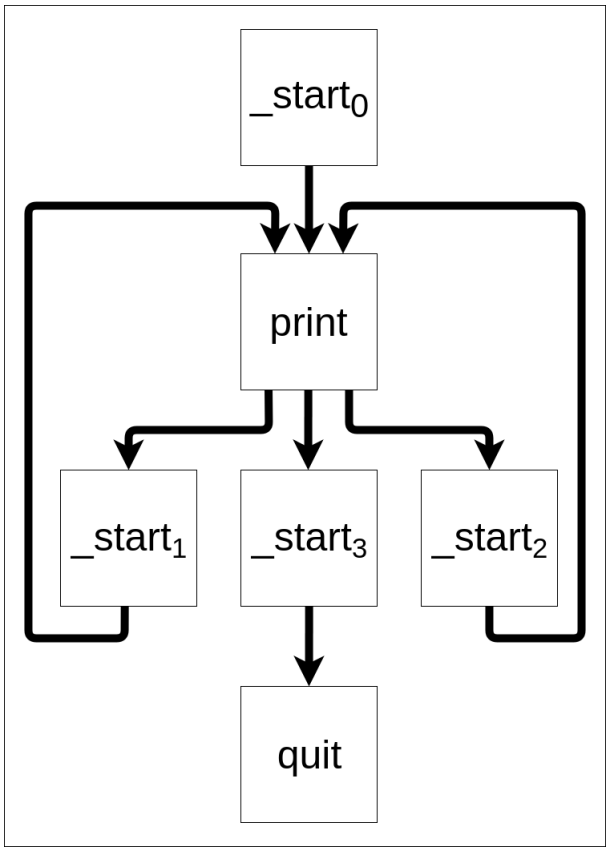


Figure 4.7: *A simplified illustration of the CFA for the small program with a large amount of possible paths.*

of the art. However, as can be observed in the example above, the number of unique paths to an unresolved location can grow rapidly with the length limit of the paths even for small examples. As such, our approach is not pragmatic unless it is possible to perform an efficient path extraction and storage.

Chapter 5

Discussion and Future Work

In this section we summarize what has been performed in the thesis as well as what conclusions could be drawn. Additionally, we discuss how our approach can be modified for other under-approximation techniques. Thereafter, we describe possible future work to our contribution. Among other things, we suggest using white-box fuzzing for under-approximation as well as evaluating other abstract domains than the constant propagation and strided interval domain.

5.1 Discussion

In this thesis, we addressed the problem of control flow reconstruction in the presence of indirect jumps. Initially, the differences between various types of instructions were explained. We then described the difference between a BBG and a CFA as well as why a CFA can be extended to be more expressive than a BBG, hence motivating our design choice of reconstructing a CFA rather than a BBG. Subsequently, the various static and dynamic analysis techniques were described and we motivated why DSE could be an interesting under-approximation candidate. Then, we introduced data flow analysis and the Jakstab tool that was applied to over-approximate control flow using abstract interpretation. Thereafter, we discussed advantages and disadvantages of adopting strided intervals as the abstract domain rather than constant propagation. Subsequently, the CPA, CPA+ and Jakstab algorithms were explained and we introduced our own modifications to the Jakstab algorithm to enable the inclusion of additional control flow edges stemming from DSE.

The modifications we performed to the Jakstab algorithm enabled the usage of under-approximation with DSE to improve the soundness of the reconstructed control flow automaton. In fact, the two versions of the ACFR algorithm that we introduced in section 3.2 abstract from the under-approximation in a modular form. As such, it is possible to integrate another under-approximation technique into the alternating algorithms simply by modifying the semantics of the DSE function at line 31 and line 33 in the pseudocodes presented in Figure 3.3 and 3.4 respectively.

After explaining the required modifications to the Jakstab algorithm, we provided and explained pseudocodes for ensuring that the symbolic execution would follow a set of provided paths and extract the successors of a set of target program locations. Then, we presented a small set of handcrafted examples that were used to illustrate the differences between the two versions of our algorithm as well as the advantages of using DSE for under-approximation of control flow. Thereafter, our abstract interpretation and DSE based alternating approach was implemented[1] and evaluated using a set of metrics derived from earlier work in the field.

