ANDREI RIMSA ÁLVARES

PRACTICAL DYNAMIC RECONSTRUCTION OF CONTROL FLOW GRAPHS

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Fernando Magno Quintão Pereira
Co-Advisor: José Nelson Amaral

Belo Horizonte

November 5, 2020

Álvares, Andrei Rimsa.

A473p Practical dynamic reconstruction of control flow graphs [manuscrito] /Andrei Rimsa Álvares. — 2020.
xxv, 88 f.; il.; 29cm.

Orientador: Fernando Magno Quintão Pereira.
Coorientador: José Nelson Amaral.

Tese (doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 83-88

1. Computação – Teses. 2. Compiladores (Programas de computador) – Teses. 3. Linguagem de programação (Computadores) – Teses. 4. Fluxo de dados (Computação) – Teses. I. Pereira, Fernando Magno Quintão. II. Amaral, José Nelson. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV. Título.

CDU 519.6*33(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg Lucas Cruz CRB 6ª Região nº 819.

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Practical Dynamic Reconstruction of Control Flow Graphs

ANDREI RIMSA ÁLVARES

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

Fernando M Q Pereira
Prof. Fernando Magno Quintão Pereira - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]
Prof. José Nelson Amaral
Department of Computing Science - University of Alberta

José Eduardo Moreira
Dr. José Eduardo Moreira
IBM Research - IBM

Rodolfo Azevedo
Prof. Rodolfo Jardim de Azevedo
Instituto de Computação - UNICAMP

[illegible]
Prof. Leonardo Barbosa e Oliveira
Departamento de Ciência da Computação - UFMG

Marcos Augusto Menezes Vieira
Prof. Marcos Augusto Menezes Vieira
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 5 de Novembro de 2020.

To my parents, brothers and wife.

Acknowledgments

I am thankful…

To my advisor Fernando for taking me as a student earlier in the masters and later in the phd. His friendship and partnership through all these years have been invaluable for my growth as a person, student and professor.

To my co-advisor Nelson for the insights and support through all of this work. His kindness is always with me.

To my friends in the compiler's laboratory for the fruitful discussions on this thesis, computers and life in general. You guys and girls are awesome.

To my fellow professors in the department of computing of CEFET-MG for the pleasant working environment. This made the persuit of this work sustainable in the long run.

To my parents, Frederico and Zilza, for my upbringing. Their unconditional support have been crucial for the accomplishment of every milestone in my life, including this one.

To my brothers, Vitor and Ivan, for being there with me all the time, even afar. This unity is paramount in my life.

And to my wife May for the love and care during the adversities of this path. I love you.

"Life begins at the end of your comfort zone."
(Neale Donald Walsch)

Resumo

A recuperação automática de informações de alto-nível de programas em formato binário é um importante problema estudado em linguagens de programação. Contudo, a maioria das soluções para esse problema são baseadas puramente em abordagens estáticas: técnicas como análise de fluxo de dados ou inferência de tipos são utilizadas para converter os bytes que constituem o executável de volta para o formato de um grafo de fluxo de controle (GFC). Esse trabalho se afasta desse tal modus operandi para mostrar que análises dinâmicas podem ser efetivas e úteis, tanto como uma técnica independente, quanto como uma forma de melhorar a precisão das abordagens estáticas. Os resultados experimentais mostram evidências que completude, ou seja, a habilidade de concluir que todos os caminhos de um GFC foram cobertos, é alcançada em muitas funções de benchmarks de nível industrial. Os experimentos também indicam que informações coletadas dinamicamente melhoram consideravelmente a habilidade de DynInst, um reconstrutor estático estado-da-arte, de lidar com códigos binários sem símbolos de depuração. Esses resultados foram obtidos com CFGgrind, um reconstrutor dinâmico de códigos binários, construído sobre a infraestrutura de valgrind. Quando aplicado sobre cBench, CFGgrind é 9% mais rápido que callgrind, uma ferramenta de valgrind capaz de rastrear alvos de chamadas de funções; e 7% mais rápido em Spec Cpu2017. CFGgrind recupera GFCs completos em 40% de todos os procedimentos invocados durante a execução padrão de programas em Spec Cpu2017, e 37% em cBench. Quando combinado com CFGgrind, DynInst encontra 15% mais GFCs para cBench e 7% mais GFCs para Spec Cpu2017. Finalmente, CFGgrind é 7 vezes mais rápido que DCFG, um reconstrutor de GFC desenvolvido pela Intel, e é 1.28 vezes mais rápido que bfTrace, um reconstrutor usado em pesquisa. CFGgrind é também mais preciso que essas duas ferramentas. Ele suporta tratamento de sinais de sistema operacional, códigos compartilhados em funções, instruções desalinhadas, programas multi-thread, profiling exato e refinamentos incrementais.

Palavras-chave: Grafo de fluxo de controle, Análise dinâmica, Instrumentação.

Abstract

The automatic recovery of a program's high-level representation from its binary version is a well-studied problem in programming languages. However, most of the solutions to this problem are based on purely static approaches: techniques such as dataflow analyses or type inference are used to convert the bytes that constitute the executable code back into a control flow graph (CFG). This work departs from such a modus operandi to show that a dynamic analysis can be effective and useful, both as a standalone technique, and as a way to enhance the precision of static approaches. The experimental results provide evidence that completeness, i.e., the ability to conclude that the entire CFG has been discovered, is achievable on many functions that are part of industry-strong benchmarks. Experiments also indicate that dynamic information greatly enhances the ability of DynInst, a state-of-the-art binary reconstructor, to deal with code stripped of debugging information. These results were obtained with CFGgrind, a new implementation of a dynamic code reconstructor, built on top of valgrind. When applied to cBench, CFGgrind is 9% faster than callgrind, valgrind's tool used to track targets of function calls; and 7% faster in Spec Cpu2017. CFGgrind recovers the complete CFG of 40% of all the procedures invoked during the standard execution of programs in Spec Cpu2017, and 37% in cBench. When combined with CFGgrind, DynInst finds 15% more CFGs for cBench, and 7% more CFGs for Spec Cpu2017. Finally, CFGgrind is more than 7 times faster than DCFG, a CFG reconstructor from Intel, and 1.28 times faster than bfTrace, a CFG reconstructor used in research. CFGgrind is also more precise than these two tools, handling operating system signals, shared code in functions, and unaligned instructions; besides supporting multi-threaded programs, exact profiling and incremental refinements.

Keywords: Control flow graph, Dynamic analysis, Code instrumentation.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The Control Flow Graph (CFG) [Aho et al., 2006, p.525] is a fundamental structure supporting the analysis and optimization of programs. A CFG is a directed graph where the vertices represent *basic blocks*. A basic block is a maximal sequence of instructions without branches, except at the end. Edges in the CFG denote possible program flows. Since its introduction in the 70's, likely due to the work of Allen [1970], CFGs have emerged as a mandatory program representation adopted in compilers, virtual machines and program verifiers. In program analyses based on source code, a CFG is produced either directly from that source code or from some high-level intermediate representation. However, there exists also much interest in recovering the CFG from the program's binary representation, as many researchers have demonstrated throughout the 90's [Cifuentes and Gough, 1995; Schwarz et al., 2002; Sites et al., 1993; Theiling, 2000]. However, while the construction of a CFG from source has a trivial solution, and is routinely performed by compilers, the reconstruction of a CFG from the binary representation is undecidable. Undecidability is easy to see: indirect branches, plus a simple extension of Rice's Theorem [Rice, 1953], hinder any algorithm from determining with certainty every possible flow in a program.

There are two ways to recover a CFG from a program's binary representation. The first approach, henceforth called *static*, tries to recover the program flow via static analysis of the binary program, i.e., from its `.text` section [Sutter et al., 2000; Kästner and Wilhelm, 2002; Bruschi et al., 2007]. This is the technique of choice employed by a number of well-known tools, such as DynInst [Meng and Miller, 2016], Bap [Brumley et al., 2011], Jakstab [Kinder and Veith, 2008], SecondWrite [Smithson et al., 2013], IDA Pro [Eagle, 2011], GNUObjdump[1], and OllyDbg[2]. The second approach, henceforth called *dynamic*, seeks to construct a CFG out of instruction traces generated during the execution of a program. The dynamic reconstruction of CFGs is not as wide-spread as its static counterpart. We know one industrial-strength tool that provides such capability: Yount's DCFG [Yount et al., 2015], a software built on top of Intel's Pin [Luk et al., 2005], and released in 2015. Dynamic CFG builders can also be found as part of different research artifacts [Gruber et al., 2019; Xu et al., 2009; Shoshitaishvili et al., 2016], a few of which are publicly available[3].

Static and dynamic approaches yield different results. Whereas the static approach gives a conservative approximation of the program's control flow, possibly containing paths that might never be traversed, the dynamic approach gives an under-approximation of the program flow. Every flow discovered by a dynamic tool is a true path within the execution of the program analyzed. However, the dynamic technique might miss paths that are not exercised by the inputs used in the reconstruction. Such differences lead to distinct applications. Static CFG reconstruction is typically used for security analyses [Song et al., 2008] and binary optimization [Panchenko et al., 2019; Zhou and Jones, 2019]. Dynamic reconstruction, in turn, is used to build dynamic slices [Agrawal and Horgan, 1990; Korel and Laski, 1988], and finds services in any situation where such slices are in need [Tip, 1994], such as malware detection, deobfuscation and profiling. Nevertheless, in spite of three decades of progress in dynamic slicing, the dynamic reconstruction of CFGs is still poorly understood, its benefits are often understated, and its engineering still leaves much room for improvement. Motivated by such observations, this work brings the following thesis:

> Dynamic CFG reconstructors can be practical tools able to augment the coverage of static reconstructors and provide users with completeness information.

1.1 Contributions

This work brings the following improvements to the recovery of CFGs from binary code:

[1]GNUObjdump is a disassembler for GNU Linux. To know more, see `https://www.gnu.org/software/binutils/`.

[2]OllyDbg is a disassembler for Microsoft Windows. To know more, see `http://www.ollydbg.de/`.

[3]As an example, tools available at `https://docs.angr.io/`, and `https://github.com/toshipiazza/LLVMCFG` provide some limited form of CFG reconstruction.

Completeness: a new definition to quantify the coverage of CFG reconstruction (Chapter 2) and an empirical evaluation (Section 5.3) that reveals that a standard execution of the Spec Cpu2017 suite yields complete CFGs for 40% of the invoked functions. For cBench this number is similar: 37%.

Precision: a suite of techniques that, once combined, yield more precise CFGs than the state-of-the-art approaches available today. Chapter 3 explains how our techniques support precise profiling information, deal with overlapping instructions and code shared by different functions, handle signals from the operating system, support multi-threaded programs and the incremental construction of CFGs from multiple inputs.

Efficiency: new algorithms (Chapter 4) that support faster reconstruction of CFGs than state-of-the-art dynamic reconstructors. Our approach is ∼7x faster than DCFG, a tool built over Intel's PinPlay, and ∼28% faster than bfTrace, the reconstructor from Gruber et al. [2019]. Our efficiency is due to extensive use of caching, as Section 5.2 shows.

Complementarity: the demonstration that static and dynamic analyses can be combined to generate more complete CFGs. Section 5.3 shows that the combination of our technique with DynInst, a state-of-the-art static CFG builder [Meng and Miller, 2016], increases coverage in cBench from 42% to 57%, and in Spec Cpu2017 from 39% to 46%.

The importance of the techniques introduced in this thesis are demonstrated in the following tools, produced as artifacts of this work:

1. CFGgrind (`https://github.com/rimsa/CFGgrind`), a dynamic CFG reconstructor that runs in valgrind's infrastructure. The internals of this tool is described in details in Chapters 2 and 4.

2. dumpcfgs (`https://github.com/rimsa/dumpcfgs`), a static CFG reconstructor that uses the DynInst API. This tool is used as a counterpart in the evaluation of the the dynamic CFGgrind tool in Chapter 5.

3. cmpcfgs (`https://github.com/rimsa/cmpcfgs`), a tool to compare the outputs produced by CFGgrind and dumpcfgs. The precision of both tools is compared in Section 5.5.

4. instrgrind (`https://github.com/rimsa/instrgrind`), a tool to count executed instructions. This tool is used in the case study presented in Chapter 6.

CFGgrind is mature enough to be used on every program of Spec Cpu2017. It supports the reconstruction of CFGs for programs that run in parallel. It also admits incremental construction of CFGs, meaning that a partial CFG built during one run of the program can be retrofitted into a new execution with different inputs in order to complement it. Thus, if new paths are traversed, more information is added to the CFG. This feature is specially important for programs that require multiple runs to construct a complete CFG. CFGgrind can be used in tandem with DynInst, a static binary analyzer, allowing it to discover the target of dynamic jumps, and to handle difficult code sections that would be missed in programs stripped of symbols and debugging information. Additionally, CFGgrind provides exact profiling information. Contrary to sampling based techniques, it tracks how many times every instruction of the target program was executed, respecting the equity of flows: the number of program flows that enter any basic block equals the number of flows that leave it.

1.2 Publications

This work led to the publication of two papers. A first version of CFGgrind with preliminary results was published in the 2019 edition of Simpósio Brasileiro de Linguagens de Programação (SBLP). Then, a more mature version of CFGgrind with several improvements was developed. This enhancements combined with stronger results and better comparison between other CFG reconstructions tools was published in Software: Practice and Experience in 2020, a reputable Wiley's journal. This latter paper is responsible for the bulk of this thesis.

- Andrei Rimsa, José Nelson Amaral, and Fernando Magno Quint ao Pereira. 2019. Efficient and Precise Dynamic Construction of Control Flow Graphs. In Proceedings of the XXIII Brazilian Symposium on Programming Languages (SBLP 2019). ACM, New York, NY, USA, 19-26. DOI: `https://doi.org/10.1145/3355378.3355383`

- Andrei Rimsa, José Nelson Amaral, and Fernando Magno Quint ao Pereira. 2020. Practical dynamic reconstruction of control flow graphs. In Softw Pract Exper. 2020; 1– 32. DOI: `https://doi.org/10.1002/spe.2907`

1.3 Outline

This thesis is structured in the following seven chapters:

Chapter 2. This chapter provides preliminary definitions. It builds from the definition of an instruction, passes through the notion of groups, and leads to the definition of what is a control flow graph. Later, the concept of CFG completeness is presented.

Chapter 3. This chapter discusses five desired features that, despite decades of dynamic CFG reconstruction knowledge, are still not supported or partially handled by many tools. A comparison against CFGgrind exposes the deficiencies of such tools.

Chapter 4. This chapter dives into how to perform the dynamic reconstruction of CFGs. First, an abstract machine that simulates the execution of a program is defined. Then, algorithms responsible for the reconstruction are provided. Also, a discussion about how to optimize them using caches and support for parallelism.

Chapter 5. This chapter handles the evaluation of the proposed solution. This evaluation is classified into five research questions, namely: how efficient is CFGgrind, how the cache impact performances, what is the ratio between complete and incomplete CFGs, what is the impact of different input sets in the incremental refinement of CFGs, and how much information CFGgrind adds to a static reconstructor.

Chapter 6. This chapter provides a case study on the visibility of instructions in the Spec Cpu2017 benchmark.

Chapter 7. This chapter compares our solution to related works. CFGgrind is analyzed against other dynamic CFG reconstruction tools. How CFGgrind relates to dynamic program slicing. Finally, how CFGgrind compares against static CFG reconstructors.

Chapter 8. This chapter provides final remarks and future directions for this work.

Chapter 2

Preliminary Definitions

The definition of a CFG is readily available in any compiler textbook; however, given its central role in this thesis, this chapter revisits it. This chapter provides preliminary definitions for an instruction (Definition 1 in Section 2.1), and a group of instructions (Definition 2 in Section 2.2). These concepts are the required for the definition of control flow graphs (Definition 3 in Section 2.3). This formalism might differ from standard definitions because it uses a number of terms that are necessary to explain the CFG reconstruction algorithm in Chapter 4. This chapter brings, to the best of our knowledge, a novel definition of completeness (Definition 4 in Section 2.4). This definition allows the classification of the reconstructed CFG in respect to coverage: if all the paths of a given CFG were explored by the execution. This characteristic of completeness is further analysed in Section 5.3 of the evaluation (Chapter 5).

2.1 Instruction

The building blocks of a CFG are instructions. In the binary representation of a program, each instruction is bound to an address. Each instruction also has an associated textual representation, e.g. *push %rbp*. An instruction can be formally defined as follows:

Definition 1. An **instruction** is a tuple $I = (@addr, size, type, text)$, where $@addr$ is the address of I in memory; $size$ is the space that I occupies, measured in bytes; $type$ represents a class to which I belongs; and $text$ is the assembly textual representation of I. For the purposes of this thesis, instructions are classified according to their effect on the flow of control of the program. Therefore an instruction belongs to one of the following types:

***standard*:** flows to the next instruction;

***jump(@target, mode: (direct | indirect))*:** unconditionally jumps to *@target* address, either directly or indirectly;

***branch(@target, @fallthrough, mode: (direct | indirect))*:** conditionally branches to *@target* or *@fallthrough* address, either directly or indirectly;

***call(@target, mode: (direct | indirect))*:** invokes the function stored at the *@target* address, either directly or indirectly;

***return*:** transfers control back to caller;

The *standard* instructions flow the execution to the instruction immediately after it. The *jump*, *branch* and *call* instructions can transfer control flow directly — the address is embedded in the instruction itself (e.g.: *jmp @addr*); or indirectly — the address is computed either from registers or memory (e.g.: *jmp %rax*). A NIL value is used as the *@target* address in case of indirect control flows. A *return* instruction transfers the execution back to the caller using the address immediately after the corresponding function call; hence, it behaves like an indirect branch. A *return* instruction is usually used to terminate a function, but it can also be used for irregular control flows, either maliciously or not. The tuple (*@*`0x400580`*, 2, branch(@*`0x40058c`*, @*`0x400582`*, direct), 'jg 0x40058c'*) is an example of an instruction.

2.2 Group of Instructions

Instructions can be logically organized in groups if they can be executed in sequence, without diverging the execution flow. A group of instructions can be formally defined as follows:

Definition 2. A **group** is an ordered sequence of instructions $S = \{I_1, I_2, \ldots, I_n\}$ containing at least one instruction ($|S| > 0$). The instructions in a group are consecutive in the program ($I_{n+1}.@addr = I_n.@addr + I_n.size$). The first instruction of a group is the *leader*. The last instruction is the *tail*. The *leader* is either the first instruction in a program, the target of a *jump*, *branch* or *call*, or the fall-through instruction of a non-taken *branch*. Instructions of type *jump*, *branch*, *call* and *return* cannot be followed by any other instruction.

Instructions are executed in order unless the program flow reaches an operation that diverts execution. Therefore, groups can be formed according to Definition 2 by tracing the sequential execution of instructions from a *leader* to a *tail*. The target of a *tail* instruction will be the *leader* of a next group to be formed. Thus, chains of groups are created during runtime. The sequence of instructions *{(@*`0x400597`*, 1, standard, 'leaveq'), (@*`0x400598`*, 1, return, 'retq')}* is an example of group, assuming that the program flow is diverted to @`0x400597` at some point during execution.

2.3 Control Flow Graph

With the definition of an instruction 1 and a group 2, we can proceed with the definition of a control flow graph (CFG):

Definition 3. A **Control Flow Graph (CFG)** is a connected, directed graph $G = (V, E)$, where:

- A node $n \in V$ must be in one of the following categories:

  ***entry*:** marks the start of the CFG.

  ***block(group, calls, signals)*:** is a basic block that contains:

  a *group*, according to Def. 2;

  a map of *calls* that associates the addresses of functions with pairs (CFG, count). The first element in the pair is the CFG of a function, and the second is the number of times that function was invoked by a call instruction in the *group*;

  a map of *signals*, similar to the map of calls, except that keys are signal ids, and the CFG, in the pair (CFG, count) is a signal handler with how many times it was invoked.

  ***phantom(@addr)*:** is an undiscovered node represented by its address.

  ***exit*:** marks the return of control to the caller of this CFG.

  ***halt*:** marks the stop of the execution of the program — no further instructions can be executed from this point forward.

- An edge $(n_1, n_2, count) \in E$ connects two nodes, n_1 and n_2 ($n_1, n_2 \in V$), with its execution count for profiling information, $count \in \mathbb{N}$, iff:

  One of the following conditions is true:

  1. The *tail* of n_1 is not an unconditional jump and the *leader* of n_2 immediately follows the tail of n_1 in program order.

2. The *leader* of n_2 is the target of a branch or jump instruction that is the *tail* of n_1.

And *count* is:

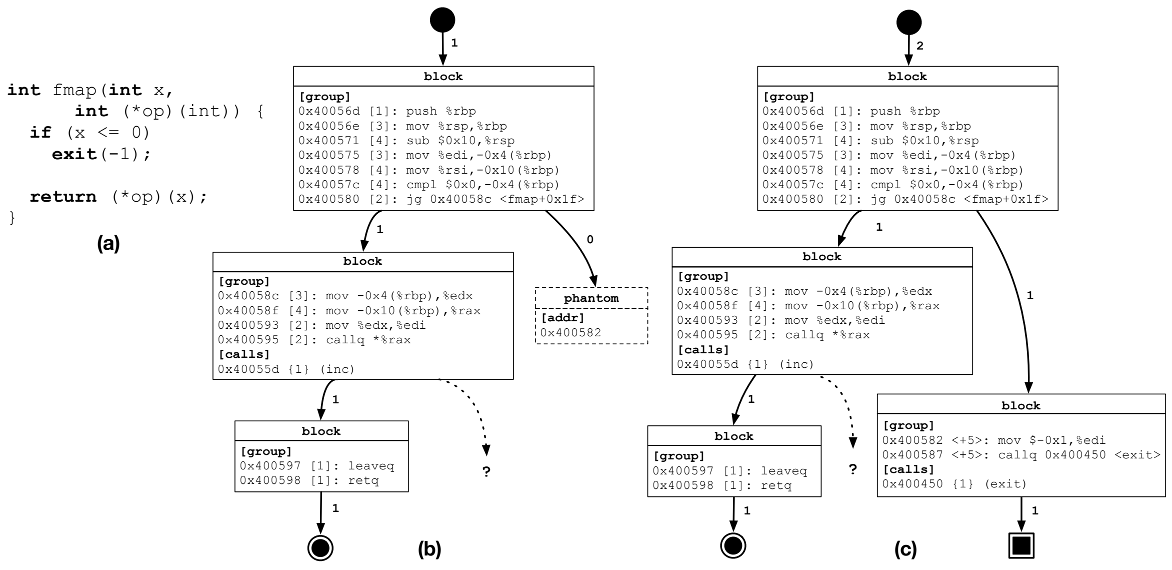
1. Zero, iff n_2 is a *phantom* node, or when profiling information is not required.
2. A positive integer with the exact count of how many times this edge was visited during execution.

- *Phantom*, *exit* and *halt* nodes have no successors. The *entry* node has no predecessors. Thus, given an edge $(n_1, n_2, count) \in E$, $n_1 \notin \{phantom, exit, halt\}$ and $n_2 \notin \{entry\}$.

During the reconstruction of a CFG, the algorithm may process branches whose un-taken target has not been visited thus far. These targets are represented by phantom nodes.

Example 2.3.1. Figure 2.1(a) shows the function FMAP written in C, and Figures 2.1(b,c) show two snapshots of FMAP's CFG. This function receives two parameters: an integer `x`; and a pointer to a function that returns an integer. It performs an indirect function call if `x` is greater than zero. For this example, consider that function `inc` — not shown in Fig. 2.1(a) — was called. The filled oval in Figures 2.1(b,c) are *entry* nodes. The double filled ovals represent *exit* nodes. A double filled square denotes the *halt* node. Note that there is only one *exit* node and/or *halt* node per CFG. Functions with multiple exit points, either that terminate the function or terminate the program, must be connected to their respective *exit* or *halt* nodes. Figure 2.1(b) contains three basic blocks, each one with a group of instructions. The block at address `@0x40058c` holds that a call to function INC was invoked one time. This target is the value stored in the function pointer `*op`. If FMAP is called with other arguments, more target functions will appear in the calls section of this block. Neither block contains invocations of signal handlers, and thus are not shown. The *phantom* nodes are represented with dashed outlines. The question mark represents possible unknown flows when the last instruction in the block is either a *jump*, *branch* or *call* node with *indirect* mode. Note that dashed edges are used to connect question marks, but they serve merely as an indication of an indirect flow for this block.

Previous works have modelled the entire program as a single CFG [Theiling, 2000; Bernat and Miller, 2012; Meng and Miller, 2016]. The boundary of functions can still be recorded in such representation, as long as edges in the CFG are marked



```
int fmap(int x,
      int (*op)(int)) {
  if (x <= 0)
    exit(-1);

  return (*op)(x);
}
```

```
[group]
0x40056d [1]: push %rbp
0x40056e [3]: mov %rsp,%rbp
0x400571 [4]: sub $0x10,%rsp
0x400575 [3]: mov %edi,-0x4(%rbp)
0x400578 [4]: mov %rsi,-0x10(%rbp)
0x40057c [4]: cmpl $0x0,-0x4(%rbp)
0x400580 [2]: jg 0x40058c <fmap+0x1f>
```

```
[group]
0x40058c [3]: mov -0x4(%rbp),%edx
0x40058f [4]: mov -0x10(%rbp),%rax
0x400593 [2]: mov %edx,%edi
0x400595 [2]: callq *%rax
[calls]
0x40055d {1} (inc)
```

```
[addr]
0x400582
```

```
[group]
0x400597 [1]: leaveq
0x400598 [1]: retq
```

```
[group]
0x400582 <+5>: mov $-0x1,%edi
0x400587 <+5>: callq 0x400450 <exit>
[calls]
0x400450 {1} (exit)
```

Figure 2.1. (a) Example program. (b) CFG after first call with positive argument for x. (c) Refined CFG after second call with negative argument for x.

as *intraprocedural* or *interprocedural*. This formalism departs from that convention: a CFG, according to Definition 3, represents the instructions of a single function. Formalizing a CFG in this way makes it easier to combine the CFG representation with information extracted from compilers such as GCC and LLVM. In particular, representing each function as a separate CFG facilitates the task of tracking the entry and exit points of procedures. In rare occasions, the binary representation of a program can be built in such a way that a set of instructions can be executed through calls to multiple addresses. Meng and Miller solved this problem by allowing a CFG to have multiple entry points [Meng and Miller, 2016]. We enforce a constraint that a CFG must have a single point of entry. Thus, in CFGGrind two different CFGs may execute a common subset of instructions. This duplication of information, however, has no penalties in the execution. Example 2.3.1 illustrate these concepts.

