

**GERAÇÃO DE CASOS DE TESTE PARA  
LINGUAGENS COM ARITIMÉTICA DE  
PONTEIRO**



FRANCISCO DEMONTIÊ DOS SANTOS JUNIOR

GERAÇÃO DE CASOS DE TESTE PARA  
LINGUAGENS COM ARITIMÉTICA DE  
PONTEIRO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: MARIZA ANDRADE DA SILVA BIGONHA  
COORIENTADOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte, MG

Janeiro de 2016



FRANCISCO DEMONTIÊ DOS SANTOS JUNIOR

**GENERATION OF TEST CASES FOR  
LANGUAGES WITH POINTER ARITHMETICS**

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: MARIZA ANDRADE DA SILVA BIGONHA  
CO-ADVISOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte, MG

January 2016

© 2016, Francisco Demontiê dos Santos Júnior  
Todos os direitos reservados

**Ficha catalográfica elaborada pela Biblioteca do IEx - UFMG**

Santos Júnior, Francisco Demontiê dos.

S237g Generation of test cases for languages with pointer  
arithmetics. / Francisco Demontiê dos Santos Júnior. —  
Belo Horizonte, 2016.  
xx, 65 f.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de Minas  
Gerais – Departamento de Ciência da Computação.

Orientadora: Mariza Andrade da Silva Bigonha  
Coorientador: Fernando Magno Quintão Pereira

1. Computação – Teses. 2. Diagrama de fluxo de dados.  
3. Compiladores (Programas de computador). 4. Software  
– Validação. I. Orientadora. II. Coorientador. III. Título.

CDU 519.6\*32(043)



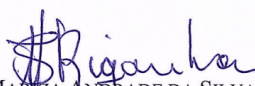
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


## FOLHA DE APROVAÇÃO

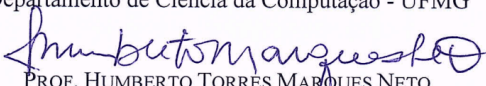
Generation of test cases for languages with pointer arithmetics

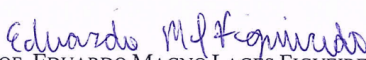
**FRANCISCO DEMONTIÊ DOS SANTOS JUNIOR**

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

  
PROFA. MARIZA ANDRADE DA SILVA BIGONHA - Orientadora  
Departamento de Ciência da Computação - UFMG

  
PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Coorientador  
Departamento de Ciência da Computação - UFMG

  
PROF. HUMBERTO TORRES MARQUES NETO  
Departamento de Ciência da Computação - PUCMG

  
PROF. EDUARDO MAGNO LAGES FIGUEIREDO  
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 29 de janeiro de 2016.





# Agradecimentos

Abusando dos clichês, agradeço primeiramente a Deus, que me deu o dom da vida, saúde e oportunidades para que eu pudesse chegar onde cheguei. Agradeço aos meus pais, Clécia e Demontiê, que além de serem exemplos de dedicação e caráter, sempre fizeram o possível e o impossível por mim e pelas minhas irmãs. Sacrificando, muitas vezes, o conforto próprio em favor do nosso. Me apoiando em cada decisão e me acolhendo sempre que precisei de colo. Por esses e outros motivos, digo e sempre direi que essa conquista é, também, deles. Muito obrigado, “painho” e “mainha”.

Agradeço à minha noiva, Izabela, que faz da minha vida algo muito mais alegre. Que sempre me apoiou, por mais doloroso que fosse, e suportou 2 anos de uma relação a distância (que não é fácil, mas, graças a ela, deu certo). Agradeço por todos os momentos em que ofereceu a mão, todos os momentos em que me fez rir, em que se alegrou com as minhas conquistas, ouviu meus desabafos. Sem ela, nada seria possível.

Agradeço às minhas irmãs, Ana Priscilla e Ana Carolina, por serem quem são. Sempre dizendo que sentem orgulho de mim e me dando forças para perseguir meus objetivos. Me alegrando sempre que estávamos juntos. Vocês são as melhores irmãs que alguém poderia ter. Também agradeço ao meu cunhado Anderson por sempre me receber de braços abertos e não medir esforços para me ajudar quando pôde. Às minhas sobrinhas, Ana Clara e Ana Vitória, que são exemplos de carinho, alegria e que sempre me surpreendem, seja com a sagacidade ou com as estripulias.

Agradeço os meus tios Vanda e Robério e meus primos Matheus e Johanna, que foram minha segunda família no início da minha graduação (o primeiro passo na minha formação acadêmica) e continuam sempre torcendo por mim. Também agradeço ao meu tio José Assis, a Juliano e a Alfredo, por terem me acolhido em Belo Horizonte, sendo sempre prestativos. Agradeço aos meus sogros Vanio e Elizabete e meu cunhado Marcos, por terem me acolhido como parte da família e terem sempre me apoiado. Por me fazerem sentir que em Maceió eu tenho uma terceira casa.

Agradeço aos meus avós Antônio Batista (*in memoriam*), Jacira, José Albino (*in memoriam*) e Eunice (*in memoriam*), pelos ensinamentos que seguirão comigo

pelo resto da vida. Um agradecimento especial ao meu primo Saulo (*in memoriam*), que, além de ter sido para mim um exemplo de simplicidade, alegria, paciência e de profissional, me apresentou a computação e me incentivou nos meus estudos. Em nome deles, agradeço a toda a minha família.

Agradeço ao professor Franklin Ramalho, que me apresentou a área de compiladores, me orientou no meu primeiro projeto de pesquisa e foi compreensivo e apoiador quando decidi fazer o mestrado em outra universidade. À minha orientadora, a professora Mariza, que sempre foi cuidadosa, compreensiva e paciente, mas também assertiva sempre que precisei. Ao meu co-orientador, o professor Fernando, que me ensinou, me acompanhou de perto e foi sempre prestativo. Além disso, agradeço-o, também, por ter sempre buscado uma relação que vai além da profissional, fazendo do meu mestrado algo muito mais divertido.

Agradeço a Douglas, Henrique, Junio, Péricles, Rubens e Victor, que, além de me ajudarem ao longo da minha pesquisa, foram essenciais na minha adaptação a Belo Horizonte. Durante esses dois anos, pude conhecê-los mais de perto, ter bons momentos de descontração, desabafar, escutar. Vocês foram muito importantes para mim e eu serei eternamente grato. Aos amigos Adam, Augusto, Tiago, Catharine, Daniel, Karol, Rodrigo, Marcela, Delano, Carlúcia, Natã e Savyo, por sempre demonstrarem preocupação e fazerem meus dias mais felizes, mesmo com toda a distância.

Agradeço a Maxtrack, na pessoa de Felipe Provenzano, empresa que financiou e que tornou esse projeto de pesquisa possível. Obrigado por todo o suporte prestado. Agradeço também à CAPES, que financiou o primeiro ano do meu mestrado. Por fim, a todos que não foram mencionados aqui e que, de alguma forma, fizeram parte da minha vida durante esses últimos anos, meu muito obrigado.

# Resumo

Testar e depurar software são tarefas difíceis. Em geral, é preciso esperar que o fluxo de execução chegue a uma função de forma a poder testá-la. Nesse sentido, muito esforço foi empregado no desenvolvimento de técnicas, tais como execução simbólica e *fuzz testing*, para geração automática de casos de teste para analisar funções de interesse. Entretanto, tais técnicas possuem limitações. Uma limitação que chama a nossa atenção é o fato de que, até onde sabemos, nenhuma técnica atual relaciona arranjos passados como entradas para funções com outras entradas que representem seus tamanhos. Isso faz com que os casos de teste gerados, em algumas situações, possam levar a acessos inválidos à memória que não aconteceriam em execuções reais do programa sendo testado. Nessa dissertação, apresentamos duas análises estáticas para inferência de tamanhos de arranjos, bem como um algoritmo para geração de casos de teste capaz de gerar testes seguros usando tais análises. Como forma de avaliar a efetividade da nossa técnica, realizamos dois experimentos. No primeiro deles, as análises estáticas foram capazes de relacionar 34.6% dos tamanhos de arranjos recebidos como parâmetro por funções dos programas contidos no SPEC CPU2006 (um conjunto de *benchmarks* contendo programas reais). No segundo, nós utilizamos a ferramenta Asymptus, que realiza análise automática de complexidade de funções e foi desenvolvida como um estudo preliminar deste mestrado, sobre os casos de teste gerados pela nossa técnica. Asymptus foi capaz de inferir corretamente a complexidade de funções desenvolvidas por nós e extraídas dos benchmarks do Polybench. Isso mostra que nossa técnica de geração de entradas é útil para a execução automática de funções de interesse.



# Abstract

Software testing and debugging are hard tasks. In general, it is necessary that the execution flow reaches a function in order to be able to test it. In this sense, much effort was employed in the development of techniques, such as symbolic execution and fuzz testing, to automatically generate test cases in a way to analyze functions of interest. However, such techniques have limitations. A limitation which catches our attention is the fact that, to the best of our knowledge, no current technique relates arrays passed as inputs for functions with other inputs that represent their sizes. Thus, existing techniques, in some cases, may generate test cases which result in invalid memory accesses that would not happen in a real execution of the program being tested. In this dissertation, we present two static analyses for inference of array sizes, as well as an algorithm for generation of test cases capable of generating safe tests by using such analyses. In order to evaluate the effectiveness of our technique, we performed two experiments. In the first of them, we found that the static analyses were able to bind 34.6% of the array sizes received as parameter by functions in the SPEC CPU2006 benchmark suite (which contains real world programs). In the second, we used Asymptus, a tool for automatic inference of function complexity which was developed during this master's research as a preliminary work, over the test cases generated by our technique. Asymptus was able to correctly infer the complexity of functions both written by us and extracted from the Polybench benchmark suite. It shows that our technique of input generation is useful for the automatic execution of functions of interest.



# List of Figures

1.1	A function which iterates over an array. The function receives an array of integers and its size. The array accesses are limited by the size argument. . . . .	4
2.1	An iterative algorithm to calculate the $n^{th}$ Fibonacci number, and its CFG. BB* identifies basic blocks. The arrows between basic blocks, say from BB0 to BB1, mean that the execution may flow from one basic block to the other. . . . .	8
2.2	The result of a liveness analysis on function <code>fibonacci</code> of Figure 2.1. The result is shown only for the basic block <code>BB4</code> . . . . .	9
2.3	Function <code>fibonacci</code> of Figure 2.1 in SSA form. The graph was automatically generated by LLVM. . . . .	10
3.1	The syntax of our core language. . . . .	20
3.2	Data flow analysis equations. . . . .	21
3.3	A code snippet and the resulting abstract state after solving each corresponding equation. . . . .	21
3.4	A struct which encapsulates arrays. . . . .	23
3.5	The relation between <code>new</code> and <code>newpos</code> is not identified by the forward analysis. . . . .	24
3.6	A function which statically allocates an array. . . . .	25
3.7	A function which iterates over an array. This is the same example of Figure 1.1. . . . .	26
3.8	Percentage of arrays with known sizes. . . . .	27
3.9	Percentage of array accesses performed over arrays with known sizes. . . . .	28
3.10	A data structure graph example. . . . .	29
3.11	A graphic representation of an array size. . . . .	30
3.12	The input generation algorithm. . . . .	31
3.13	A function which iterates over an array. This is the same example of Figure 1.1. . . . .	32
3.14	The LLVM's IR of the code snippet in Figure 3.13. . . . .	33
3.15	Code produced by the input generator to test function <code>sum</code> . Lines 2, 3, 4 and 8 are produced by the slice for the code snippet in Figure 3.13. . . . .	33

4.1	Matrix multiplication – the running example that we shall use to explain our contributions. . . . .	37
4.2	Gprof output for a simple program containing our example function. . . . .	38
4.3	The output produced by the aprof input sensitive profiler. . . . .	39
4.4	A function to print duplicate lines containing a given key. The second loop has a conditional execution. . . . .	42
4.5	(a) Program with a multi-path loop. (b) The cost-graph of the program. Nodes represent program points and the edges’ weights represent the number of executed instructions between two points. (c) The cost of each loop iteration. . . . .	44
4.6	(a) Polynomials found for the loop at lines 18-25 of Figure 4.1. (b) Polynomials found for the loop nest at lines 7-15. In each figure, the first curve that fits the points in the verification set is marked in gray. . . . .	45
4.7	Percentage of loops per benchmark of Rodinia that we could analyze. The correctness of all these results have been checked manually. . . . .	46
4.8	An example of LLVM’s IR and a code snippet of an LLVM pass. . . . .	47
4.9	The architecture of our implementation. . . . .	48



# List of Tables

3.1	Reasons for ineffectiveness of the forward analysis on randomly chosen functions. . . . .	26
-----	---	----



# Contents

<b>Agradecimientos</b>	<b>ix</b>
<b>Resumo</b>	<b>xi</b>
<b>Abstract</b>	<b>xiii</b>
<b>List of Figures</b>	<b>xv</b>
<b>List of Tables</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Publications . . . . .	5
<b>2 Literature Review</b>	<b>7</b>
2.1 Data-flow Analysis . . . . .	7
2.2 Static Single Assignment Form . . . . .	9
2.3 Automatic Inference of Program Complexity . . . . .	10
2.4 Program Slicing . . . . .	12
2.5 Data Structure Graph . . . . .	12
2.6 Symbolic Execution . . . . .	13
2.7 Fuzz Testing . . . . .	15
2.8 Final Remarks . . . . .	17
<b>3 Generation of Test Cases for Languages with Pointer Arithmetics</b>	<b>19</b>
3.1 Array Size Inference in C . . . . .	19
3.1.1 Forward Size Analysis . . . . .	20
3.1.2 Backward Size Analysis . . . . .	24
3.2 Test Case Generation . . . . .	27
3.2.1 Data Structure Graph . . . . .	28
3.2.2 Input Generation . . . . .	29

3.2.3	Slicing Technique . . . . .	31
3.3	Conclusion . . . . .	32
<b>4</b>	<b>Case Study</b>	<b>35</b>
4.1	Overview . . . . .	37
4.2	Automatic Inference of Loop Complexity through Polynomial Interpolation . . . . .	39
4.2.1	Input Analysis . . . . .	40
4.2.2	Loop Dependence Analysis . . . . .	41
4.2.3	Code Instrumentation . . . . .	43
4.2.4	Polynomial Interpolation . . . . .	44
4.3	Evaluation . . . . .	45
4.3.1	An LLVM Pass . . . . .	46
4.3.2	Experiment . . . . .	48
4.4	Conclusion . . . . .	49
<b>5</b>	<b>Conclusion</b>	<b>51</b>
5.1	Contributions . . . . .	52
5.2	Future Work . . . . .	52
	<b>Bibliography</b>	<b>53</b>
	<b>Appendix A Functions for the Input Generation Experiment</b>	<b>59</b>



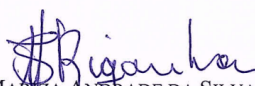
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


## FOLHA DE APROVAÇÃO

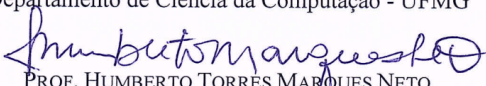
Generation of test cases for languages with pointer arithmetics

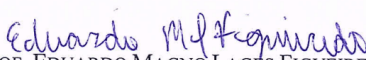
**FRANCISCO DEMONTIÊ DOS SANTOS JUNIOR**

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

  
PROFA. MARIZA ANDRADE DA SILVA BIGONHA - Orientadora  
Departamento de Ciência da Computação - UFMG

  
PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Coorientador  
Departamento de Ciência da Computação - UFMG

  
PROF. HUMBERTO TORRES MARQUES NETO  
Departamento de Ciência da Computação - PUCMG

  
PROF. EDUARDO MAGNO LAGES FIGUEIREDO  
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 29 de janeiro de 2016.



# Chapter 1

## Introduction

Software testing and debugging are hard tasks. In general, to properly perform such tasks, it is necessary to analyze all possible execution paths in the program. In this sense, previous works [Cadaru et al., 2008; Godefroid et al., 2005a; Godefroid, 2007; Saxena et al., 2009] have proposed different approaches to automatically generate inputs (test cases) for programs. One of the most common techniques to this end is *symbolic execution* [Cadaru and Sen, 2013]. Usually, it is performed by means of a virtual machine which replaces the operations that manipulate concrete values by ones which operate over symbolic values. The program starts executing with symbolic values as inputs. Whenever the execution flow reaches a branch, both sides are symbolically executed “simultaneously” and the conditions to execute such paths (called path conditions/constraints) are stored. At the end of the execution, or when it finds a bug, a constraint solver finds concrete input values which follow the same execution flow as the symbolic values which produced a particular set of path conditions.