Throughout the thesis, we presented a set of synthetic binaries. These binaries were used for an initial evaluation of the tool. The results of the experiments with the synthetic binaries suggested that our algorithm can be advantageous for multiple challenging binaries. Furthermore, we then proceeded to evaluate our alternating approach on the five smallest binaries of the GNU coreutils.

The results of the GNU coreutils evaluations showed no difference between the two version of our algorithm. Moreover, they suggested that the strided interval analysis suffered in performance when encountering particularly challening loops and that the requirement of extracting all possible paths from the program entry to a set of targets was not feasible in practice. Additionally, for constant propagation, it showed that the soundness could be increased by applying constant propagation with DSE rather than pure constant propagation. Finally, we explained the challenging type of loops that could thwart

[1] `https://github.com/tpetersonkth/AlternatingControlFlowReconstruction`

the strided interval based analyses and described why a large amount of possible paths was the reason for the high memory requirements of our algorithm.

From our results, a first conclusion was that our modifications to the widening operator resulted in strided interval analyses which could not reach a fixpoint within a reasonable amount of time when the subject program contained loops with a different amount of *call* and *ret* instructions. This suggested that strided intervals are not suitable for over-approximations of CFAs unless they use an overly conservative widening operator.

A second conclusion we drew was that alternating approaches which are dependent on extractions of paths are very limited since the amount of possible paths in the CFA can be very large. In fact, this conclusion implies that any approach which requires path extraction in a CFA will be very limited as the number of paths can scale exponentially in the maximum path length. Hence, future work should probably be focused on approaches which do not require paths towards unresolved locations. A possibility to avoid the path explosion problem could be to only study a program slice. This would require to manually identify the path conditions for the entry of the program slice. Then, a path extraction algorithm could be initialized with the path conditions and the search for paths could be limited to only the program slice.

5.2 Future Work

As the strided interval based analyses struggled with loops with a different amount of *call* and *ret* instructions, it could be interesting to detect and exclude those. This would require the execution of an algorithm for detection of cycles in a graph such as Tarjan's or Kosaraju's algorithm [35][36]. Once a cycle is found, it could trivially be determined if it should be excluded by counting its *call* and *ret* instructions. This would, however, add the assumption that the binary does not contain any loops with a different amount of *call* and *ret* instructions since they have to be assumed to be a product of the over-approximation. This kind of assumption would exclude malware and other types of malicious binaries as viable target binaries.

Another approach could be to identify these loops and force the widening operator to over-approximate the possible *esp* values to $\top$. This would enable the analysis to continue without getting stuck infinitely. However, this would require some heuristic to determine at which instruction the widening operator should be over-approximated to $\top$. For example, a loop might contain a different number of *call* and *ret* instructions but there might exist a path that uses only a small portion of the loop and continues the execution with some other instructions.

Thus, at the execution of each instruction, it would be required to know if an instruction in a problematic loop is being revisited. However, to determine if this is the case, it would be required to know previously executed instructions leading to the current one. As this does not scale well in terms of memory usage, some heuristic would be required that can determine if the instruction in a problematic loop is revisited with less saved information than all previously executed instructions. For example, a CFA node could consists of program location and a call-stack which only contains a fixed number of entries.

An additional problem that was encountered was the large memory requirement as a result of unrolling the loops in the CFA into a large amount of length limited paths. To limit this memory usage, one alternative could be to determine how many iterations are possible for each loop and only extract the paths with that particular number of iterations. This could possibly be conducted by identifying loop variants and bounding these. However, this is an undecidable problem in general since its solution would mean a solution to the halting problem. As such, loops which can not be determined to have a fixed number of possible iterations would require unrolling. Nevertheless, it could be interesting to investigate to what extent this method could decrease the memory requirements of our approach as it could still decrease memory usage even if it could not determine the number of possible iterations for all loops.

Another alternative could be to find a more efficient storage method than to simply store all paths shorter than a fixed length as done in this thesis. For example, it could be interesting to utilize a cycle detection algorithm to detect the cycles to avoid multiple entries for paths which

only different in the number of loop iterations performed. Thus, the loop would only have to be stored once together with some meta data marking the sequence of instructions as a loop. This would require further modifications to the pseudocodes presented in section 3.3 but would decrease the storage requirements of the paths.