2.4 Completeness

When a CFG contains an indirect jump, an indirect branch, or an indirect call, it is not possible to ensure that all possible execution paths have been discovered. Future executions of the program with different workloads may follow new execution paths. Also, the presence of phantom nodes indicates the existence of paths that have not yet been discovered. The concept of CFG completeness, for the purposed of dynamic reconstruction, can be defined as follows:

Definition 4. Given a control flow graph $G = (V, E)$ (Definition 3), G is said to be **complete** iff, V contains an *entry* and at least an *exit* or a *halt* node; and $\forall n \in V$, the following conditions are true:

1. the successors of n are in V.
2. $n \neq phantom$.
3. if n is a *block* then the *mode* of the *tail* of the *group* of n is *direct*.

A CFG is complete if all its paths are known. Definition 4 uses a more restrictive notion of completeness: even if all indirections are proven to be constrained inside the same CFG, the existence of indirect jumps still classifies this CFG as incomplete.

Example 2.4.1. The examples in Figures 2.1(b-c) present an incomplete CFG because they contains both a *phantom* node and a *block* with an indirect call. Note that edges connecting phantom nodes, although known to exist, are never executed. Thus, they have a count of zero.

This classification is useful in the evaluation of complex code executions. For instance, malware programs may deliberately hide some part of their execution. In order to do so, they rely on constructs such as indirect jumps or calls to avoid exposing the address of the offending code. In such cases, CFGgrind will mark the CFGs as incomplete. In other words, code that contains indirect control flow will invariably contain either phantom nodes or an indirect marking — the question mark in Figure 2.1. Further executions of the same program, with different inputs, might improve coverage; hence, reducing the number of incomplete CFGs. The reconstruction of dynamic CFGs is based on successive refinements. Example 2.4.2 shows how re-execution refines CFGs.

Example 2.4.2. Figure 2.1(c) shows the CFG that results from a new activation of the same function, but with different arguments. In this case, the branch at @`0x400580` is not taken and leads to the discovery of the block at address @`0x400582`. In this example, the *phantom* node becomes a *block* node that is connected to the *halt* node.

2.5 Conclusion

This chapter covered important definitions — instruction (Def. 1), group of instruction (Def. 2), and control flow graph (Def. 3) — that are used by the dynamic reconstruction of CFGs by the algorithms discussed in Chapter 4. A new definition of completeness (Def. 4) was given to quantify the coverage of CFG reconstruction. The completeness of CFGs are evaluated in Section 5.3 (Chapter 5).

Chapter 3

The Need for CFG Reconstruction Tools

There are at least two tools that perform the dynamic reconstruction of control flow graphs, namely, DCFG [Yount et al., 2015] and BFTRACE [Gruber et al., 2019]. DCFG[1] is part of PINPLAY[2] — a framework for deterministically replaying a program execution. PINPLAY is publicly available, albeit closed-source. BFTRACE, in turn, is the first part of a four-staged implementation of dependence analysis [Gruber et al., 2019]. It builds intraprocedural control flow graphs and interprocedural call graphs. Revisiting these technologies, the need for further work in this area stems from two simple observations about state-of-the-art tools. On the one hand, the most precise of these tools, DCFG, incurs a heavy performance slowdown that makes its usage prohibitive in programs with long execution traces. On the other hand, BFTRACE, the faster dynamic analyzer, leaves too much information out from the CFGs that it reconstructs — namely, precise profiling data. This thesis shows that it is possible to reconstruct CFGs faster than BFTRACE, and still more exactly than DCFG. Chapter 5 provides empirical evidence to support this efficiency claim. This chapter explains why CFGGRIND's CFGs are more complete than similar structures produced by the other tools.

Table 3.1 presents a summary comparison of the three tools and indicates the section where each feature is discussed. Beware, however, that these tools are not strictly equivalent: being conceived with different goals, each of them has a distinct representation for CFGs. For instance, BFTRACE is part of a larger system whose purpose is to track dependencies between memory regions in order to advise for or

[1]DCFG: https://software.intel.com/en-us/articles/pintool-dcfg

[2]https://software.intel.com/en-us/articles/program-recordreplay-toolkit

against program parallelization. Nonetheless, bfTrace is a standalone application whose sole purpose is to reconstruct a program's CFGs and call graph. DCFG is also part of a larger system, PinPlay, which logs program state to allow re-execution, e.g., to support debugging. The code of DCFG is not open; hence, we cannot affirm that its only purpose is to reconstruct CFGs for PinPlay. Nevertheless, from what we could infer from DCFG's documentation, such seems to be the case.

Table 3.1. Qualitative comparison of the different tools considered in this work.

Feature	CFGgrind	bfTrace	DCFG	Section
Completeness	Reported	Absent	Absent	3.1
Program exit	Present	Absent	Absent	3.1
OS Signals	Tracked	Absent	Imperfect	3.1
Edge count	Present	Absent	Present	3.2
Flow equity	Present	Absent	Imperfect	3.2
Incremental analysis	Present	Absent	Absent	3.3
Multi-threading	Handled	Not handled	Handled	3.4
Overlapping instructions	Different	Different	Split	3.5
Shared code in functions	Duplicated	Duplicated	Shared	3.5

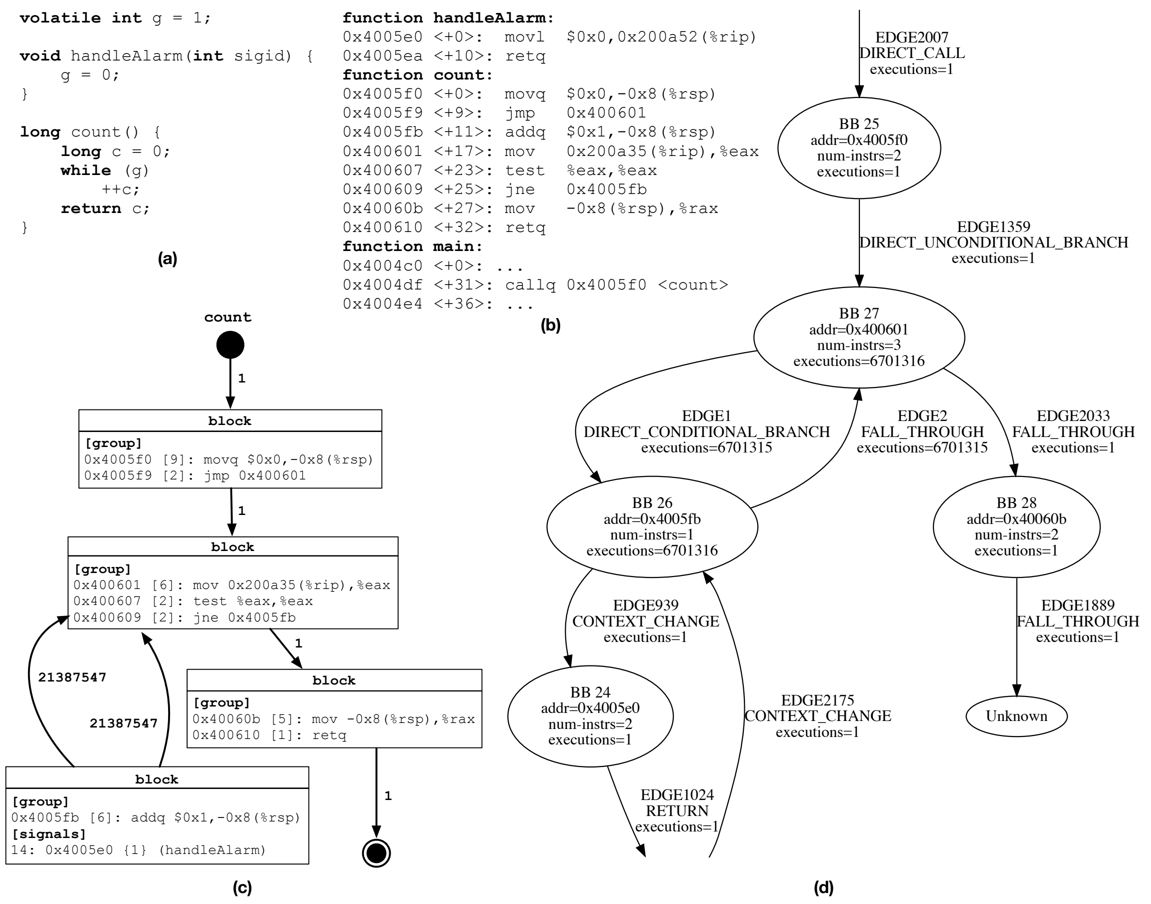
3.1 On the Precise Representation of CFGs

bfTrace, DCFG and CFGgrind adopt different representations for the program's control flow graph. CFGgrind and bfTrace associate a CFG for each identified program function, while DCFG provides a single, flattened, CFG for the entire program. However, the CFGs produced by CFGgrind have a few features that are absent from the CFGs produced by at least one, and sometimes both, of the other tools.

First, CFGgrind reports the **completeness**, a notion formalized in Definition 4, of a CFG. Neither DCFG nor bfTrace let users know if a CFG had all its basic blocks visited during the execution of the program. CFGgrind provides this functionality by augmenting the concept of a CFG with *phantom* nodes and annotations for indirect flows.

Second, the precise recognition of **exit points** is another feature missing in DCFG and bfTrace. These tools, like CFGgrind, track paths between different functions along the program's call graph. However, in both DCFG and bfTrace it is not possible to know if a basic block ends only a function, or terminates the entire program. Our experience using CFGGrind as a debugger tells us that such differentiation is important to correctly identify the points where no other instructions can be executed.

Third, CFGGrind tracks signal events that may occur during the program execution. Signals are particularly difficult to handle because they do not originate from specific instructions, e.g., `call` or `jmp`. Some instructions, such as `div`, `mod`, `store` and `load` can produce signals (`SIGSEGV`, `SIGILL`, `SIGFPE`, etc). Signals can come from outside the program, e.g., due to interruptions (`SIGINT`), or can be scheduled to happen, e.g., due to alarms (`SIGALRM`). Example 3.1.1 compares the support for this feature in CFGGrind in contrast with the other tools.



```
volatile int g = 1;

void handleAlarm(int sigid) {
    g = 0;
}

long count() {
    long c = 0;
    while (g)
        ++c;
    return c;
}
```

(a)

```
function handleAlarm:
0x4005e0 <+0>:   movl  $0x0,0x200a52(%rip)
0x4005ea <+10>:  retq
function count:
0x4005f0 <+0>:   movq  $0x0,-0x8(%rsp)
0x4005f9 <+9>:   jmp   0x400601
0x4005fb <+11>:  addq  $0x1,-0x8(%rsp)
0x400601 <+17>:  mov   0x200a35(%rip),%eax
0x400607 <+23>:  test  %eax,%eax
0x400609 <+25>:  jne   0x4005fb
0x40060b <+27>:  mov   -0x8(%rsp),%rax
0x400610 <+32>:  retq
function main:
0x4004c0 <+0>: ...
0x4004df <+31>: callq 0x4005f0 <count>
0x4004e4 <+36>: ...
```

(b)

```
block
[group]
0x4005f0 [9]: movq $0x0,-0x8(%rsp)
0x4005f9 [2]: jmp 0x400601

block
[group]
0x400601 [6]: mov 0x200a35(%rip),%eax
0x400607 [2]: test %eax,%eax
0x400609 [2]: jne 0x4005fb

block
[group]
0x40060b [5]: mov -0x8(%rsp),%rax
0x400610 [1]: retq

block
[group]
0x4005fb [6]: addq $0x1,-0x8(%rsp)
[signals]
14: 0x4005e0 {1} (handleAlarm)
```

Figure 3.1. (a) Example program with alarm handler. (b) Assembly code for this example. (c) CFG obtained with CFGGrind. (d) Simplified CFG obtained with DCFG, showing “unknown" node that emerges after signal handling.

Example 3.1.1. Figure 3.1 shows a sample program that has a signal handler (a) with its respective assembly code (b), and shows how signals are processed by CFGGrind

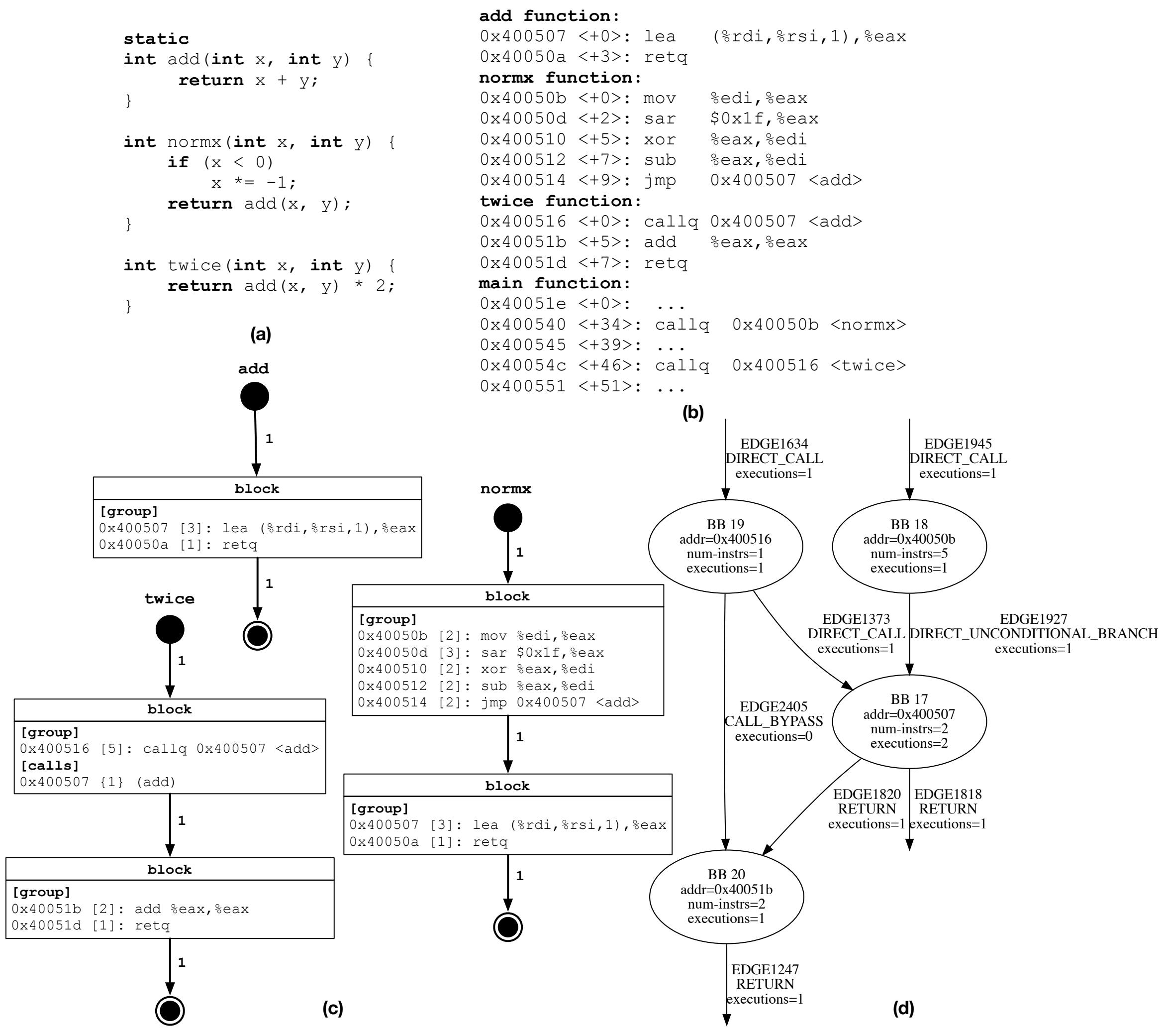
(c) and DCFG (d). When a signal handler is activated, bfTrace crashes and is unable to produce the CFGs for such a program. CFGgrind records the address of the function handler called with its associated signal id at the basic block where the event originated. DCFG creates a special edge marking the function handler as a context switch, but without an associated signal id. Also, DCFG fails to track reliably the correct execution flow after the return of the signal handler. In Figure 3.1(d) the edge at address @`0x400610` in `BB 28`, is misidentified as a fall-through edge, whereas it should have been marked as a return edge. Furthermore, the target of this edge points to a special `Unknown` node due to an invalid target address calculated at this point. Note that if a signal handler is never activated, none of the three tools are able to find its CFG. Also, the signal registration is not relevant to the context of the control flow graph, and thus is not tracked by any of the these tools.

3.2 On Exact Profiling Information

A profiler provides users with either exact or approximate information. In the latter category we have all the *sampling-based profilers*. In the former, we have *instrumentation* and *emulation* based profilers. CFG reconstructors can be used as a supporting infrastructure to build exact profilers. To fulfill this goal, three features are desirable: **edge count**, **call count** and **signal count**. Edge count gives the number of times each edge in the CFG was traversed by the program flow. Call count provides the number of times each function has been called during the execution of the program. Signal count holds similar information, but for signal handlers instead of functions calls. Both, CFGgrind and DCFG provide these three features. They are absent in bfTrace.

Edge counts, when available, should be subject of the **Law of Flows**, which Tarjan [1974], among other graph theoreticians, have postulated as: "the sum of incoming flows must equal the sum of outgoing flows on each vertex of a directed graph, except on its start and end nodes." In the context of this work, the count in the incoming edges must add up to the sum of the counts of the outgoing edges for any basic block traversed during program execution. The two exceptions are the program entry point, whose in-degree is zero, and the program exit point, whose out-degree is zero. This principle is true for CFGgrind; however, it is not entirely true for DCFG.

Example 3.2.1. Figure 3.2(a) shows an example program where the compiler can optimize the invocation of function `add` depending on its calling context. As can be observed by the assembly code produced in Fig 3.2(b), the call in function `normx` was



```
static
int add(int x, int y) {
    return x + y;
}

int normx(int x, int y) {
    if (x < 0)
        x *= -1;
    return add(x, y);
}

int twice(int x, int y) {
    return add(x, y) * 2;
}
```

(a)

```
add function:
0x400507 <+0>: lea    (%rdi,%rsi,1),%eax
0x40050a <+3>: retq
normx function:
0x40050b <+0>: mov    %edi,%eax
0x40050d <+2>: sar    $0x1f,%eax
0x400510 <+5>: xor    %eax,%edi
0x400512 <+7>: sub    %eax,%edi
0x400514 <+9>: jmp    0x400507 <add>
twice function:
0x400516 <+0>: callq 0x400507 <add>
0x40051b <+5>: add    %eax,%eax
0x40051d <+7>: retq
main function:
0x40051e <+0>:  ...
0x400540 <+34>: callq  0x40050b <normx>
0x400545 <+39>: ...
0x40054c <+46>: callq  0x400516 <twice>
0x400551 <+51>: ...
```

(b)

```
add
block
[group]
0x400507 [3]: lea (%rdi,%rsi,1),%eax
0x40050a [1]: retq

twice
block
[group]
0x400516 [5]: callq 0x400507 <add>
[calls]
0x400507 {1} (add)
block
[group]
0x40051b [2]: add %eax,%eax
0x40051d [1]: retq

normx
block
[group]
0x40050b [2]: mov %edi,%eax
0x40050d [3]: sar $0x1f,%eax
0x400510 [2]: xor %eax,%edi
0x400512 [2]: sub %eax,%edi
0x400514 [2]: jmp 0x400507 <add>
block
[group]
0x400507 [3]: lea (%rdi,%rsi,1),%eax
0x40050a [1]: retq
```

(c)

1
EDGE1634 DIRECT_CALL executions=1
EDGE1945 DIRECT_CALL executions=1
BB 19 addr=0x400516 num-instrs=1 executions=1
BB 18 addr=0x40050b num-instrs=5 executions=1
EDGE1373 DIRECT_CALL executions=1
EDGE1927 DIRECT_UNCONDITIONAL_BRANCH executions=1
EDGE2405 CALL_BYPASS executions=0
BB 17 addr=0x400507 num-instrs=2 executions=2
EDGE1820 RETURN executions=1
EDGE1818 RETURN executions=1
BB 20 addr=0x40051b num-instrs=2 executions=1
EDGE1247 RETURN executions=1

(d)

Figure 3.2. (a) Example program with two distinct calls to function `add`. (b) Assembly code with a tail call optimization for this example. (c) CFGs obtained with CFGGrind for each function. (d) Simplified CFG obtained with DCFG for this program.

optimized to use a jump instruction. However, the compiler was unable to use the same strategy for the call in function `twice`. Thus, function `add` is used in two distinct contexts. Similar situation is commonly observed in code in general. For example, libgfortran (version 3.0.0) has some data transfer functions, e.g. `transfer_integer` or `transfer_real`, to copy data between different container types. These functions are used externally — using call instructions — , but are also used internally — using jump instructions after tail call optimization. Therefore, CFG reconstruction tools must be able to handle properly such cases. The three CFGs in Fig. 3.2(c) were produced both by CFGGrind and by bfTrace. Each CFG has its own distinctive copy of a shared block — the block with address `@0x400507` is duplicated in the CFGs for `normx` and `add`. DCFG on the other hand uses a single block (`BB 17`) in both contexts. Consequently, it is not possible to determine if the transfer of control happened due to a function call or due to an unconditional branch. In Figure 3.2(d), there are two return edges going out of `BB 17`, but only one incoming call — the other return edge is due to the jump from `BB 18`.

3.3 On the Incremental Construction of CFGs

Dynamic analyses require good datasets: the more inputs are available for a program, the more information can be inferred from the program's behavior. This principle applies to the dynamic reconstruction of CFGs. However, neither DCFG nor bfTrace support the **incremental construction** of CFGs. In other words, it is not possible to combine events observed in two different executions of a program to build a refined version of its CFGs. CFGgrind provides this functionality, as Example 2.3.1 illustrates. Thus, additional program inputs lead to successive refinements of this program's CFG; hence, increasing code coverage. Section 5.4 quantifies the benefits of incremental construction in the cBench suite. Note that this capability is only supported if, for each execution, the program is always loaded in the same memory region. Security protections, such as Address Space Layout Randomization (ASLR), must be disabled to achieve incremental construction coverage. This is not an issue for CFGgrind since valgrind manages its own memory mappings when emulating programs execution. More details on how CFGgrind supports incremental constructions of CFGs can be found in Section 4.2.

3.4 On the Execution of Multi-Threaded Programs

A parallel program can span multiple threads during its execution. Both CFGgrind and DCFG supports tracking the execution of such threads; however, bfTrace crashes in this scenario. DCFG provides detailed profiling information, where each edge in the control flow graph contains the execution count for each thread separately; CFGgrind compounds the result of all threads as a total for each edge.

CFGgrind leverages the serialization performed natively by valgrind, where the execution of multi-threaded programs is converted into a single-threaded application by using valgrind's own scheduling policy. CFGgrind tracks each thread's context switch to account for the correct execution flow of programs. More details about the implementation of this feature can be found in Section 4.3. It is unclear how DCFG works internally to support this feature.

3.5 On Other Assembly Idiosyncrasies

Although low-level assembly code is usually derived from high-level languages via a compilation chain, some aggressive optimizations can dramatically change the structure of the target code. For instance, optimizations might force code sharing between multiple functions. Also, some sections of assembly code can have overlapping instructions. Overlapping happens mostly in hand-crafted code, which either implements some optimization or encodes malware. In this last category, we have examples of return oriented programming attacks [Shacham, 2007]. In all these cases, binary code presents idiosyncrasies that a reconstructor must handle.

Figure 3.3. Objdump snippet for functions *__read* and *__read_nocancel*, from glibc 2.17, that share code. To see that overlapping happens, notice that $[eb86a_{16}, eb86a_{16} + 22_{16}] \cap [eb860_{16}, eb860_{16} + 77_{16}] \neq \emptyset$.

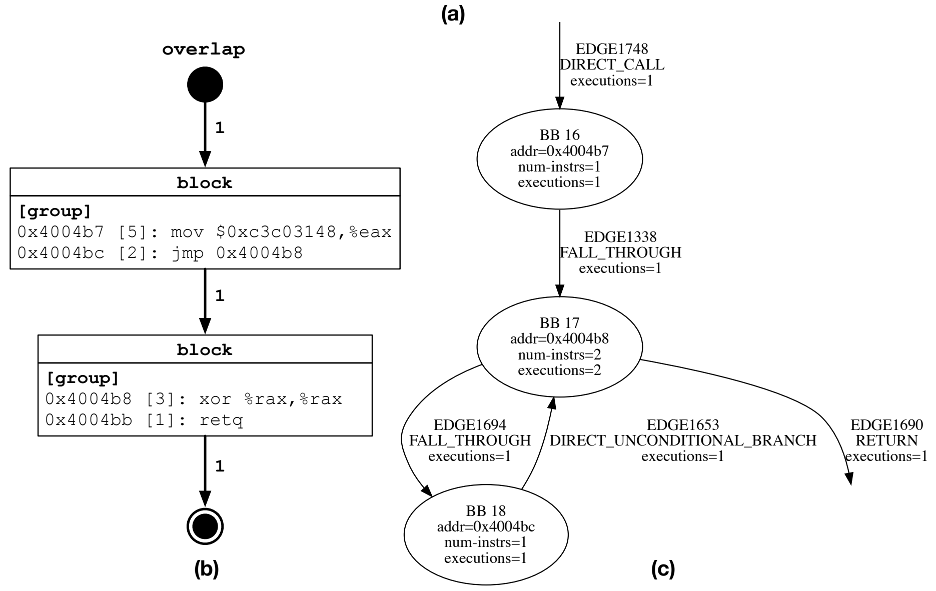
```
000eb86a l     F .text  00000022    __read_nocancel
000eb860  w    F .text  00000077    __read
```

Example 3.5.1. Figure 3.3 exemplifies the first situation: instructions shared between functions. A snippet of an object dump of mapped symbols available in glibc (version 2.17) for two function: *__read* and *__read_nocancel*. The former function is mapped between addresses @`@0xeb86a`-@`0xeb88c`; while the latter is mapped between @`0xeb860`-@`0xeb8d7`. Since there is an overlap of these two ranges, some instructions are shared by these functions. DCFG approaches this situation using an unique node that is shared across multiple parts of the entire control flow graph. This node can be interpreted as if a section of code has multiple access points. On the other hand, CFGgrind and bfTrace build a CFG for each function, which means that each CFG has its own copy of a block that contains these shared instructions. The same approach is employed by CFGgrind to support functions with multiple entry points. Each entry point spawns a different CFG with its own copies of the shared instructions. Thus, every CFG in CFGgrind has only a single entry point.