Another used approach is the *Fuzz Testing* (or fuzzing), which was introduced in the 90’s [Miller et al., 1990] with the purpose of testing programs as a black box providing random inputs. This technique was later improved by Godefroid et al. [2008] with the ability of getting a feedback of the execution for some initial random inputs and finding new input values which follow different paths. They called this new technique as *Whitebox Fuzz Testing*. Although different techniques are shown to be effective on finding bugs or achieving high code coverage, there exist limitations. A limitation which catches our attention is regarding to the size of memory regions. Unlike some strongly typed languages, allocated memory in C has no meta information. Consider the example in Figure 1.1. If we pass to function *sum* an array with 10 elements, the value of *size* has to be less than or equals to 10. If it is not the case, we will end up reaching a bug which may never happen in a actual execution of the function in its

context - i.e. a *false positive*. For the best of our knowledge, no existing technique handle it properly.

```

1: int sum(int *A, int size) {
2:   int s = 0;
3:   for (int i=0; i < size; i++) {
4:     s += A[i];
5:   }
6:   return s;
7: }
```

Figure 1.1: A function which iterates over an array. The function receives an array of integers and its size. The array accesses are limited by the size argument.

In this work we aim to solve this limitation by using static analyses to bind meta information to memory regions. Our hypotheses is the following:

It is possible to reduce the false positive rate of current automatic testing techniques if we can determine symbolic meta information for memory regions which are inputs for the testing unit.

In this dissertation we present two static analyses to bind allocated memory regions with their sizes. The first one, which we call *forward size analysis*, gets information about the memory allocation instructions and propagates it forward. The second, *backward size analysis*, gets information about the memory accesses and propagate it backward. While the former is more precise to our goal, the latter has shown to be able to bind more pairs of allocated memory and size.

In order to measure the effectiveness of the size analyses, we have executed experiments over the benchmarks found at SPEC CPU2006. We have been able to bind sizes for 22.7%, in average, of all arrays in SPEC passed as argument to functions. Also, we have counted that for all array accesses in SPEC that are performed over arguments of functions, 28% of them perform over arrays which we have been able to find a size. It tells us that generating inputs for arrays using the result of this analysis avoids memory errors in up to 28% of all array accesses.

We have purposed an algorithm to generate test cases, which is similar to the one used in DART (Directed Automated Random Testing) [Godefroid et al., 2005a]. Basically, we first identify the interface of a function that we want to test. The interface consists of the types of a function's inputs, which are its parameters and global variables that are read before written inside the function. The types are represented by a graph, which we call *data structure graph*. We then systematically and recursively generate



random values for each input. When our size analyses are able to bind an array with its size, our algorithm generates values which are consistent for both arguments.

We have implemented our techniques on top of the LLVM compiler infrastructure [Lattner and Adve, 2004b]. Our approach has several possible uses. For instance, we have used it to improve the effectiveness of a program complexity inference tool, called *Asymptus* [Demontiê et al., 2015]. *Asymptus* is a tool for automatic inference of loop complexity, which was developed during this master’s research as a preliminary work. It can be described in four main steps: (1) static analysis, (2) code instrumentation, (3) dynamic information extraction and (4) polynomial interpolation. Since this technique is novel, we have published a paper about it, which is partially restated in Chapter 4. One of the *Asymptus*’ limitations comes from the fact that it needs to execute a function a certain number of times, with different inputs, in order to be able to analyze it. However, it may not happen for several reasons (lack of data-sets, hardcoded inputs for functions, functions called only for specific inputs, among others). We have executed *Asymptus* together with our input generator on functions that have arrays, matrices and recursive data-structures as inputs and found that the technique presented in this dissertation is effective on the execution of interest functions.

The main application of our work is in the design of a testing infra-structure for Maxtrack, a Brazilian company which builds trackers for trucks. This infra-structure is meant to be used to test, in a Unix environment, the software developed to be embedded into trackers. Because of that, we do not have the bodies of some library functions and they are automatically generated to return a random value based on the function’s return type. We execute the test drivers generated for chosen functions together with Valgrind [Nethercote and Seward, 2007a] in order to catch memory corruption errors. When an error is found, we log the seed of the random functions, allowing the user to later reproduce the test.

The rest of this dissertation is organized as follows. Section 1.1 shows the list of publications related to the completion of the intended degree. Chapter 2 presents the concepts needed to understand this work and a literature review about the related themes. Chapter 3 presents our solution to the problem stated in this dissertation. We show the array size analyses in Section 3.1 and our input generation approach in Section 3.2. Chapter 4 presents a case study of our input generator using *Asymptus*. We make our final thoughts in Chapter 5.

## 1.1 Publications

- Francisco Demontiê, Filipe de Lima Arcaño, and Mariza A. S. Bigonha. Um Algoritmo para Emparelhamento de Chamadas de Função. Brazilian Symposium on Programming Languages (SBLP). 2014.
- Francisco Demontiê, Junio Cezar, Mariza A. S. Bigonha, Frederico Campos, Fernando Magno Quintão Pereira, Automatic Inference of Loop Complexity Through Polynomial Interpolation. Brazilian Symposium on Programming Languages (SBLP), pp. 1-15. 2015.
  - This paper was chosen as the 3<sup>rd</sup> (third) best paper of the Brazilian Symposium on Programming Languages 2015.
- Junio Cezar, Francisco Demontiê, Mariza A. S. Bigonha, and Fernando Magno Quintão Pereira, Asymptus - A Tool for Automatic Inference of Loop Complexity. CBSOft, Tools Session, pp. 89-96. 2015.

# Chapter 2

## Literature Review

This chapter presents a literature review about the subjects and concepts related to this work. We start with the necessary background for the reader in order to better understand this dissertation and end presenting the works that are most related to ours.

### 2.1 Data-flow Analysis

To infer the sizes of arrays, we resort to data-flow analysis. To better understand the concept of data-flow analysis, it is necessary to know what is a Control Flow Graph (CFG) [Allen, 1970]. A CFG is a program representation consisting of a directed graph where nodes are *basic blocks* and there is an edge between two basic blocks BB1 and BB2 if the execution can flow to BB2 right after the end of BB1. A basic block is the maximum set of consecutive instructions with basically two properties: (i) the execution of a basic block only starts from the first instruction (there are no jumps to the middle of a basic block) and (ii) the execution of a basic block only ends in the last instruction (there are also no jumps from the middle of a basic block). It means that any branch instruction is the end of a basic block and the target of any branch is the first instruction of a basic block. Figure 2.1 shows a function, which implements the iterative algorithm to calculate the  $n^{th}$  Fibonacci number, and a graphical representation of its CFG.

Data-flow analysis [Kildall, 1973] is a technique for analyzing an interest property at several points of a program. The goal is to approximate properties of the dynamic behavior of a program by analyzing it statically. The properties are associated to variables and this association is called *abstract state*. Data-flow analyses make use of the program's CFG, to determine how the data flows in the program, and are expressed

```

1: int fibonacci(int n) {
2:   if (n <= 1)
3:     return n;
4:
5:   int n_1 = 1, n_2 = 0, fib;
6:   for (int i=2; i <= n; i++) {
7:     fib = n_1 + n_2;
8:     n_2 = n_1;
9:     n_1 = fib;
10:  }
11:  return fib;
12: }

```

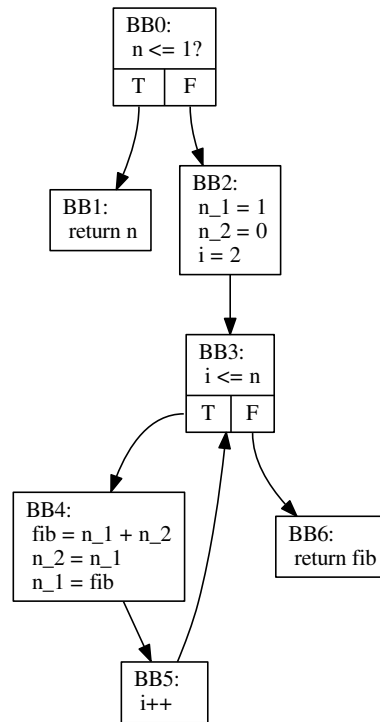


Figure 2.1: An iterative algorithm to calculate the  $n^{\text{th}}$  Fibonacci number, and its CFG. BB\* identifies basic blocks. The arrows between basic blocks, say from BB0 to BB1, mean that the execution may flow from one basic block to the other.

as transfer functions. A transfer function computes the abstract state of a variable at a program point using as input the variables involved in the analyzed instruction and the abstract state calculated at the neighbor instructions.

A classic example of data-flow analysis is the *liveness analysis* which calculates what variables are alive at any program point. A variable is alive at a program point if it was already defined, is not out of scope and will still be used in a further program point. Thus, the liveness analysis starts with the last instruction, in a topological order, and go backward. At the initial program point, after the last instruction, all variables are dead. During the analysis, the variables used as operands in a instruction are considered to be alive in the program point right above it, while the assigned variable, if there is one, is considered dead.

It is possible to describe the liveness analysis using two transfer functions:

$$\llbracket p \rrbracket_{in} = (\llbracket p \rrbracket_{out} - \{v\}) \cup \text{vars}(E) \quad (2.1)$$

$$\llbracket p \rrbracket_{out} = \bigcup_{p_s \in \text{succ}(p)} \llbracket p_s \rrbracket_{in} \quad (2.2)$$

where  $p$  is a program point of the form  $p : v = E$  (with  $v$  being a variable and  $E$

an expression),  $\llbracket p \rrbracket_{in}$  and  $\llbracket p \rrbracket_{out}$  the abstract state of  $p$  (i.e., the set of alive variables right before and right after  $p$ , respectively),  $vars(E)$  the variables that appear in  $E$ ,  $succ(p)$  the set of program points which are immediate successors of  $p$ . Notice that, from the Equation 2.2, when a point with multiple successors is found, e.g. a branch instruction of an if-then-else, we get as result the union of the abstract states of the successors. We do it because if a variable may be alive in at least one possible path from the program point  $p$  to the end of the program, it has to be considered alive right after  $p$ . Otherwise, some compiler optimizations could make a wrong decision. Figure 2.2 shows the result of a liveness analysis for the instructions of the basic block  $BB_4$  of Figure 2.1. The values between curly brackets are the alive variables in each program point. For instance, the variable  $n\_2$  is not alive right before the instruction  $n\_2 = n\_1$ , since this instruction redefines the variable and any information about the variable is lost. Obviously, it is alive after the definition.

<p><b>BB0:</b></p> <p style="padding-left: 40px;"><math>\{n\_1, n\_2, n\}</math></p> <p><math>fib = n\_1 + n\_2</math></p> <p style="padding-left: 40px;"><math>\{fib, n\_1, n\}</math></p> <p><math>n\_2 = n\_1</math></p> <p style="padding-left: 40px;"><math>\{fib, n\_2, n\}</math></p> <p><math>n\_1 = fib</math></p> <p style="padding-left: 40px;"><math>\{fib, n\_1, n\_2, n\}</math></p>
--

Figure 2.2: The result of a liveness analysis on function `fibonacci` of Figure 2.1. The result is shown only for the basic block  $BB_4$ .

If in a data-flow analysis the abstract state has to be associated to each variable at each program point, we say that the analysis is dense. Otherwise, if the abstract state of a variable is the same for the whole program, the analysis is sparse. Sparse analyses are faster than dense ones and require less space in memory. Often, there are ways to make a dense analysis to become sparse by changing the program.

## 2.2 Static Single Assignment Form

A common approach to make a data-flow analysis sparse is to break the live range of variables in interest points, in order to be able to assign results for variables. To this purpose, Cytron et al. [1989] have proposed the Static Single Assignment Form (SSA). A program representation is in the SSA form if, and only if, each variable has exactly one assignment and every variable is defined before being used. Thus, the main idea is to split a variable into two versions whenever a reassignment is found. However, it is

possible that we have different versions of the same original variable in different sides of a branch. In this case, we need to have a way to identify the value reaching a use. This is done by *PHI* ( $\phi$ ) *functions*. A  $\phi$  function is only a concept (it does not exist in the concrete implementation) which receives a list of versions of a variable - one for each different path reaching the point - and returns the proper value for a given execution. Figure 2.3 shows the example of Figure 2.1 in SSA form. It is worth to mention that all the analyses presented in this dissertation are performed over LLVM's intermediate representation in SSA form.

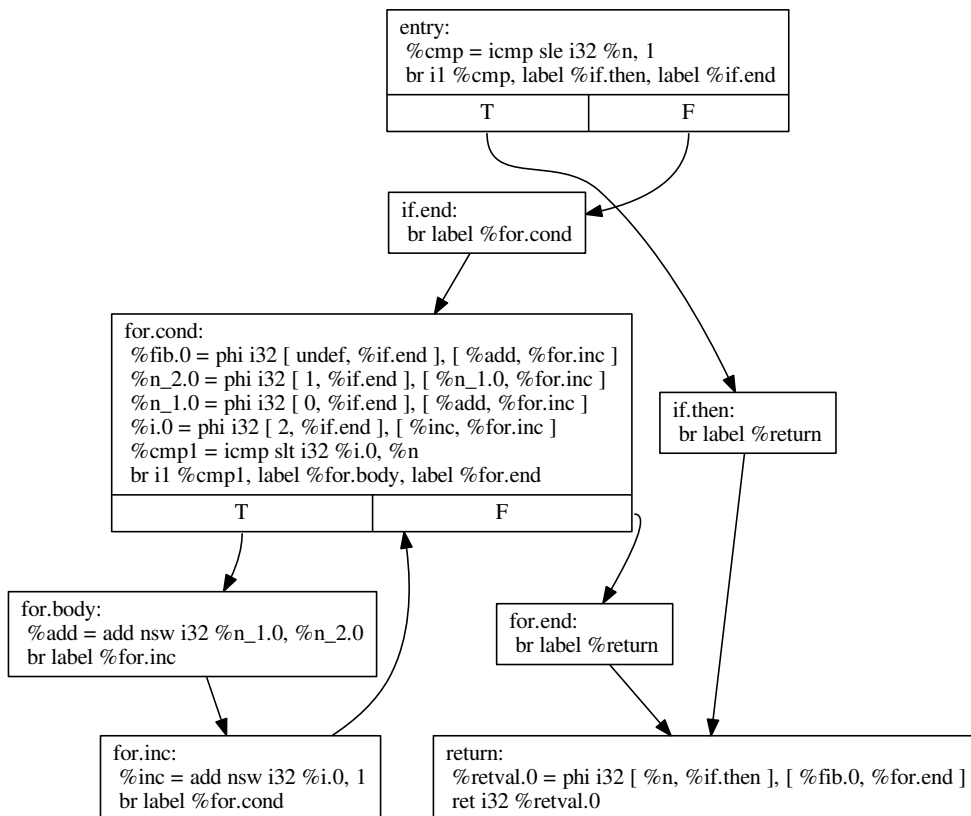


Figure 2.3: Function `fibonacci` of Figure 2.1 in SSA form. The graph was automatically generated by LLVM.

## 2.3 Automatic Inference of Program Complexity

In this section we describe works related to Asymptus, the program complexity inference tool, developed as a preliminary work during this master's research, that we use to evaluate the effectiveness of our input generation approach. Recent work has attempted to improve the state of the art on complexity analysis. Particularly, profiler-based approaches have been able to give interesting results. Goldsmith et al. [2007]

proposed a technique which consists in executing the target program over workloads with different orders of magnitude and tracking how many times each program location was executed. They use polynomial regression to fit the data into a linear or power-law model. However, the user has to specify, for each workload, the value of features - a feature is an input property which affects the algorithm execution, e.g. the size of an array or the height of a tree. Our technique is able to automatically infer loops' inputs; hence, it does not require this type of user intervention.

Zaparanuks and Hauswirth [2012] proposed the concept of algorithmic profiler. Their approach consists in grouping the basic blocks of a loop and the functions which make a cycle in the call-graph into the so called *repetition nodes*. Those nodes are then combined in units that they have named *algorithms*. The technique is able, for example, to identify if an algorithm is modifying or traversing a list or an array. In order to estimate the complexity of an algorithm, they retrieve the size of the inputs and some performance metrics for each execution of the repetition nodes. This modus operandi leads to a significant overhead, since the analyzer iterates over the entire data structure to calculate its size. The automatic reconstruction of data-structures is still an incipient area of research. Therefore, Zaparanuks *et al.* have implemented a prototype which, up to this point, can analyze only toy examples. We cannot reconstruct recursive data-structures as Zaparanuks does; however, our approach is able to infer the complexity of most of the loops in a real-world benchmark suite.