In addition to studying how to avoid the storage of a large amount of paths, it could alse be interesting to study different abstract domains for the over-approximation. When the ACFR algorithms were developed, it was ensured that they would not disable the usage of abstract domains which requrie precision elements and precision refinement. As such, it could be interesting to evaluate our alternating approach using some of the other abstract domains that were originally developed for Jakstab. Furthermore, Jakstab also includes the possibility to combine multiple abstract domains into a combined abstract domain. As such, it might also be of interest to study which groups of abstract domains perform best with DSE.

Moreover, it could also be of value to study other under-approximation approaches than DSE in the context of our alternating algorithms. As briefly discussed in the previous section, this should not be more complicated than to modify the semantics of the DSE function. For example, the DSE funciton could instead utilize the provided paths and targets for performing white box fuzz testing [37] to acquire successor locations of the target locations. It could then be interesting to evaluate if the white-box fuzzing can determine more successors than DSE.

Bibliography

[1] Liang Xu, Fangqi Sun and Zhendong Su. 'Constructing Precise Control Flow Graphs from Binaries'. *University of California, Davis, Tech. Rep* (2010).

[2] Claud Xiao. *Novel Malware XcodeGhost Modifies Xcode, Infects Apple iOS Apps and Hits App Store*. 2015. URL: `https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/` (visited on 03/08/2019).

[3] Dawn Song et al. 'BitBlaze: A New Approach to Computer Security via Binary Analysis'. *Proceedings of the 4th International Conference on Information Systems Security*. ICISS '08. Springer-Verlag, 2008, pp. 1–25.

[4] David Brumley et al. 'BAP: A Binary Analysis Platform'. *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Springer Berlin Heidelberg, 2011, pp. 463–469.

[5] Andreas Lindner, Roberto Guanciale and Roberto Metere. 'TrABin: Trustworthy analyses of binaries'. *Science of Computer Programming* 174 (2019), pp. 72–89.

[6] Gogul Balakrishnan et al. 'CodeSurfer/x86—A Platform for Analyzing x86 Executables'. *Compiler Construction*. Ed. by Rastislav Bodik. Springer Berlin Heidelberg, 2005, pp. 250–254.

[7] Yan Shoshitaishvili et al. 'SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis'. *IEEE Symposium on Security and Privacy*. 2016.

[8] Nicholas Nethercote and Julian Seward. 'Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation'. *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100.

[9] Thomas Reinbacher and Jörg Brauer. 'Precise control flow reconstruction using Boolean logic'. *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*. Oct. 2011, pp. 117–126.

[10] Johannes Kinder and Dmitry Kravchenko. 'Alternating Control Flow Reconstruction'. *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI'12. Philadelphia, PA: Springer-Verlag, 2012, pp. 267–282.

[11] Johannes Kinder. 'Static Analysis of x86 Executables'. PhD thesis. Technische Universität Darmstadt, Nov. 2010.

[12] Bogdan Mihaila. 'Control flow reconstruction from PowerPC binaries'. MA thesis. Technical University of Munich, 2009.

[13] Thomas A. Henzinger et al. 'Lazy Abstraction'. *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '02. Portland, Oregon: ACM, 2002, pp. 58–70.

[14] Martín Abadi et al. 'Control-flow Integrity Principles, Implementations, and Applications'. *ACM Trans. Inf. Syst. Secur.* 13.1 (Nov. 2009), 4:1–4:40.

[15] Bjorn De Sutter et al. 'On the Static Analysis of Indirect Control Transfers in Binaries'. *Proceedings of the international conference on parallel and distributed processing techniques and applications*. Vol. 2. 2000, pp. 1013–1019.

[16] Manish Prasad and Tzi-cker Chiueh. 'A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks.' *USENIX Annual Technical Conference, General Track*. 2003, pp. 211–224.

[17] Benjamin Schwarz, Saumya Debray and Gregory Andrews. 'Disassembly of executable code revisited'. *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* Nov. 2002, pp. 45–54.

[18] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, Inc., 2005.

[19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 2009.

[20] Srinivas Raman, Vladimir Pentkovski and Jagannath Keshava. 'Implementing streaming SIMD extensions on the Pentium III processor'. *IEEE Micro* 20.4 (July 2000), pp. 47–57.