Example 3.5.2. Figure 3.4 exemplifies the second situation: a block of contiguous bytes can be interpreted as different sequences of assembly instructions. Such situation occurs when there is a jump or call to an unaligned target address. Fig. 3.4(b) shows that CFGGrind and bfTrace obtained the same CFG, while Fig. 3.4(b) shows that DCFG splits the nodes incorrectly, leading to an unrealistic execution flow. A call to address @`0x4004b7` activates `Sequence 1` with two instructions. Its last instruction is a relative jump to the unaligned address @`0x4004b8`. Thus, `Sequence 2` is activated. Of the three instructions of this sequence, the last one is never executed due to the return instruction. CFGGrind and bfTrace capture the correct behavior by treating the instructions individually in the blocks. However, DCFG treats the block as a range of addresses, disregarding how the instructions are read inside this range. This modus-operandi leads to the flawed split at node `BB 16`. Although seemly artificial, the unaligned access that this example illustrate is a key component in several real-world ROP-based program exploits, some of which are catalogued in the CVE database [Gorelik, 2018; Seebug, 2018; Alvarez-Perez, 2017].

Address	Byte	Sequence 1	Sequence 2
0x4004b7	0xb8	mov $0xc3c03148, %eax	
0x4004b8	0x48		xor %rax, %rax
0x4004b9	0x31		
0x4004ba	0xc0		
0x4004bb	0xc3		retq
0x4004bc	0xeb	jmp .-4	jmp .-4
0x4004bd	0xfa		

(a)



```
[group]
0x4004b7 [5]: mov $0xc3c03148,%eax
0x4004bc [2]: jmp 0x4004b8
```

```
[group]
0x4004b8 [3]: xor %rax,%rax
0x4004bb [1]: retq
```

Figure 3.4. (a) Hand-crafted example of two overlapping sequences of assembly instructions. (b) CFG obtained with CFGGrind and bfTrace. (d) Simplified CFG obtained with DCFG.

3.6 Conclusion

In this chapter CFGgrind was compared against two other CFG reconstruction tools found in the literature: DCFG [Yount et al., 2015] and bfTrace [Gruber et al., 2019]. An in-depth analysis of these three tools was conducted over several features concerning control flow graphs. Such features include how precise is the representation of the CFGs (Section 3.1), how exact are the profiling information provided (Section 3.2), how CFGs can be incrementally improved (Section 3.3), how multi-threaded programs are supported (Section 3.4), and how to handle specific assembly idiosyncrasies that can be generated by compilers (Section 3.5). We show that CFGgrind is the most feature-rich tool available.

Chapter 4

Dynamic Reconstruction of CFGs

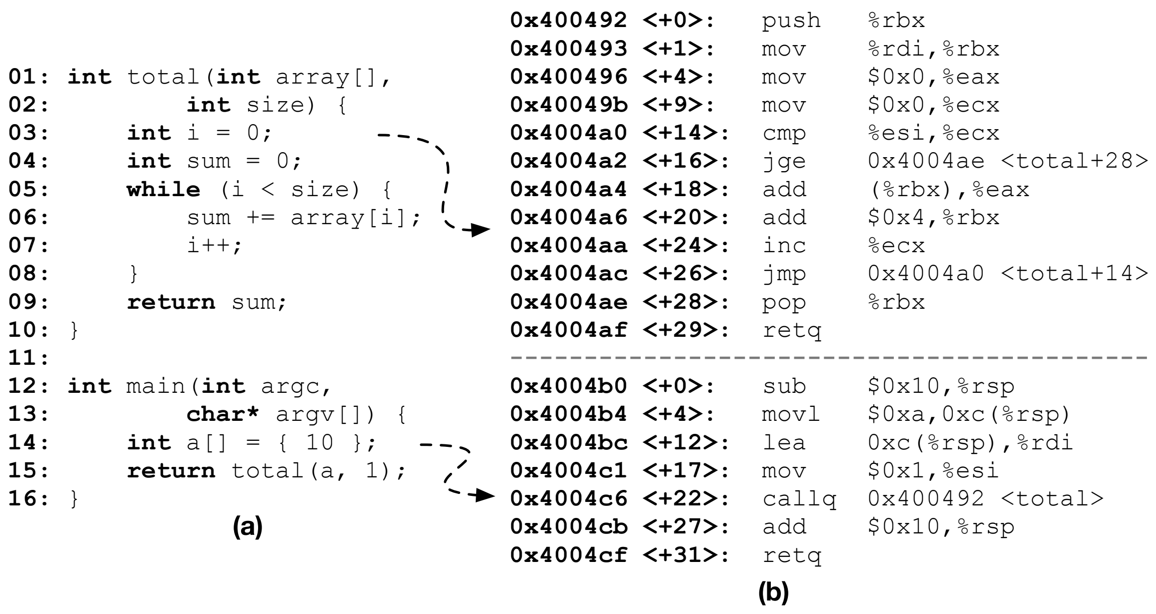
This chapter uses pseudo-code to explain the dynamic reconstruction of CFGs. CFGGRIND, the tool that prototypes the ideas presented in this thesis, is implemented in C, on top of VALGRIND. However, for ease of understanding, the algorithms in this chapter are presented in a Python-like format. Executable versions of these algorithms can be downloaded from CFGGRIND's repository.

Before diving into the algorithms, this chapter defines an abstract machine that is capable of simulation the execution of a program (Section 4.1). Then, the basic algorithm for the dynamic reconstruction of control flow graphs is presented (Section 4.2). Later, this algorithm is extended with features that highlight CFGGRIND's improved capabilities when compared to other evaluated tools (Section 4.3). These extensions are:

- the caching strategy employed by CFGGRIND that makes it the fastest tool among its competitors;
- the support for multi-threaded programs that makes CFGGRIND apart from BFTRACE [Gruber et al., 2019];
- and the effectiveness of the signal events handling that makes CFGGRIND more precise than DCFG [Yount et al., 2015].

4.1 The Machine

In the context of this work, a *machine* is any technology, be it based on interpretation, emulation or instrumentation, that produces traces representing the execution of programs. Typical machines include tools such as QEMU [Bellard, 2005], PIN [Luk et al., 2005], GDB and VALGRIND [Nethercote and Seward, 2007]. The instructions that appear in a trace are partitioned into groups according to Definition 2. Traces can be processed online, as soon as they are produced by the machine; or offline, as a *post-morten* analysis. The algorithm described in Section 4.2 is agnostic to this processing mode. CFGGRIND, implemented in VALGRIND, uses the online approach. The following example illustrates the notion of a trace.

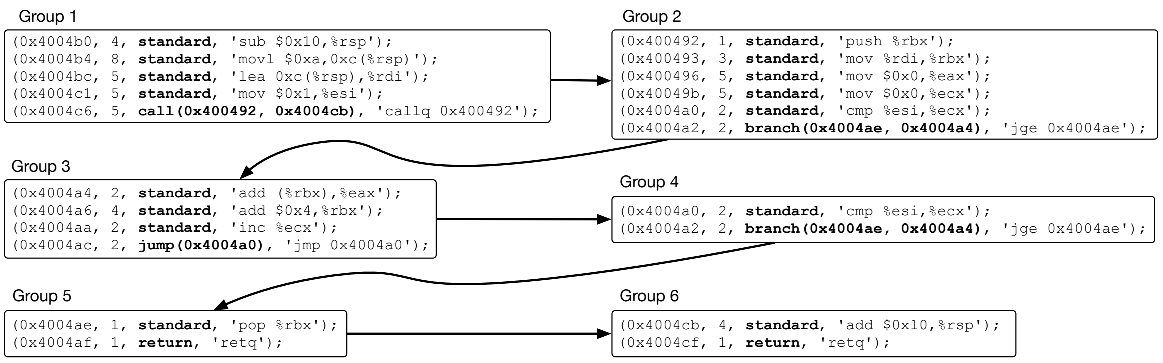


```
int total(int array[],
          int size) {
    int i = 0;
    int sum = 0;
    while (i < size) {
        sum += array[i];
        i++;
    }
    return sum;
}

int main(int argc,
          char* argv[]) {
    int a[] = { 10 };
    return total(a, 1);
}
```

```
0x400492 <+0>:   push   %rbx
0x400493 <+1>:   mov    %rdi,%rbx
0x400496 <+4>:   mov    $0x0,%eax
0x40049b <+9>:   mov    $0x0,%ecx
0x4004a0 <+14>:  cmp    %esi,%ecx
0x4004a2 <+16>:  jge    0x4004ae <total+28>
0x4004a4 <+18>:  add    (%rbx),%eax
0x4004a6 <+20>:  add    $0x4,%rbx
0x4004aa <+24>:  inc    %ecx
0x4004ac <+26>:  jmp    0x4004a0 <total+14>
0x4004ae <+28>:  pop    %rbx
0x4004af <+29>:  retq
-------------------------------------------
0x4004b0 <+0>:   sub    $0x10,%rsp
0x4004b4 <+4>:   movl   $0xa,0xc(%rsp)
0x4004bc <+12>:  lea    0xc(%rsp),%rdi
0x4004c1 <+17>:  mov    $0x1,%esi
0x4004c6 <+22>:  callq  0x400492 <total>
0x4004cb <+27>:  add    $0x10,%rsp
0x4004cf <+31>:  retq
```

Figure 4.1. (a) Program written in C. (b) static assembly representation of the program.



```
(0x4004b0, 4, standard, 'sub $0x10,%rsp');
(0x4004b4, 8, standard, 'movl $0xa,0xc(%rsp)');
(0x4004bc, 5, standard, 'lea 0xc(%rsp),%rdi');
(0x4004c1, 5, standard, 'mov $0x1,%esi');
(0x4004c6, 5, call(0x400492, 0x4004cb), 'callq 0x400492');
```

```
(0x400492, 1, standard, 'push %rbx');
(0x400493, 3, standard, 'mov %rdi,%rbx');
(0x400496, 5, standard, 'mov $0x0,%eax');
(0x40049b, 5, standard, 'mov $0x0,%ecx');
(0x4004a0, 2, standard, 'cmp %esi,%ecx');
(0x4004a2, 2, branch(0x4004ae, 0x4004a4), 'jge 0x4004ae');
```

```
(0x4004a4, 2, standard, 'add (%rbx),%eax');
(0x4004a6, 4, standard, 'add $0x4,%rbx');
(0x4004aa, 2, standard, 'inc %ecx');
(0x4004ac, 2, jump(0x4004a0), 'jmp 0x4004a0');
```

```
(0x4004a0, 2, standard, 'cmp %esi,%ecx');
(0x4004a2, 2, branch(0x4004ae, 0x4004a4), 'jge 0x4004ae');
```

```
(0x4004ae, 1, standard, 'pop %rbx');
(0x4004af, 1, return, 'retq');
```

```
(0x4004cb, 4, standard, 'add $0x10,%rsp');
(0x4004cf, 1, return, 'retq');
```

Figure 4.2. Execution trace of the program in Figure 4.1. Instructions are grouped according to Def. 2. Arrows show order in which groups are processed.

Example 4.1.1. Figure 4.1 shows a program (a) with two functions and its assembly representation (b). The execution of this program in a machine produces a trace formed by those assembly instructions. Such trace represents the paths traversed by the execution of the program. Figure 4.2 shows the different groups formed by the analysis of this execution trace. The *jump*, *branch* and *call* instructions in this trace are all direct, and thus the *mode* of each instruction is omitted. In this example, the body of the while loop in function `total` (Lines 5-8) executes only once.

4.2 The Algorithm

Central to the understanding of Algorithms 1-3, is the notion of a *state*, defined as follows:

Definition 5. A **state** is a tuple $S = (current, callstack)$. *Current* is a pair (*cfg*, *working*), where *cfg* is the CFG ($G = (V, E)$, Def. 3) currently being reconstructed, and *working* is one of this CFG's nodes ($working \in V$). The *callstack* is a stack of (*current*, *@ret_ addr*) pairs, where *@ret_ addr* is a return address. The *callstack*'s *current* pair is similar to the one in the *state*, except *working* must perform a function call (*working.tail.type* is *call*), and the *@ret_ addr* is the address of the instruction immediately after this call (*working.tail.type.fallthrough*).

During the reconstruction of CFGs, the algorithms discussed in this section operate on a *state*. The processing of groups, such as those shown in Fig. 4.2, leads to changes in this state. Thus, Algorithms 1-3 are state-transition functions that map a state-group pair into another state ($state \times group \mapsto state$). When the algorithm processes a *working* node in the *current* CFG, another node becomes the *working* node. When the algorithm processes a function call, the *current* pair is pushed onto the *callstack* and its return address is set. A function return to an address matching a *@ret_ addr* in the *callstack* causes the stack to pop elements until this point is reached. The *current* pair associated with this return address is then restored as the *current* pair of the *state*. At initialization, the *current* is set to NIL and the *callstack* is empty ($S = (\text{NIL}, [])$).

Example 4.2.1. Figure 4.3 shows the *state* after each one of the six groups in Figure 4.2 is processed. In this multi-layer representation, the front layer presents the *current* state, e.g., (*cfg*, *working*). Underneath layers represent the state's *callstack*. The front layer in Figures 4.3(a) and 4.3(f) represents the `main` function. The front layer in Figure 4.3(b-e) corresponds to the `total` function.

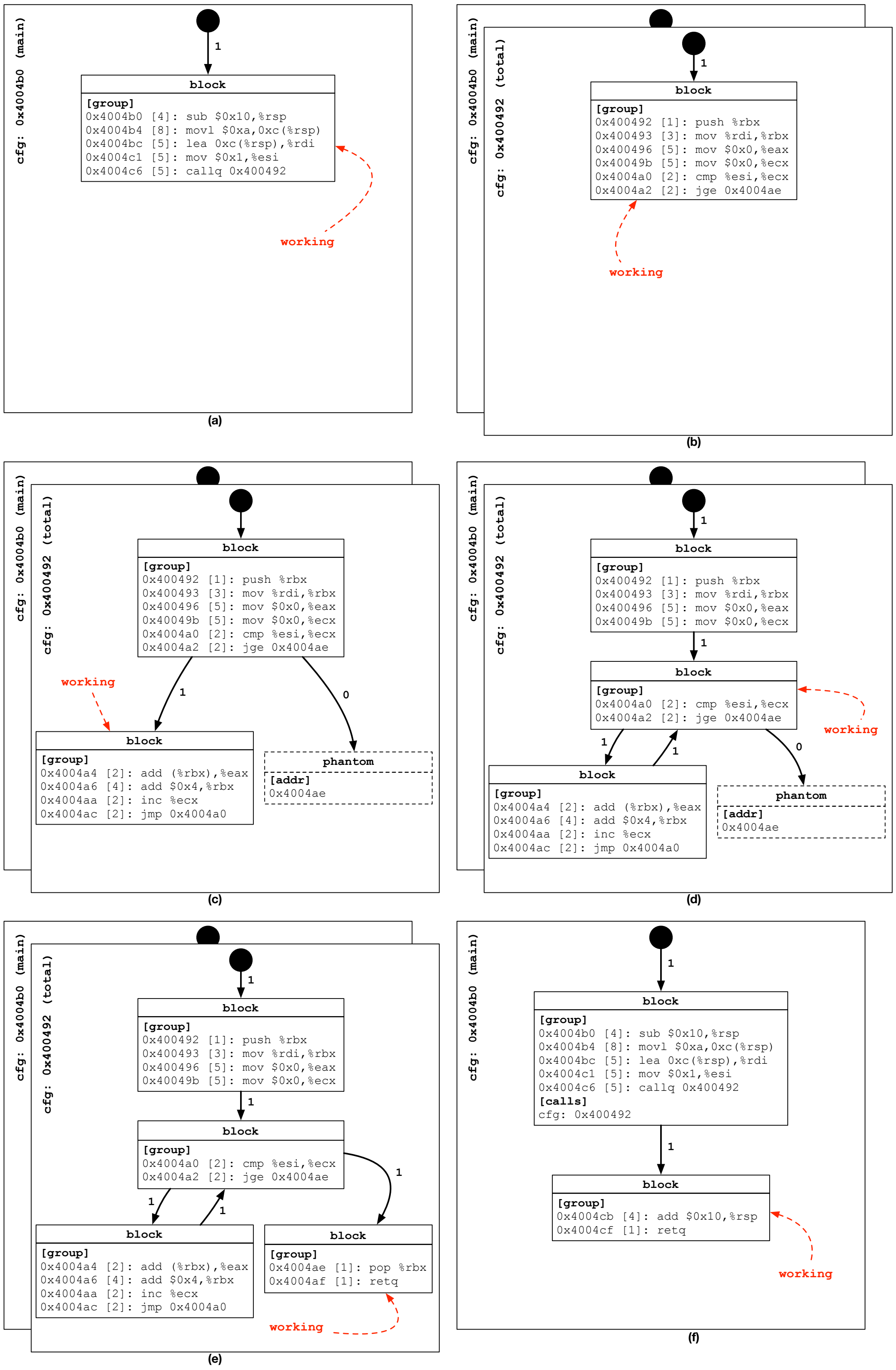
The algorithms discussed in this section use a core data structure, the *cfg*, with the following operations:

- *add_node(node)*: adds a new node to the *cfg* if this new node is not already there.
- *add_edge(src, dst, count)*: adds a new edge to the *cfg* from node *src* to node *dst* with *count* as the number of executions. If the edge already exists, increment the previous execution count by the value of *count*.
- *find_node_with_addr(@addr)*: searches for a block node with instruction at *@addr*, or a phantom node at *@addr*; returns NIL if not found.
- *phantom2block(phantom_node, block_node)*: replaces the phantom node with the block node, including moving its predecessors edges to the new node.
- *split(block_node, @addr)*: finds instruction i_j with address *@addr* in the group of the block node such that $i_1 < i_j \leq i_n$, moves the instructions $\{i_1, \ldots, i_{j-1}\}$, and its predecessor to a new block node, and finally connects them with a new edge.

4.2.1 Processing Programs

Algorithm 1 is the entry point for the process of CFG reconstruction. The algorithm assumes the existence of a global *state*, initialized as (NIL, []), that is readily available during processing. This global *state* can be externally manipulated to support features such as multi-thread programs and signal handlers (Sec. 4.3). The algorithm receives a *machine* and a *mapping* of CFGs indexed by their addresses. The *mapping* can be either empty or pre-populated with CFGs loaded from a previous run. This is they key to support incremental construction of CFGs as described in Section 3.3. By loading previously computed CFGs, the algorithms described in this section can further improve them, as they continue to refine the CFGs as new paths are explored during the execution.

Algorithm 1 expects a sequence of groups generated by the machine to reconstruct the CFGs dynamically. Each group is then processed individually (Lines 2-12) by this algorithm. Once the machine halts, i.e. no more groups are generated, the algorithm finalizes the remaining CFGs by connecting the *working* nodes, of the *state*'s *current* pair or of the *callstack* if present, to the *halt* node (Lines 13-17). Finally, Algorithm 1 returns the updated mapping with all reconstructed CFGs at line 17.
For each group (Lines 2-12), Algorithm 1 manipulates the state in two phases:



```
[group]
0x4004b0 [4]: sub $0x10,%rsp
0x4004b4 [8]: movl $0xa,0xc(%rsp)
0x4004bc [5]: lea 0xc(%rsp),%rdi
0x4004c1 [5]: mov $0x1,%esi
0x4004c6 [5]: callq 0x400492
```

```
[group]
0x400492 [1]: push %rbx
0x400493 [3]: mov %rdi,%rbx
0x400496 [5]: mov $0x0,%eax
0x40049b [5]: mov $0x0,%ecx
0x4004a0 [2]: cmp %esi,%ecx
0x4004a2 [2]: jge 0x4004ae
```

```
[group]
0x4004a4 [2]: add (%rbx),%eax
0x4004a6 [4]: add $0x4,%rbx
0x4004aa [2]: inc %ecx
0x4004ac [2]: jmp 0x4004a0
```

```
[addr]
0x4004ae
```

```
[group]
0x400492 [1]: push %rbx
0x400493 [3]: mov %rdi,%rbx
0x400496 [5]: mov $0x0,%eax
0x40049b [5]: mov $0x0,%ecx
```

```
[group]
0x4004a0 [2]: cmp %esi,%ecx
0x4004a2 [2]: jge 0x4004ae
```

```
[group]
0x4004ae [1]: pop %rbx
0x4004af [1]: retq
```

```
[group]
0x4004b0 [4]: sub $0x10,%rsp
0x4004b4 [8]: movl $0xa,0xc(%rsp)
0x4004bc [5]: lea 0xc(%rsp),%rdi
0x4004c1 [5]: mov $0x1,%esi
0x4004c6 [5]: callq 0x400492
[calls]
cfg: 0x400492
```

```
[group]
0x4004cb [4]: add $0x10,%rsp
0x4004cf [1]: retq
```

Figure 4.3. State after processing each of the six groups listed by Figure 4.1.

Algorithm 1 Process program by handling each group of instructions generated by a machine during execution.

```
    global: state
    input: machine, mapping
    output: mapping
 1: function PROCESS_PROGRAM(machine, mapping)
 2:     for group in machine.RUN() do
 3:         @addr = group.leader.addr
 4:         if not state.current then
 5:             initial = mapping.GET(@addr) if mapping.HAS(@addr) else mapping.PUT(@addr, CFG())
 6:             state.current = (initial, initial.entry)
 7:         else
 8:             assert state.current.working instanceof Block
 9:             mapping = PROCESS_TYPE(mapping, state.current.working.group.tail.type, @addr)
10:         end if
11:         state.current.working = PROCESS_GROUP(state.current.cfg, state.current.working, group)
12:     end for
13:     while state.current do
14:         state.current.cfg.ADD_EDGE(state.current.working, state.current.cfg.halt, 1)
15:         state.current = state.callstack.POP() if not state.callstack.EMPTY() else nil
16:     end while
17:     return mapping
18: end function
```

Phase 1 (Lines 4-10): takes an action based on the previous *working* node. In the absence of the *working* node, initializes the first CFG (Lines 4-6). The initial CFG is either fetched from the mapping based on the address of the *group*'s leader instruction if existent, or it is created and set in the mapping (Line 5). Then, the state's *current* pair is configured with this CFG and its entry node (Line 6). Otherwise, ensures that *working* node is a basic block (Line 8) and activates Algorithm 2 (Line 9) passing the type of the *tail* instruction of the *working* node and the address of the next instruction of the group.

Phase 2 (Line 11): activates Algorithm 3. This algorithm is responsible for building a new path or following an existing one in the CFG. It may create or split nodes in this process, but it will never transition between CFGs. At the end, Algorithm 3, sets the *working* node to the node which its *tail* is last instruction of the processed group.

Example 4.2.2. Each one of the six frames in Figure 4.3 is a snapshot of the state after each iteration of Algorithm 1. Snapshots are taken immediately after the processing of the group by Algorithm 3 (Line 11).

Group 1: (Figure 4.3(a)) In phase 1, the CFG for function MAIN is created with its *entry* node set as the *working* node. In phase 2 this group is processed leading to the creation of the block with address `@0x4004b0` with all the instructions of the group. A new edge was created with execution count of 1 from the previous *working* node, i.e. *entry* node, to the current *working* node, i.e. the newly created block.

Group 2: (Figure 4.3(b)) In phase 1, the pending call of the previous block is processed. The CFG for function TOTAL at address @`0x400492` is created and inserted into *mapping*. This CFG is added to the call map of the *working* node (block @`0x4004b0`). Then, the *state*'s *current* pair is pushed onto the *state*'s *call-stack* with the return address @`0x4004cb` — the fall-through of the instruction call. Finally, there is a switch to the new CFG by setting the *state*'s *current* pair with this CFG and its entry node. In phase 2, the second group is processed in a similar fashion as the previous. A block with address @`0x400492` is created, connected from the *entry* with execution count of one, and set as the new *working* node.

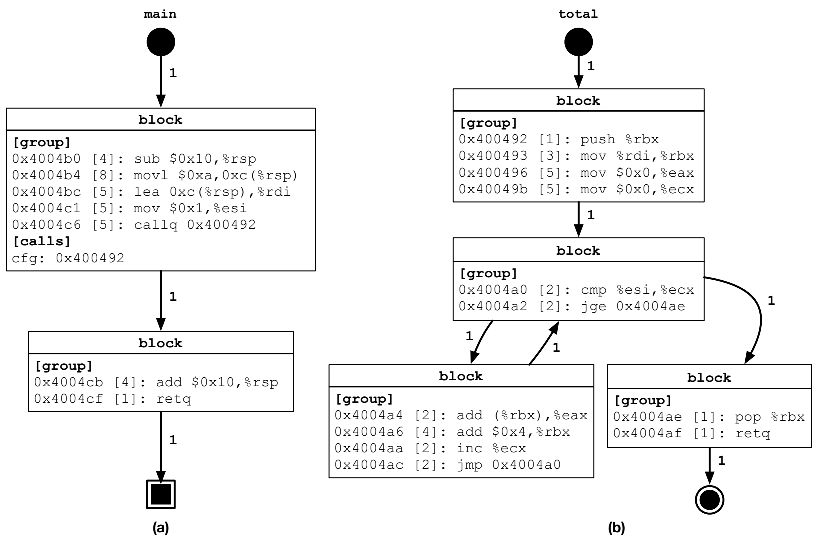
Group 3: (Figure 4.3(c)) In phase 1, the pending branch of the previous block is processed. The algorithm creates a phantom node with address @`0x4004ae` for the target address of this branch. Note that no *phantom* node is created for this branch's fall-through address, since this path will be covered in phase 2 for this group. Thus, in phase 2 the block @`0x4004a4` is created, connected, and set as the *working* node.