The work that is more related to the technique described in Chapter 4 is Coppa et al. [2012]'s input sensitive profiler. This work has materialized itself into a tool called *aprof*. Core to *aprof*'s work is the notion of Read Memory Size (RMS). This metric represents the number of memory locations which are read before they have been written inside a function. *Aprof* was implemented as a *Valgrind* [Nethercote and Seward, 2007b] extension. We believe that *aprof* is the most practical tool available nowadays to infer the complexity of general purpose programs. Nevertheless, it has the following shortcomings: (i) the granularity of results is at the function, not at the loop, level; (ii) users have to fit equation by hand in *aprof*'s results to find the complexity of a function; and (iii) results are given in terms of RMS, which may not be significant to the developer. Our technique is capable of addressing these drawbacks.

There exists a plethora of work related to the static estimation of complexity of code [Alves et al., 2015b; Danielsson, 2008; Gulavani and Gulwani, 2008; Gulwani et al., 2009b; Monniaux and Gonnord, 2011]. Our work is essentially different from these approaches, because our results are based on program behavior observed at runtime. In other words, our approach is dynamic: we execute and profile the program to infer its computational complexity. The downside of our approach is that we are not able

to *prove* properties about the program's complexity: there are no guarantees that we will be able to observe every possible execution path within the program code. The upside is precision: our approach is able to reason about typical programming language features such as dynamically allocated memory, multiple paths in loops, non-structured control flow graphs and pointer arithmetics. So far, these real-world constructs have been challenging adversaries to the purely static analyses.

## 2.4 Program Slicing

Program slicing, as defined by Weiser [1981], consists of segmenting a program extracting only the instructions which affect the computed value at an interest point, called slicing criterion. There are basically two kinds of slicing techniques: static and dynamic slicing. The static techniques [Horwitz et al., 1988; Danicic et al., 1995; Reps, 1998], in general, statically generate a dependence graph [Ottenstein and Ottenstein, 1984] for a program's instructions and, starting from the slicing criterion, traverse the graph getting the instructions which affect it. The dynamic approaches [Korel and Laski, 1988, 1990; Agrawal and Horgan, 1990; Kamkar et al., 1993] use runtime information to compute the instructions which affect the slicing criterion in a particular execution. Dynamic slices tend to be smaller and more significant than the static ones for some programs/inputs and are useful to reduce the search space of a bug known to happen for a given input. Another technique which aims to reduce the size of a slice is called *conditioned slicing* [De Lucia et al., 1996]. Firstly, it computes a reduced version of the program containing only the reachable instructions given an initial input set - using, for instance, symbolic execution [Jaffar et al., 2012]. Then, the slicing is performed in the so called *conditioned program*. The different program slicing techniques are used for decades in different areas of computer science for several ends, as debugging, refactoring, parallelism and compiler optimizations. We have used static program slicing when generating test cases in order to reuse a program's instructions which calculate the size of an array.

## 2.5 Data Structure Graph

Lattner and Adve [2003] have proposed a data structures analysis in order to enable transformations which need to disambiguate entire instances of data structures, such as lists, trees or graphs. For this analysis, they proposed a data structure graph. We borrowed the idea of a data structure graph in order to identify the types of a function's



inputs and generate test cases. However, besides some structural differences of our graph, our approach to build it differs from Lattner and Adve [2003]’s. Their approach iterates over each instruction of a function. For each instruction of memory allocation, it creates a graph node with a flag corresponding to its memory region (heap or stack). For loads and stores it merges the nodes involved in the operation. For instructions corresponding to a struct field access or an array indexing, it updates type information of the accessed node. With this technique they are able to have precise information about the memory regions. Our approach, described in Section 3.2.1, is not focused on memory and we use the LLVM type information to generate the data structure graph for the inputs of a function.

## 2.6 Symbolic Execution

Symbolic Execution, although introduced more than 3 decades ago [Boyer et al., 1975; King, 1976], has been used in the past years for automatic test case generation, mostly because of the improvements on constraint solving. Techniques based on symbolic execution are capable of finding concrete inputs which execute a large part of the possible execution flows of a program. The classic symbolic execution consists of execution a program using symbolic values as inputs, instead of concrete ones. The operations in the program are changed to deal with symbolic value, generating as output symbolic expressions which are represented as functions of its inputs. The results of the execution are kept as *path conditions* or *path constraints*, which represents the conditions (input values) for an execution flow to be taken. Such constraints are written as first-order logic formulas, thus it is possible to use constraint solvers in order to find concrete inputs to take the same execution flows. More recent techniques [Godefroid et al., 2005a; Godefroid, 2007; Cadar et al., 2008], however, perform symbolic execution and concrete execution side-by-side, what is called *concolic testing*;

Cadar et al. [2008] noticed that previous work usually did not allow interaction with the external environment, or, when they did, the execution of external functions where limited to use only concrete inputs. In this context, Cadar et al. [2008] implemented KLEE (KLEE LLVM Execution Engine), a symbolic execution tool which aims to improve the interaction with the external environment making the use of symbolic execution more effective in real systems. The method used by KLEE consists on modeling the system call API at the C language’s standard library level. Basically, the necessary standard library functions are modified to treat symbolic values. For instance, KLEE has a file system with a single directory. The number of files and their

sizes are specified by the user which wants to make use of it during the symbolic execution. If the function `fopen`, which originally opens a file in the system's file system and returns a pointer to it, is called using concrete parameters, it is executed normally. Otherwise, if the call is performed using symbolic values as parameters, the function's behavior is simulated using KLEE's symbolic file system. KLEE uses a version (modified by the authors) of `uClibc`<sup>1</sup>, a small implementation of the C's standard library. Therefore, in order to make a proper use of KLEE on programs which use the standard library, it is necessary to compile the program pointing to the modified `uClibc`.

Besides modeling the system's API, using search heuristics and representing symbolic states in a compact way allow the use of KLEE on real systems to generate test cases with high code coverage and to find non-trivial bugs. KLEE was executed over the 89 programs of *GNU Coreutils*<sup>2</sup> for a period of time limited to 60 minutes each. KLEE was able to generate test cases which covered, in average, 90% of the programs' code lines and found 10 bugs which were undiscovered. However, the tool was not executed over large systems and it is not clear its real effectiveness and efficiency. Also, to ensure covering the several paths of a function, the execution flow has to reach this function. Although, in some cases KLEE need to exercise a large number of program paths until it find inputs to make the execution flow to reach a particular program point.

The closest related work to ours was done by Godefroid et al. [2005a]. They proposed a hybrid technique for test input generation called *Directed Automated Random Testing* (DART). Their technique consists of three main steps: (i) statically identify the input interface of a program with its external environment, (ii) randomly generate inputs for the program based on its interface, and (iii) analyze the dynamic execution of the program to generate new inputs in order to exercise other program paths. For step (ii), Godefroid et al. [2005a] have implemented an algorithm which takes a memory location and a type as inputs. If the type is a primitive type, the algorithm initializes the memory location with as many random bits as the type requires. If it is a pointer to another type `T`, the algorithm will either lets the pointer to be *null* or allocate a new memory region. In the second case, the algorithm will recurse passing the newly allocated region and `T`. If the type is a struct or an array, every sub-element is initialized recursively.

For step (iii), DART instruments the program and executes it both concretely and symbolically. With this side-by-side execution, DART generates path constraints on the fly depending on how predicates of branches evaluate. The constraints gen-

---

<sup>1</sup>[www.uclibc.org](http://www.uclibc.org)

<sup>2</sup>[www.gnu.org/software/coreutils](http://www.gnu.org/software/coreutils)

erated during an execution represent an equivalence class of all input vectors which will drive the program through the executed paths. DART then begins to change predicates in the path constraint to guide the generation of inputs for the next execution, which will take a different path. Godefroid et al. [2005a] claim that, using a dynamic analysis to generate new inputs, DART is able to improve code coverage when compared to pure random testing. However, since this technique suffers of path explosion, Godefroid [2007] proposed an extension of DART, which is called *Systematic Modular Automated Random Testing* (SMART). Basically, SMART executes functions isolated and generates summaries in form of preconditions and postconditions. The summaries of lower level functions are re-used when testing higher level functions to avoid re-testing functions.

Our approach is mostly based on DART, in the sense that our algorithm to generate random values is similar to DART's. However, we have addressed some limitations. The main limitation is that DART cannot relate an array with its size when generating inputs for a function call. It means that it can generate test cases which lead to invalid memory accesses that would not occur in practice. It will increase the number of false positives when looking for bugs. In our approach, we statically identify such relations using either a forward analysis (which propagates information starting on memory allocations) or a backward analysis (which calculates symbolic upper bounds for variables which indexes a given array). Also, according to the algorithm, DART only treats arrays with fixed size. In our approach, we identify pointer arithmetics and thus can treat pointers as arrays. In this case, we allocate a memory space of random size.

In order to address some of the limitations of DART, Godefroid et al. [2008] have implemented SAGE (Scalable Automated Guided Execution). The main contribution of their work is a so called *generational search*, which aims to improve code coverage faster and enable analyzing larger programs. The key idea behind SAGE's generational search is to systematically expand each constraint in a path constraint in order to generate new inputs. They start symbolically executing the program with a random input, what generates a path constraint. Then, each constraint is systematically negated and solved to find a new input, called child input. The same process is done recursively for the children inputs. In order to avoid redundancy, if a child input was generated when the constraint  $j$  was negated, the search algorithm will start expand this child input from the constraint  $j+1$ . Another contribution of this work is that they do not perform a source-code-based symbolic execution. Rather, their constraints generator analyzes (off-line) traces of execution of x86 machine-code, being language-independent. Our work differs from Godefroid et al. [2008]'s because our approach is source-code-based. It is not a goal this work to improve code coverage.

## 2.7 Fuzz Testing

Random testing is basically the ability of automatically testing programs using random factors. Some approaches randomly generate sequences of method calls to test object-oriented programs [Pacheco et al., 2007; Oriol and Tassis, 2010], create test cases (argument values for functions, for example) [Miller et al., 1990; Godefroid et al., 2005a; Forrester and Miller, 2000], or even insert failures for hardware simulation [Groce et al., 2007]. Although it is already known that random testing usually leads to low code coverage [Chen et al., 2013], it was shown that this technique can be effective to find bugs in a short period of time [Duran and Ntafos, 1984; Forrester and Miller, 2000; Ciupa et al., 2007, 2008]. Also, random testing is much simpler than more sophisticated approaches, such as symbolic execution. The use of random testing to generate test cases, which is the goal of this work, is called *fuzz testing* (or *fuzzing*). The literature classifies the fuzzing approaches onto two categories: blackbox and whitebox fuzzing. The blackbox approaches make use only of the program's interface, while whitebox fuzzers also analyze the executed instructions in order to improve code coverage.

Fuzz testing was firstly introduced by Miller et al. [1990] with the goal of testing the reliability of UNIX utilities, such as `cat`, `grep` and `awk`. They basically implemented a random character generator to generate inputs for the UNIX utilities by command line. A similar work was done by Forrester and Miller [2000] to test Windows NT applications. For doing this, they implemented a tool which randomly generates valid keyboard and mouse events. Both tools had practical use and were well succeeded to find bugs in real applications. However, while the former is very simple and works only for programs which receives inputs from the command line, the latter is very specific to its use. Our approach identifies the input interface of interest functions in a program and automatically generates random inputs, thus being more general.

Some effort has been applied to improve testing regarding to its effectiveness of finding bugs and its code coverage [Chen et al., 2005; Godefroid et al., 2005a; Oriol and Tassis, 2010]. Chen et al. [2005] has proposed a method to select input values so that the test cases will be spread over the input domain. They showed that using their technique, they improve code coverage and find more bugs [Chen et al., 2013]. Ahmad and Oriol [2014] have also shown improvement on finding bugs going to the opposite way. When a test case leads to a bug, they select surrounding values as interest values for the next inputs. They have implemented their technique on top of YETI [Oriol and Tassis, 2010]. YETI was shown to be useful to find bugs, but it is not effective on creating instances of complex data structures. It is not the goal of this work to improve code coverage, yet it would be possible to apply some of these techniques on

top of our work.

Godefroid [2014] proposed the concept of Micro Execution [Godefroid, 2014] as the ability of executing a code fragment (function) without user intervention. The key idea behind this technique is to dynamically identify the Input/Output interface of a target function and to provide input values on demand. In order to identify inputs he defined a default policy, where an input is a value read from an uninitialized function argument or a dereference of another input. Since it is done dynamically in executable code, function arguments are stored in memory addresses above the stack pointer (esp). Whenever an instruction tries to read an uninitialized input value, a value is generated using one of four strategies: (i) filling all inputs with zeros (useful for debugging), (ii) randomly generating values, (iii) using a process dump, where inputs are filled with values present in the dump (useful to reproduce test cases) or (iv) using a whitebox fuzzer called SAGE, which uses symbolic execution in order to cover more program paths.

Although micro execution has some applications, it has limitations. It generates each input value independently (except if it uses the fourth strategy of value generation, using SAGE, making the execution slower). Because of this, it may fail to perform useful executions when the inputs are data structures such as arrays, trees or linked lists. Our technique solves this problem by statically identifying data structures and generating code to create instances of them. It also cannot identify memory corruption bugs due to input data structures, such as buffer overflows. It happens because some information (such as buffer size) may not be known inside the function. We use a symbolic analysis to bind program symbols (variables) with the size of arrays. With this, we are able to detect such memory errors. Finally, it is not clear the real applicability of micro execution to find bugs on real world applications.

## 2.8 Final Remarks

There exists a large set of works that automatically generates test cases for functions. One of the most used techniques is symbolic execution [Cadar et al., 2008; Godefroid et al., 2005a, 2008]. Although this technique is effective to improve coverage, it can be take too long to generate inputs for some interest functions, since the execution flow has to reach them. Also, KLEE [Cadar et al., 2008] is not able to generate recursive data structures (e.g. linked lists), while our work, similarly to DART [Godefroid et al., 2005a], implements an algorithm to generate such data structures with random size. Fuzz testing was shown to generate inputs faster than classic symbolic execution while

still achieves high code coverage. One of the most recent works in this area was done by Godefroid [2014], which dynamically generates random inputs for functions. However, each input is generated independently, what may lead to ineffective executions when inputs are data structures, for example. From the best of our knowledge, there is no technique for test case generation that relate memory regions with variable that represent meta information about their sizes. Thus, existing techniques may generate test cases which result in invalid memory accesses that would not happen in a real execution of the same program. The work described in Chapter 3 does it by using two static array size analyses. Is not a goal of our work to maximize code coverage.

# Chapter 3

## Generation of Test Cases for Languages with Pointer Arithmetics

This chapter describes a method for test case generation. Firstly, we describe two approaches to identify symbolic sizes of memory regions. Such approaches are used to generate test cases that do not end in invalid memory accesses. The use of the array size analysis while generating test cases is the major contribution behind the work done for this dissertation.

### 3.1 Array Size Inference in C

Aiming to generate test cases which do not lead to memory corruption errors not yet present in the program, we proposed the idea of binding arrays with their sizes when generating function arguments. In this section we describe two approaches for array size analysis. Firstly, we would like to adopt a core language which is much smaller than C but still expressive enough for our goal, which we present in Figure 3.1. This language was introduced by Nazaré et al. [2014] and has the constructions that we need to present our analysis. Nazaré et al. [2014] have described the semantics of their language in details using operational semantics [Plotkin, 1981]. We choose to informally describe the semantics of the instructions in order to make it easier to the reader.

A program  $P$  is composed by instructions which have two kinds of operands: variables (V) and constants (C). Each instruction has a label (in order to allow jumps). We denote binary operation as a  $\oplus$  because it does not make any difference which operation it is. Stores and loads have the usual semantics. A store  $*v_1 = v_3$  means that the value in  $v_3$  is stored into the memory location addressed by  $v_1$ . A load

Programs (P)	::=	$\ell_1 : I_1, \ell_2 : I_2, \dots, \ell_n : \text{end}$
Labels (L)	::=	$\{\ell_1, \ell_2, \dots\}$
Variables (V)	::=	$\{v_1, v_2, \dots\}$
Constants (C)	::=	$\{c_1, c_2, \dots\}$
Operands (O)	::=	$V \cup C$
Instructions (I)	::=	
– Assignment		$v = o$
– Input		$v = \bullet$
– Binary operation		$v_1 = v_2 \oplus v_3$
– $\phi$ -function		$\bar{v} = \phi(\bar{v}_1, \dots, \bar{v}_n)$
– Store into memory		$*v_1 = v_3$
– Load from memory		$v_1 = *v_2$
– Allocate memory		$v_1 = \text{alloc}(v_2)$
– Liberate memory		$\text{free}(v)$
– Branch if zero		$\text{br}(v, \ell)$
– Unconditional jump		$\text{jmp}(\ell)$
– Halt execution		$\text{end}$

Figure 3.1: The syntax of our core language.