[21] Giovanni Vigna. 'Static disassembly and code analysis'. *Malware Detection*. Springer, 2007, pp. 19–41.

[22] Valentin J. M. Manès et al. 'Fuzzing: Art, Science, and Engineering'. *CoRR* abs/1812.00140 (2018).

[23] Patrick Cousot. *Abstract Interpretation in a Nutshell*. 2019. URL: https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html (visited on 14/07/2019).

[24] Encylopedia of Mathematics. *Partially ordered set*. 2017. URL: https://www.encyclopediaofmath.org/index.php/Partially_ordered_set (visited on 14/07/2019).

[25] Ferdinand Börner. 'Basics of Galois Connections'. *Complexity of Constraints: An Overview of Current Research Themes*. Ed. by Nadia Creignou, Phokion G. Kolaitis and Heribert Vollmer. Springer Berlin Heidelberg, 2008, pp. 38–67.

[26] Agostino Cortesi. 'Widening operators for abstract interpretation'. *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*. IEEE. 2008, pp. 31–40.

[27] Dirk Beyer, Thomas A. Henzinger and Grégory Théoduloz. 'Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis'. *International Conference on Computer Aided Verification*. Springer. 2007, pp. 504–518.

[28] Jean-François Collard and Jens Knoop. 'A Comparative Study of Reaching-definitions Analyses' (1998).

[29] Mark Probst, Andreas Krall and Bernhard Scholz. 'Register Liveness Analysis for Optimizing Dynamic Binary Translation'. *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE. 2002, pp. 35–44.

[30] Todd C. Mowry. *Introduction to Data Flow Analysis*. 2003. URL: https://www.cs.cmu.edu/afs/cs/academic/class/15745-s03/public/lectures/L4_handouts.pdf (visited on 29/08/2019).

[31] Johannes Kinder. *jkinder/jakstab: The Jakstab static analysis platform for binaries*. 2017. URL: https://github.com/jkinder/jakstab (visited on 03/06/2019).

[32] Felipe Manzano. personal communication. 30 Oct. 2019.

[33] Dirk Beyer, Thomas A. Henzinger and Grégory Théoduloz. 'Program Analysis with Dynamic Precision Adjustment'. *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE '08. IEEE Computer Society, 2008, pp. 29–38.

[34] Harold Pashler and Eric–Jan Wagenmakers. 'Editors' Introduction to the Special Section on Replicability in Psychological Science: A Crisis of Confidence?' *Perspectives on Psychological Science* 7.6 (2012). PMID: 26168108, pp. 528–530.

[35] Robert E. Tarjan. 'Depth-First Search and Linear Graph Algorithms'. *SIAM J. Comput.* 1 (1972), pp. 146–160.

[36] Micha Sharir. 'A strong-connectivity algorithm and its applications in data flow analysis'. *Computers & Mathematics with Applications* 7.1 (1981), pp. 67–72.

[37] Patrice Godefroid, Michael Y. Levin and David Molnar. 'Automated Whitebox Fuzz Testing'. *NDSS*. Vol. 8. Citeseer. 2008, pp. 151–166.

Appendix A

Benchmarking

This appendix contains tables presenting the execution times for all the experiments which were presented in the Results chapter. Each table consists of seven columns. The first column is the name of the binary and also contains a $\triangle_T$ if the analysis had to be interrupted after 2 hours or a $\triangle_M$ if the analysis experienced an out of memory error. The second column represents the analysis mode. A c denotes constant propagation while an i represents strided interval analysis. Furthermore, a cD and iD denotes constant propagation with DSE and strided interval analysis with DSE respectively. Finally, the last four columns show the time spent executing the over-approximation, DFS, DSE and time spent executing everything outside of these three components, in milliseconds.