Group 4: (Figure 4.3(d)) In phase 1, there is no action for the jump instruction of the previous block, since the jump target will be handled by this group. In Phase 2, there is a jump to the instruction @`0x4004a0` that is inside block @`0x4004a0`. Therefore, this block must be split in two blocks: block @`0x400492` with four instructions and block @`0x4004a0` with two instructions. Then, a new edge with one execution is created between blocks @`0x4004a4` and @`0x4004a0`. All the instructions of this group are matched against the ones in block @`0x4004a0`; thus no new information is added at this point. Afterwards, this block becomes the *working* node.

Group 5: (Figure 4.3(e)) In phase 1, the branch of instruction @`0x4004a2` is processed again, but both paths it can follow have already been covered; thus nothing is changed for this CFG. The execution followed the target of the branch, which lead to this group. In phase 2, the leader of this group matches the address of the phantom node at @`0x4004ae`. Thus, the *phantom* node is converted to a *block* node and it is populated with the instructions of this group and the update count of the edge increased by one. Finally, this new block becomes the *working* node.

Group 6: (Figure 4.3(f)) In phase 1, a function return occurs, because the tail instruction of the *working* node is a return. First, the *working* node is connected to the *exit* node with an edge count of one. Then, the algorithm checks if there is a return address in the *callstack* that matches the address of the leader of this group. In this case, the leader address of this group @`0x4004cb` matches the top of the stack. Thus, the *state*'s *current* pair is restored by popping the top of the stack. At this point, the current *working* node is block @`0x4004b0`, and the *cfg* the CFG of the MAIN function. In phase 2, the block @`0x4004cb` is created, connected, and set as the *working* node.

After processing all the groups, Algorithm 1 connects the state's *working* node to the *halt* node to conclude the execution (Lines 13-16). Then, Algorithm 1 returns the *mapping* containing the functions MAIN and TOTAL that were invoked during the execution of this program (Line 17). The final CFG for both functions can be seen in Figure 4.4.



```
[group]
0x4004b0 [4]: sub $0x10,%rsp
0x4004b4 [8]: movl $0xa,0xc(%rsp)
0x4004bc [5]: lea 0xc(%rsp),%rdi
0x4004c1 [5]: mov $0x1,%esi
0x4004c6 [5]: callq 0x400492
[calls]
cfg: 0x400492
```

```
[group]
0x4004cb [4]: add $0x10,%rsp
0x4004cf [1]: retq
```

```
[group]
0x400492 [1]: push %rbx
0x400493 [3]: mov %rdi,%rbx
0x400496 [5]: mov $0x0,%eax
0x40049b [5]: mov $0x0,%ecx
```

```
[group]
0x4004a0 [2]: cmp %esi,%ecx
0x4004a2 [2]: jge 0x4004ae
```

```
[group]
0x4004a4 [2]: add (%rbx),%eax
0x4004a6 [4]: add $0x4,%rbx
0x4004aa [2]: inc %ecx
0x4004ac [2]: jmp 0x4004a0
```

```
[group]
0x4004ae [1]: pop %rbx
0x4004af [1]: retq
```

Figure 4.4. The resulting CFGs for functions MAIN (a) and TOTAL (b) in the *mapping* produced by Algorithm 1.

4.2.2 Processing the Type of a Group's Tail Instruction

Algorithm 2, invoked at Line 9 of Algorithm 1, performs an action based on the *type* of the *tail* instruction of the previously processed group. This *tail* instruction is obtained from the last instruction of the *working* node, which is always a basic block. The function PROCESS_TYPE of Algorithm 2 receives a *mapping* of all CFGs discovered so far, the *type* of the *tail* instruction of the previous group, and the target address (*target_ addr*) of the *leader* instruction of the next group obtained from the machine. This function returns the updated *mapping*, in case new control flow graphs are discovered. Note that this function may also affect the global *state*.

Algorithm 2 Process the type of the tail instruction of a group.

```
     global: state
     input: mapping, type, @target_ addr
     output: mapping
 1: function PROCESS_TYPE(mapping, type, @target_ addr)
 2:     if type instanceof Jump then
 3:         # do nothing
 4:     else if type instanceof Branch then
 5:         addrs = [type.fallthrough]
 6:         if type.direct then
 7:             addrs.APPEND(type.target)
 8:         end if
 9:         for @addr in addrs do
10:             if $@addr \neq @target\_addr$ then
11:                 node = state.current.cfg.FIND_NODE_WITH_ADDR(@addr)
12:                 if node then
13:                     if node instanceof Block and $node.group.leader.addr \neq @addr$ then
14:                         node = state.current.cfg.SPLIT(node, @addr)
15:                     end if
16:                 else
17:                     node = state.current.cfg.ADD_NODE(Phantom(@addr))
18:                 end if
19:                 state.current.cfg.ADD_EDGE(state.current.working, node, 0)
20:             end if
21:         end for
22:     else if type instanceof Call then
23:         called = mapping.GET(@target_ addr) if mapping.HAS(@target_ addr)
                else mapping.PUT(@target_ addr, CFG())
24:         state.current.working.ADD_CALL(called, 1)
25:         state.callstack.PUSH(state.current, type.fallthrough)
26:         state.current = (called, called.entry)
27:     else if type instanceof Return then
28:         pops = state.callstack.POPS_COUNT(@target_ addr)
29:         while $pops > 0$ do
30:             state.current.cfg.ADD_EDGE(state.current.working, state.current.cfg.exit, 1)
31:             state.current = state.callstack.POP()
32:             pops−−
33:         end while
34:     else
35:         error "Unreachable code"
36:     end if
37:     return mapping
38: end function
```

According to Definition 2 the *type* of the *tail* instruction of a group must be either *jump*, *branch*, *call*, or *return*. If the *type* is an unconditional *jump*, then no special action is required (Lines 2-3). In this case, only one program flow is possible in the CFG and it will be handled when processing the next group. If the *type* is a conditional *branch* then Algorithm 2 models the possible execution flows for this instruction (Lines 4-21). First, it builds a list of the possible target addresses: the branch's target if it is known — in case of a direct branch — , and the branch's fall-through address (Lines 5-8).

Then, for each target *@addr* (Line 9) that is not the *target_addr* of the next block, Algorithm 2 either: (1) splits its block, if *@addr* is not in the first instruction (Lines 12-15); or (2) creates a new phantom node, if *@addr* does not belong to a known block (Line 16-18). Regardless of the case, the *working* node is connected to this new node without updating its execution count, since this path has not been traversed yet (Line 19).

Example 4.2.3. Figure 4.3(c) shows that Algorithm 2 created the phantom node @`0x4004ae`. Said node corresponds to the target of the branch `jge` at the end of group @`0x400492` that was not taken.

If the *type* is a *call*, then a different CFG will be visited (Line 22-26). First, the CFG is obtained either from the mapping if it already exists, or a new instance is created otherwise (Line 23). Then, this CFG is added to the call list of the *working* node (Line 24) with the execution count incremented by one. Later, Algorithm 2 pushes the *current* pair with the *cfg* and *working* node onto the state's *callstack* with the expected return address, at the fall-through of this call after it is completed (Line 25). Finally, the *state*'s *current* pair is updated with the called cfg and its entry node (Line 26).

Example 4.2.4. Figure 4.3 shows the transition in the *state* for the CFGs that happens when the function `main` (Fig. 4.3(a)) makes a call to another function `total` (Fig. 4.3(b)). At this point, the *working* node points to the entry node of function `total` — situation prior to the Figure 4.3(b). Also, the called CFG is added to the call list of block @`0x4004b0` of function `main`, as seen in Figure 4.4(a).

If the *type* is a *return*, then Algorithm 2 restores the *state*'s *current* pair if the target address matches the return address of an entry in the call stack (Lines 27-33). First, the Algorithm 2 calls the CFG auxiliary function POPS_COUNT to scan the state's *callstack*, from top to bottom, searching for an entry whose *@ret_addr* is the same as the *@target_addr*. It returns how many pops, or hops, are necessary to find the matching entry. Then, while the pop count is positive (Line 29), Algorithm 2 adds an edge from *working* to *exit*, or increments that edge's counter by one (Line 30). The *state*'s *current* pair is restored with a pop in the call stack (Line 31). Also, the pop count is decremented by one (Line 32). If there is no entry matching the target address with the return address in the call stack, the return is treated as an unconditional jump. In this case, no further action is performed.

Example 4.2.5. Figure 4.3(e) shows the moment before the return at block @`0x4004ae` is processed by Algorithm 2. The target address @`0x4004cb` matches the top of the call stack, hence a pop is required. The *working* node of Figure 4.3(e) is connected to the exit node as can be seen in Figure 4.4(b). Then, the *current* pair is restored to the CFG of MAIN function as in Figure 4.3(f), but with working node at @`0x4004b0` and before the creation of block @`0x4004eb`.

Algorithm 2 ensures that variable *type* can only be one of: *jump*, *branch*, *call*, or *return* according to Definition 2. Any other type results in an error (Lines 34-35). In the end, Algorithm 2 returns the *mapping*, which might have been updated.

4.2.3 Processing Groups of Instructions

Algorithm 3 processes each instruction in a group in the order defined by their addresses. When processing instructions, Algorithm 3 either builds a new path in the current CFG, follows an existing path, or does a combination of both. While the same group might pertain to different CFGs [Meng and Miller, 2016], a CFG cannot contain only part of a group. Therefore, the only part of the state that matters to Algorithm 3 is the *current* pair — the *cfg* and the *working* node. Algorithm 3 is parameterized by these two arguments, plus the group to be processed. The algorithm follows the instructions of the group, updating the *working* node in the process. It returns a block that has the tail instruction of the group, which Algorithm 1 uses to update the *working* node (Alg. 1-Line 15).

Algorithm 3 processes instructions individually (Line 3). There is an auxiliary variable *curr_instr* to ensure that the first instruction of the group belongs to the successor of the *working* node (Line 2). When *curr_instr* is defined, Algorithm 3 takes no action (Lines 4-5). In this case, the *instr* already exists in the basic block of the *working* node. When *curr_instr* is NIL, there is a switch from one basic block to another. Such event happens in several scenarios, each one implying different actions:

1. The program flow is moving onto a phantom node. Algorithm 3 "resurrects" it, that is to say, turns this phantom node into a basic block (Lines 9-11). A new edge is created between the *working* node and the revived block with the execution count increased by one (Line 16). The new block becomes the current *working* node (Line 17).

2. The program flow is moving onto the middle of a sequence of instructions previously thought to be a single basic block. Algorithm 3 splits this block (Lines

Algorithm 3 Process group by handling each instruction individually.

```
    input: cfg, working, group
    output: working
 1: function PROCESS_GROUP(cfg, working, group)
 2:     curr_instr = nil
 3:     for instr in group.INSTRS() do
 4:         if curr_instr then
 5:             assert curr_instr == instr
 6:         else
 7:             node = cfg.FIND_NODE_WITH_ADDR(instr.addr)
 8:             if node then
 9:                 if node instanceof Phantom then
10:                     node = cfg.FIND_NODE_WITH_ADDR(instr.addr)
11:                     node = cfg.PHANTOM2BLOCK(node, Block(Group(instr)))
12:                 else if node.group.leader ≠ instr then
13:                     node = cfg.SPLIT(node, instr.addr)
14:                 end if
15:                 assert node.group.leader == instr
16:                 cfg.ADD_EDGE(working, node, 1)
17:                 working = node
18:             else
19:                 if (instr ≠ group.leader) and (working instanceof Block) and (not working.calls.EMPTY())
20:                         and (not working.signals.EMPTY()) and (not cfg.SUCCS(working).EMPTY()) then
21:                     assert (working.group.tail.addr + working.group.tail.size) == instr.addr
22:                     working.group.ADD_INSTR(instr)
23:                 else
24:                     node = cfg.ADD_NODE(Block(Group(instr)))
25:                     cfg.ADD_EDGE(working, node, 1)
26:                     working = node
27:                 end if
28:             end if
29:         end if
30:         curr_instr = working.group.NEXT(instr)
31:     end for
32:     return working
33: end function
```

12-13). Then, the same steps of the previous case happens to connect the *working* node to this block and make it the new *working* node (Lines 16-17).

3. The program flow is visiting an instruction that should belong to the *working* node. Algorithm 3 appends the new instruction to the *working* node (Lines 20-23), if: (1) the instruction is not be the leader of the group — group leaders must always be the first instruction of a block; (2) the *working* node has no calls nor signal handlers to other functions, and neither does it have successors nodes.

4. The program flow is visiting a block leader for the first time. Algorithm 3 creates a new node to represent this block of instructions, and sets the *working* pointer to it (Lines 25-28).

When there is a mismatch between an instruction of the group (*instr*) with the tracker pointer (*curr_instr*), Algorithm 3 takes one of two possible actions. (1) a block must be created (event 4) or modified (events 1 and 2) which guarantees *instr* is the leader. Edges may be added to connect the previous *working* block with this new block. (2) *instr* must be the tail of the *working* block (event 3).

Example 4.2.6. Figure 4.3(e) shows that the phantom node @`0x400492` in Figure 4.3(d) was replaced with an actual basic block. This happens during the processing of Group 5 in Figure 4.3, when the branch `jge` is visited. Figure 4.3(d) shows the splitting of block @`0x400492` into two new blocks: @`0x400492` now with four instructions and @`0x4004a0` with the remaining two instructions. This happens during the processing of Group 4 in Figure 4.3, because of the jump to @`0x4004a0`.

4.3 Extensions to the Basic Algorithm

The core algorithms described in Section 4.2 support extensions for performance and precision. Regarding efficiency, Algorithm 1 admits a caching strategy to avoid unnecessary recomputations. Regarding precision, the algorithm supports multi-threaded programs and signal handlers. These extensions are key for CFGGRIND to provide the functionalities described in Chapter 3.

4.3.1 Caching Strategy

By Definition 2, once the program flow reaches the leading instruction of a group g, every instruction within g will be executed. As a consequence of this observation, it is not necessary to invoke Algorithm 3 on groups that have already been visited in the same context. In other words, the outcome of function PROCESS_GROUP (Alg. 3), invoked at Line 11 of Algorithm 1, is always the same for a given triple (*cfg*, *working*, *group*). Therefore, as an optimization, the algorithm associates a cache in each node of the *cfg*. When a pair formed by a *working* node and a group is processed for the first time, the algorithm caches the next *working* node. This cache is a table $working_{src} \times group \mapsto (working_{dst}, count)$. The execution *count* is updated in case of cache hits, and flushed in case of cache misses. This optimization is implemented by Algorithm 4, which augments Algorithm 1 with a cache.

Example 4.3.1. The cache avoids work due to repeated loop iterations. The loop in Figure 4.1(a) iterates once, because its input is a single-element array. However, running this program with a longer array, only the first loop iteration would change the structure of the CFG. The other iterations would just update the execution counters. In this case, the *working* pointers would be moving between the loop condition (block

Algorithm 4 Algorithm 1 update to use a caching strategy to avoid recomputation of function PROCESS_GROUP.

```
    global: state
    input: machine, mapping
    output: mapping
 1: function PROCESS_PROGRAM(machine, mapping)
 2:     for group in machine.RUN() do
 3:         addr = group.leader.addr
            ...
11:         idx = addr mod CACHE_SIZE
12:         (cached_group, cached_working, cached_count) = state.current.working.cache[idx]
13:         if cached_group == group then
14:             state.current.working.cache[idx] = (cached_group, cached_working, cached_count + 1)
15:             state.current.working = cached_working
16:         else
17:             if cached_count > 0 then
18:                 state.current.cfg.FLUSH_COUNTS(
                        state.current.working, cached_group, cached_working, cached_count)
19:             end if
20:             prev_working = state.current.working
21:             state.current.working = PROCESS_GROUP(state.current.cfg, state.current.working, group)
22:             prev_working.cache[idx] = (group, state.current.working, 0)
23:         end if
24:     end for
        ...
29:     for (addr, cfg) in mapping do
30:         for src in cfg.NODES() do
31:             for (group, dst, count) in src.cache do
32:                 if count > 0 then
33:                     cfg.FLUSH_COUNTS(src, group, dst, count)
34:                 end if
35:             end for
36:         end for
37:     end for
38:     return mapping
39: end function
```

@0x4004a0) and loop body (block @0x4004a4) without generating new information, except updating its execution count.

The cache avoids the $O(i)$ cost of Algorithm 3, where i is the number of instruction in the group, upon cache hits. The performance evaluation in Chapter 5 indicates that such situations abound. To support this optimization, the CFG is augmented with the following operation:

- *flush_counts(src, group, dst, count)*: flush execution counts of *group* for *count* times by following the edges starting from *src* node until it reaches the *dst* node.

Every *entry* and *block* node has a cache with n triples like (*group*, *working*, *count*). Entries are indexed by the *leader* address of the group (Alg. 4, Lines 11). The cache size n is configurable at compile-time. The *working* node is updated without invoking Algorithm 3 if the current *group* plus its *working* node gives us a cache hit (Lines 13-15). The cache is updated; hence, increasing by one its execution count.

If we have a cache miss, then a number of actions must be taken, before a new entry is added to the cache. In particular, Algorithm 4 needs to update counters associated with paths stored in the cache. This “flush" is necessary because the cache avoids processing instructions, including updating edge counters. Flushing happens at Lines 17-19 of Algorithm 4. After flushing the cache, Algorithm 4 processes the new group and updates the cache (Lines 20-22). The first entry of a group in the cache is associated with a counter of zero (Line 22), because this first information is recorded directly in the current CFG’s edges when PROCESS_GROUP is invoked at Line 21 of Algorithm 4.

4.3.2 Support to Multi-Threaded Programs

The prototype of CFGGRIND, implemented in VALGRIND, has support for multi-threaded programs. VALGRIND natively serializes the execution of such programs into a single-threaded application by implementing its own scheduling policy. CFGGRIND leverages this feature by maintaining a *state* per thread of execution. In Algorithm 5, the *state* is the same global variable used by Algorithms 1-3. The global variable *current_thread* keeps all the information regarding the execution of the active thread, including its ID. The global map *thread_states* holds the states for all the threads indexed by the thread IDs. Initially, each thread state is initialized with an empty state *(*NIL*, [])*, similarly to how the global *state* is configured prior to program execution (Sec. 4.1). A context switch occurs as an event external to the process that runs Algorithms 1-3. Thus, this operation is of no consequence to the inner workings of these algorithms.

Algorithm 5 Context switch from the *current_thread* to another *next_thread*.

```
    global: state, current_thread, thread_states
    input: next_thread
1: procedure SWITCH_CONTEXT(next_thread)
2:     assert $current\_thread \neq next\_thread$
3:     $thread\_states[current\_thread.id] = state$
4:     $state = thread\_states[next\_thread.id]$
5:     $current\_thread = next\_thread$
6: end procedure
```

In a context switch from the active thread (*current_thread*) to a different thread (*next_thread*), Alg. 5 saves the *state* of *current_thread* in the map *thread_states*, indexing it by *current_thread*’s ID (Line 3). Then, the previously saved state of *next_thread* is restored to the global *state* (Line 4). Finally, Alg. 5 updates variable *current_thread* to refer to *next_thread* (Line 5).

Although programs are emulated as is, the instrumentation runtime environment adds an overhead to the execution that may affect its behavior. Some specific timed constraints may never be reached, and thus some code parts never executed. This is a limitation of dynamic binary frameworks in general, not the algorithms presented in this section.

4.3.3 Handling Signal Events

The prototype of CFGgrind has the ability to precisely track signal events. CFGgrind relies on valgrind's capabilities to identify when an event is raised by the machine and when this event is properly handled by the program. To support signal events, CFGgrind employs a strategy similar to the one employed for multi-threaded programs: it manipulates the global *state* externally. Algorithm 6 is responsible to prepare the state when a signal event is raised, whereas Algorithm 7 is responsible to recover the state once the signal was handled by the program. Both Algorithms 6 and 7 hold a global variable for the active thread (*current_thread*), and a global map of state's queue indexed by thread IDs (*signal_states*). Signals occur in the scope of a thread; thus each thread must have its own signal handlers. A queue is required to hold the thread state in case multiple signals are raised simultaneously — they must be treated separately and in sequence.

Algorithm 6 Process a raised signal event.

```
    global: state, current_thread, signal_states
    input: mapping, sigid, @target_addr
1: procedure ENTER_SIGNAL(mapping, sigid, @target_addr)
2:     called = mapping.GET(@target_addr) if mapping.HAS(@target_addr)
           else mapping.PUT(@target_addr, CFG())
3:     assert state.current.working instanceof Block
4:     state.current.working.ADD_SIGNAL(sigid, called, 1)
5:     signal_states[current_thread.id].ENQUEUE(state)
6:     state = (nil, [])
7: end procedure
```

In Algorithm 6, ENTER_SIGNAL receives the map of CFGs (*mapping*), the ID of the signal raised (*sigid*), and the address of the first instruction to be executed afterwards (*@target_addr*). This address is the entry point of a function that will be called to handle the signal. Algorithm 6 obtains the called CFG for the signal handler based on *@target_addr* (Line 2). Then, it adds this CFG to the list of signal handlers associated with the *working* node with an execution count of one (Lines 3-4). Notice that this *working* node must be of the *block* type. Later, the current *state* is pushed onto the *signal_states* queue for the active thread (Line 5). Finally, the state is initialized as empty to proceed with the execution of Algorithms 1-3.

Algorithm 7 Process a handled signal event.

```
    global: state, current_thread, signal_states
1: procedure LEAVE_SIGNAL( )
2:     while state.current do
3:         state.current.cfg.ADD_EDGE(state.current.working, state.current.cfg.exit, 1)
4:         state.current = state.callstack.POP() if state.callstack else nil
5:     end while
6:     assert not signal_states[current_thread.id].EMPTY()
7:     state = signal_states[current_thread.id].DEQUEUE()
8: end procedure
```

When leaving a signal handler, Algorithm 7 connects the *working* node of the current *state* and all the *working* nodes in its *callstack* if present, to the *exit* node with execution count of one (Line 2-6). This step is similar to the one present in Algorithm 1, Lines 13-16: it is used to ensure consistency of CFGs. Then, the *state* is restored by popping the *signal_states* queue for the *current_thread*.

4.4 Conclusion

This chapter develops the foundation of the algorithms implemented by CFGgrind, a tool built on top of valgrind [Nethercote and Seward, 2007]. The dynamic binary instrumentation framework valgrind plays the role of the machine described in Section 4.1. The basic algorithm, described Section 4.2, and its extensions, described in Section 4.3, are publicly available for use in CFGgrind's repository (`https://github.com/rimsa/CFGgrind`). A working python prototype of these algorithms is also available in this repository.

Chapter 5

Evaluation

The techniques introduced in this thesis are integrated into a tool, CFGGRIND, which is publicly available at https://github.com/rimsa/CFGgrind. Although a research artifact, CFGGRIND's implementation is sufficiently solid to enable the exploration of several research questions:

RQ1 How efficient is CFGGRIND, when compared with tools with similar purpose?

RQ2 What is the impact of the cache (Algorithm 4) on the performance of CFGGRIND?

RQ3 What is the ratio between complete and incomplete CFGs in large programs?

RQ4 What is the impact of different input sets in the incremental refinement of CFGs?

RQ5 How much information does CFGGRIND add onto a static CFG reconstructor?

RQ6 What is the time complexity of CFGGRIND in practice?

Benchmarks. This evaluation of CFGGRIND uses two benchmarks, CBENCH (http://ctuning.org/) and SPEC CPU2017 (https://www.spec.org/). CBENCH contains 32 C programs. Each program has 20 available input sets, except `bzip2d` and `bzip2e` with 32 inputs each. The CBENCH programs were modified to compile and run in a 64-bit architecture. SPEC CPU2017 contains 43 programs, written in C, C++ and Fortran. They are organized in 4 categories: integer and floating point, separated into single- and multi-thread versions. Multi-thread programs were configured to execute as single-thread, since VALGRIND serializes the execution. Thus, executing the programs with a single thread in one core for the experiments is sufficient because the performance of CFGGRIND is not affected by the number of threads in an execution (Sec.

3.4). All programs in cBench and Spec Cpu2017 are compiled with gcc at the -O2 optimization level. In the comparison with DynInst (**RQ5** in Section 5.5), the code was stripped from debugging symbols.

Runtime Setup. The current version of CFGgrind has been implemented in valgrind version 3.15.0. Results reported for cBench were produced on a 16-core Intel(R) Xeon(R) E5-2630 at 2.40GHz with 16GB of RAM running CentOS 7.5. For Spec Cpu2017, the results were obtained on an 8-core Intel(R) Core(TM) i7-4790 at 3.60GHz with 16GB of RAM running CentOS 7.6. We use two machines to run the experiments in parallel.

Measurement Methodology. Performance numbers for cBench are the average of three executions for each program. On average, a run on the entire cBench is completed in ∼22.5h. The difference between the fastest and slowest of the three runs is less than one minute. Due to this small difference — one minute in 22.5 hours, we shall not report standard deviations in our results. Performance numbers for Spec Cpu2017 were measured only once because of the long run times. Executing a single set of experiments for `intrate` takes ∼30.1h, `fprate` ∼35.5h, `intspeed` ∼40.6h, and `fpspeed` ∼295.6h. The experimental evaluation used all the inputs available in both benchmarks for the simulations. To answer RQ1 5.1 and RQ2 5.2, the average is computed using the geometric mean. The variance between each program run times is high: the fastest program in cBench executes in ∼2s, while the slowest in ∼20s; in Spec Cpu2017 the fastest runs in ∼4m, while the slowest in ∼57m. To answer RQ3 5.3, the average is computed using the arithmetic mean. The total number of complete, incomplete, and unreached CFGs is divided by the total number of CFGs in the benchmark suite.