$v_1 = *v_2$  means that the variable  $v_1$  will keep the value contained into the memory location addressed by  $v_2$ . The “branch if zero” instruction changes the control flow of the program to the label  $\ell$  if the predicate is zero. A  $\phi$ -function, explained in Section 2.2, is necessary since all programs in this language are in SSA. The *alloc* instruction is the most important to our analysis. This instruction allocate  $v_2$  bytes in memory and returns a pointer to it.

### 3.1.1 Forward Size Analysis

The forward size analysis is an interprocedural analysis which starts on memory allocation instructions. We iterate over each function of the program in topological order of a call-graph. It makes us to analyze the called functions before analyzing their callees. For each function, we iterate over the instructions looking for memory allocation instructions. Such instructions are LLVM *allocas* (which allocate memory on the stack) and call instructions for memory allocation functions. We add the *alloca*’s size or the arguments of the function call to the abstract state of the left-value of the instruction. We also handle pointer alias information and pointer arithmetics.

Figure 3.2 shows the equations of our data-flow analysis. In this analysis, the abstract state of an array is a set of variables which are related to its size, and the abstract state of a variable is a set of arrays with the size related to it. Equation 3.1 says that if we find a *malloc* with a constant value as argument, the abstract state of



$$v = \text{malloc}(c) \vdash \llbracket v \rrbracket = \{c\} \quad (3.1)$$

$$v = \text{malloc}(v_1) \vdash \llbracket v \rrbracket = \{v_1\}, \llbracket v_1 \rrbracket \cup = \{v\} \quad (3.2)$$

$$v_1 = v_2 + c \vdash \frac{v'_2 \in \llbracket v_2 \rrbracket}{\llbracket v_1 \rrbracket = \llbracket v_2 \rrbracket, \llbracket v'_2 \rrbracket \cup = \{v_1\}} \quad (3.3)$$

$$v_1 = v_2 \odot v_3 \vdash \frac{v'_2 \in \llbracket v_2 \rrbracket, v'_3 \in \llbracket v_3 \rrbracket}{\llbracket v_1 \rrbracket = \llbracket v_2 \rrbracket \cup \llbracket v_3 \rrbracket, \llbracket v'_2 \rrbracket \cup = \{v_1\}, \llbracket v'_3 \rrbracket \cup = \{v_1\}} \quad (3.4)$$

$$v = \phi(v_1, \dots, v_n) \vdash \frac{v'_1 \in \llbracket v_1 \rrbracket, \dots, v'_n \in \llbracket v_n \rrbracket}{\llbracket v \rrbracket = \bigcup_{1 \leq i \leq n} \llbracket v_i \rrbracket, \llbracket v'_1 \rrbracket \cup = \{v\}, \dots, \llbracket v'_n \rrbracket \cup = \{v\}} \quad (3.5)$$

Figure 3.2: Data flow analysis equations.

the resulting array is a set containing the constant (i.e., the size of  $v$  is  $c$ ). Equation 3.2 is similar. However, since it deals with a variable, which may already be the size of another array, we add the resulting array to the abstract state of the variable. Notice that the operator  $\cup =$  represents the addition of the elements in the set on the right to the set on the left. Equation 3.3 says that if we have pointer arithmetics, the resulting array has the same abstract state as the original one, i.e., everything related to the size of  $v_2$  is also related to the size of  $v_1$ . Also, in this case, we add the new array to the abstract state of any variable in the abstract state of the original array, in order to keep it consistent. Equation 3.4 says that any operation involving a variable which is related to the size of arrays, makes the resulting value to be also related to the same arrays. Finally, Equation 3.5 says that if we have a  $\phi$ -function, the abstract state of the resulting value is the union of the abstract states of all operands. Figure 3.3 shows a code snippet with the resulting abstract state after solving the equations.

<pre><b>int</b> *v = (<b>int</b>*) malloc(10);</pre>	<b>Equation 1:</b> $\llbracket v \rrbracket = 10$
<pre><b>int</b> *v1 = (<b>int</b>*) malloc(n1);</pre>	<b>Equation 2:</b> $\llbracket v1 \rrbracket = \{n1\}$ and $\llbracket n1 \rrbracket = \{v1\}$
<pre><b>int</b> *v2 = v1 + 1;</pre>	<b>Equation 3:</b> $\llbracket v2 \rrbracket = \{n1\}$ and $\llbracket n1 \rrbracket = \{v1, v2\}$
<pre><b>int</b> n2 = n1 + x;</pre>	<b>Equation 4:</b> $\llbracket n2 \rrbracket = \{v1, v2\}$

Figure 3.3: A code snippet and the resulting abstract state after solving each corresponding equation.

We also handle user-defined memory allocation functions which make use of other memory allocation functions. We start with a set of memory allocation functions (in C, *malloc*, *calloc*, *valloc*, etc.). Then, in the topological order traversal of the call graph,

we analyze return instructions of functions. If the returned value is an array which has a function argument as its abstract state, then we mark this function as a memory allocation function. This way, we achieve higher effectiveness.

### 3.1.1.1 Evaluation

We wanted to measure the effectiveness of the forward array size analysis to find pairs of function parameters which are composed by arrays and their size. The goal is to understand the impact of using such analysis in order to reduce the false positive rate of test case generators. For this experiment we executed the analysis over the benchmarks of SPEC CPU2006, which is a benchmark suite containing real-world programs, such as GCC<sup>1</sup> and Bzip<sup>2</sup>. Since our analysis find pairs of parameters, we only considered functions containing at least two parameters, one of them being a pointer. The metrics of our interest, considering arrays that are received as parameter by functions, are:

1. The proportion of arrays with known sizes.
2. The proportion of array accesses performed over arrays with known sizes.

We have only analyzed functions which potentially have array sizes, i.e., functions containing at least one pointer and one scalar-type parameter. It gives us 7,586 functions in SPEC, that represent 18% of all functions. The forward analysis only found array sizes for two benchmarks, 6.9% for GCC and 0.5% for HMMER. In total, it only found 1.3% of all array sizes in SPEC. We have also analyzed the single-source and multi-source benchmarks present in LLVM's test-suite. In both benchmark suites, the forward analysis found almost 3% of the array sizes, in average. For some benchmarks, it could even reach an expressive rate: 50% for Shootout's *heapsort* and 66.6% for FreeBench's *analyzer*, for instance. Although these numbers are better than what we got for SPEC, it is still much less than what we expected.

In order to understand the ineffectiveness of our forward analysis, we randomly chose, for manual inspection, 12 SPEC functions which have array sizes found by the backward analysis but not by the forward analysis. Among the reasons, we were able to find four main patterns of code where our analysis cannot succeed.

1. **The size is calculated with no relation to the array allocation.** In this case there is nothing our forward analysis could do. We leave such cases for the backward analysis.

---

<sup>1</sup><https://gcc.gnu.org/>

<sup>2</sup><http://www.bzip.org/>

```

1: typedef struct network
2: {
3:   char inputfile[200];
4:   char clustfile[200];
5:   long n, n_trips;
6:   long max_m, m, m_org, m_impl;
7:   long max_residual_new_m, max_new_m;
8:
9:   long primal_unbounded;
10:  long dual_unbounded;
11:  long perturbed;
12:  long feasible;
13:  long eps;
14:  long opt_tol;
15:  long feas_tol;
16:  long pert_val;
17:  long bigM;
18:  double optcost;
19:  cost_t ignore_impl;
20:  node_p nodes, stop_nodes;
21:  arc_p arcs, stop_arcs;
22:  arc_p dummy_arcs, stop_dummy;
23:  long iterations;
24:  long bound_exchanges;
25:  long checksum;
26: } network_t;

```

```

1: long price_out_impl( network_t *net )
2: {
3:   [...]
4:   long new_arcs = 0;
5:   [...]
6:   arcnew = net->stop_arcs;
7:   [...]
8:   if( new_arcs < net->max_residual_new_m )
9:   {
10:    insert_new_arc( arcnew, new_arcs, tail,
11:                  head, arc_cost,
12:                  red_cost );
13:    new_arcs++;
14:   }
15:   [...]
16: }

```

a: The struct *network* contains several fields which are treated as arrays. For instance, the field *stop\_arcs* (line 21) is an array of another struct.

b: The array passed to function *insert\_new\_arc* (line 10) is the field *stop\_arcs* of struct *network*, while the array size (*new\_arcs*) is calculated independently. The lines containing "[...]" are only to suppress code.

Figure 3.4: A struct which encapsulates arrays.

2. **The array (and maybe its size) is encapsulated inside a struct.** Figure 3.4 shows a struct *network*, of the benchmark *429.mcf*, which encapsulates an array. It also shows a call of function *insert\_new\_arc*, of Figure 3.5, where the array size is calculated independently of the array size.
3. **The array is statically allocated (on the stack) but the size passed to the function is not the allocated size.** This pattern can be found on function *save\_call\_clobbered\_regs* when calling function *insert\_restore*, shown in Figure 3.6 (line 29). Notice that the array *save\_mode* is statically allocated but the size argument *regno* is dynamically calculated;
4. **The array is received as a function argument.** Although our analysis is inter-procedural, it does not propagate allocation information through functions.

Table 3.1 describes the reasons why our forward analysis did not relate sizes for the 12 SPEC functions.

```

1: void insert_new_arc(arc_t *new, long newpos,
2:                    node_t *tail, node_t *head,
3:                    cost_t cost, cost_t red_cost)
4: {
5:     long pos;
6:
7:     new[newpos].tail = tail;
8:     new[newpos].head = head;
9:     new[newpos].org_cost = cost;
10:    new[newpos].cost = cost;
11:    new[newpos].flow = (flow_t)red_cost;
12:
13:    pos = newpos+1;
14:    while(pos-1 && red_cost >
15:         (cost_t)new[pos/2-1].flow)
16:    {
17:        new[pos-1].tail = new[pos/2-1].tail;
18:        new[pos-1].head = new[pos/2-1].head;
19:        new[pos-1].cost = new[pos/2-1].cost;
20:        new[pos-1].org_cost = new[pos/2-1].org_cost;
21:        new[pos-1].flow = new[pos/2-1].flow;
22:
23:        pos = pos/2;
24:        new[pos-1].tail = tail;
25:        new[pos-1].head = head;
26:        new[pos-1].cost = cost;
27:        new[pos-1].org_cost = cost;
28:    }
29: }

```

Figure 3.5: The relation between *new* and *newpos* is not identified by the forward analysis.

### 3.1.2 Backward Size Analysis

The forward analysis is very precise for our goal and helps us to generate test cases which follow the patterns of real executions. However, the portion of arrays we could analyze compared to the total number of arrays is small. Among the reasons for such inefficiency, we have: arrays allocated in the stack using constant sizes and arrays encapsulated into structs. In this sense, we have used a backward size analysis for the cases where our forward analysis does not find a size to the array. The backward analysis was proposed by Alves et al. [2015a]. Basically, it extracts information about array accesses that have basic induction variables of a loop as offset. A variable is considered to be a basic induction variable of a loop if its redefinitions inside the loop represent an increment or a decrement of its value by a loop invariant. Usually, we call such variables *counters*. Thus, the backward size analysis makes use of symbolic range analysis [Nazaré et al., 2014] on induction variables that access an array to bound its size.

Let's get back to the example in the introduction, presented again in Figure 3.7 for better readability. Firstly, the analysis collects all (dereferences) of an array. In the

```

1: void save_call_clobbered_regs()
2: {
3:   struct insn_chain *chain, *next;
4:   enum machine_mode save_mode [FIRST_PSEUDO_REGISTER];
5:
6:   CLEAR_HARD_REG_SET (hard_regs_saved);
7:   n_regs_saved = 0;
8:
9:   for (chain = reload_insn_chain; chain != 0; chain = next)
10:  {
11:    rtx insn = chain->insn;
12:    enum rtx_code code = GET_CODE (insn);
13:
14:    next = chain->next;
15:
16:    if (chain->is_caller_save_insn)
17:      abort ();
18:
19:    if (GET_RTX_CLASS (code) == 'i')
20:    {
21:      if (n_regs_saved)
22:      {
23:        int regno;
24:
25:        [...]
26:
27:        for (regno = 0; regno < FIRST_PSEUDO_REGISTER; regno++)
28:          if (TEST_HARD_REG_BIT (referenced_regs, regno))
29:            regno += insert_restore(chain, 1, regno,
30:                                  MOVE_MAX_WORDS, save_mode);
31:      }
32:
33:      [...]
34:
35: }

```

Figure 3.6: A function which statically allocates an array.

example, the array  $A$  is accessed only with the induction variable  $i$ . Then, it calculates the bounds for induction variables using a symbolic range analysis. In this case, the symbolic range for  $i$  is  $[0, \max(0, size - 1)]$ . Finally, it applies the symbolic ranges to the access expressions to find the bounds of arrays ( $[0, \max(0, size - 1)]$  in this case). Since this analysis was originally used to disambiguate pointers, it is necessary to keep both the lower and the upper bound of an array. In our case, we get only the upper bound of arrays which are parameters of functions and check if any value in the expression is related to another function argument. For example,  $size$  itself is a function parameter. We then let  $size$  to be the size of  $A$ .

### 3.1.2.1 Evaluation

We executed the same experiments described in Section 3.1.1.1 in order to evaluate the effectiveness of the backward array size analysis. For SPEC CPU2006, the backward analysis found 33.9% of all array sizes. Figure 3.8 shows the number of arrays with

Function	Benchmark	Reason
S_qsortsv	400.perlbench	The array and size are received as function arguments and the function is not directly called.
BZ2_hbMakeCodeLengths	401.bzip2	The array is a struct field.
parse_options_and_default_flags	403.gcc	The array and size are received as function arguments.
insert_restore	403.gcc	The array is statically allocated.
merge_dependencies	403.gcc	The array is statically allocated and the size is calculated with no relation to the array allocation.
insert_new_arc	429.mcf	The array is a struct field.
swap_points_and_codes	445.gobmk	The array is received as a function argument and the size is dynamically calculated.
propose_edge_moves	445.gobmk	The array is statically allocated and size is dynamically calculated.
DisplayPlan7PostAlign	456.hmmmer	The array and size are received as function arguments.
PositionBasedWeights	456.hmmmer	The function is never called.
order_moves	458.sjeng	The array is a slice of a statically allocated buffer.
accumArrayUTF	483.xalancbmk	The array is a C++ string and the size is calculated with function length.

Table 3.1: Reasons for ineffectiveness of the forward analysis on randomly chosen functions.

```

1: int sum(int *A, int size) {
2:   int s = 0;
3:   for (int i=0; i < size; i++) {
4:     s += A[i];
5:   }
6:   return s;
7: }

```

Figure 3.7: A function which iterates over an array. This is the same example of Figure 1.1.

known sizes for each benchmark of SPEC. For the single-source benchmarks, which contain simpler programs, the backward analysis was surprisingly effective on them, finding the size of 74.5% of all arrays received as parameters. Particularly, for Polybench [Pouchet, 2012], one of the single-source benchmark suites which have polynomial

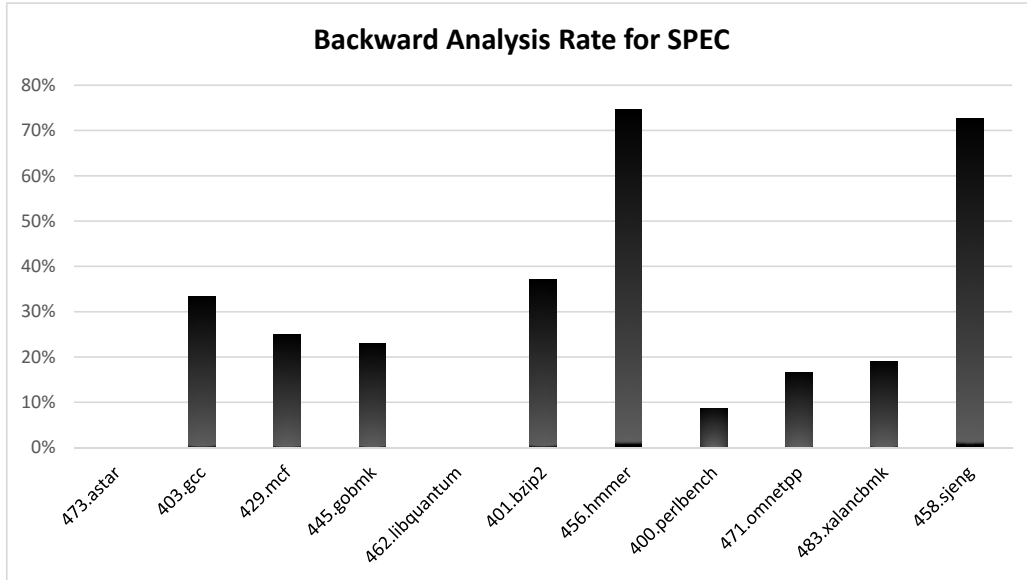


Figure 3.8: Percentage of arrays with known sizes.

functions that receive arrays and matrices as parameters, this analysis found almost 100% of the array sizes. For the multi-source benchmarks, this number is similar to what we got for SPEC: 39.8%. Together, the forward and the backward analyses were able to find 34.6% of all array sizes for SPEC. Figure 3.9 shows the proportion of array accesses that were performed over sized arrays. Together, for 33% of all considered accesses in SPEC, our analyses could find the size of the array, in average. It means that we may reduce the chance of finding false positive bugs due to array out-of-bound in up to 33% of the accesses.