Binary	Mode	Instructions	ms			
			$T_{Jakstab}$	T_{DFS}	T_{DSE}	T_{Other}
badCall	c	4	293	0	0	113
badCall	cD	7	264	0	680	198
badCall	i	7	256	0	0	145
badCall	iD	7	256	0	0	216
branch	c	17	323	0	0	123
branch	cD	17	463	1	684	204
branch	i	17	471	0	0	119
branch	iD	17	474	0	0	210
callRet	c	2	266	0	0	136
callRet	cD	6	267	3	657	208
callRet	i	6	254	0	0	148
callRet	iD	6	250	0	0	213
cjmp	c	11	291	0	0	127
cjmp	cD	11	290	0	0	203
cjmp	i	11	279	0	0	173
cjmp	iD	11	269	0	0	252
cjmpBitOp	c	7	277	0	0	122
cjmpBitOp	cD	7	278	0	0	194
cjmpBitOp	i	8	267	0	0	142
cjmpBitOp	iD	8	252	0	0	214
conditionalTop	c	11	303	0	0	129
conditionalTop	cD	11	285	1	766	211
conditionalTop	i	11	312	0	0	136
conditionalTop	iD	11	277	1	896	241
doubleCallRet	c	17	304	0	0	149
doubleCallRet	cD	22	301	0	794	223
doubleCallRet	i	22	287	0	0	166
doubleCallRet	iD	22	288	0	0	238
espAssignment	c	9	324	0	0	124
espAssignment	cD	9	277	4	528	343
espAssignment	i	9	326	0	0	129
espAssignment	iD	9	273	13	558	361
indMemJmp	c	6	269	0	0	123
indMemJmp	cD	6	261	0	0	196
indMemJmp	i	6	255	0	0	138
indMemJmp	iD	6	239	0	0	228
infiniteLoop	c	10	310	0	0	127
infiniteLoop	cD	10	287	0	0	205
infiniteLoop$\triangle_T$	i	10	7141883	0	0	63
infiniteLoop$\triangle_T$	iD	10	7141893	0	0	105
int	c	9	257	0	0	138
int	cD	9	265	0	0	190
int	i	9	239	0	0	141
int	iD	9	233	0	0	216
kindConditionalTop	c	8	285	0	0	120
kindConditionalTop	cD	8	282	0	659	190
kindConditionalTop	i	8	306	0	0	116
kindConditionalTop	iD	8	250	1	694	234

Table A.1: *Execution times for the synthetic evaluation with the first ACFR algorithm (Part 1). Analyses which had to be interrupted as they did not finish within 2 hours are marked with* $\triangle_T$

Binary	Mode	Instructions	ms			
			$T_{Jakstab}$	T_{DFS}	T_{DSE}	T_{Other}
longRegisterEquals	c	14	315	0	0	119
longRegisterEquals	cD	14	302	0	702	189
longRegisterEquals	i	14	319	0	0	142
longRegisterEquals	iD	14	266	1	559	219
memCallRet	c	4	287	0	0	116
memCallRet	cD	8	285	1	589	196
memCallRet	i	8	261	0	0	141
memCallRet	iD	8	255	0	0	224
minimalistic	c	5	276	0	0	132
minimalistic	cD	11	280	0	632	215
minimalistic	i	11	328	0	0	132
minimalistic $\triangle_T$	iD	15	7148928	9	626	291
nestedCallRet	c	3	272	0	0	122
nestedCallRet	cD	8	270	1	1160	199
nestedCallRet	i	8	260	0	0	154
nestedCallRet	iD	8	238	0	0	231
registerEquals	c	8	301	0	0	127
registerEquals	cD	8	285	1	605	214
registerEquals	i	8	299	0	0	122
registerEquals	iD	8	261	3	761	230
sequentialCallRet	c	2	276	0	0	133
sequentialCallRet	cD	3	258	0	773	188
sequentialCallRet	i	7	271	0	0	167
sequentialCallRet	iD	7	260	0	0	231
stackBO	c	13	303	0	0	138
stackBO	cD	13	291	0	933	210
stackBO	i	22	281	0	0	151
stackBO	iD	22	278	0	0	232
trueTop	c	7	291	0	0	132
trueTop	cD	7	273	1	1053	201
trueTop	i	7	289	0	0	125
trueTop	iD	7	261	1	810	222
twoVars	c	7	305	0	0	137
twoVars	cD	11	278	0	632	210
twoVars	i	7	305	0	0	140
twoVars	iD	11	247	1	652	268
xyz	c	10	299	0	0	139
xyz	cD	13	290	1	742	218
xyz	i	10	309	0	0	112
xyz	iD	13	262	9	753	220