5.1 RQ1: Efficiency

Dynamically reconstructing CFGs with CFGgrind during the execution of a program results in significant overhead. For instance, the execution of the 32 programs of cBench with CFGgrind is ∼19 times slower than an equivalent non-instrumented baseline execution. For the 43 programs in Spec Cpu2017, CFGgrind has a slowdown of ∼29 times. This runtime cost is on par with other tools built on top of valgrind, whose manual we quote below [Seward, 2019]:

> ‘The amount of instrumentation code added varies widely between tools. At one end of the scale, `Memcheck` adds code to check every memory access and every value computed, making it run 10-50 times slower than natively.

At the other end of the spectrum, the minimal tool, called `Nulgrind`, adds no instrumentation at all and causes in total “only" about a 4 times slowdown.’

The empirical results in this section evidence that the instrumentation overhead is also high for other tools that reconstruct CFGs. Figure 5.1 presents, in logarithmic scale, a comparison of the slowdown for three different tools that reconstruct CFGs: CFGgrind, bfTrace [Gruber et al., 2019] and DCFG [Yount et al., 2015]. The baseline for these comparisons is the original program. Figure 5.1 also shows the slowdown caused by callgrind, a valgrind tool that builds the call graph of a program. Results for the 32 programs available in cBench are reported; however, we omit for SPEC CPU2017 because DCFG takes a prohibitively long time to process the larger Spec Cpu2017 suite.

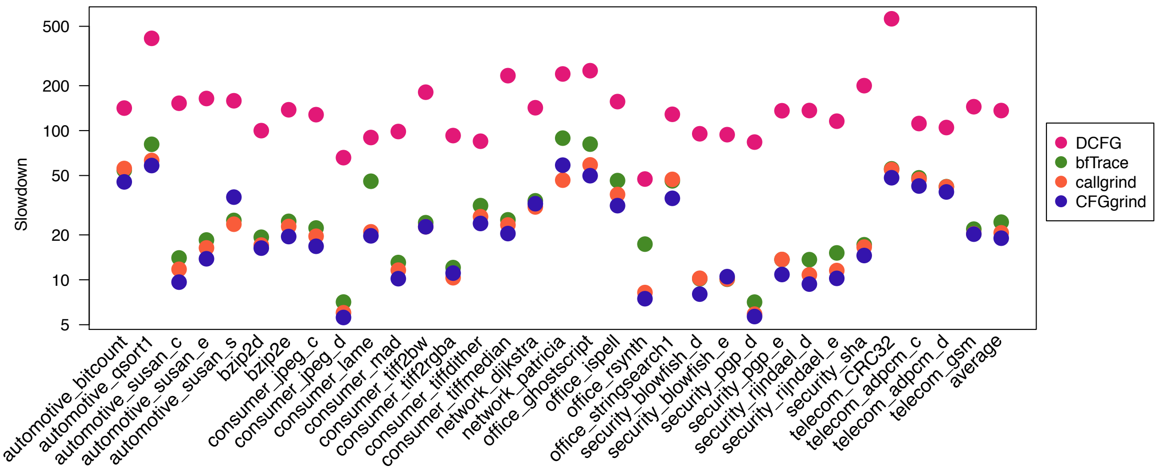


Figure 5.1. Slowdown of different tools that reconstruct CFGs relative to the original program execution without instrumentation for cBench.

The CFG-reconstruction tools compared in Fig. 5.1 are not equivalent (see Chapter 3). callgrind is not a CFG reconstructor, but it is included in the comparison because it runs on valgrind, like CFGgrind does. Hence, callgrind serves as a performance baseline for readers that are familiar with the valgrind’s ecosystem. The other three tools in Fig. 5.1 reconstruct CFGs. However, their outputs, although similar, are not directly comparable because each tool uses its own program representation. For instance, DCFG produces a single CFG for the entire program; CFGgrind and bfTrace, in turn, splits it per function.

Figure 5.1 indicates that CFGgrind and bfTrace are much faster than DCFG. DCFG, built onto PinPlay, saves program state for posterior re-execution — an overhead absent on the other tools. On average, DCFG is ∼7x slower than CFGgrind. CFGgrind runs faster than bfTrace, although by a lower margin: ∼28%. CFGgrind is also faster than callgrind: ∼9%. Figure 5.1 also shows the runtime for the original programs. Binaries analyzed by CFGgrind experiment a slowdown of ∼19x when compared to the original programs — viz., without any emulation. To put these numbers in perspective, DCFG causes a slowdown of ∼136x and bfTrace, ∼24x. valgrind, without any tool, slows cBench down by ∼3.6x on average; however, for the sake of readability, we omit this result from Figure 5.1.

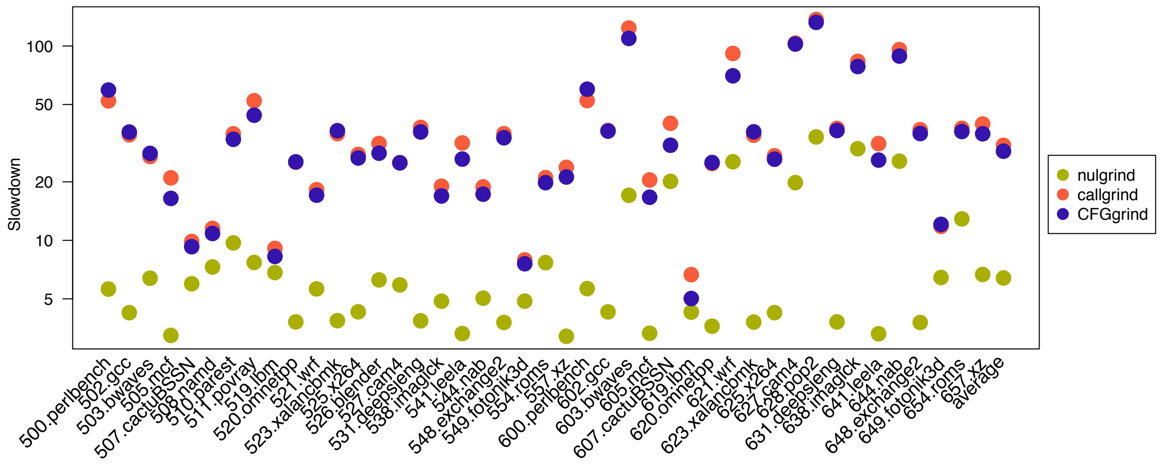


Figure 5.2. Slowdown of builtin tools in valgrindand CFGgrind relative to the original program execution without instrumentation for Spec Cpu2017.

Figure 5.2 compares the runtime of CFGgrind for the Spec Cpu2017 suite against the non-instrumented baseline program and other valgrind builtin tools. These results indicate that CFGgrind has a performance on par with callgrind — CFGgrind is actually ∼7% faster than callgrind. Even though CFGgrind has a slowdown of ∼29x in relation to the original program, it is only ∼4.5x slower than running valgrind without any tool (nulgrind). Notice that CFGgrind delivers substantial more information than callgrind. CFGgrind's CFGs encapsulates the call graph of programs, in addition to all the instructions and paths traversed during the program flow. CFGgrind is as suitable as other tools in the valgrind ecosystem for practical use.

5.2 RQ2: The Impact of the Cache

The cache implemented by Algorithm 4 is key to boost CFGGRIND's performance. As explained in Chapter 4, the caching strategy avoids the re-execution of Algorithm 3 for a previously visited pair (*working*, *group*). A cache hit enables the algorithm to move directly to the next *working* node without processing all the instructions of the group. In CFGGRIND, the size of the cache is configurable at compilation time: for each *working* node there are n entries indexed by different group addresses. However, increasing n past a certain value results in diminishing returns. Figure 5.3 illustrates this trend on the training set for the `intrate` part of the SPEC CPU2017 suite.

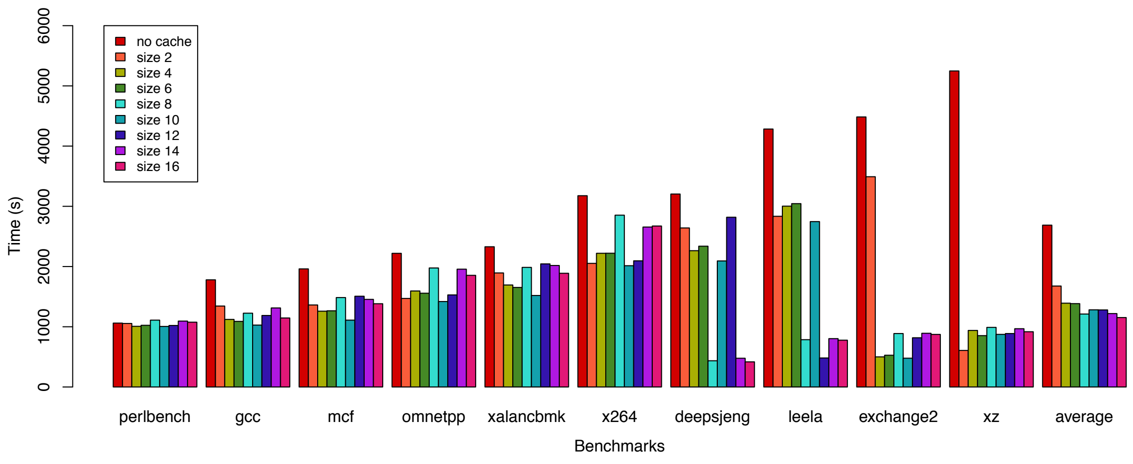


Figure 5.3. The impact of cache size on the runtime of CFGGRIND.

Figure 5.3 makes it clear that the cache is important: the average performance improvement from the introduction of a cache with $n = 2$ is ∼1.6 times. This benefit is substantial in loop-intensive programs, such as `xz`. Setting $n > 2$ produces mixed results because most of the basic blocks in a program have only one or two successors. Exceptions to this rule are due to indirect jumps, such as those used to implement switch statements. Thus, although the last column of Figure 5.3 tends to report increasingly better results, improvements past $n = 8$ are too small to be of practical consequence. Larger cache sizes might even provoke slowdowns due to heavier memory usage. Based on these results, the experiment performed to answer **RQ1** (Sec. 5.1) used a fixed cache size of $n = 8$, which provides a good balance between efficiency and memory requirements.

5.3 RQ3: CFG Completeness

When used to support software testing, CFGGrind accurately recovers the portions of code traversed by test inputs, with exact profiling information (as explained in Chapter 3). And, contrary to classic approaches [Fraser and Arcuri, 2011; Lemos et al., 2006], it also recovers the CFG of library code. Program coverage through dynamic reconstruction of CFGs can be estimated by answering the following question: "how many functions had all their instructions executed at least once by a particular input?". These functions are called *complete*, as stated in Definition 4. This section provides an answer to this research question.

Determining Functions of Interest. Even though CFGGrind can track the execution of dynamically shared libraries, this study of completeness considers only functions available in the source code of each benchmark. This restriction enables the computation of a ratio of completeness because the total number of functions that can be invoked is available when the source code is accessible. The `.text` section of binary files, compiled with debugging symbols, is used to identify source-code functions. Spec Cpu2017 has 172,268 functions scattered across 43 programs; cBench has 7,250 functions in 32 programs.

Invocation Ratio. The *invocation ratio* of a set of inputs for a benchmark suite is the number of functions that are invoked over the total number of functions in the programs in the benchmark suite. For the Spec Cpu2017, with all the reference inputs, the invocation rate is ∼25%, while for cBench, with 20 inputs, the invocation rate is ∼38%.

Completeness Ratio. Figures 5.4-5.5 show, in logarithmic scale, the number of complete, incomplete, and unreached CFGs for Spec Cpu2017 and cBench, respectively. Both were executed with the benchmark's reference inputs. The data collection for both figures, from a single run of the entire suite, required 402 hours (almost ∼17 days) for Spec Cpu2017 and 22.5 hours for cBench. The *completeness ratio* for a benchmark suite with a given workload is the number of functions for which the entire CFG was discovered divided by the number of functions that were invoked with that workload. For the Spec Cpu2017 suite with the reference inputs, the completeness ratio is ∼40%, and for the cBench suite the completeness ratio is ∼37%.

Figure 5.6 shows the correlation between completeness ratio and (a) number of blocks, or (b) number of instructions per CFG. To improve readability, the graph shows

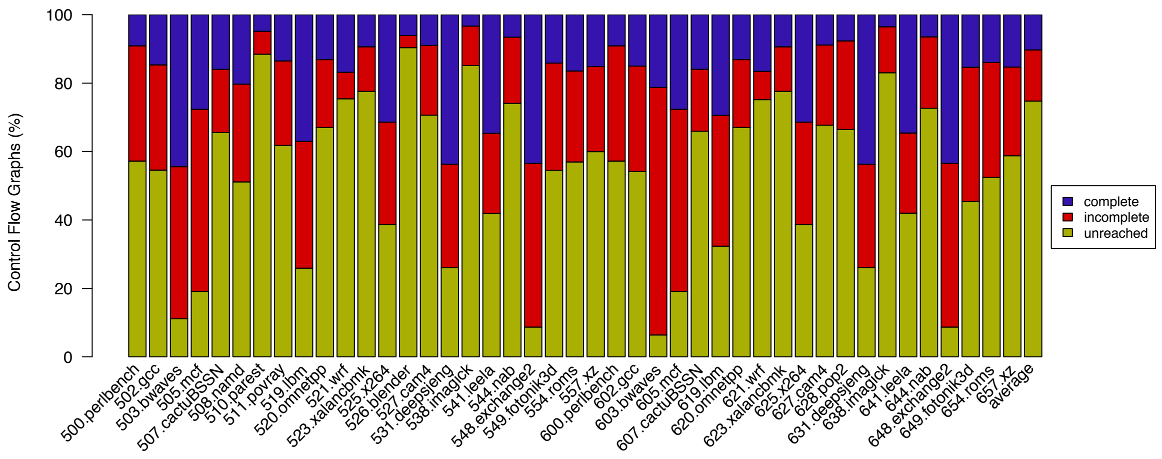


Figure 5.4. Number of complete/incomplete/unreached flow graphs for each of the 43 programs in Spec Cpu2017.

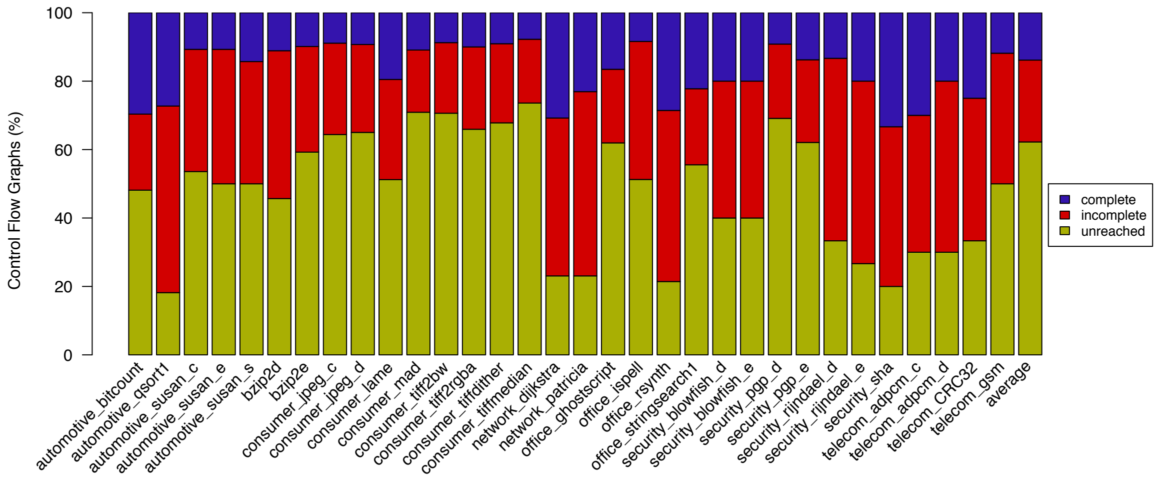


Figure 5.5. Number of complete/incomplete/unreached flow graphs for each of the 32 programs in cBench.

results only for CFGs with up to 100 blocks (a) and up to 1,000 instructions (b). For example, Spec Cpu2017 has 392 CFGs that contain exactly 20 block nodes. Out of those, 50 are complete and 342 are incomplete. The same is true for instructions: out of the 15 CFGs of Spec Cpu2017 with exactly 200 instructions, 4 are complete and 11 are not. Most of the CFGs in programs in the Spec Cpu2017 suite are small; hence, the negative slopes in Figure 5.6(a-b). A similar behaviour is observed in cBench, although not shown in this manuscript. This decreasing rate is much more accentuated for complete CFGs. This trend indicates that, as expected, the probability of finding complete CFGs decreases as the size of the CFG increases.

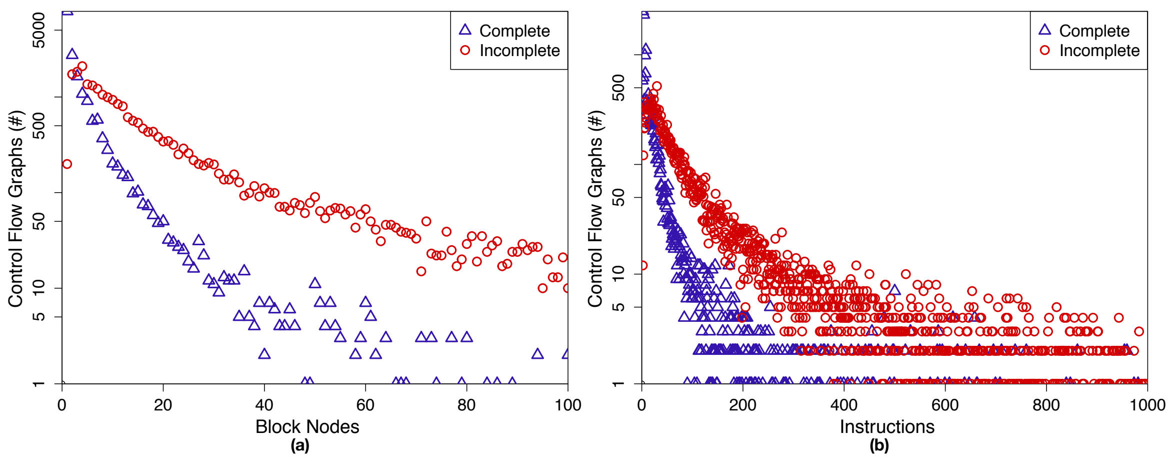
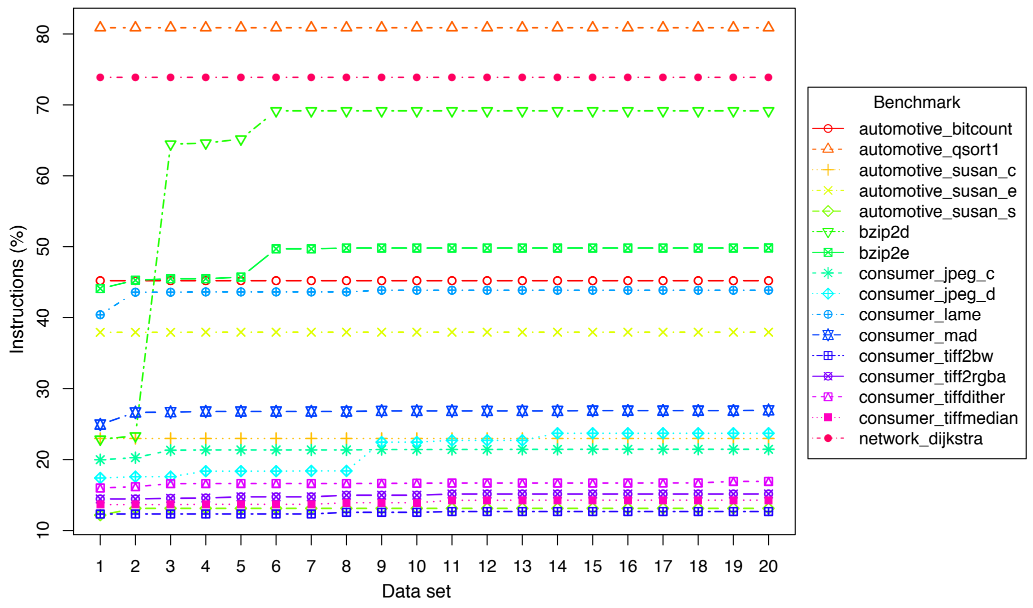


Figure 5.6. Relation between the number of complete/incomplete CFGs (y-axis, ln scale) per number of blocks (a) and instructions (b) for the Spec Cpu2017 benchmark. X-axis shows number of blocks (a) and instructions (b).

5.4 RQ4: Incremental Construction of CFGs

Multiple invocations of the same function during a single run of a program might lead to more complete CFGs when new paths are explored. To capitalize on this observation, the results produced by a run of CFGgrind can be forwarded as input to another run. If the new execution flows into unexplored program areas, this information will be added to the CFGs produced. Entire CFGs can be included when new functions are called, and existing CFGs can be expanded when phantom nodes or unmapped areas are visited. The more inputs are given to CFGgrind, the more complete the reconstruction of the program's control flow. Note that neither bfTrace nor DCFG support incremental construction of CFGs, as discussed in Section 3.3.

Figure 5.7 shows how extra inputs contribute to augment the number of visited instructions in cBench. This benchmark is well suited for this experiment because each program comes with 20 data sets, except for `bzip2d` and `bzip2e` that come with 32 inputs each. In this case, the 32 inputs were evenly distributed as 20 inputs in Figure 5.7. Each tick in the X-axis of Figure 5.7 shows the number of instructions observed up to the n^{th} execution of a program $(1 \leq n \leq 20)$. Following the methodology used in Section 5.3, library functions are excluded from this analysis. Considering all 32 cBench programs, the 19 extra inputs augment the number of instructions visited from 127,016 to 163,750 — an increase of ∼29%. The largest growths were



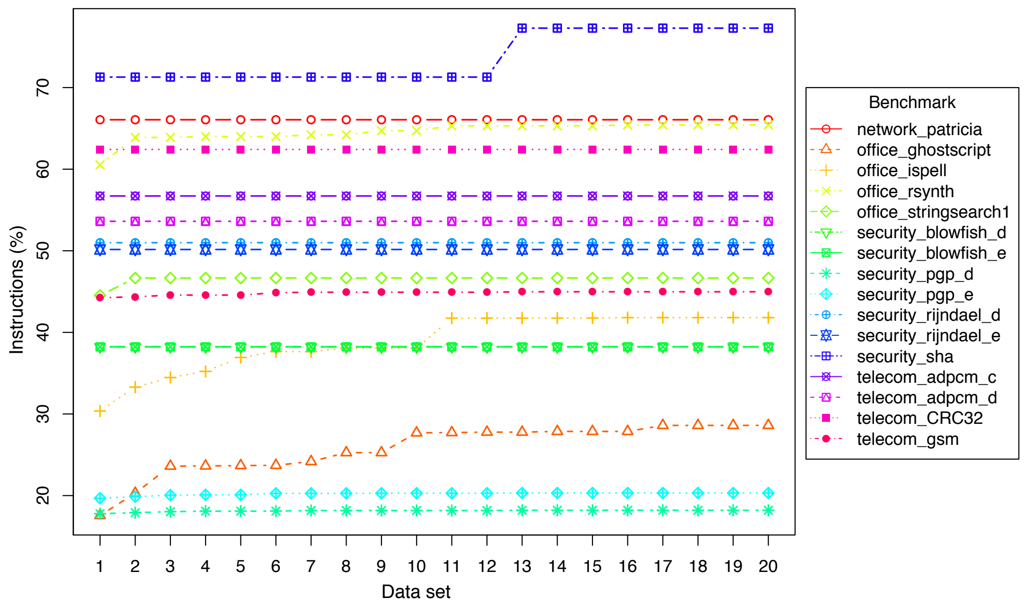


Figure 5.7. Evolution of instruction coverage (y-axis) due to incremental execution of inputs for cBench.

observed in `bzip2d`, 19 new CFGs were added of the 81 available CFGs for the entire program (∼23%), and `office_ghostscript`, 312 new CFGs added onto 3,488 (∼9%). Comparing the first and last executions of all the programs reveals that CFGGrind was able to identify 378 new CFGs — a growth of 5.21% over a universe of 7,250 CFGs available in the text section of cBench. Applying the same principles for instructions, 36,736 new unique instructions were executed — a growth of 6.01% upon 601,345 instructions in cBench. This experiment indicates that, at least for cBench, extra inputs have a mild effect on code coverage: they provide new information about the program execution. Although a great extent of each program was already observed in the first execution. This is not a limitation of CFGGrind, further coverage could be achieved with improved tests inputs.

5.5 RQ5: Combining Static and Dynamic CFG Reconstruction

DynInst, a state-of-the-art static CFG reconstructor [Meng and Miller, 2016], can be used to extend CFGGrind's coverage, and vice-versa. This section uses these two tools in tandem to analyze cBench and Spec Cpu2017. For this experiment, we have compiled the benchmarks without debugging information — the typical way in which production code is distributed. Table 5.1 shows the result of this comparison for cBench. The invocation ratio of CFGGrind is ∼38%; hence, it identifies 2,738 out of 7,250 possible CFGs. The invocation ratio of DynInst is 42%; hence, it finds 3,049 CFGs. The two techniques found 1,633 common CFGs, i.e., ∼23% of the total. Similarly, Table 5.2 shows results for Spec Cpu2017. The invocation ratio of CFGGrind is ∼25%. This percentage means that it identifies 43,485 out of 172,268 CFGs. The invocation ratio of DynInst is ∼39%; hence, it finds 66,552 of the CFGs. The two techniques found 30,825 common CFGs in Spec Cpu2017, i.e., ∼18% of the total.