## 3.2 Test Case Generation

In this section we describe a technique for random test case generation which uses the array size analyses of Section 3.1. We start presenting our data structure graph, which is used to represent a function’s interface, in Section 3.2.1. Section 3.2.2 describes the algorithm for input generation. Section 3.2.3 describes a slicing technique used to get the instructions of the original program that calculate array sizes, which are used by our test case generator.

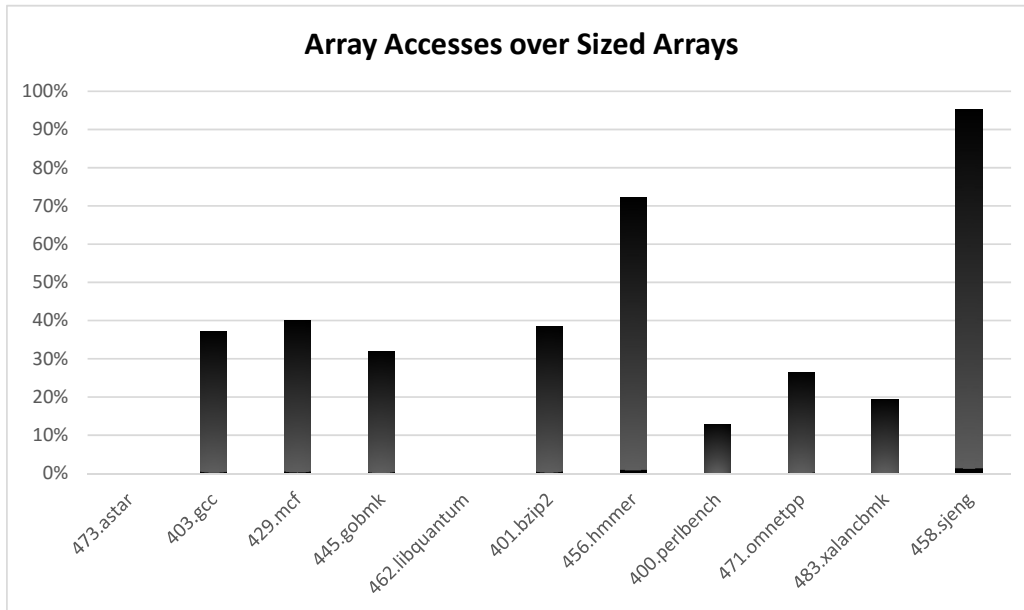


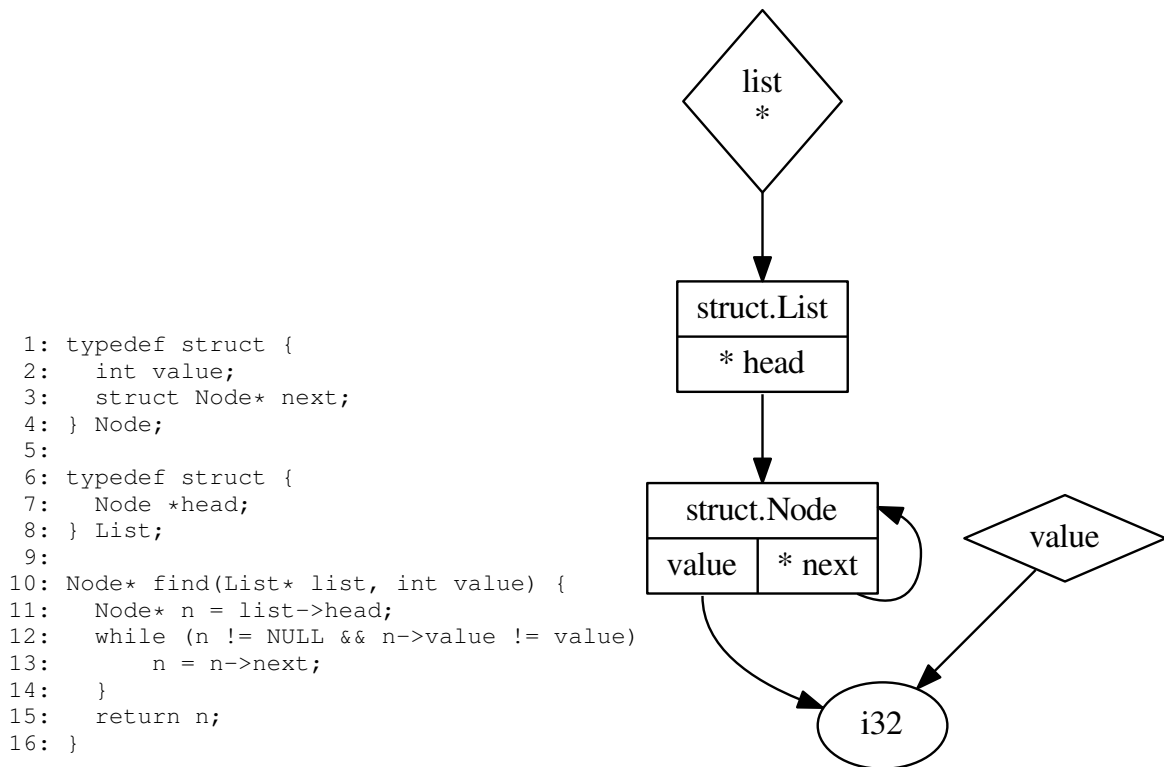
Figure 3.9: Percentage of array accesses performed over arrays with known sizes.

### 3.2.1 Data Structure Graph

A data structure graph is a possibly cyclic graph which contains information about a function’s inputs types. We consider as inputs the function’s parameters and the global variables which are read before being written inside the function. A data structure node may represent a type. Nodes may have a list of fields. The fields are basically edges between the owned node and a node which represents a contained type. For instance, Figure 3.10b shows the graphic representation of the data structure graph for the parameters of function `find` in Figure 3.10a. The diamonds represent the parameters, the square nodes represent structs and the round nodes represent primitive types (in this case, *i32*, i.e., a 32-bits integer). Notice that `Node` is a recursive struct, so the recursive field (`next`) points to the struct itself.

Fields also have attributes to indicate if it is a pointer or an array. Pointers are represented with a star (as shown in Figure 3.10b). In case of arrays, the field may have an attribute containing its size. If the array is statically allocated, the array size is a numeric value. If it is dynamically allocated, the array size is a pointer to another field representing a variable, if our analyses described in Section 3.1 is able to find a





a: C code snippet. Function `find` finds a node in a linked list.

b: Data structure graph of the parameters of function `find`.

Figure 3.10: A data structure graph example.

relation. Figure 3.11b shows an example of data structure graph with an array. The parameter pairs of function `closestPoint` of Figure 3.11a can be easily related to the parameter `size`. We graphically represent this relation by a dashed arrow. Having this relation in a graph helps us to generate test cases easier.

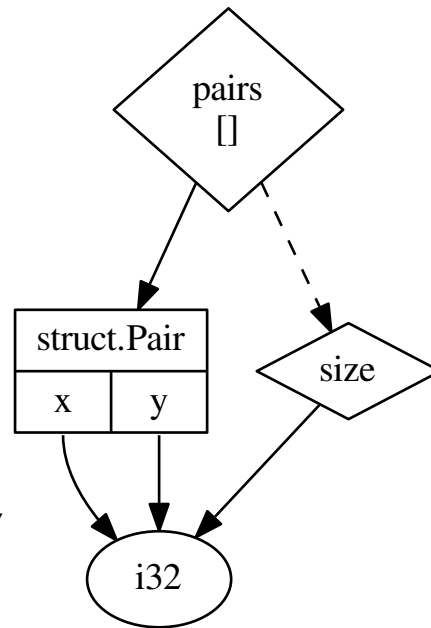
### 3.2.2 Input Generation

Many approaches for test case generation have already been proposed. Some of them focus on finding bugs faster and increasing code coverage [Godefroid et al., 2005a, 2008; Cadar et al., 2008; Chen et al., 2013; Ahmad and Oriol, 2014]. We focus on the generation of sound test cases, which do not introduce memory corruption bugs that were not present originally in the code. To accomplish this goal, we use the two static array size analyses presented in Section 3.1 to improve a black-box fuzzer. Our algorithm to generate random values for function inputs is similar to Godefroid et al. [2005b]. The most significant difference is regarding to arrays. First, we identify dynamically allocated arrays by looking for memory allocation instructions and pointer arithmetics. Also, we have two approaches to relate arrays with their sizes when

```

1: struct {
2:   int x;
3:   int y;
4: } Pair;
5:
6: void closestPoint(Pair *pairs, int size) {
7:   int min = INFINITY;
8:   Pair closest;
9:   for (int i=0; i < size; i++) {
10:    Pair p = pairs[i];
11:    int dist = sqrt(pow(p.x, 2)
12:                  + pow(p.y, 2));
13:    if (dist < min) {
14:      min = dist;
15:      closest = p;
16:    }
17:  }
18:  printf("Closest = (%d, %d)\n", closest.x,
19:         closest.y);
20: }

```



a: Function `closestPoint` finds the point of an array which is closer to the origin of a Cartesian coordinate plane.

b: Data structure graph of the parameters of function `closestPoint`. Notice that `size` is the size of `pairs`.

Figure 3.11: A graphic representation of an array size.

generating values for function arguments. When it is possible to relate array sizes using the forward analysis described in Section 3.1.1, we reuse program's instructions to calculate the array size and the related argument value. Such technique uses program slicing and is described in Section 3.2.3. When the relation is only captured by the backward analysis, we simply use the same random value as the array size and the argument related to the array. Notice that the second approach is more conservative.

Figure 3.12 describes our algorithm. The values are not generated in compile time. Instead, we instrument with instructions to generate such random values according to the input type. However, to simplify the reading, we decided to represent the input generation in a higher level. The function receives a field of the data structure graph as input. For the initial calls, this field represents a function input. From lines 3 to 11 the algorithm generates code to initialize pointers. It flips a coin to decide whether the pointer will be *null* or not. If it is not null, memory is allocated and the algorithm recurses to generate value for the pointed type. Lines 13 to 20 create arrays using the previously mentioned approach. Firstly, we try to get size information from the array size analyses of Section 3.1. If it is not possible, we generate a random value for it with no relation to another function argument. The function `create_array_factory` generates code to allocate *size* bytes in memory and fill it with random values of the correct type. Lines 22 to 26 creates structs by allocating the proper amount of memory

and recursively calls the function for each struct field. Notice that this algorithm is able to generate recursive structs with random sizes, because of the combination of the pointer and struct generation. Lines 28 to 33 only generate values for primitive types. However, if the field is known to represent a size of an array, we reuse the already computed value in order to make it consistent.

```

1: generate_value(field):
2:     node = field.get_node()
2:     if (field.is_pointer() && !field.is_array()):
3:         probability = random_float()
4:         if (flip_coin() < probability):
5:             return NULL
6:         else:
7:             m = malloc(sizeof(node.get_type()))
8:             field.set_pointer(false)
9:             v = generate_value(field)
10:            *m = v
11:            return m
12:     else if (field.is_array()):
13:         factory = create_array_factory(node.get_type())
14:         if (forward_analysis.found_size(field)):
15:             slice = get_slice(field)
16:             size = slice.get_size()
17:         else if (backward_analysis.found_size(field)):
18:             size = backward_analysis.get_size(field)
19:         else:
20:             size = random_int()
21:         array = create_call(factory, size)
22:     else if (node.get_type().is_struct()):
23:         m = malloc(sizeof(node.get_type()))
24:         for (s_field : node.get_fields()):
25:             v = generate_value(s_field)
26:             *(m + s_field.get_offset()) = v
27:         return m
28:     else:
29:         if (forward_analysis.is_size(field)):
30:             slice = get_slice(field)
31:             return slice.get_size()
32:         else if (backward_analysis.is_size(field)):
33:             return backward_analysis.get_size()
34:     return random_value(node.get_type())

```

Figure 3.12: The input generation algorithm.

### 3.2.3 Slicing Technique

We have implemented two static analysis to bind an array argument with the argument representing its size - if existing. With this information in hand, we are able to use the correct values in a function call and avoid memory corruption errors that should not happen. However, in order to generate test cases which better reproduce the real context in which functions are called, we want to use - whenever possible - instructions present in the program to compute an array size. To do that, we have implemented a program slicing approach. Figure 1.1 shows as example a snippet of C code which

creates an array and calls a function `f`. To recreate the array size and the argument, we produce two slices. The first one focuses on calculating the array size - which in this case is the argument of the `malloc` call. The second focuses on the argument of `f` which is related to the array size.

```
1: int size = x * 2;
2: int *A = (int*) malloc(sizeof(int)*size);
3: for (int i=0; i < size; i++) {
4:   A[i] = rand();
5: }
6: sum(A, size/2);
```

Figure 3.13: A function which iterates over an array. This is the same example of Figure 1.1.

To illustrate how the slicing technique works, we will use the code snippet in Figure 3.13, which creates an array of size  $x*2$  filling it with random values. Figure 3.14 shows the LLVM's intermediate representation of the code snippet. We are interested on the instructions which calculates the argument of `malloc` (underlined instructions in the intermediate representation) and argument of `f` which is related to the size of the array (bolded instructions in the IR). Note that the first instruction is present in both slices. As result, we get the union of the two slices and use the calculated values to create arrays of random sizes and call the target function `sum`. Figure 3.15 shows the resulting code produced by our input generator using the program slice of the code snippet. Notice that `x` is assigned with the result of a random integer, because `x` is the first variable with only one use, then treated as input for the slice. The function `fillArrayi32` is automatically created by our input generator.

### 3.3 Conclusion

In this chapter we have presented a novel approach for automatic generation of test cases, which makes use of static analyses that bind arrays with function arguments that represent their sizes. The use of such analyses helps our approach to generate safer test cases than existing techniques. Although the forward analysis have shown to still be ineffective (due to several reasons), the backward analysis is very effective on binding meta information about memory regions. We suppose that, despite the limitations of the forward analysis, using our array size analyses can consistently reduce the false positive rate on the state-of-the-art techniques for automatic test case generation. Our test case generator is also able to generate recursive data structures, such as lists

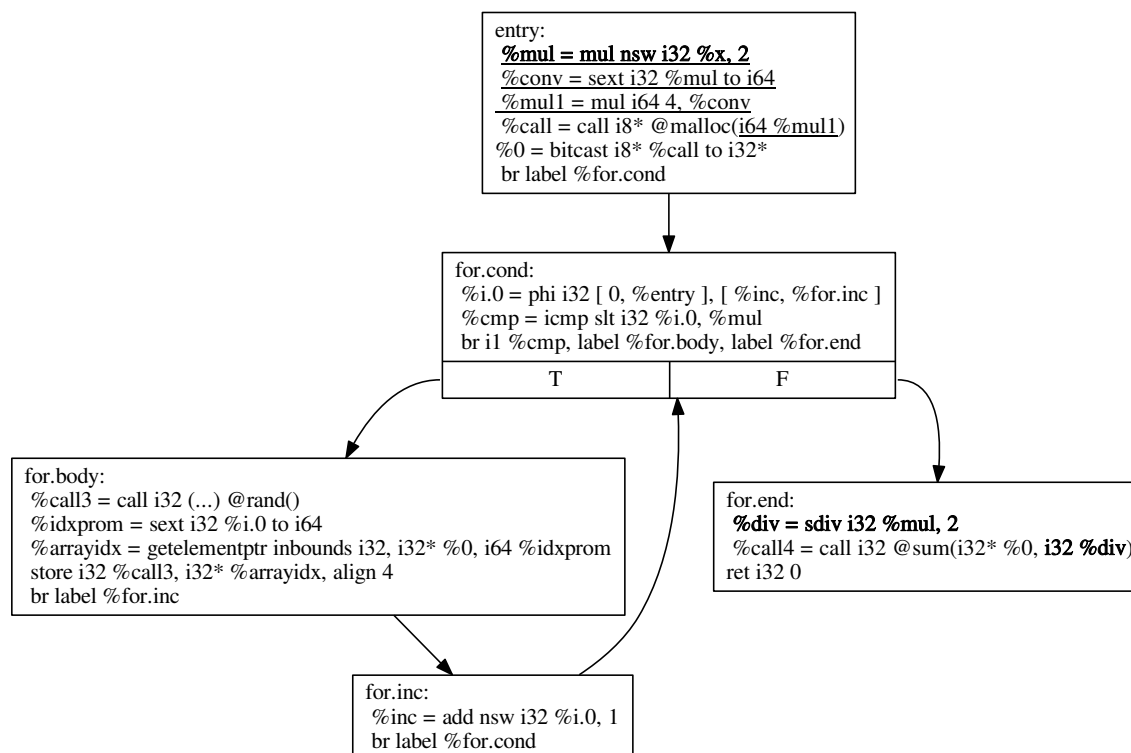


Figure 3.14: The LLVM's IR of the code snippet in Figure 3.13.

```

1: %x = i32 call @rand_i32();
2: %mul = mul nsw i32 %x, 2
3: %conv = sext i32 %mul to i64
4: %mull = mul i64 4, %conv
5: %call = call noalias i8* @malloc(i64 %mull)
6: %a = bitcast i8* %call to i32*
7: call void @fillArrayi32(i32* %a, i64 %mull)
8: %div = sdiv i32 %mul, 2
9: call void @sum(i32* %a, i32 %div)

```

Figure 3.15: Code produced by the input generator to test function sum. Lines 2, 3, 4 and 8 are produced by the slice for the code snippet in Figure 3.13.

or trees. We also discover pointers that are used as arrays by looking for pointer arithmetics.