Table A.2: *Execution times for the synthetic evaluation with the first ACFR algorithm (Part 2). Analyses which had to be interrupted as they did not finish within 2 hours are marked with* $\triangle_T$

Binary	Mode	Instructions	ms			
			$T_{Jakstab}$	T_{DFS}	T_{DSE}	T_{Other}
badCall	c	4	301	0	0	110
badCall	cD	7	269	0	1202	203
badCall	i	7	254	0	0	139
badCall	iD	7	247	0	0	207
branch	c	17	320	0	0	123
branch	cD	17	442	2	2019	177
branch	i	17	477	0	0	121
branch	iD	17	473	0	0	208
callRet	c	2	280	0	0	134
callRet	cD	6	274	1	1152	183
callRet	i	6	248	0	0	146
callRet	iD	6	248	0	0	213
cjmp	c	11	290	0	0	122
cjmp	cD	11	282	0	0	209
cjmp	i	11	278	0	0	150
cjmp	iD	11	270	0	0	226
cjmpBitOp	c	7	273	0	0	129
cjmpBitOp	cD	7	291	0	0	183
cjmpBitOp	i	8	260	0	0	146
cjmpBitOp	iD	8	247	0	0	222
conditionalTop	c	11	312	0	0	121
conditionalTop	cD	11	285	1	1584	209
conditionalTop	i	11	319	0	0	126
conditionalTop	iD	11	270	0	2025	234
doubleCallRet	c	17	318	0	0	127
doubleCallRet	cD	22	296	0	1593	223
doubleCallRet	i	22	304	0	0	164
doubleCallRet	iD	22	296	0	0	225
espAssignment	c	9	311	0	0	126
espAssignment	cD	9	281	5	592	328
espAssignment	i	9	338	0	0	138
espAssignment	iD	9	282	13	578	363
indMemJmp	c	6	269	0	0	122
indMemJmp	cD	6	270	0	0	202
indMemJmp	i	6	246	0	0	151
indMemJmp	iD	6	247	0	0	229
infiniteLoop	c	10	297	0	0	132
infiniteLoop	cD	10	299	0	0	189
infiniteLoop$\triangle_T$	i	10	7141982	0	0	63
infiniteLoop$\triangle_T$	iD	10	7141871	0	0	114
int	c	9	268	0	0	123
int	cD	9	273	0	0	197
int	i	9	247	0	0	148
int	iD	9	239	0	0	210
kindConditionalTop	c	8	300	0	0	121
kindConditionalTop	cD	8	274	1	1400	194
kindConditionalTop	i	8	291	0	0	122
kindConditionalTop	iD	8	261	1	1374	205

Table A.3: *Execution times for the synthetic evaluation with the second ACFR algorithm (Part 1). Analyses which had to be interrupted as they did not finish within 2 hours are marked with* $\triangle_T$

Binary	Mode	Instructions	ms			
			$T_{Jakstab}$	T_{DFS}	T_{DSE}	T_{Other}
longRegisterEquals	c	14	303	0	0	133
longRegisterEquals	cD	14	301	1	949	193
longRegisterEquals	i	14	308	0	0	126
longRegisterEquals	iD	14	258	2	605	246
memCallRet	c	4	284	0	0	124
memCallRet	cD	8	271	0	1092	182
memCallRet	i	8	255	0	0	161
memCallRet	iD	8	245	0	0	218
minimalistic	c	5	268	0	0	146
minimalistic	cD	35	574	44	4615	725
minimalistic	i	11	365	0	0	130
minimalistic$\triangle_T$	iD	15	7148204	7	686	293
nestedCallRet	c	3	271	0	0	133
nestedCallRet	cD	8	279	1	1760	199
nestedCallRet	i	8	253	0	0	154
nestedCallRet	iD	8	278	0	0	211
registerEquals	c	8	291	0	0	133
registerEquals	cD	8	268	1	624	206
registerEquals	i	8	299	0	0	116
registerEquals	iD	8	263	1	640	216
sequentialCallRet	c	2	269	0	0	121
sequentialCallRet	cD	7	277	22	3644	655
sequentialCallRet	i	7	271	0	0	150
sequentialCallRet	iD	7	264	0	0	222
stackBO	c	13	309	0	0	132
stackBO	cD	13	281	0	1620	221
stackBO	i	22	278	0	0	154
stackBO	iD	22	264	0	0	218
trueTop	c	7	283	0	0	131
trueTop	cD	7	265	0	1847	202
trueTop	i	7	275	0	0	109
trueTop	iD	7	256	0	2121	209
twoVars	c	7	293	0	0	143
twoVars	cD	11	283	2	1292	201
twoVars	i	7	298	0	0	145
twoVars	iD	11	249	1	1388	249
xyz	c	10	317	0	0	118
xyz	cD	13	280	0	1418	212
xyz	i	10	297	0	0	139
xyz	iD	13	262	0	1471	242