	CFGGrind (A)	DynInst (B)	$A \cap B$	$A \setminus B$	$B \setminus A$
CFGs	2,738	3,049	1,633 (59.6%/53.6%)	1,105 (40.4%)	1,416 (46.4%)
Basic blocks	33,316	76,456	23,608 (70.9%/30.9%)	9,708 (29.1%)	52,848 (69.1%)
Edges	52,980	111,732	37,345 (70.5%/33.4%)	15,635 (29.5%)	74,387 (66.6%)
Instructions	163,752	332,189	124,338 (75.9%/37.4%)	39,414 (24.1%)	207,851 (62.6%)
Calls	7,596	18,728	4,120 (54.2%/22.0%)	3,476 (45.8%)	14,608 (78.0%)

Table 5.1. Comparison between CFGGrind and DynInst for cBench. In the column between the intersection of CFGGrind and DynInst, the percentage is given in relation to CFGGrind and DynInst, respectively.

	CFGGrind (A)	DynInst (B)	$A \cap B$	$A \setminus B$	$B \setminus A$
CFGs	43,485	66,552	30,825 (70.9%/46.3%)	12,660 (29.1%)	35,727 (53.7%)
Basic blocks	939,568	3,466,454	714,309 (76.0%/20.6%)	225,259 (24.0%)	2,752,145 (79.4%)
Edges	1,429,277	5,098,624	1,096,006 (76.7%/21.5%)	333,271 (23.3%)	4,002,618 (78.5%)
Instructions	4,968,718	17,712,186	4,161,470 (83.8%/23.5%)	807,248 (16.2%)	13,550,716 (76.5%)
Calls	302,929	3,160,257	198,561 (65.5%/06.3%)	104,368 (34.5%)	2,961,696 (93.7%)

Table 5.2. Similar to Table 5.1, but for Spec Cpu2017.

Binaries without debugging information hurt static analyses, whereas dynamic analyses require good program inputs to be effective. Combining these two techniques can improve code coverage.

The combined analyses find 4,154 CFGs for cBench — an invocation rate of ∼57%. CFGgrind finds 1,105 new CFGs that DynInst was unable to recover statically. In other words, CFGgrind adds ∼15% more CFGs onto the collection observed by DynInst. Similarly, the combined analysis for Spec Cpu2017 yields 79,212 CFGs — an invocation rate of ∼46%. Of those, 12,660 (∼7%) were previously unknown to DynInst. The remaining metrics in Tables 5.1-5.2, e.g., blocks, edges, instructions and calls, collide in ways that are hard to quantify. For instance, DynInst identifies instructions that are never executed, such as those used for padding. Some CFG edges mark impossible paths — they arise due to conservative estimates of indirect branches, for instance. Also, a basic block in one analysis can intersect partially with one or more basic blocks in other analysis. Thus, because there is no one-to-one correspondence between these four metrics in both analyses, the numbers presented must be understood as approximate results.

5.6 RQ6: Empirical Estimate of Asymptotic Complexity

The reconstruction of CFGs increases the complexity of program execution because of accesses to the cache discussed in Section 4.3. The cache is implemented as a hash-table. In the absence of collisions, the next working node is retrieved in constant time, i.e., with an overhead for this access of $O(1)$. However, collisions might happen. The current implementation of CFGgrind minimizes collisions via a simple expedient. If occupation of the hash-table reaches 80% of its size, then a new table, twice as large is allocated, and data is copied from the old cache to the new one. If collisions happen, then CFGgrind uses a list to store multiple entries. We have opted for a list, instead of a balanced tree, for two reasons. First, the list has lower startup cost; hence, it outperforms the balanced trees for a small number of elements. Second, the resize-and-copy procedure tends to reduce the number of collisions; thus, in practice, it is unlikely that CFGgrind's cache will contain a large number of entries with the same hash code. The other components of the algorithms discussed in Section 4.2 contribute only a constant factor to the processing of each instruction. The algorithms follow instructions in the order in which they are executed. The loop in Lines 2-12 of Algorithm 1 processes each group in order, while the loop in Lines 3-31 of Algorithm 3 processes each instruction of a group sequentially. Algorithm 2 processes the tail instruction of the group, and runs in $O(1)$ for jumps, calls and returns. For branches, the algorithm needs to find a successor in a list, but since most blocks have a small number of successors, the search cost is low. Furthermore, operations that add a node or an edge, or replace a phantom node with a block node run in $O(1)$. The operation to find the node with a specific address, or the exact program point where to split a node requires a search in a hash-table; but this operation tends to run in constant time due to lower collisions. Therefore, in practice it is still possible to reconstruct the CFG of programs with a constant time per instruction; or, in other words, with a linear cost in terms of number of executed instructions. Figure 5.8 supports this observation with empirical data.

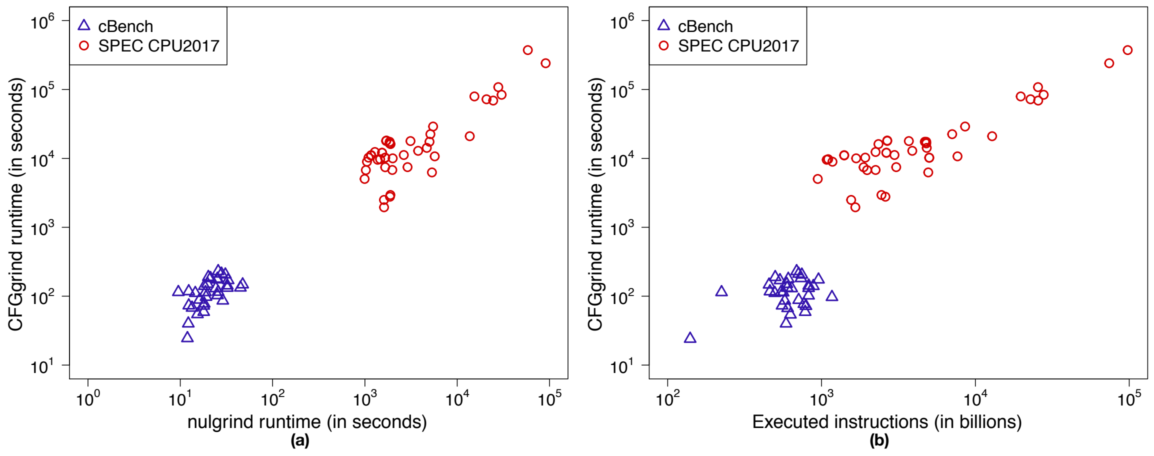


Figure 5.8. Relation between execution time of non-instrumented programs with NULGRIND (emulation only) and programs instrumented with CFGGRIND, for CBENCH and SPEC CPU2017 (a); Relation between the number instructions observed during the execution of CBENCH and SPEC CPU2017 programs, and the running time of these programs, when instrumented with CFGGRIND (b).

Figure 5.8(a) correlates the running time of programs executed with NULGRIND and the running time of programs instrumented with CFGGRIND. NULGRIND runs VALGRIND on the target program without instrumentation, as an emulation only tool. Visual inspection of the figure indicates strong linear correlation. Indeed, the coefficient of determination between these two running times is 0.905: very strong evidence of linear behavior. The linear relation between the number of instructions that are fetched during program execution, and the running time of CFGGRIND is even stronger. Figure 5.8(b) supports this statement by presenting such relation for CBENCH and SPEC CPU2017 programs. In this case, the coefficient of determination is 0.990: very close to 1.0, which would be a perfect linear relation.

5.7 Conclusion

CFGgrind enabled the exploration of six research questions as discussed in this chapter. RQ1 (Section 5.1) shows that CFGgrind outperforms DCFG by ∼7 times and bfTrace by ∼28% in execution efficiency. RQ2 (Section 5.2) shows that, by employing a caching strategy to the algorithms, CFGgrind's execution speed is significantly improved: more than 2 times when a size-8 cache is used. RQ3 (Section 5.3) shows that, the invocation ratio — invoked functions divided by all available functions — for Spec Cpu2017 is ∼25% and for cBench is ∼38%. In contrast, the completeness ratio — complete CFGs divided by all invoked CFGs — for Spec Cpu2017 is ∼40% and for cBench is ∼37%. RQ3 also shows that, the larger the CFG — more instructions or blocks —, smaller it is its probability of being complete. RQ4 (Section 5.4) shows that executing cBench with extra inputs discovered ∼5% new control flow graphs and augmented the number of visited instructions, previously unexplored, by ∼29%. RQ5 (Section 5.5) shows that DynInst finds 15% more CFGs for cBench, and 7% more CFGs for Spec Cpu2017 when combined with CFGgrind. RQ6 (Section 5.6) shows that the algorithms described in Chapter 4 exhibit a linear behavior in terms of number of executed instructions; hence, CFGgrind executes with a constant time per instruction.

CFGgrind is mature enough to be practically used, as demonstrated with its application in reputable benchmarks such as Spec Cpu2017 and cBench. The data that support the findings of this study are openly available in CFGgrind's repository at `https://github.com/rimsa/CFGgrind`.

Chapter 6

Case Study

This chapter provides a case study on the capabilities of dynamic instrumentation tools. This study is built on the concept of instruction visibility [Leobas et al., 2018], where instructions in a program's execution can be classified as visible — derived from source code — or invisible — derived from library code (Section 6.2). The relation between a program's visible parts compared to its entire execution, i.e. its *visibility ratio*, can be used to measure how such program is affected by compiler optimizations. Larger visible sections are expected to be more sensitive to these optimizations, and thus have higher impact on its execution. Thus, this case study provides an analysis on the visibility of instruction for the Spec Cpu2017 benchmark when applied to different optimization levels. The key insight is that optimizations gains obtained by this benchmark should only be expected in programs that share a similar visibility ratio. More about the motivations of this work can be found in Section 6.1.

The visibility ratio must be measured by instrumenting a program's execution to count how many instructions are visible and invisible (Section 6.3). CFGgrind can be used for this purpose: the instructions count can be extracted from the profiling information gathered in the process of reconstructing the CFGs. However, the cost to build the CFGs for this application can be avoided. Therefore, a new dynamic instrumentation tool, instrgrind, was conceived (Section 6.3). This tool is derived from CFGgrind, where most of its CFG reconstruction functionalities were stripped in favor of a more lightweight version. This adjustment is important, otherwise the experiments in this section would take a prohibitively long time to run (Section 6.4). Thus, this chapter shows how a relatively heavy weight tool can inspire a much lighter counterpart in order to keep dynamic analysis *practical*.

6.1 Introduction

The Standard Performance Evaluation Corporation (SPEC) was founded in 1988 to create standard benchmark suites to evaluate the performance of computer systems [Dixit, 1993]. Since then, SPEC has evolved to provide benchmarks for Cloud Computing, Graphics, Java, Storage, Virtualization, Web services and Power Consumption, including SERT, a suite widely used to evaluate computer server efficiency. For the compiler and computer architecture research community, SPEC has been the dominant source of programs used to predict the effects of innovation on the performance of computing systems [Phansalkar et al., 2007a]. This suite focuses on compute-intensive applications that stress the CPU and the higher levels of the memory hierarchy. The first collection of SPEC CPU benchmarks in this series was announced in October 1989 [Dixit, 1991]. The most recent release is the SPEC CPU2017 benchmark suite. SPEC CPU2017 is divided into four suites: `intspeed`, `fpspeed`, `intrate` and `fprate`. Programs in these collections are implemented in either C, C++ or Fortran.

SPEC CPU has been fundamental to the standardization of performance measurement in the computing industry. This impact has, unsurprisingly, been of great consequence to the development of computer architectures. As testimony to this statement, the many editions of Patterson and Hennessy's classic textbook, including its latest version [Patterson and Hennessy, 2013], rely on different releases of SPEC CPU to justify hardware design decisions. For computer-architecture simulation studies, many efforts to subset SPEC benchmark suites led to the capture of the behavior of the programs with shorter simulations [Phansalkar et al., 2007b; Nair and John, 2008; KleinOsowski and Lilja, 2002]. Therefore, compiler writers and computer architects tend to employ SPEC CPU as a beacon: good code optimizations should perform well in this benchmark collection. Such importance, however, elicits one question from a compiler's perspective: how much of the gains obtained in SPEC CPU can be extrapolated to benefits into actual applications?

To answer the above question, this case study analyzes the provenance of machine instructions processed during the execution of SPEC CPU2017 programs. This study uses INSTRGRIND, a new dynamic profiler, based on VALGRIND [Nethercote and Seward, 2007], that counts executed instructions. In an effort to summarize results, this case study classifies the instructions produced by INSTRGRIND as *visible* and *invisible*. The former comes from the source code of the benchmark; the latter, from libraries invoked during its execution. Visible instructions determine the proportion of the program that the compiler can modify.

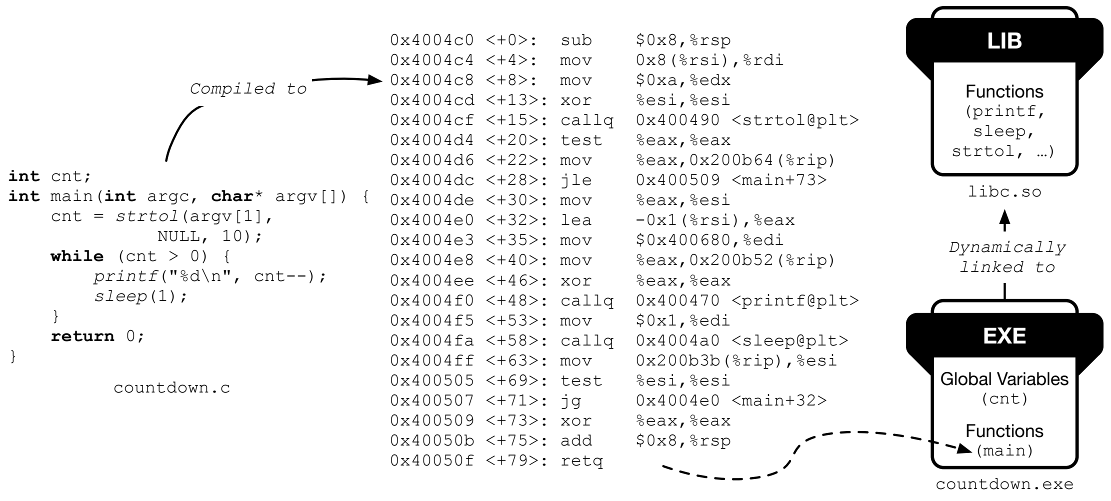
To assess the representativeness of SPEC CPU2017, this case study uses instrgrind to measure instruction provenance in the GNU CoreUtils library. GNU CoreUtils consists of 106 utility programs used in UNIX-like operating systems. Examples include `cat`, `mv` and `ls`. The GNU CoreUtils programs are the epitome of system applications: they interact heavily with the operating system, are input bound, and are massively used in production environments. The comparison of instruction provenance in Spec Cpu2017 and GNU CoreUtils leads to the following results:

- The proportion of visible instructions found in Spec Cpu2017 when compiled using gcc with -O0 is, on average, 0.922.
- The same ratio in GNU CoreUtils, when compiled with gcc at -O0 is, on average, 0.108; hence, substantially smaller than Spec Cpu2017's. Thus, whereas most of Spec Cpu2017's executed instructions are visible, most of GNU CoreUtils' are invisible.
- Optimization levels maintain the disparity between these ratios. For instance, the ratio observed in Spec Cpu2017 compiled using gcc with -O2 is 0.701. Similar behavior holds for GNU CoreUtils: the proportion of visible instructions is still low, at 0.073.

Following SPEC's recommended methodology [Dixit, 1993], all the averages are geometric means. Measurements in Spec Cpu2017 use the combined benchmarks' reference inputs. Modified versions of the scripts distributed with GNU CoreUtils are used to execute programs in this collection. Benchmarks were compiled with gcc. Results reported in this case study indicate that there is greater potential for compiler-based code transformations to impact the performance of Spec Cpu2017 than the performance of applications from other domains. This observation is confirmed by the total execution time on different levels of compiler optimization. The execution time of Spec Cpu2017 at level 2 (gcc with -O2) is 45% lower than at level 0 (gcc with -O0). For GNU CoreUtils this reduction is 1%. The results of this case study concern a particular architecture, operating system and compiler; however, instrgrind—itself a contribution of this work—can be used in tandem with other such triples.

6.2 Visible and Invisible Instructions

In a computer application, only part of the instructions executed come from the application's source code. The rest of the instructions comes from dynamically linked



```
int cnt;
int main(int argc, char* argv[]) {
    cnt = strtol(argv[1],
           NULL, 10);
    while (cnt > 0) {
        printf("%d\n", cnt--);
        sleep(1);
    }
    return 0;
}
```

```
0x4004c0 <+0>:  sub    $0x8,%rsp
0x4004c4 <+4>:  mov    0x8(%rsi),%rdi
0x4004c8 <+8>:  mov    $0xa,%edx
0x4004cd <+13>: xor    %esi,%esi
0x4004cf <+15>: callq  0x400490 <strtol@plt>
0x4004d4 <+20>: test   %eax,%eax
0x4004d6 <+22>: mov    %eax,0x200b64(%rip)
0x4004dc <+28>: jle    0x400509 <main+73>
0x4004de <+30>: mov    %eax,%esi
0x4004e0 <+32>: lea    -0x1(%rsi),%eax
0x4004e3 <+35>: mov    $0x400680,%edi
0x4004e8 <+40>: mov    %eax,0x200b52(%rip)
0x4004ee <+46>: xor    %eax,%eax
0x4004f0 <+48>: callq  0x400470 <printf@plt>
0x4004f5 <+53>: mov    $0x1,%edi
0x4004fa <+58>: callq  0x4004a0 <sleep@plt>
0x4004ff <+63>: mov    0x200b3b(%rip),%esi
0x400505 <+69>: test   %esi,%esi
0x400507 <+71>: jg     0x4004e0 <main+32>
0x400509 <+73>: xor    %eax,%eax
0x40050b <+75>: add    $0x8,%rsp
0x40050f <+79>: retq
```

Figure 6.1. Example code in C (left) compiled to x86-64 assembly instructions (middle) produced the executable file that is dynamically linked to the `libc` library (right).

libraries and routines added by the compiler, such as initialization (pre-main code) and finalization (post-main code). This distinction is formalized by Definition 6.

Definition 6 (Visibility)**.** Given a program P with source code S, and a compiler C, the *visible* instructions of P are the instructions that were produced by C's code generator for statements that appear in S. Every other instruction required for the execution of P is an *invisible* instruction.

Example 6.2.1 (Visibility)**.** Figure 6.1 shows `countdown.c`, a C program whose function `main` invokes three functions that belong to the `libc` library: `strtol`, `printf` and `sleep`. This program is dynamically linked to the `libc` library file in Figure 6.1. The executable file contains the translated x86-64 assembly instructions of the `main` function, while the library file contains many functions, including the three functions invoked by this program. According to Definition 6, the instructions in the executable file that originated from the `main` function are *visible*, every other instruction is *invisible*.

The concept of visibility leads to the notion of *Visibility Ratio* of a program execution. Visibility is a static property of an instruction, established by its provenance. However, the visibility ratio of a program execution is determined dynamically, because it depends on the number of instructions from each category (visible or invisible) that is processed.

Group Data Structure	Group Instrumentation

G1 {
  instrs = [0x4004c0..0x4004cf]
  execs_count
}
	0x4004c0: **G1.execs_count++**
	          sub $0x8,%rsp
	0x4004c4: mov 0x8(%rsi),%rdi
	0x4004c8: mov $0xa,%edx
	0x4004cd: xor %esi,%esi
	0x4004cf: callq 0x400490 <strtol>

G2 {
  instrs = [0x4004d4..0x4004dc]
  execs_count
}
	0x4004d4: **G2.execs_count++**
	          test %eax,%eax
	0x4004d6: mov %eax,0x200b64(%rip)
	0x4004dc: jle 0x400509

G3 {
  instrs = [0x4004de..0x4004f0]
  execs_count
}
	0x4004de: **G3.execs_count++**
	          mov %eax,%esi
	0x4004e0: lea -0x1(%rsi),%eax
	0x4004e3: mov $0x400680,%edi
	0x4004e8: mov %eax,0x200b52(%rip)
	0x4004ee: xor %eax,%eax
	0x4004f0: callq 0x400470 <printf>

G4 {
  instrs = [0x4004f5..0x4004fa]
  execs_count
}
	0x4004f5: **G4.execs_count++**
	          mov $0x1,%edi
	0x4004fa: callq 0x4004a0 <sleep>

G5 {
  instrs = [0x4004ff..0x400507]
  execs_count
}
	0x4004ff: **G5.execs_count++**
	          mov 0x200b3b(%rip),%esi
	0x400505: test %esi,%esi
	0x400507: jg 0x4004e0

G6 {
  instrs = [0x4004e0..0x4004f0]
  execs_count
}
	0x4004e0: **G6.execs_count++**
	          lea -0x1(%rsi),%eax
	0x4004e3: mov $0x400680,%edi
	0x4004e8: mov %eax,0x200b52(%rip)
	0x4004ee: xor %eax,%eax
	0x4004f0: callq 0x400470 <printf>

G7 {
  instrs = [0x400509..0x40050f]
  execs_count
}
	0x400509: **G7.execs_count++**
	          xor %eax,%eax
	0x40050b: add $0x8,%rsp
	0x40050f: retq

Figure 6.2. (Left) data-structures used to represent groups of instructions: a list of instruction addresses, plus a counter. (Right) dynamic instrumentation, in boldface, that precedes the execution of a group of instructions.

Definition 7 (Visibility Ratio)**.** Let V be the number of *visible* and I be the number of *invisible* instructions executed during a run of a program P with workload W. The visibility ratio of (P, W) is $\frac{V}{V+I}$.

6.3 Analyzing Instruction Provenance

The measurement of the visibility ratio of a program execution requires instrumentation that counts how many times each instruction is executed. This instrumentation must be dynamic—it cannot be statically inserted into the code because a compiler only has access to visible instructions. Thus, INSTRGRIND, a tool built on top of VALGRIND [Nethercote and Seward, 2007], is used to perform the instrumentation. The choice for VALGRIND is due to our familiarity with it. Any other dynamic binary instrumentation framework could have been used instead, such as PIN [Luk et al., 2005] or QEMU [Bellard, 2005].

6.3.1 Instrumenting Groups of Instructions

Although every instruction in an execution must be accounted for, there is no need to instrument all of them. Instrumenting all the instructions would lead to an impractical runtime overhead. Instead, INSTRGRIND instruments groups of instructions that are executed as a unit. These groups, as described in Section 2.2, are obtained by INSTRGRIND the same way they are obtained by CFGGRIND when reconstructing control flow graphs. The groups found in programs instrumented with INSTRGRIND for this study contain, typically, six to seven instructions. Thus, grouping reduces the instrumentation overhead up to sevenfold.

Example 6.3.1. Figure 6.2 shows how the instructions observed during the execution of function `main`, seen in Example 6.2.1, are divided into groups. This division happens while instructions are executed, not statically. When a group is discovered, an auxiliary structure is created for it, as seen in the left column of Figure 6.2. This structure holds a counter, incremented via binary instrumentation, as seen in the right column of Figure 6.2 in boldface. Thus, whenever a group of instructions is visited by the program flow, its execution counter is incremented by one. An instruction may belong to more than one group. For instance, the operation at address *@0x4004e0* belongs to groups *G3* and *G6*.

6.3.2 Classifying Instructions

Instructions are classified into visible or invisible using as reference their location in memory. In order to classify instructions, INSTRGRIND uses a mapping that associates

```
/usr/lib64/libc-2.17.so:0x4c459a0:1376079
/usr/lib/valgrind/vgpreload_core-amd64-linux.so:0x4a24580:568
/usr/lib/valgrind/instrgrind-amd64-linux:0x58000150:1764106
/usr/lib64/ld-2.17.so:0x4000ad0:111936
/home/user/countdown:0x4004c0:402
```

Figure 6.3. Files mapping onto memory with range addresses specified as base address and size.

object files with the range of addresses that they cover, once loaded into memory. A range of addresses is defined by a base address and a size. As noted in Definition 6, instructions located in ranges belonging to source files are classified as visible and all other instructions are classified as invisible. Example 6.3.2 shows how this classification works in practice.

Example 6.3.2. Figure 6.3 shows the mapping that INSTRGRIND creates to analyze the execution of the `countdown` program seen in Example 6.2.1. Visible instructions start at the address `0x4004c0`, and cover the next 402 bytes (Line 5). Instructions that exist in VALGRIND's address space (Lines 2-3 of Fig. 6.3) are not counted because they are not part of the normal execution of the program. Instructions from `libc` (Line 1) and from `ld` (Line 4) are classified as invisible.

In some large computer applications part of the application code, which must be classified as visible, is offloaded to shared libraries. In such cases, an analyst must identify which address ranges in memory corresponds to visible or invisible instructions. For all the benchmarks analyzed in this study, the determination of visible and invisible address ranges is automatic because the files that compose the compiled program originate the instructions marked as visible.

6.4 Measuring Visibility Ratio

This section answers three research questions:

- **RQ1**: What is the visibility ratio of Spec Cpu2017's execution with reference inputs?
- **RQ2**: What is the visibility ratio in GNU CoreUtils' execution with its standard test inputs?
- **RQ3**: How code optimizations impact these ratios?

All the experiments were executed on a 16-core Intel(R) Xeon(R) E5-2630 at 2.40GHz with 16GB of RAM running Linux CentOS 7.5. Programs are compiled with gcc v7.3.1.