# Chapter 4

## Case Study

In this chapter, we describe Asymptus, a tool for automatic inference of loop complexity, which was developed during this master’s research as a preliminary work. Asymptus has shown to be very precise to analyze polynomial functions. However, in order to be analyzed, a function has to be executed a certain number of times with different inputs. It may not happen in real world applications due to several reasons. For example, the inputs of a function may not change regardless the initial inputs of the program (e.g. hardcoded arguments). We have used our test case generator in order to overcome this limitation and to show the efficiency of our technique to generate valid inputs for functions.

Complexity analyses show how algorithms scale as a function of their inputs. Its importance stems from the fact that such a technique helps program developers to uncover performance bugs which are hard to find. In addition to this, complexity analysis supports the decision of offloading or not computation to the cloud or GPU. Finally, this kind of technique has implications to the theoretical computer science community, as it provides data that corroborate the formal asymptotic analysis of algorithms. Given this importance, it comes as no surprise that, since the 70s [Wegbreit, 1975], large amounts of effort have been spent in the design and improvement of empirical methodologies to infer code complexity.

Over the time, different static approaches were proposed to analyze programs in functional [Wegbreit, 1975; Le Métayer, 1988; Debray and Lin, 1993] and imperative [Gulavani and Gulwani, 2008; Gulwani et al., 2009b,a] languages. Although the static approaches have the benefit of running fast and may give correct upper bounds, this methodology has shortcomings. Static analyses may yield imprecise – or even incorrect – results. This imprecision happens due to the inherently inability of purely static approaches to capture the dynamic behavior of programs. In order to circumvent

this limitation of static approaches, the programming language community has resorted to profiling-based methodologies [Goldsmith et al., 2007; Zaparanuks and Hauswirth, 2012; Coppa et al., 2012]. However, even these dynamic techniques are not free of limitations.

The main drawback of a profiling-based complexity analysis is the fact that it is usually ineffective to relate the symbols in the program text to the result that it delivers. For instance, the state-of-the-art tool in this field is `aprof` [Coppa et al., 2012]. `Aprof` furnishes programmers with a table that relates input sizes with the number of operations performed. This *modus operandi* has two problems, in our opinion. First, the input is provided as a number of memory cells read during the execution of a function. This number may not be meaningful to the programmer, as we will clarify in Section 4.1. Second, it works at the granularity of functions. However, developers are often more interested in knowing the computational complexity of small regions within a function. Such regions can be, for instance, performance-intensive loops. Our technique addresses these two limitations of input sensitive profiling.

The main contribution of our work is a novel hybrid technique to perform complexity analysis on imperative programs, which we describe in Section 4.2. Our technique is hybrid because it combines static analysis with dynamic profiling. First, we use static analysis to determine loop inputs and to find algebraic relations between these loops. Then, we use a dynamic profiler, plus polynomial interpolation, to infer the complexity of each loop in a function. Our technique is capable of generating symbolic expressions that denote the complexity of each loop, instead of the whole function. Furthermore, we combine and simplify these expressions to make them even more meaningful to the software engineer. We believe that this granularity can help developers to have a deeper understanding of a function’s behaviour; hence, it provides them with the means to detect and solve performance bugs more efficiently. We also show that our technique is simpler than previous work while producing more useful results.

We have designed, tested, and implemented a tool on top of the LLVM compilation infrastructure [Lattner and Adve, 2004b] to infer, automatically, the complexity of loops within programs. We ran our tool over the Polybench [Pouchet, 2012] and Rodinia [Che et al., 2009] benchmark suites. Our results indicate that we are capable of correctly inferring the complexity of 99.7% of the Polybench loops and 69.18% of the Rodinia loops. All the equations that we output, as explained in detail in Section 4.1, are written as functions of the symbols, i.e., variable names, present in the program code – that is an improvement on top of `aprof` and similar tools. Moreover, we have found that 38% of all functions in the benchmarks that we analyzed have at least two



independent loops. In this case, tools that only report complexity information for entire functions may miss important details about the asymptotic behaviour of smaller regions of code.

## 4.1 Overview

In this section, we give an overview of the challenges Asymptus addresses. Figure 4.1 shows the example we will use to illustrate our technique. Function *multiply* is a routine that performs matrix multiplication of two square matrices. For pedagogical purposes, our function does not return the resulting matrix; instead, it prints the result. We chose to implement the function in such a way to show how our technique behaves on functions with multiple loops.

As developers, we would like to know the computational cost to execute this function. For instance, knowing the complexity of each part of the target function, we can find out performance bottlenecks and improve its implementation. Looking at the *multiply* function we can easily identify the linear behavior of the loop on line 4 and the cubic behavior of the nested loops beginning at line 7. However, a quick visual inspection on the loop at line 18 may not capture its quadratic complexity.

We can use profilers to find out where the program is spending most of its resources. However, traditional tools lack the ability to show how the program scales as a function of its inputs. For instance, Figure 4.2 shows the output that Gprof [Graham et al., 1982] – the most well-known profiler in the Unix systems – produces for our example. This profiler does not give us any information regarding the asymptotic complexity of the program in Figure 4.1. Instead, it produces a table describing where the program spends more time during its execution.

There exist profilers that have been designed specifically to provide developers with an idea about the asymptotic complexity of programs [Goldsmith et al., 2007; Zapanuks and Hauswirth, 2012; Coppa et al., 2012]. Nevertheless, *aprof* [Coppa et al., 2012], the state-of-the-art approach in this field, is also not very useful in this example. For instance, only looking at Figure 4.3, which shows *aprof*'s results for the function *multiply*, the user may not fully understand about the function behaviour: this table shows numbers, but do not relate these numbers with symbols in the program text. Moreover, the complexity curve seems to be linear, since *aprof* considers the whole matrices as inputs ( $n^2$ ) – usually, developers describe asymptotic complexity in terms of the matrices dimensions ( $n$ ). Finally, the result generated by *aprof* describes the whole function. We believe that this granularity is too coarse, because it makes it

```

1: void multiply(int **matA, int **matB, int n){
2:     int i, j, k, sum;
3:     int **result = (int**) malloc(n * sizeof(int*));
4:     for (i = 0; i < n; i++)
5:         result[i] = (int*) malloc(n * sizeof(int));
6:
7:     for (i=0; i < n; i++) {
8:         for (j=0; j < n; j++) {
9:             sum = 0;
10:            for (k=0; k < n; k++) {
11:                sum += matA[i][k] * matB[k][j];
12:            }
13:            result[i][j] = sum;
14:        }
15:    }
16:
17:    j = 0;
18:    for (i = 0; i < n;) {
19:        if (j >= n) {
20:            j = 0;
21:            i++;
22:            printf("\n");
23:        } else {
24:            printf("%8d", result[i][j++]);
25:        }
26:    }
27:    printf("\n");
28: }

```

Figure 4.1: Matrix multiplication – the running example that we shall use to explain our contributions.

index	% time	self	children	name
[1]	100.0	0.00	0.03	main [1]
		0.03	0.00	multiply(int**, int**, int) [2]
		0.00	0.00	initArray(int**, int, int) [9]
		0.00	0.00	free_all(int**, int**, int) [10]
-----				
		0.03	0.00	main [1]
[2]	100.0	0.03	0.00	multiply(int**, int**, int) [2]
-----				

Figure 4.2: Gprof output for a simple program containing our example function.

very difficult for the user to verify the behavior of particular parts of the function.

We can do better: the technique that we describe in this chapter produces one

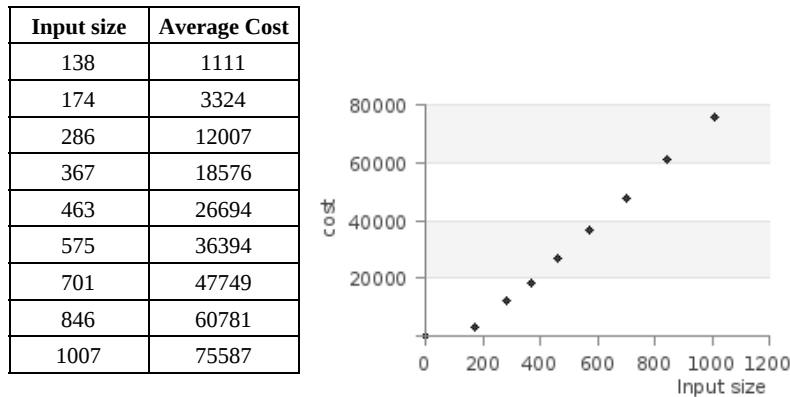


Figure 4.3: The output produced by the aprof input sensitive profiler.

polynomial for each loop in the function. These polynomials range on symbols defined in the program text, e.g., the names of variables. Therefore, we claim that our output is clearer to the developer. For instance, considering the loop in line 7, we will state – automatically – that its complexity polynomial is:  $n + 1$ . Furthermore, considering the loop nest starting in line 18, we produce the following equation to denote its complexity polynomial:  $n^2 + n + 1$ .

Our result is on a finer granularity, so we can combine them to generate an equation that expresses the asymptotic behavior of the whole target function. For the function in the Listing 4.1, our approach generates the following simplified equation, in big O, to denote the function’s complexity:

$$O(n^3)$$

We claim that this notation, which uses the names of variables present in the program, is more meaningful to the application developer than the output produced by traditional profilers, such as *gprof* or *aprof*.

## 4.2 Automatic Inference of Loop Complexity through Polynomial Interpolation

We can describe our technique in four main steps: (1) static analysis, (2) code instrumentation, (3) dynamic information extraction and (4) polynomial interpolation. In this section we describe each one of these steps. However, before delving into the details of our technique, we shall introduce some notation, which will guide our explanations henceforth.

**Loop Jargon.** Let  $S$  be a subset of nodes of a control flow graph  $G$ .  $S$  contains a special node  $H$ , which we shall call *header*, or *entry point*. Following Appel and Palsberg [Appel and Palsberg, 2002, pp.376], we say that  $S$  is a *natural loop* if, and only if, it presents the following three properties:

1. there exists a path from any node in  $S$  to  $H$ ;
2. there exists a path from  $H$  to any node in  $S$ ;
3. there is no path from a node of  $G$  to a node of  $S$  that does not go across  $H$ .

The last property defines  $S$  as a *single-entry* region, following Ferrante's nomenclature [Ferrante et al., 1987]. An edge between any node in  $S$  to  $H$  is called a *back-edge*. We adopt Wolfe's definition of *trip count* [Wolfe, 1996, pp.200]: the number of times any back-edge of a natural loop has been traversed by the program flow within a single execution of the loop. Hence, a loop that exits the first time it is executed has a trip count of zero. The number of times  $H$  is visited is one more than the trip count of the loop. We estimate the *complexity* of a loop as the product of its trip count by the number of operations in its longest path.

We call a node  $L \in S$  a *latch*, or *exit point*, if there exists an edge from  $L$  to a node  $N$ ,  $N \in G$ ,  $N \notin S$ . We say that  $L$  is a *natural latch* if one of these two conditions applies:

- $L = H$ . In this case we have a *while loop*;
- $L \neq H$ , and any edge from  $L$  either leaves  $S$  or leads to  $H$ . In this case we have a *repeat loop*.

If  $S$  contains only one latch, then we call it *single exit*. In this work we consider multiple exit loops featuring only one natural latch. Code generated from typical programming language constructs, i.e., `for`, `while` and `repeat` has this property, as long as the command `goto` is not used.

Any latch contains a *stop condition*: a boolean expression whose evaluation either keeps the program flow in  $S$  or leads away from it. If the natural latch contains a stop condition that uses only one operator, which can be either  $<$ ,  $\leq$ ,  $>$  or  $\geq$ , then we call  $S$  an *interval loop*. We let the operands of the stop condition be the *limits* of the interval. For instance, in the interval loop `for (i = 0; i < N; i++)`, we have the stop condition  $i < N$ , whose limits are  $i$  and  $N$ . Our technique handles any loop with only one input, and interval loops with up to two inputs  $i_1$  and  $i_2$ . In this case, we consider as the input size the difference  $|i_1 - i_2|$ .

### 4.2.1 Input Analysis

We start the process of inferring the complexity of code with a static analysis phase. The static analysis determines the inputs of each loop in the function. We qualify as *loop input* any data that:

- influences the stop condition of the loop; and,
- is not defined within the loop.

For instance, the loop at line 7 in Figure 4.1 is controlled by  $i < n$ . Variable  $i$  has two definitions: one outside the loop, which we shall call  $i_0$ , and another inside, which we shall call  $i_1$ . The former is initialized with the constant zero, which is thus considered a loop input. Variable  $n$  is a parameter of the function; hence, it is considered a symbolic input. Therefore, the two inputs of the loop that exists at line 7 are  $\{0, n\}$ . Concretely, we detect inputs through a *backward* analysis, that starts at the variables used in the loop's stop condition, and ends at the definitions of variables that lay outside the loop body. To determine the complexity of a loop, we will plot the number of operations executed by the loop for each value bound to one of its inputs that we have observed during a profiling step. We shall describe this profiling in Section 4.2.3

### 4.2.2 Loop Dependence Analysis

Our profiler outputs the complexity of all the loops within a program. We must combine this information to have a snapshot of the program's complexity. However, combining the complexity of all the loops that constitute a program is not a straightforward problem. One of the main difficulties that we face in this case is how to deal with loops that may, or may not, execute, depending on the path that the program follows. In order to provide meaningful answers to the user, we propose an algebra to simplify the equations that we produce. Our algebra has three operators: *plus* ( $+$ ), *times* ( $\times$ ) and *expander* ( $\oplus$ ). The *plus* and *times* operators have the usual semantics of asymptotic analysis. The expander was proposed by us as an alternative to describe the complexity of code that may or may not execute, depending on the program's flow. Its semantics is defined in the equations 4.1 and 4.2:

$$O(x^a \oplus y^b) = O(x^a) + O(x^b), \quad \{a, b\} \in \mathbb{N} \quad (4.1)$$

$$\Omega(x^a \oplus y^b) = \Omega(x^a), \quad \{a, b\} \in \mathbb{N} \quad (4.2)$$

```

1: void printDups(std::vector<std::string> lines, std::string key) {
2:     std::vector<std::string> result;
3:     for (int i=0; i < lines.size(); i++) {
4:         if (lines[i].find(key) != std::string::npos) {
5:             result.push_back(lines[i]);
6:         }
7:     }
8:
9:     if (result.empty()) return;
10:
11:     // find dups in a naive way
12:     for (int i=0; i < result.size()-1; i++) {
13:         for (int j=i+1; j < result.size(); j++) {
14:             if (i != j && result[i] == result[j])
15:                 std::cout << result[i] << std::endl;
16:         }
17:     }
18: }

```

Figure 4.4: A function to print duplicate lines containing a given key. The second loop has a conditional execution.