Table A.4: *Execution times for the synthetic evaluation with the second ACFR algorithm (Part 2). Analyses which had to be interrupted as they did not finish within 2 hours are marked with* $\triangle_T$

Binary	Mode	Instructions	ms			
			$T_{Jakstab}$	T_{DFS}	T_{DSE}	T_{Other}
echo	c	10	293	0	0	302
echo $\triangle_M$	cD	59	1315	32	4064	957
echo $\triangle_T$	i	1253	7157728	0	0	241
echo $\triangle_T$	iD	1346	7157956	0	0	570
false	c	10	293	0	0	322
false $\triangle_M$	cD	59	1396	56	4302	968
false $\triangle_T$	i	1253	7153636	0	0	206
false $\triangle_T$	iD	1346	7156047	0	0	280
make-prime-list	c	10	297	0	0	303
make-prime-list $\triangle_M$	cD	59	1142	35	3633	644
make-prime-list $\triangle_T$	i	1346	7153572	0	0	207
make-prime-list $\triangle_T$	iD	1346	7156736	0	0	294
printenv	c	10	283	0	0	328
printenv $\triangle_M$	cD	59	1164	32	3582	639
printenv $\triangle_T$	i	1253	7153785	0	0	216
printenv $\triangle_T$	iD	1346	7156810	0	0	296
true	c	10	300	0	0	302
true $\triangle_M$	cD	59	1192	36	3459	755
true $\triangle_T$	i	1253	7158645	0	0	221
true $\triangle_T$	iD	1346	7157823	0	0	290

Table A.5: *Execution times for the evaluation of the 5 smallest GNU coreutils binaries using the first ACFR algorithm. Analyses which had to be interrupted after 2 hours and analyses that terminated because of insufficient memory are marked with $\triangle_T$ and $\triangle_M$ respectively.*

Binary	Mode	Instructions	ms			
			$T_{Jakstab}$	T_{DFS}	T_{DSE}	T_{Other}
echo	c	10	288	0	0	310
echo $\triangle_M$	cD	59	1147	47	3983	687
echo $\triangle_T$	i	1253	7155590	0	0	218
echo $\triangle_T$	iD	1346	7154816	0	0	308
false	c	10	324	0	0	304
false $\triangle_M$	cD	59	1531	39	4840	975
false $\triangle_T$	i	1253	7154597	0	0	221
false $\triangle_T$	iD	1346	7153839	0	0	285
make-prime-list	c	10	291	0	0	277
make-prime-list $\triangle_M$	cD	59	1111	34	3503	666
make-prime-list $\triangle_T$	i	1346	7154939	0	0	210
make-prime-list $\triangle_T$	iD	1346	7153194	0	0	270
printenv	c	10	287	0	0	312
printenv $\triangle_M$	cD	59	1196	34	4041	755
printenv $\triangle_T$	i	1253	7153311	0	0	211
printenv $\triangle_T$	iD	1346	7154729	0	0	354
true	c	10	303	0	0	305
true $\triangle_M$	cD	59	1185	29	3795	715
true $\triangle_T$	i	1253	7153809	0	0	219
true $\triangle_T$	iD	1346	7156424	0	0	289

Table A.6: *Execution times for the evaluation of the 5 smallest GNU coreutils binaries using the second ACFR algorithm. Analyses which had to be interrupted after 2 hours and analyses that terminated because of insufficient memory are marked with $\triangle_T$ and $\triangle_M$ respectively.*

TRITA -EECS-EX-2019:835