6.4.1 RQ1: Visibility ratio in Spec Cpu2017

Figure 6.4 shows the visibility ratio of instructions executed for the Spec Cpu2017 benchmark suite with reference inputs. Each benchmark was compiled with -O0 and

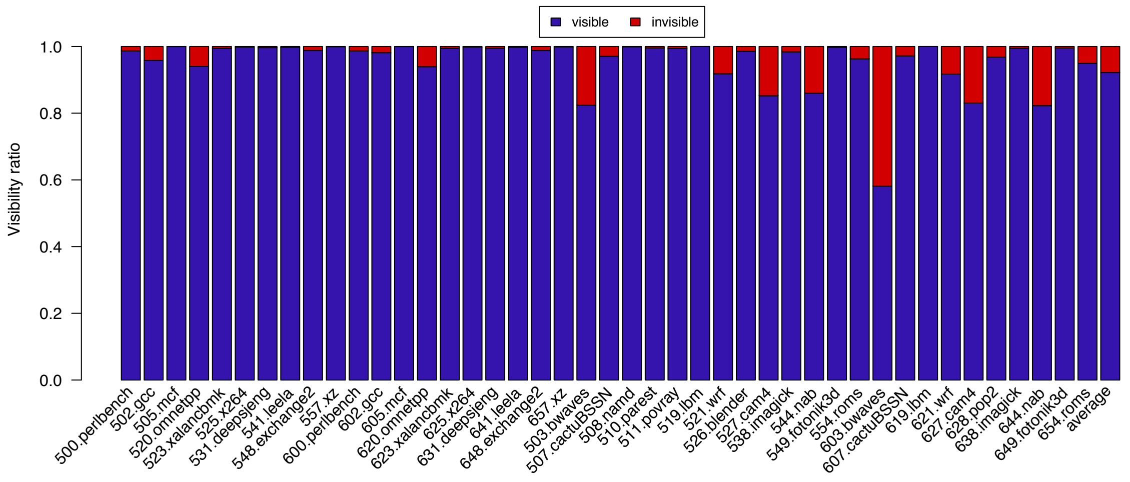


Figure 6.4. Visibility ratio of Spec Cpu2017 programs compiled using gcc with -O0, running with reference inputs.

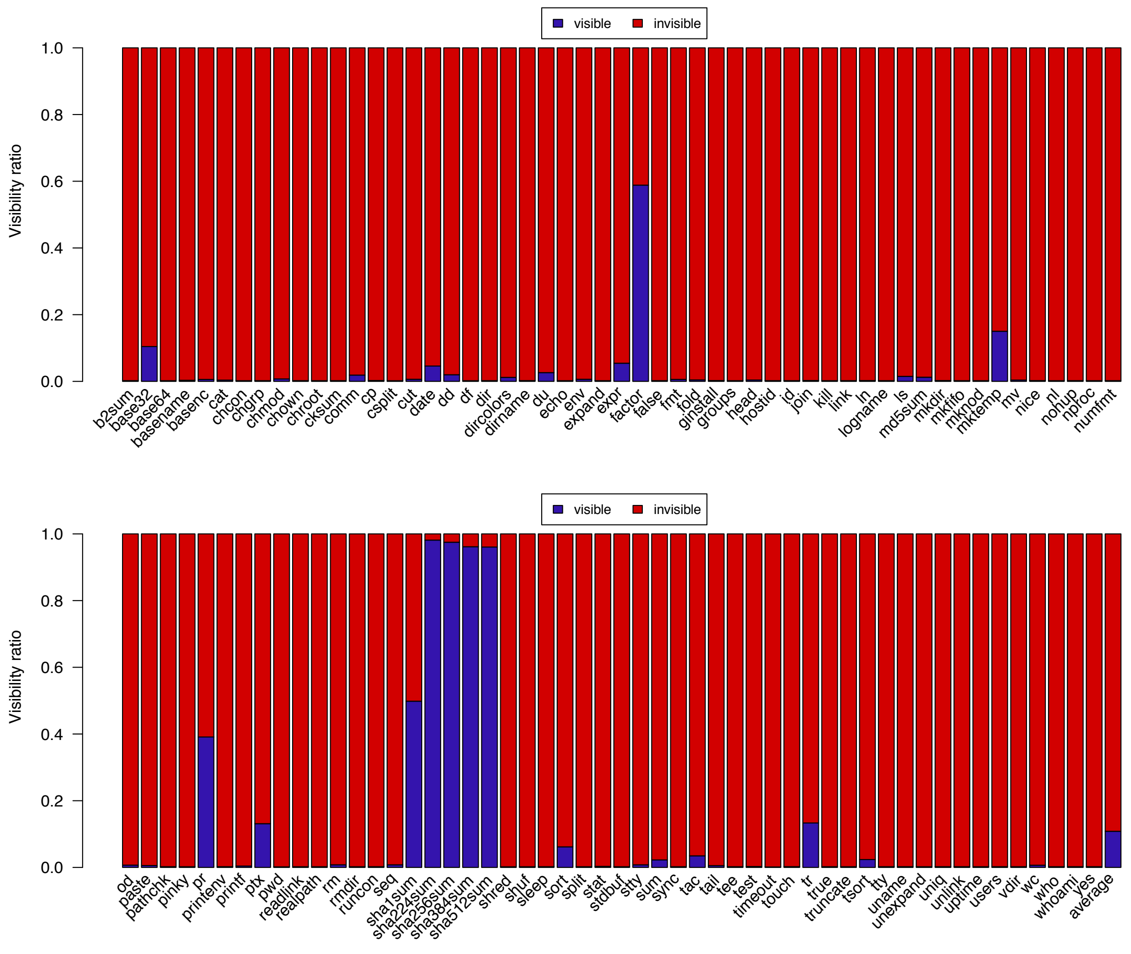


Figure 6.5. Visibility ratio of GNU CoreUtils programs compiled using gcc with -O0.

executed once using INSTRGRIND with its reference input. This experiment took approximately 9 days to complete for the entire Spec Cpu2017 suite.

Most of the instructions observed by INSTRGRIND are visible. For the 43 programs available in Spec Cpu2017, 37 of them had visibility ratio higher than 0.9; 5 more of them higher than 0.8. The only divergence was `603.bwaves`, with a visibility ratio of about 0.6. The average visibility ratio reported in the charts is computed by summing the number of visible instructions over all the benchmark executions and dividing by the total number of instructions executed by all the benchmarks in the suite. For Spec Cpu2017 this average is 0.922.

Most of the instructions observed in a typical run of Spec Cpu2017 are *visible*. This result confirms our expectations given that Spec Cpu2017 focuses on CPU performance and the curation of the benchmark suite must ensure that the entire code can be distributed under a SPEC license agreement. Thus, benchmark authors tend to avoid extensive library usage, so to avoid having to deal with the legal process of obtaining permission to use them.

6.4.2 RQ2: Visibility Ratio in GNU CoreUtils

Figure 6.5 shows the visibility ratio observed during the execution of GNU CoreUtils with the standard inputs available in its test suite. All programs were compiled with -O0. This test suite was modified to invoke each program, with its reference input, using INSTRGRIND. The entire suite was executed once. The numbers that follow represent the results for 105 of the 106 programs available in GNU CoreUtils. The exception is the *[* (open bracket) program that was not exercised by the GNU CoreUtils' test suite. The average visibility ratio is 0.108. A few calculation-oriented programs, such as `factor` and the cryptographic routines, break this tendency, as Figure 6.5 shows. However, these programs are exceptions within GNU CoreUtils. The conclusion of this experiment is that instructions processed in a typical run of GNU CoreUtils' programs are, in general, invisible.

6.4.3 RQ3: the Impact of Compiler Optimizations

When producing code for a program P, a compiler has an effect only on the source code of P; that is, on its visible instructions (Def. 6). Hence, it is natural to assume that the larger the visibility ratio of P, averaged across different executions, the larger the impact of compiler optimizations on P. Figures 6.6 and 6.7 confirm this intuition.

Figure 6.6 shows the speedups obtained by the different optimization levels of GCC when applied onto Spec Cpu2017. Results are the average of three executions. Deviations are insignificant; thus, not reported. In total, one run of all unoptimized programs for Spec Cpu2017 takes approximately 45 hours. The following average speedups (geometric mean) are observed for Spec Cpu2017: $-O1 = 2.277$x, $-O2 = 2.437$x and $-O3 = 2.463$x.

Figure 6.7 shows similar data, this time considering the programs from GNU CoreUtils. The test suite was invoked 25 times, each invocation taking approximately 4 minutes. Deviations are, again, negligible; hence, omitted. Speedups are too low to be considered statistically significant[1]. On average (geomean), GCC -O3 delivers a speedup of 1.011 across the GNU CoreUtils' programs. From this last experiment, we conclude that the visibility ratio strongly determines the effect of compiler optimizations onto programs.

The mild effect of optimizations onto GNU CoreUtils, and its important impact onto Spec Cpu2017 is due to a simple observation: the visibility ratio disparity is maintained in face of different optimization levels. Figures 6.8 and 6.9 show the visibility ratio for optimized versions of the Spec Cpu2017's and GNU CoreUtils' programs, respectively. The optimized version (compiled with -O2) has less visible instructions than the non-optimized (compiled with -O0): a visibility ratio of 0.701 against 0.108. However, this difference is small to be of consequence. Optimizations bear a more noticeable effect onto Spec Cpu2017. Programs optimized with -O2 show a visibility ratio of 0.701, against 0.922 in the unoptimized programs. This difference is natural: Spec Cpu2017 has a larger code base, and a much larger visibility ratio than GNU CoreUtils; therefore, the compiler has more opportunities to optimize code.

6.4.4 Reflection on the Results

Are these visibility ratios an experimental confirmation of the expectation of compiler developers or are they new information for some of them? An informal consultation from July of 2018 indicated that there was no consensus about visibility. At that time we asked ten researchers what proportion of a program in Spec Cpu2006 is visible[2]. These engineers and academics have experience with compilers: four of them have been

[1]Because the runtime differences produced by the three optimization levels were small, this experiment averages 25 samples, in contrast to the one used to produce the data in Figure 6.6, which averages 3.

[2]We chose Spec Cpu2006 because this suite was more well-known than Spec Cpu2017 at the time when the questions were sent.

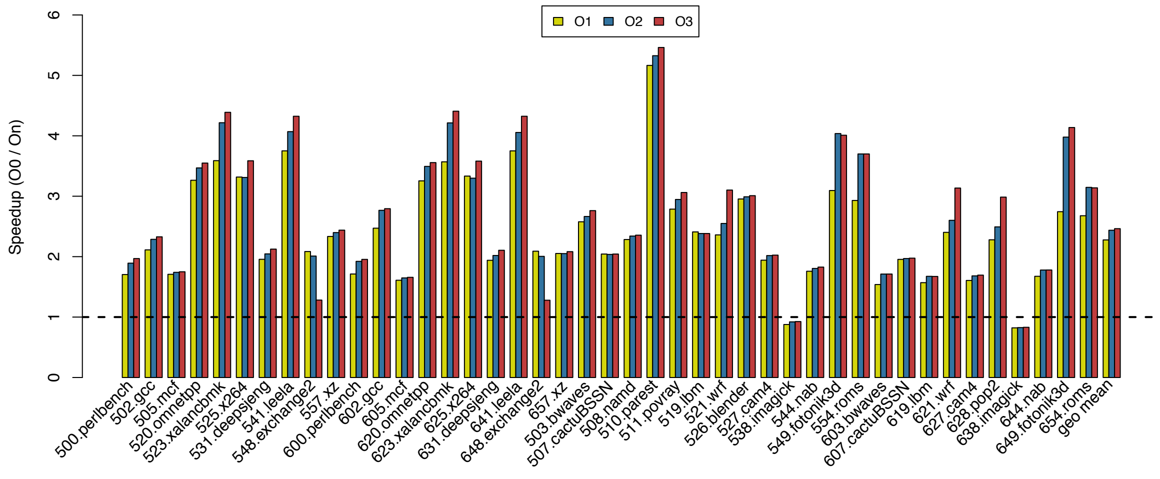


Figure 6.6. Speedup produced by different optimization levels of GCC, when applied onto the Spec Cpu2017 programs.

part of CGO's program committee. The other six were professionals actively working with the following compilers, within different companies: LLVM (Apple, Facebook and ARM), Visual Studio (Microsoft), JavaScript Core (STMicroelectronics), and V8 (Google). We quote below the e-mail sent to the different researchers and professionals, asking them about their feeling on the proportion of invisible instructions:

> We are working on a project to count the number of "visible" and "invisible instructions" in a program. A visible instruction is, for instance, the instruction that we can see in the LLVM IR. An invisible instruction is an operation that we cannot see, for instance, instructions inserted to implement the ABI, or that are part of some external library. We have already counted this ratio in several benchmarks. I've decided to write to the compiler experts that I know, asking them what is the proportion between:
>
> - **VS:** Stores in the source code of a C program, and
> - **TS:** all the stores produced during the execution of the program.

Answers came out all over the spectrum. The average was **VS/TS** = 0.575. This value is substantially lower than the average (arithmetic mean) observed for Spec Cpu2017: 0.92. Variance in the answers was high: the standard deviation was 0.31. Only two researchers gave us answers greater than 0.9. Two answers were less than 0.3. The highest ratio someone guessed was 0.95; the lowest was 0.2.

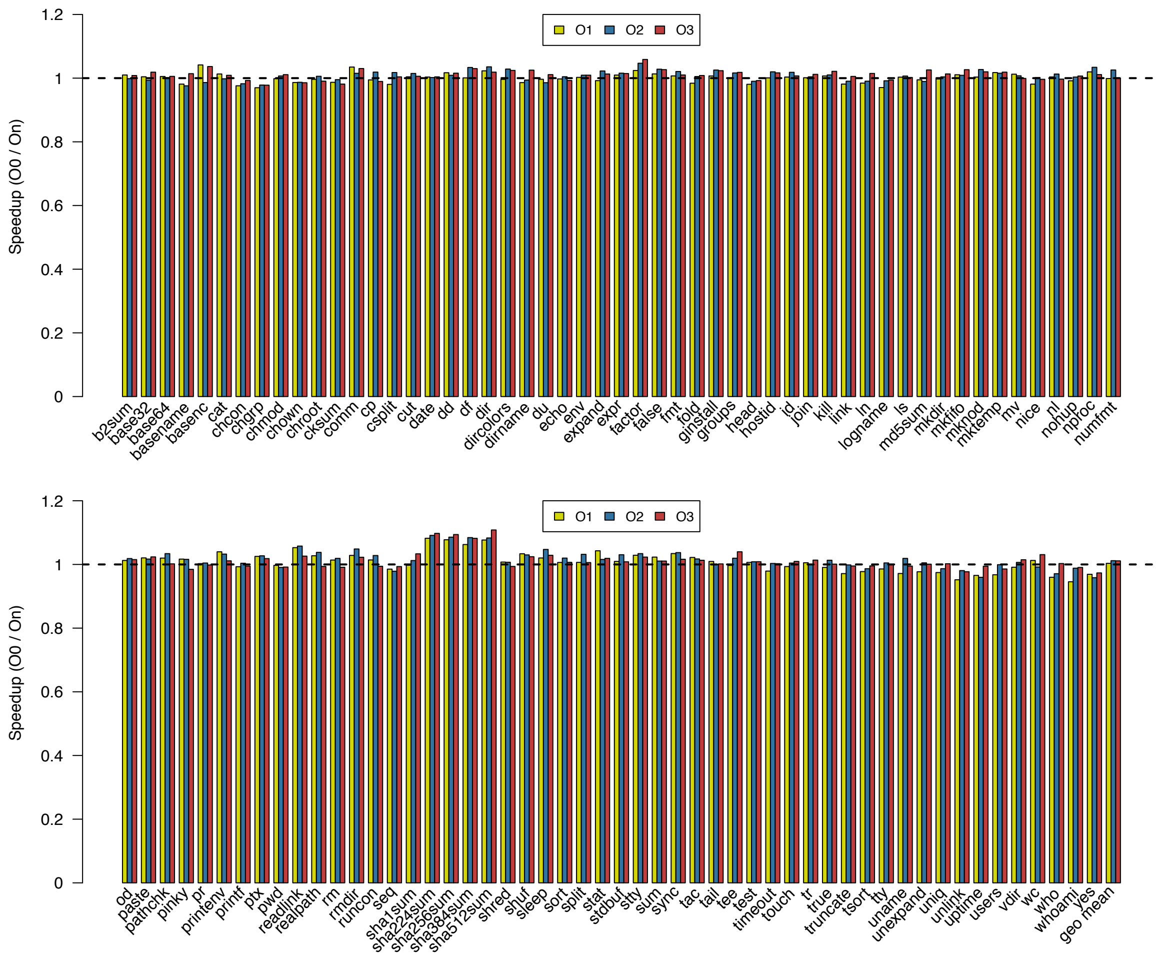


Figure 6.7. Speedup produced by different optimization levels of gcc, when applied onto the GNU CoreUtils programs.

6.5 Conclusion

This case study has measured the visibility ratio of instructions executed by benchmarks in Spec Cpu2017, when run with reference inputs on a Linux-based processor with the x86 architecture. This study reveals that most of the executed instructions are visible—they originate from code available to the compiler. In contrast, most of the instructions executed in a typical run of programs in GNU CoreUtils are invisible. The origin of invisible instructions are libraries that these applications invoke during their execution. These findings confirm the intuition of some developers but will be surprising to others. Visibility measurements are important because architectures and compilers are often evaluated with Spec Cpu2017, which shows high visibility ratio; however, they are applied onto applications like GNU CoreUtils, which have low

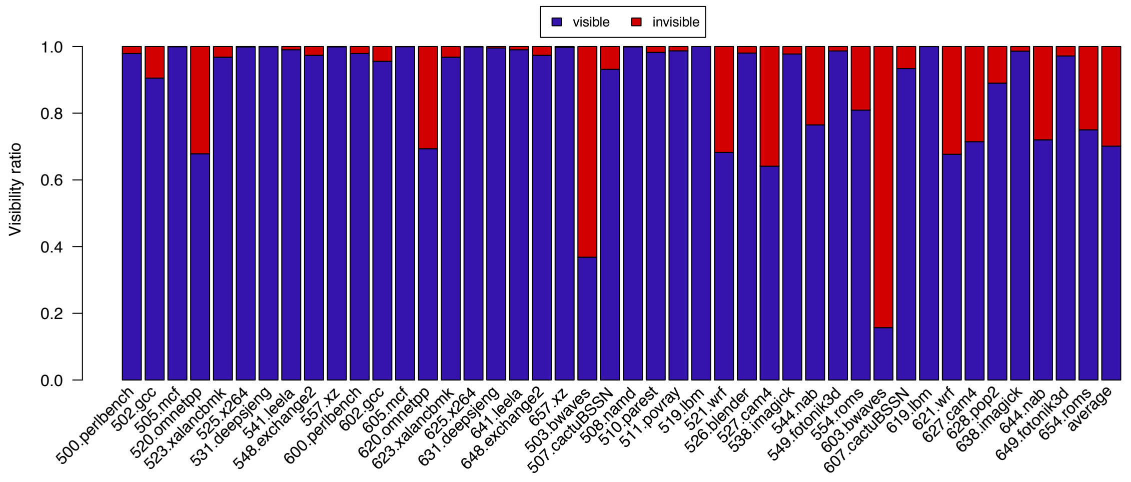


Figure 6.8. Visibility ratio of Spec Cpu2017 programs compiled using gcc with -O2, running with reference inputs.

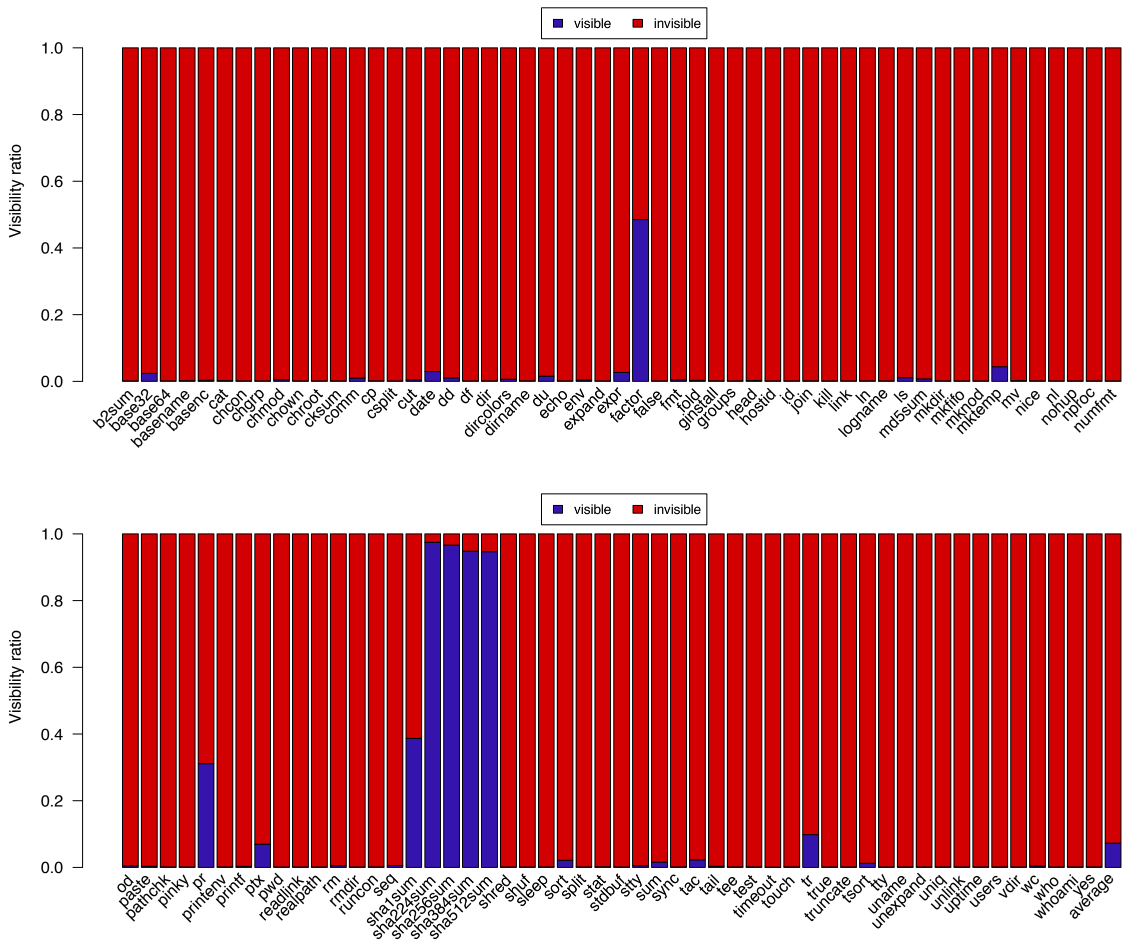


Figure 6.9. Visibility ratio of GNU CoreUtils programs compiled using gcc with -O2.

visibility ratio. Thus caution should be exercised when projecting the effects of code transformations observed in Spec Cpu2017 onto other applications.

Chapter 7

Related Work

This chapter visits related works on three main fronts: on reconstruction of control flow graphs (Section 7.1), on instruction categorization (Section 7.2 and on binary instrumentation (Section 7.3). The first topic discusses how control flow graphs can be obtained from a binary, either using a dynamic approach — by running the code and following its execution flow — or using a static approach — by analyzing its instructions and inferring the possible execution paths. An in-depth comparison of the three dynamic reconstructions tools that were used to evaluate our solution can be found in Chapter 3. The second topic gives an overview of related works on instructions Categorization, mainly comparing the visible/invisible relation as tackled in our case study (Chapter 6). The last topic addresses how binary programs can be instrumented for profiling purposes. This instrumentation can be performed by frameworks using different approaches. A static profiler modifies the internals of a binary program: it can include, modify or remove its instructions to perform the profiling. A dynamic profiler in the other hand does not change a program's structure, but emulates its runtime environment to allow introspection of its instruction, to observe or change its behavior during execution time. A hybrid profiler combines both approaches.

7.1 On Control Flow Graphs

Related research includes the reconstruction of CFGs for the analysis of binary programs; two alternative approaches for dynamic reconstruction of CFGs; the reconstruction of CFGs in dynamic program slicing, which is typically done through program instrumentation; and several approaches for the static reconstruction of CFGs. This section reviews these related topics.

Dynamic Reconstruction of CFGs. Dynamic analyses of binary programs have been used to detect malware [Moser et al., 2007], to improve test coverage [Godefroid, 2014], to de-obfuscate programs [Zhen, 2014], to locate out-of-bounds memory accesses [Kimball, 2013], and to detect memory dependences [Gruber et al., 2019]. All these uses of dynamic analysis of binaries had to reconstruct the CFG in order to perform the analysis. However, the description of these analyses do not detail the method used to reconstruct the CFG. Therefore, it is difficult to discern the advantages and shortcomings of the CFG reconstruction in each of them. Moreover, none of them provide a publicly available artifact that would allow for an evaluation or comparison with the approach described in this thesis. To the best of our knowledge, only three tools focus on dynamic CFG reconstruction: FXE by Xu et al. [2009], bfTrace by Gruber et al. [2019] and PinPlay by Yount et al. [2015]. In this thesis, we have experimented with the last two of them.

PinPlay and bfTrace are the only implementations of dynamic CFG reconstruction that are available for public scrutiny. The experimental results presented in Chapter 5 indicate that the techniques described in this thesis improve on both tools, in terms of efficiency and completeness. Indeed, many of the design decisions in the development of CFGgrind were motivated by the possibility to use it to augment the precision of DynInst, a static CFG reconstructor. Integration with static analyzers is not a driving force behind neither PinPlay's implementation nor bfTrace's; hence, such possibility is not discussed in the papers that introduce those tools.