As a reminder, the big-Omega notation indicates a lower asymptotic bound:  $\Omega(f)$  denotes a function whose growth is less than or equal to the growth of  $f$ . Expansion denotes the complexity of code that executes conditionally. Figure 4.4 provides an example of a situation where the expander operation is useful. The function *printDups* prints the duplicate lines containing a given substring in a naive way. Because of the conditional branch in line 9, the loop starting on line 12 may or may not execute. Because of this, the complexity of this function is  $\Omega(n)$  - best case, when no line contains the key - and  $O(n^2)$ , where  $n$  is the size of the vector. If  $C(L)$  denotes the asymptotic complexity of a given code region, then we let  $C(\textit{printDups}) = C(L_{3-7}) \oplus C(L_{12-17}) = O(n \oplus n^2)$ , where  $L_{3-7}$  is the loop at lines 3 to 7 in Figure 4.4, and  $L_{12-17}$  is the loop at lines 12 to 17.

As usual, addition and multiplication in the big-O notation are associative and commutative. Multiplication is also distributive with regard to addition. On the other hand, *expansion* is only associative, due to Equation 4.2. These properties let us use typical simplification rules to provide users of our tool with more palatable results. Notice, once again, that expansion is non-commutative, and simplification only applies if the first operand has higher complexity than the second:

$$\frac{C(L) = O(x^a) + O(x^b), a \geq b}{C(L) = O(x^a)}$$

$$\frac{C(L) = O(x^a) \times O(x^b)}{C(L) = O(x^{a+b})}$$

$$\frac{C(L) = O(x^a) + O(x^b), a < b}{C(L) = O(x^b)}$$

$$\frac{C(L) = O(x^a) \oplus O(x^b), a \geq b}{C(L) = O(x^a)}$$

The simplification process is guaranteed to terminate, as it always reduces the size of the resulting expression. Looking back to Figure 4.1 it is easy to see that the complexity is  $C(\text{multiply}) = C(L_{4-5}) + C(L_{7-15}) \times C(L_{8-14}) \times C(L_{10-12}) + C(L_{18-26})$ , which gives us:  $O(n + n * n * n + n^2)$ . Using the above equations we can recursively simplify this expression. Firstly, we can simplify  $n * n$  with  $n^2$ . We have now  $O(n + n^2 * n + n^2)$  and we can use the same rule to simplify the remaining multiplication, resulting in  $n^3$ . It is easy to see that we can use the two rules of *plus* to simplify the two additions. Then, the resulting complexity is  $O(n^3)$ , as expected. Notice that  $n$  is a symbol produced by the input analysis of Section 4.2.1.

### 4.2.3 Code Instrumentation

We infer the complexity of code by analyzing profiling data. We produce this data through code instrumentation. To be able to extract dynamic information, we instrument the target program to output: (i) the values of the loop inputs immediately before the loop execution and (ii) the number of operations performed by each loop. Loop inputs are determined by the analysis seen in Section 4.2.1. The execution cost is measured in terms of instructions executed. We have implemented this instrumentation framework within the LLVM compiler infrastructure.

Care must be taken with regard to loops with multiple paths. Different paths may yield different costs, a fact that could hinder our interpolator from finding a perfect polynomial fit. Figure 4.5 illustrates this shortcoming. The program seen in part (a) of the figure contains two loops, at lines 2 and 4. The loop at line 4 contains two execution paths. Let's assume that during execution, our profiler has observed that for  $M = 1$ , that loop executed 44 instructions, and for  $M = 2$ , it always took the cheapest path; hence, executing  $3 + 3$  operations. These points,  $(1, 42)$ ,  $(2, 6)$  would confuse our interpolator, which expects more operations for larger inputs. To avoid this problem, we consider that the cost of a loop is determined by its path of highest cost, which we estimate statically. To obtain a conservative estimate of this path, we resort to a modified version of Dijkstra's algorithm, to solve the single-source largest path problem for an acyclic graph with non-negative weights assigned to edges [Dijkstra, 1959]. To build an acyclic graph, we consider all the paths from the loop header  $H$  to its natural

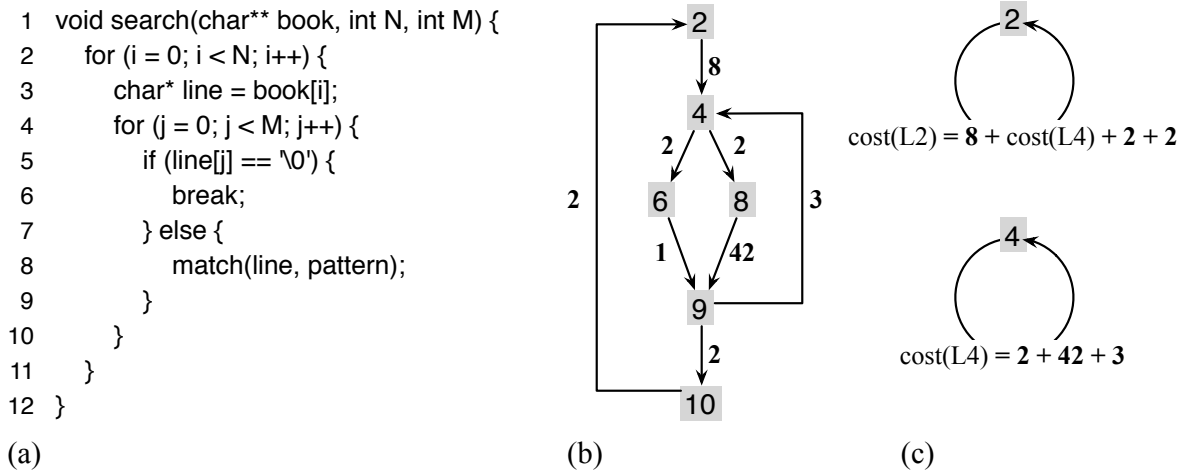


Figure 4.5: (a) Program with a multi-path loop. (b) The cost-graph of the program. Nodes represent program points and the edges' weights represent the number of executed instructions between two points. (c) The cost of each loop iteration.

latch  $L$ .

Once we have determined – statically – the cost of a loop iteration, we instrument it. To this end, we create a counter variable at the loop's header, and increment it by the estimated cost. Notice that incrementing this counter at the loop header will account for one more iteration than the real execution. Nevertheless, it will not affect our cost analysis. We chose to do it like this because the loop header is unique, and is always executed, independent on the way the program flows within the loop body. Figure 4.5 (c) shows the cost expressions that we create for each loop. In the figure, edges represent paths within the loop, and the nodes are the headers of those loops. Each one of these values is added once per iteration of the loop. Once we have instrumented the program, we execute it. As mentioned before, each execution of an instrumented program outputs the values of each loop input, together with the number of operations executed within that loop.

#### 4.2.4 Polynomial Interpolation

We log the output of our profiler and parse it to extract pairs: input value  $\times$  execution cost. With these points, we execute a polynomial interpolation method to find the curve that best fits into this set. Our interpolation works as follows: we test different polynomials, starting from a line (degree 1) upwards until  $n - 1$ , where  $n$  is the number of points available. At step  $i$  we need  $i + 1$  points to determine a polynomial. Any group of  $i + 1$  different points fits this purpose. We call this group of points the *guiding set*.



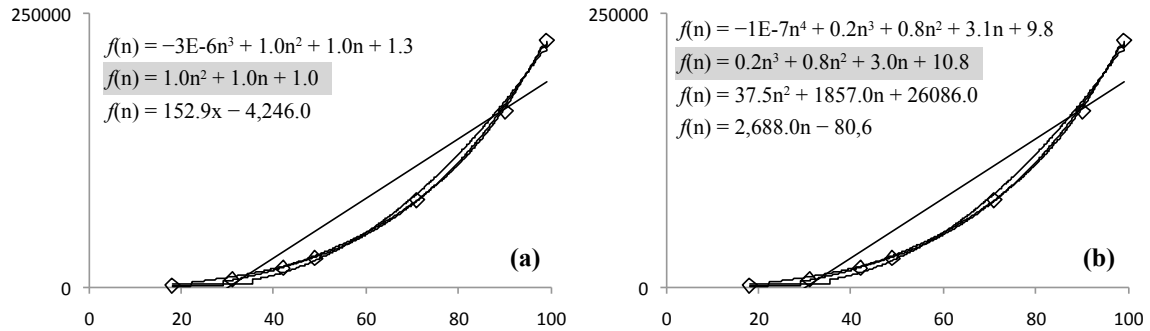


Figure 4.6: (a) Polynomials found for the loop at lines 18-25 of Figure 4.1. (b) Polynomials found for the loop nest at lines 7-15. In each figure, the first curve that fits the points in the verification set is marked in gray.

We use the points that are left to check if we have found the correct polynomial. These remaining points are called the *verification set*. We stop interpolation if, upon finding a polynomial  $p$ , of degree  $k$ ,  $k < n - 1$ , we notice that the  $n - k$  points in the verification set fit perfectly into  $p$ . Our interpolation only works for single-variable polynomials, but we can infer the complexity of nests of loops by multiplying symbolically their individual complexities.

Figure 4.6 illustrates this process for the program seen in Figure 4.1. The figure has two blocks of loops; thus, we produce two polynomials. Let us take a deeper look into the polynomial that we produce for the loop that exists at lines 18-25 of Figure 4.1. This curve is shown in Figure 4.6 (a). In this example, we assume that we have obtained, after profiling the program with eight different inputs, the following pairs of size  $\times$  cost: (13, 183), (50, 2,551), (72, 5,257), (80, 6,481), (98, 9,704), (115, 13,341), (139, 19,461). To derive a polynomial that describes the complexity of this loop, we try to interpolate a line across those points using, as our guiding set, only the first two pairs, e.g., (13, 183) and (50, 2,551). This line does not contain the other six points, which form the verification set. Thus, we move on to try a polynomial of degree two, this time, adding also the pair (72, 5,257) to our guiding set. The new polynomial,  $n^2 + n + 0.8$  contains the points in our verification set. Hence, we let it denote the computational cost of the loop. The complexity of the loop is then  $O(n^2)$ , where  $n$  is the only symbolic input of the loop under analysis, as we have explained in Section 4.2.1. We perform similar process to discover the polynomial that characterizes the loop nest at lines 7-15 of Figure 4.1. However, this time our search stabilizes in a third-degree polynomial. Figure 4.6 (b) shows this curve.

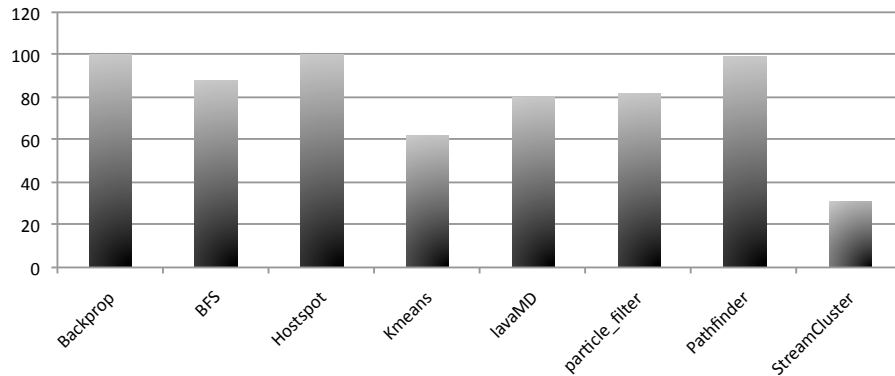


Figure 4.7: Percentage of loops per benchmark of Rodinia that we could analyze. The correctness of all these results have been checked manually.

### 4.3 Evaluation

To examine the real applicability of our technique, we have implemented it as a prototype tool. We have used the LLVM compilation infrastructure to perform the static analysis and code instrumentation phases mentioned in Sections 4.2.1, 4.2.2 and 4.2.3. The goal of the experiment is to find out how effective is the technique when applied to the loops found in real-world programs. We have executed Asymptus on the Polybench Pouchet [2012] and Rodinia Che et al. [2009] benchmark suites. We have checked, manually, the answers produced by our tool for every loop in these benchmarks. This exercise shows that we are able to correctly analyze 99.7% of the loops in Polybench. The remaining 0.3% is due to a single loop which is constant for the first two points, and varies for larger inputs. This behavior makes it impossible for us to get a perfect polynomial match. For Rodinia – a much bigger and general benchmark suite – our tool correctly analysed 63.58% of the loops. However, the execution flow never reached some functions during our profiling phase so we could not generate data for them. If we ignore those functions, our success rate increases to 69.18%.

Figure 4.7 shows the percentage of loops that we could analyze per Rodinia benchmark. We do not show a chart for Polybench, because we believe that this chart is not interesting. It would have almost only bars at 100% of precision. Our results are worse for Rodinia because of four reasons: (1) some loops are not polynomial, but we use a polynomial interpolation; (2) some loops iterate over structures that our technique does not handle, such as strings or files; (3) some loops have 3 or 4 inputs that bound their execution; and (4) some functions are not executed enough times with different inputs to enable the interpolation. The goal of this experiment is to help

Asymptus to overcome the 4<sup>th</sup> limitation.

### 4.3.1 An LLVM Pass

We have decided to implement the techniques presented in Chapter 3 on top of the LLVM compiler infrastructure Lattner and Adve [2004a]. LLVM is composed by a set of modular and reusable libraries and comes with several tools to compile, analyze and optimize code. It provides an SSA-based compilation strategy, which makes it easier to analyze and modify programs in the LLVM's intermediate representation. Also, writing a pass to LLVM is very simple. It provides a well-designed API with full support to navigate over the instructions, collect static information and modify the program. A pass is a small program which runs in an optimization/analysis pipeline. Figure 4.8 shows a function written in C and its corresponding intermediate representation generated by Clang 3.7 (the LLVM's front end for C). It also shows a code snippet exemplifying how to iterate over the instructions of a function.

<pre> 1: unsigned fib(unsigned n) { 2:   if (n &lt;= 1) 3:     return n; 4:   return fib(n-1) + fib(n-2); 5: }</pre>	<pre> define i32 @fib(i32 %n) #0 { entry:   %cmp = icmp ule i32 %n, 1   br i1 %cmp, label %if.then, label %if.end  if.then:   br label %return  if.end:   %sub = sub i32 %n, 1   %call = call i32 @fib(i32 %sub)   %sub1 = sub i32 %n, 2   %call12 = call i32 @fib(i32 %sub1)   %add = add i32 %call, %call12   br label %return  return:   %retval.0 = phi i32 [ %n, %if.then ],                   [ %add, %if.end ]   ret i32 %retval.0 }</pre>
<p>a: A function written in C.</p> <pre> 1: for (auto &amp;BB : F) { 2:   for (auto &amp;I : BB) { 3:     if (isa&lt;CallInst&gt;(&amp;I)) 4:       I.dump(); 5:   } 6: }</pre>	<p>c: The function <code>fib</code> in LLVM's IR.</p>
<p>b: Code snippet of an LLVM pass which prints all the call instructions in a function.</p>	

Figure 4.8: An example of LLVM's IR and a code snippet of an LLVM pass.

We have implemented a set of LLVM passes for our analyses and the input generator. The first step was to design the data structure graph, and its construction algorithm, in a way to have all the information needed and to be easy to manipulate. Then, we designed the algorithm to generate inputs and the data-flow equations for the forward array size analysis. With the design completed, we started implementing the LLVM passes. The main pass is the input generator. It depends on other analyses and implements our algorithm. It is also responsible to get the program slices presented in

Section 3.2.3, since it is part of the input generation. Some simple analyses (such as the graph creation and the forward size analysis) were not implemented as passes. Instead, we just created C++ classes and invoked methods inside the main pass. Figure 4.9 shows a high-level architecture of our implementation.