FXE combines static and dynamic analysis [Xu et al., 2009]. Like bfTrace, FXE interprets a program using Qemu [Bellard, 2005], whereas PinPlay uses Pin [Luk et al., 2005], and CFGgrind uses valgrind [Nethercote and Seward, 2007]. However, instead of simply interpreting each instruction with the state produced by the normal execution of the program, FXE tries to *force* the execution of each branch that it finds while building the program's CFG; hence, a CFG produced by FXE does not correspond to a dynamic slice of a program's execution. In other words, upon finding a phantom node, FXE saves the current state at that program point, and marks it as active. While there are active branches, FXE backtracks, and re-evaluates the branch condition, forcing the visit of the phantom block. Although elegant, this approach has a much higher runtime complexity. Therefore, to keep reconstruction practical, FXE foregoes the analysis of library code, which is a serious limitation for its practical use. According to Xu et al. [2009]:

> "When FXE detects a function call pointing to external code, it forces the execution to immediately return to the call site and continue along the

fall-through." [Xu et al., 2009]

Dynamic Program Slicing. Much of the literature on the dynamic reconstruction of CFGs was influenced by the notion of *Dynamic Program Slicing*. This concept was introduced by Korel and Laski [1988]. Yet, the formulation of Agrawal et al. [1993], introduced five years later, seems to be the most standard today. If P is a program, $I \in P$ is an instruction of P and ι is an input of P, then the dynamic slice S is a subset of P's instructions that, when executed, always causes the interpretation of I as in P. Dynamic program slicing has been the focus of much research, and remains a trendy topic even today [Hu et al., 2018; Lin et al., 2018].

A survey of the literature on Dynamic Slicing reveals that most work on the area relies on code instrumentation. In contrast, CFGGrind, PinPLAYand bfTrace rely on program emulation. Code instrumentation has a key advantage: it simplifies the task of linking runtime events with source code. On the other hand, it has a major disadvantage: it requires the availability of the source code; hence, it is unable to handle library code.

Static Reconstruction of CFGs. Most papers about the analysis of binary code deal with the static reconstruction of control flow graphs. Seminal work on binary code analysis, such as Cifuentes' [Cifuentes and Gough, 1995], Gao's [Gao et al., 2008] and Balakrishnan's [Balakrishnan and Reps, 2004], used static reconstruction of CFG. More recent techniques to reconstruct CFGs also use static reconstruction. For example, the binary optimizers that appeared in 2019, such as BOLT [Panchenko et al., 2019] and Janus [Zhou and Jones, 2019]. As discussed in Chapter 1, the static methodology has advantages and disadvantages over its dynamic counterpart. This thesis presents a dynamic CFG reconstruction technique that improved the precision of DynInst, a static CFG reconstructor, created by Meng and Miller [2016]. To the best of our knowledge, DynInst is the most precise static CFG reconstructor to date.

7.2 On Instructions Categorization

The categorization of dynamic instances of program instructions is not a novel endeavor. In their classic textbook, Patterson and Hennessy's discuss the dynamic count of opcodes executed for the twelve integer benchmarks in the Spec Cpu2006 benchmark suite [Patterson and Hennessy, 2013]. Similarly, Guthaus et al. [2001] present a comprehensive analysis of the distribution of instructions across MiBench and in the Spec Cpu2000 benchmark suite. In this analysis, instructions are split into four major types: control, memory, integer arithmetics and floating point arithmetics. However, previous work has not distinguished visible from invisible instructions.

Historically, SPEC CPU has been the subject of several statistical studies [Nair and John, 2008; Phansalkar et al., 2007b,a]. Since the release of Spec Cpu2017, numerous research groups have already performed analyses on it [Bucek et al., 2018; Limaye and Adegbija, 2018; Panda et al., 2018; Singh and Awasthi, 2019; Wu et al., 2018]. These studies focus on some particular aspect of that benchmark suite, such as potential redundancies [Panda et al., 2018], code coverage [Limaye and Adegbija, 2018; Wu et al., 2018] or memory characterization [Singh and Awasthi, 2019]. None of these papers analyze the visibility ratio in Spec Cpu2017, nor the impact of compiler optimizations on these benchmarks. Amaral et al. [2018] curated the Alberta Workloads for the Spec Cpu2017 Benchmark Suite. Based on their analysis of how program behavior varies with workloads, the expectation is that the visibility ratios will not change significantly when the benchmarks are run with different inputs.

Leobas et al. [2018] classify an instruction as visible if it corresponds directly to operations in the LLVM representation of the program. Instructions inserted by the compiler, such as loads and stores used to spill variables, are classified as invisible. Their goal is to estimate the overhead caused by the code that the compiler creates outside of a programmer's control. Their definition is in contrast to Definition 6, in which the category of an instruction depends on its location in memory. Differently than the work of Leobas et al. [2018], this case study supports the assessment of the impact that compiler transformations might have on programs.

7.3 On Binary Instrumentation

Instrumentation is a powerful technique to profile, monitor and even modify programs [Aho et al., 2006; Appel and Palsberg, 2002]. It is specially useful when applied directly to binary programs, where the source code is not available or may be protected. The execution of a program can be inspected for numerous applications. For code coverage, profilers are useful to verify if test cases covered most of a program's execution paths [Eustace and Srivastava, 1995; Tikir and Hollingsworth, 2002]. For performance, profilers aid in the discovery of program's hotspots that can be targeted for potential compiler optimizations [Nethercote et al., 2006]. For debugging, profilers can check for programming errors, such as bugs in memory management, race conditions in multi-threads programs,, detailed simulations in caching latencies, among others [Seward and Nethercote, 2005; Nethercote et al., 2006; Nethercote and Seward, 2007]. For security, profilers can help identify possible security problems [Tymburibá et al., 2016], and even actively prevent attacks from affecting the runtime environment [Kiriansky et al., 2002]. For malware analysis, profilers can search for signature to identify malicious infections in programs [Brumley et al., 2006; Pewny et al., 2015]. Binary instrumentation brings endless possibilities for program analysis.

These applications are supported by binary instrumentation frameworks. Some of them can work statically, i.e., can infer properties without actually executing a program. Others can work dynamically, i.e., can monitor its execution environment extracting information or actively performing actions at runtime. There are also hybrid frameworks that combine designs of these two. These frameworks, with its advantages and disadvantages, are explored as follows.

Static Binary Frameworks. Static binary frameworks enable verification or transformation, such as instrumentation or code rewriting, by analyzing or manipulating the instructions of a program without the need of executing it. This avoids the cost of running such program with added instrumentation or code manipulation [Meng and Miller, 2016]. Because of that, static analysis tends to be faster, albeit less accurate. This enables more complex analysis, such as whole program path coverage. However, assembly constructs with indirections, such as indirect calls or jumps, hurt the precision of the static analysis. Stripped binaries, i.e., binaries without debugging symbols, are specially difficult to analyze since it is hard to pinpoint exactly where functions begin and end. Thus, static analysis tends to be conservative: it may raise many false positives due to the unavailability of runtime information [Shoshitaishvili et al., 2016]. This has a direct impact on the accuracy of recovered control flow graph, as can be observed in the discussion in Section 5.5.

Despite its shortcomings, many frameworks are available for writing custom static analysis tools for binaries [Eustace and Srivastava, 1995; Laurenzano et al., 2010; Brumley et al., 2011]. They can be used to recover high-level structural information, such as complex control and data-flow, dependency analyses, optimizations and security.

Dynamic Binary Frameworks. Dynamic binary frameworks are powerful when information about the runtime environment can not be inferred statically. The execution of the code is required for a specific action to take place in order to be observed or manipulated. For example, it is easy for a dynamic analysis to follow an indirect call or jump since its target address is calculated during runtime. However, it is hard to tell all of its potential targets. Hence, dynamic analysis is limited to the code sections that are seen during the execution, while static analysis may have a global view of the entire program. Despite this limitations, dynamic analysis are critical for analyzing programs with signal handlers, calls to shared libraries, dynamically generated code or self-modifying code.

Dynamic binary frameworks support applying instrumentation or transformation into a code while it is executing. There are many frameworks available to support dynamic analysis, where some of the most popular are valgrind [Nethercote and Seward, 2007], Pin [Luk et al., 2005], DynamoRIO [Bruening et al., 2003], and Qemu [Bellard, 2005]. These frameworks support building tools within its infrastructure to perform dynamic analyses, such as to find parallelization opportunities [Bach et al., 2010; Gruber et al., 2019], memory errors [Seward and Nethercote, 2005; Bruening and Zhao, 2011], and to protected against control-flow based attacks [Kiriansky et al., 2002].

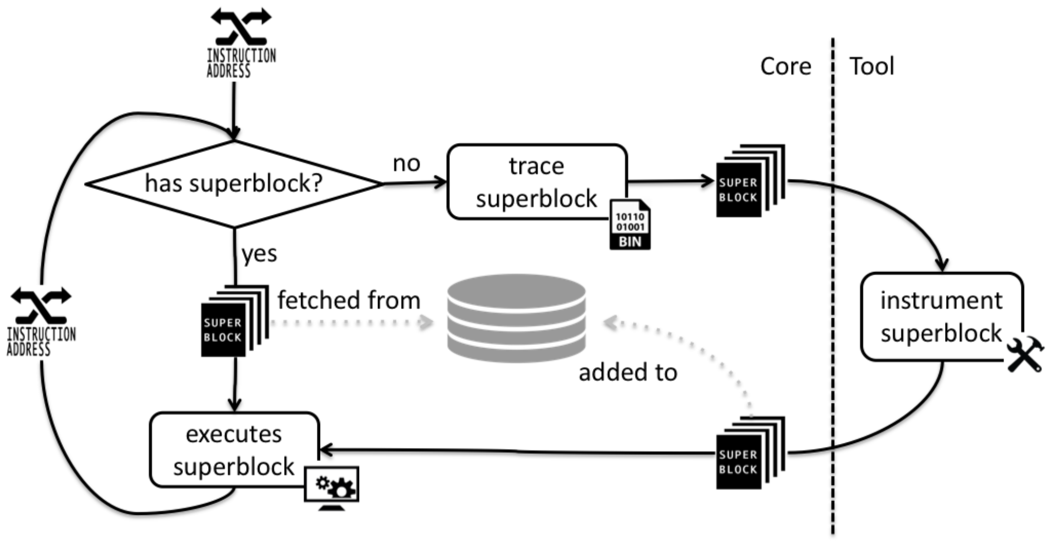


Figure 7.1. valgrind's execution workflow.

Both CFGgrind (Chapter 4) and instrgrind (Chapter 6) were built using valgrind's infrastructure [Nethercote and Seward, 2007]. valgrind's architecture is divided into two main parts: the core that provides low-level support for instrumentation and useful services, and the tool that actually performs the instrumentation. Figure 7.1 shows a diagram of valgrind's execution workflow. At the core, valgrind first identifies a sequence of instructions that can be executed as a unit. If this sequence is viewed for the first time, then they are disassembled into a custom IR called VEX to form a superblock. This superblock is thus passed to the tool to perform the instrumentation. Afterwards, the superblock is returned to valgrind's core to be stored in a cache for fast retrieval. Finally, this superblock is translated back to the host's assembly instruction set using a just in time compiler for execution. In case of a previously seen sequence of instructions, VALGRIND recovers its cached instrumented superblock and proceed with its execution.

Hybrid Binary Frameworks. A hybrid binary framework combines aspects of static and dynamic frameworks to achieve a more complete and powerful tool. The weaknesses of both the static and dynamic approaches are mitigated by building a hybrid tool on its strengths, which are complementary to each other.

BitBlaze [Song et al., 2008] and angr [Shoshitaishvili et al., 2016] are two examples of frameworks that support static analysis and dynamic analysis via symbolic execution [Majumdar and Sen, 2007]. The former was built specifically for security applications, whereas the latter can be used to detect vulnerabilities and recover control flow graphs. Angr supports two distinct strategies to reconstruct control flow graphs: *CFGFast* which uses static analysis and *CFGEmulated* which uses dynamic analysis. The static analysis suffers from the same imprecisions, while the dynamic analysis takes prohibitively long times, even for small and simple programs.

DynInst [Meng and Miller, 2016] is another hybrid framework that supports static and dynamic analysis. It allows for dynamic binary rewriting and can be used for profiling and performance analysis. It has a builtin static analysis algorithm to recover control flow graphs that was used in Section 5.5 when compared against CFGgrind. However, there are no similar algorithms using a dynamic approach. Albeit one could use the foundations elicited in Chapter 4 on top of DynInst to build CFGs dynamically. Janus [Zhou and Jones, 2019] is also a hybrid framework. It supports complex modifications of the original program during runtime. It uses DynamoRIO [Bruening et al., 2003] to perform binary parallelization while the code is executing.

7.4 Conclusion

This chapter reviewed related works in respect to three subjects. First, CFGgrind is compared against other works on its capability of reconstructing control flow graphs in Section 7.1. Then, the approached employed by INSTRGRIND to classify the visibility of instructions is compared against other works in Section 7.2. Finally, a discussion on how binary frameworks can be used to support different analysis, such as the instrumentation performed by these two tools, was provided in Section 7.3.

Chapter 8

Conclusion

This thesis provided evidence that the dynamic reconstruction of CFGs from the execution of a program can result in more precise CFGs than the ones obtained solely via static analyses. However, to correctly reconstruct CFGs in this fashion, it was necessary to revisit the definition of CFGs to account for phantom nodes and signals. New algorithms had to be engineered into an efficient and robust tool. This tool, CFGGRIND (`https://github.com/rimsa/CFGgrind`), was used to analyze several large programs. The experimental evaluation determined that CFGGRIND outperforms, in terms of precision and efficiency, two other tools that support dynamic CFG reconstruction: DCFG and BFTRACE. CFGGRIND also improves the precision of DYNINST, a state-of-the-art static binary analyzer by augmenting it with the ability to handle binaries stripped of debugging information. The experimental results also evidenced that typical data sets distributed with benchmarks already let a dynamic reconstructor completely recover a substantial part of all the active functions in large programs. Although intrinsically dependent on program inputs, this complete recovery has been observed in a large number of programs, including SPEC CPU2017 and CBENCH.

Future work that stems from this thesis includes the use of CFGGRIND in different scenarios. First, CFGGRIND's ability to track non-aligned and overlapping instructions in the binary representation of a program gives can be useful to reconstruct return-oriented programming attacks [Shacham, 2007], even when they are built via Checkoway et al. [2010]'s approach based on indirect jumps. Second, CFGGRIND's exact profiler is likely to give binary optimizers, such as BOLT [Panchenko et al., 2019], more information to improve the instruction layout of programs. The performance improvements that can be derived from this extra information remains to be evaluated. Finally, CFGGRIND's dynamic approach can also be useful for the recovery of the control flow of programs obfuscated with control flow flattening [Blazy and Trieu, 2016], the nemesis of static deobfuscation. How much information can be recovered via CFGgrind when it is used to analyze obfuscated programs is an open question.

Bibliography

Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 246--256, New York, NY, USA. ACM.

Agrawal, R., Imielinski, T., and Swami, A. N. (1993). Mining association rules between sets of items in large databases. In *SIGMOD Conference*, pages 207–216, New York, NY, USA. ACM.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Boston, Massachusetts, USA.

Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, 5:1--19.

Alvarez-Perez, D. (2017). In depth analysis of malware exploiting cve-2017-11826. https://www.gradiant.org/noticia/analysis-malware-cve-2017/.

Amaral, J. N., Borin, E., Ashley, D., Benedicto, C., Colp, E., Hoffman, J. H. S., Karpoff, M., Ochoa, E., Redshaw, M., and Rodrigues, R. E. (2018). The alberta workloads for the SPEC CPU 2017 benchmark suite. In *ISPASS*, pages 159–168, Belfast, Northern Ireland.

Appel, A. W. and Palsberg, J. (2002). *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition.

Bach, M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., Luk, C., Lyons, G., Patil, H., and Tal, A. (2010). Analyzing parallel programs with pin. *Computer*, 43(3):34–41.

Balakrishnan, G. and Reps, T. W. (2004). Analyzing memory accesses in x86 executables. In *Compiler Construction (CC)*, pages 5--23, Berlin, Germany. Springer.

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *ATEC*, page 41, USA. USENIX.

Bernat, A. R. and Miller, B. P. (2012). Structured binary editing with a CFG transformation algebra. In *Working Conference on Reverse Engineering*, pages 9–18, New York, NY, USA. ACM. ISSN 2375-5369.

Blazy, S. and Trieu, A. (2016). Formal verification of control-flow graph flattening. In *CPP*, page 176–187, New York, NY, USA. ACM.

Bruening, D., Garnett, T., and Amarasinghe, S. (2003). An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, page 265–275, USA. IEEE Computer Society.

Bruening, D. and Zhao, Q. (2011). Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, page 213–223, USA. IEEE Computer Society.

Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. (2011). Bap: A binary analysis platform. In Gopalakrishnan, G. and Qadeer, S., editors, *Computer Aided Verification*, pages 463--469, Berlin, Heidelberg. Springer Berlin Heidelberg.

Brumley, D., Newsome, J., Song, D., Hao Wang, and Somesh Jha (2006). Towards automatic generation of vulnerability-based signatures. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 15 pp.–16.

Bruschi, D., Cavallaro, L., and Lanzi, A. (2007). Static analysis on x86 executables for preventing automatic mimicry attacks. In M. Hämmerli, B. and Sommer, R., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 213--230, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bucek, J., Lange, K.-D., and v. Kistowski, J. (2018). SPEC CPU2017: Next-generation compute benchmark. In *ICPE*, page 41–42, New York, NY, USA. ACM.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *CCS*, pages 1–14. ACM.

Cifuentes, C. and Gough, K. J. (1995). Decompilation of binary programs. *Softw. Pract. Exper.*, 25(7):811--829.

Dixit, K. M. (1991). The SPEC benchmarks. *Parallel Computing*, 17(10-11):1195--1209.

Dixit, K. M. (1993). Overview of the SPEC benchmarks. In Gray, J., editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*, pages 489--521. Morgan Kaufmann.

Eagle, C. (2011). *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA. ISBN 1593272898, 9781593272890.

Eustace, A. and Srivastava, A. (1995). Atom: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, page 25, USA. USENIX Association.

Fraser, G. and Arcuri, A. (2011). Evosuite: Automatic test suite generation for object-oriented software. In *ESEC/FSE*, pages 416--419, New York, NY, USA. ACM.

Gao, D., Reiter, M. K., and Song, D. (2008). Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security (ICICS)*, pages 238--255, Berlin, Heidelberg. Springer-Verlag.

Godefroid, P. (2014). Micro execution. In *International Conference on Software Engineering (ICSE)*, pages 539--549, New York, NY, USA. ACM.

Gorelik, M. (2018). [critical alert] cve-2018-4990 acrobat reader dc double-free vulnerability. `http://blog.morphisec.com/critical-alert-cve-2018-4990-acrobat-reader-dc-double-free-vulnerability`.

Gruber, F., Selva, M., Sampaio, D., Guillon, C., Moynault, A., Pouchet, L.-N., and Rastello, F. (2019). Data-flow/dependence profiling for structured transformations. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 173--185, New York, NY, USA. ACM.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). Mibench: A free, commercially representative embedded benchmark suite. In *WWC-4*, pages 3--14. IEEE.

Hu, Y., Zhang, Y., Li, J., Wang, H., Li, B., and Gu, D. (2018). BinMatch: A semantics-based hybrid approach on binary code clone analysis. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 104–114, Madrid, Spain. IEEE.

Kästner, D. and Wilhelm, S. (2002). Generic control flow reconstruction from assembly code. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/SCOPES '02, pages 46--55, New York, NY, USA. ACM.

Kimball, W. B. (2013). *A Formal Approach to Vulnerability Discovery in Binary Programs*. PhD dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH, USA.

Kinder, J. and Veith, H. (2008). Jakstab: A static analysis platform for binaries. In *Computer Aided Verification (CAV)*, pages 423--427, Berlin, Heidelberg. Springer-Verlag.

Kiriansky, V., Bruening, D., and Amarasinghe, S. P. (2002). Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, page 191–206, USA. USENIX Association.

KleinOsowski, A. J. and Lilja, D. J. (2002). MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *IEEE Computer Architecture Letters*, 1(1):7–7.

Korel, B. and Laski, J. (1988). Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155--163.

Laurenzano, M. A., Tikir, M. M., Carrington, L., and Snavely, A. (2010). Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 175–183.

Lemos, O. A. L., Ferrari, F. C., Masiero, P. C., and Lopes, C. V. (2006). Testing aspect-oriented programming pointcut descriptors. In *Workshop on Testing Aspect-Oriented Programs (WTAOP)*, pages 33--38, New York, NY, USA. ACM.

Leobas, G. V., Guimarães, B. C. F., and Pereira, F. M. Q. (2018). More than meets the eye: Invisible instructions. In *SBLP*, page 27–34, New York, NY, USA. Association for Computing Machinery.

Limaye, A. and Adegbija, T. (2018). A workload characterization of the SPEC CPU2017 benchmark suite. In *ISPASS*, pages 149--158, New York, NY, USA. IEEE.

Lin, Y., Sun, J., Tran, L., Bai, G., Wang, H., and Dong, J. (2018). Break the dead end of dynamic slicing: Localizing data and control omission bug. In *Automated Software Engineering (ASE)*, pages 509--519, New York, NY, USA. ACM.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, page 190–200, New York, NY, USA. ACM.

Majumdar, R. and Sen, K. (2007). Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, page 416–426, USA. IEEE Computer Society.

Meng, X. and Miller, B. P. (2016). Binary code is not easy. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 24--35, New York, NY, USA. ACM.

Moser, A., Kruegel, C., and Kirda, E. (2007). Exploring multiple execution paths for malware analysis. In *SP*, pages 231--245, Washington, DC, USA. IEEE Computer Society.

Nair, A. A. and John, L. K. (2008). Simulation points for SPEC CPU 2006. In *ICCD*, pages 397–403. IEEE.

Nethercote, N. and Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 89--100, New York, NY, USA. ACM.

Nethercote, N., Walsh, R., and Fitzhardinge, J. (2006). "building workload characterization tools with valgrind". In *2006 IEEE International Symposium on Workload Characterization*, pages 2–2.

Panchenko, M., Auler, R., Nell, B., and Ottoni, G. (2019). BOLT: A practical binary optimizer for data centers and beyond. In *Code Generation and Optimization (CGO)*, pages 2--14, Piscataway, NJ, USA. IEEE Press.

Panda, R., Song, S., Dean, J., and John, L. K. (2018). Wait of a decade: Did SPEC CPU 2017 broaden the performance horizon? In *HPCA*, pages 271–282, New York, NY, USA. IEEE.

Patterson, D. A. and Hennessy, J. L. (2013). *Computer Organization and Design MIPS Edition: The Hardware/Software Interface.* Newnes.

Pewny, J., Garmany, B., Gawlik, R., Rossow, C., and Holz, T. (2015). Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724.

Phansalkar, A., Joshi, A., and John, L. K. (2007a). Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *ISCA*, page 412–423, New York, NY, USA. ACM.

Phansalkar, A., Joshi, A., and John, L. K. (2007b). Subsetting the SPEC CPU2006 benchmark suite. *SIGARCH Comput. Archit. News*, 35(1):69–76. ISSN 0163-5964.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(1):358--366.

Schwarz, B., Debray, S., and Andrews, G. (2002). Disassembly of executable code revisited. In *WCRE*, pages 45--, Washington, DC, USA. IEEE Computer Society.

Seebug, M. (2018). Tenda ac15 router - unauthenticated remote code execution(cve-2018-5767). https://vulners.com/seebug/SSV:97161.

Seward, J. (2019). The valgrind manual. valgrind.org/docs/manual.

Seward, J. and Nethercote, N. (2005). Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 2, USA. USENIX Association.

Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*, pages 552--561. ACM.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. (2016). Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157.

Singh, S. and Awasthi, M. (2019). Memory centric characterization and analysis of spec cpu2017 suite. In *ICPE*, page 285–292, New York, NY, USA. ACM.

Sites, R. L., Chernoff, A., Kirk, M. B., Marks, M. P., and Robinson, S. G. (1993). Binary translation. *Commun. ACM*, 36(2):69--81.

Smithson, M., Elwazeer, K., Anand, K., Kotha, A., and Barua, R. (2013). Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Working Conference on Reverse Engineering (WCRE)*, pages 52--61, Koblenz, Germany. IEEE.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. (2008). Bitblaze: A new approach to computer security via binary analysis. In Sekar, R. and Pujari, A. K., editors, *Information Systems Security*, pages 1--25, Berlin, Heidelberg. Springer Berlin Heidelberg.

Sutter, B. D., Bus, B. D., Bosschere, K. D., Keyngnaert, P., and Demoen, B. (2000). On the static analysis of indirect control transfers in binaries. In *In PDPTA*, pages 1013--1019.

Tarjan, R. E. (1974). Testing graph connectivity. In *STOC*, pages 185--193, New York, NY, USA. ACM.

Theiling, H. (2000). Extracting safe and precise control flow from binaries. In *Real-Time Computing Systems and Applications (RTCSA)*, pages 23–30, New York, NY, USA. ACM. ISSN 1530-1427.

Tikir, M. M. and Hollingsworth, J. K. (2002). Efficient instrumentation for code coverage testing. *SIGSOFT Softw. Eng. Notes*, 27(4):86–96. ISSN 0163-5948.

Tip, F. (1994). A survey of program slicing techniques. Technical report, IBM T. J. Watson Research Center, Amsterdam, The Netherlands, The Netherlands.

Tymburibá, M., Moreira, R. E. A., and Quintão Pereira, F. M. (2016). Inference of peak density of indirect branches to detect rop attacks. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 150–159, New York, NY, USA. Association for Computing Machinery.

Wu, Q., Flolid, S., Song, S., Deng, J., and John, L. K. (2018). Hot regions in SPEC CPU2017. In *IISWC*, pages 71--77, New York, NY, USA. IEEE. Invited Paper for the Hot Workloads Special Session.

Xu, L., Sun, F., and Su, Z. (2009). Constructing precise control flow graphs from binaries. Technical report, University of California, Davis, Davis, CA, USA.

Yount, C., Patil, H., Islam, M. S., and Srikanth, A. (2015). Graph-matching-based simulation-region selection for multiple binaries. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 52--61, Washington, DC, USA. IEEE Computer Society.

Zhen, L. (2014). Control flow graph based attacks: In the context of flattened programs. Master's thesis, KTH, Stockholm, Sweden.

Zhou, R. and Jones, T. M. (2019). Janus: Statically-driven and profile-guided automatic dynamic binary parallelisation. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 15–25. IEEE Press.