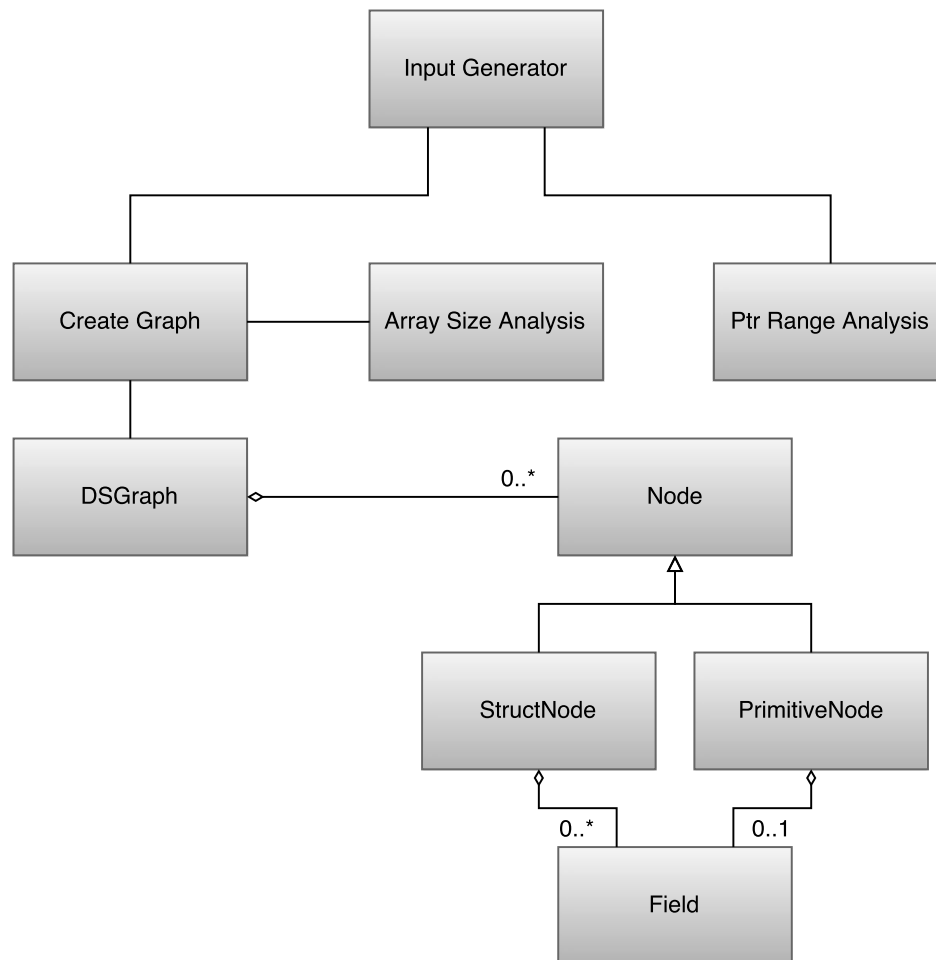


Figure 4.9: The architecture of our implementation.

*Input Generator* is the main module, responsible for the generation of test cases. In order to get the interface of functions, it uses *Create Graph*. This module is responsible for the creation of a data structure graph and also uses *Array Size Analysis* which implements our forward analysis. The backward analysis is implemented by *Ptr Range Analysis*. The other modules represent the structure of our data structure graph. The graph is composed by *Nodes*, which may be either a *StructNode* or a *PrimitiveNode*. *StructNodes* may have several *Fields*, while the *PrimitiveNodes* may have only one.

### 4.3.2 Experiment

In order to understand the applicability of our input generator on real-world systems, we made use of Asymptus. As mentioned before, one of Asymptus' limitations is that it has to execute a function a certain number of times with different inputs in order to be able to analyze it. For instance, we have executed Asymptus over the benchmarks of Rodinia and found that there is a function which is always executed with the same hardcoded input, regardless the inputs provided to the program. Since the technique describe in this dissertation is able to generate random inputs for specific functions, we decided to see whether it could be used to help Asymptus' on analyzing particular functions.

For this experiments, we have implemented 10 functions receiving arrays, matrices and recursive data-structures as parameters. We have also modified some functions of Polybench, since they originally use constant values defined at compile time. The functions' source code can be found in Appendix A. We have used our LLVM pass to generate test cases for each function and executed Asymptus over the produced programs. We have successfully executed all of the functions with several different inputs. Asymptus could correctly analyze all functions. It shows us that our technique is useful in the execution of functions of interest.

## 4.4 Conclusion

We have shown that our input generation technique is useful for the execution of interesting functions without having the context on which the function is called. Thus, our technique has different uses, from the creation of a testing infrastructure to the improvement of Asymptus. This chapter also has presented a new technique, based on a combination of profiling and static analysis, to infer the complexity of code. Static analysis gives us the names of variables that bound the trip count of loops. Profiling lets us associate these variables with the number of operations in the loops that they control. We believe that our approach, whenever applicable, yields results that are more meaningful to the application developer than the state-of-the-art tools that are currently available. A tool that implements the technique is publicly available<sup>1</sup> for use.

---

<sup>1</sup><http://demontiejr.github.io/asymptus>



# Chapter 5

## Conclusion

In this dissertation we have presented a technique to automatically create test drivers for functions, providing random values as inputs. The key contribution of our work is the use of two static analyses which are able to bind arrays and their sizes. Our assumption was that such techniques can contribute to decrease the false positive rate of automatic input generators. We have implemented our techniques on top of the LLVM compiler infrastructure with two main goals: (i) implement a testing infrastructure to test embedded systems of Maxtrack, a Brazilian company which produces truck trackers, and (ii) evaluate the effectiveness of our techniques.

During our experiments with the SPEC CPU 2006 benchmark suite, we have found that the array size analyses, together, could correctly find sizes for 22.7% of the arrays received as parameter by functions. However, for the forward analysis only, this number is much smaller: 0.8%. It shows us that, although this analysis is very precise for our goal, it has small applicability in real-world programs. We have manually inspected the code in order to find out why the numbers were smaller than what we expected before. The main reasons for this ineffectiveness are due to (i) arrays allocated with constant sizes (which are not variables in the program), and (ii) arrays stored into a more complex structure. The second case happens, for example, when the program allocates a large global buffer and passes parts of it to functions.

After analyzing the results, we concluded that handling sizes of arrays when generating inputs can be an effective way to reduce the false positive rate of existing approaches. However, the analyses that we have nowadays are not very effective to do this job. It opens space for future work, such as properly dealing with alias, handling statically allocated arrays and extending the work to deal with C++ code. We also concluded that the techniques presented in this work are useful for different applications. One of them, as mentioned, is the construction of a test infra-structure

for embedded systems. To this end, we also have to generate code for functions not present in the source code of the program in order to be able to properly compile and link it.

## 5.1 Contributions

The main contributions of this master's dissertation are:

1. The design of a forward data flow analysis to find sizes of arrays.
2. The design of an algorithm to generate inputs for functions, including the generation of complex data structures, that makes use of array size analyses in order to generate safe test cases, in the sense of memory access.
3. The identification of pointers used as arrays by looking for pointer arithmetics.
4. The development of a tool, Asymptus, for program complexity inference which, making use of the input generator, is very effective and precise on analyzing the complexity of polynomial functions.
5. The development of a testing infrastructure which makes use of our input generation technique to test mobile applications of Maxtrack.

## 5.2 Future Work

1. Extend the input generator to calculate the complete upper bound of array accesses in order to always allocate the correct amount of memory needed.
2. Extend the forward array size analysis in order to properly deal with alias information and with statically allocated arrays.
3. Extend the forward array size analysis to track arrays over calls (inter-procedural analysis).
4. Create an analysis to infer sizes of recursive data structures (such as linked lists).



# Bibliography

- Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6):246--256.
- Ahmad, M. A. and Oriol, M. (2014). Dirt spot sweeping random strategy. *Lecture Notes on Software Engineering*, 2(4):294.
- Allen, F. E. (1970). Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1--19, New York, NY, USA. ACM.
- Alves, P., Gruber, F., Doerfert, J., Lamprineas, A., Grosser, T., Rastello, F., and Pereira, F. M. Q. a. (2015a). Runtime pointer disambiguation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*, pages 589--606, New York, NY, USA. ACM.
- Alves, P. R. O., Rodrigues, R. E., de Souza, R. M., and Pereira, F. M. Q. (2015b). A case for a fast trip count predictor. *Inf. Process. Lett.*, 115(2):146--150.
- Appel, A. W. and Palsberg, J. (2002). *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition.
- Boyer, R. S., Elspas, B., and Levitt, K. N. (1975). Select&mdash;a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234--245, New York, NY, USA. ACM.
- Cadar, C., Dunbar, D., and Engler, D. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08*, pages 209--224, Berkeley, CA, USA. USENIX Association.

- Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90. ISSN 0001-0782.
- Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54. IEEE.
- Chen, T., Leung, H., and Mak, I. (2005). Adaptive random testing. In Maher, M., editor, *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, volume 3321 of *Lecture Notes in Computer Science*, pages 320–329. Springer Berlin Heidelberg.
- Chen, T. Y., Kuo, F.-C., Liu, H., and Wong, W. (2013). Code coverage of adaptive random testing. *Reliability, IEEE Transactions on*, 62(1):226–237. ISSN 0018-9529.
- Ciupa, I., Leitner, A., Oriol, M., and Meyer, B. (2007). Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07*, pages 84–94, New York, NY, USA. ACM.
- Ciupa, I., Meyer, B., Oriol, M., and Pretschner, A. (2008). Finding faults: Manual testing vs. random+ testing vs. user reports. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 157–166. ISSN 1071-9458.
- Coppa, E., Demetrescu, C., and Finocchi, I. (2012). Input-sensitive profiling. In *PLDI*. ACM.
- Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1989). An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89*, pages 25–35, New York, NY, USA. ACM.
- Danicic, S., Harman, M., and Sivagurunathan, Y. (1995). A parallel algorithm for static program slicing. *Inf. Process. Lett.*, 56(6):307–313. ISSN 0020-0190.
- Danielsson, N. A. (2008). Lightweight semiformal time complexity analysis for purely functional data structures. In *POPL*, pages 133–144. ACM.
- De Lucia, A., Fasolino, A., and Munro, M. (1996). Understanding function behaviors through program slicing. In *Program Comprehension, 1996, Proceedings., Fourth Workshop on*, pages 9–18. ISSN 1092-8138.

- Debray, S. K. and Lin, N.-W. (1993). Cost analysis of logic programs. *ACM Trans. Program. Lang. Syst.*, 15(5):826--875. ISSN 0164-0925.
- Demontiê, F., Cezar, J., Bigonha, M., Campos, F., and Magno Quintão Pereira, F. (2015). Automatic inference of loop complexity through polynomial interpolation. In Pardo, A. and Swierstra, S. D., editors, *Programming Languages*, volume 9325 of *Lecture Notes in Computer Science*, pages 1–15. Springer International Publishing.
- Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.
- Duran, J. W. and Ntafos, S. C. (1984). An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438--444. ISSN 0098-5589.
- Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319--349.
- Forrester, J. E. and Miller, B. P. (2000). An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, pages 6--6, Berkeley, CA, USA. USENIX Association.
- Godefroid, P. (2007). Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47--54, New York, NY, USA. ACM.
- Godefroid, P. (2014). Micro execution. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 539--549, New York, NY, USA. ACM.
- Godefroid, P., Klarlund, N., and Sen, K. (2005a). Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213--223, New York, NY, USA. ACM.
- Godefroid, P., Klarlund, N., and Sen, K. (2005b). Dart: directed automated random testing. In *PLDI*, pages 213--223. ACM.
- Godefroid, P., Levin, M. Y., Molnar, D. A., et al. (2008). Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151--166.

- Goldsmith, S. F., Aiken, A. S., and Wilkerson, D. S. (2007). Measuring empirical computational complexity. In *FSE*, pages 395–404. ACM.
- Graham, S. L., Kessler, P. B., and McKusick, M. K. (1982). gprof: a call graph execution profiler (with retrospective). In *Best of PLDI*, pages 49–57.
- Groce, A., Holzmann, G., and Joshi, R. (2007). Randomized differential testing as a prelude to formal verification. In *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, pages 621–631, Washington, DC, USA. IEEE Computer Society.
- Gulavani, B. and Gulwani, S. (2008). A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, volume 5123 of *LNCS*, pages 370–384. Springer.
- Gulwani, S., Jain, S., and Koskinen, E. (2009a). Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385. ACM.
- Gulwani, S., Mehra, K. K., and Chilimbi, T. (2009b). SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM.
- Horwitz, S., Reps, T., and Binkley, D. (1988). Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88*, pages 35–46, New York, NY, USA. ACM.
- Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. (2012). Path-sensitive backward slicing. In *Proceedings of the 19th International Conference on Static Analysis, SAS'12*, pages 231–247, Berlin, Heidelberg. Springer-Verlag.
- Kamkar, M., Fritzson, P., and Shahmehri, N. (1993). Three approaches to interprocedural dynamic slicing. In *Nineteenth EUROMICRO Symposium on Microprocessing and Microprogramming on Open System Design : Hardware, Software and Applications: Hardware, Software and Applications*, EUROMICRO 93, pages 625–636, Amsterdam, The Netherlands, The Netherlands. Elsevier Science Publishers B. V.
- Kildall, G. A. (1973). A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '73*, pages 194–206, New York, NY, USA. ACM.

- King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385--394. ISSN 0001-0782.
- Korel, B. and Laski, J. (1988). Dynamic program slicing. *Information Processing Letters*, 29(3):155--163.
- Korel, B. and Laski, J. (1990). Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187--195.
- Lattner, C. and Adve, V. (2003). Data structure analysis: An efficient context-sensitive heap analysis. Technical report, Tech. Report UIUCDCSR-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign.
- Lattner, C. and Adve, V. (2004a). Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75--86.
- Lattner, C. and Adve, V. S. (2004b). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75--88. IEEE.
- Le Métayer, D. (1988). Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248--266. ISSN 0164-0925.
- Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32--44. ISSN 0001-0782.
- Monniaux, D. and Gonnord, L. (2011). Using bounded model checking to focus fixpoint iterations. In *SAS*, pages 369--385. Springer.
- Nazaré, H., Maffra, I., Santos, W., Barbosa, L., Gonnord, L., and Quintão Pereira, F. M. (2014). Validation of memory accesses through symbolic analyses. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 791--809, New York, NY, USA. ACM.
- Nethercote, N. and Seward, J. (2007a). Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89--100, New York, NY, USA. ACM.
- Nethercote, N. and Seward, J. (2007b). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89--100. ACM.

- Oriol, M. and Tassis, S. (2010). Testing .net code with yeti. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 264–265.
- Ottenstein, K. J. and Ottenstein, L. M. (1984). The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 177–184, New York, NY, USA. ACM.
- Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. (2007). Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA. IEEE Computer Society.
- Plotkin, G. D. (1981). A structural approach to operational semantics.
- Pouchet, L.-N. (2012). Polybench: The polyhedral benchmark suite. URL: <http://www.cs.ucla.edu/~pouchet/software/polybench/>. Last access: April, 2015.
- Reps, T. (1998). Program analysis via graph reachability. *Information and software technology*, 40(11):701--726.
- Saxena, P., Poosankam, P., McCamant, S., and Song, D. (2009). Loop-extended symbolic execution on binary programs. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 225–236, New York, NY, USA. ACM.
- Wegbreit, B. (1975). Mechanical program analysis. *Commun. ACM*, 18(9):528--539. ISSN 0001-0782.
- Weiser, M. (1981). Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA. IEEE Press.
- Wolfe, M. (1996). *High Performance Compilers for Parallel Computing*. Adison-Wesley, 1st edition.
- Zaparanuks, D. and Hauswirth, M. (2012). Algorithmic profiling. In *PLDI*, pages 67--76. ACM.

# Appendix A

## Functions for the Input Generation Experiment

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void bubble_sort(int *list , int n) {
    for (int c = 0 ; c < (n - 1); c++) {
        for (int d = 0 ; d < n - c - 1; d++) {
            if (list[d] > list[d+1]) {
                int t = list[d];
                list[d] = list[d+1];
                list[d+1] = t;
            }
        }
    }
}

void selection_sort(int *list , int n) {
    for (int i = 0; i < (n-1); i++) {
        int min = i;
        for (int j = (i+1); j < n; j++) {
            if(list[j] < list[min])
                min = j;
        }
        if (i != min) {
```

```

    int aux = list[i];
    list[i] = list[min];
    list[min] = aux;
}
}
}

void insertion_sort(int *list, int n) {
    for (int i = 0; i < n; i++) {
        int current = list[i];
        int j = i - 1;
        while ((j >= 0) && (current < list[j])) {
            list[j + 1] = list[j];
            j = j - 1;
        }

        list[j + 1] = current;
    }
}

typedef struct Node {
    struct Node *next;
    int value;
} Node;

typedef struct List {
    Node *head;
} List;

List* sub_list(List *l, int elements) {
    if (!l)
        return NULL;

    List *newList = (List*) malloc(sizeof(List));
    Node *prev = NULL;
    Node *n = l->head;
    for (int i=0; i < elements; i++) {
        if (!n) break;
        Node *newNode = (Node*) malloc(sizeof(Node));

```



```

newNode->value = n->value;
if (prev)
    prev->next = newNode;
else
    newList->head = newNode;
prev = newNode;
n = n->next;
}
return newList;
}

struct Pair {
    int i;
    int j;
};

struct Pair closest_point(struct Pair *pairs, int size) {
    int min = INFINITY;
    struct Pair closest;
    for (int i=0; i < size; i++) {
        struct Pair p = pairs[i];
        int dist = sqrt(pow(p.i, 2) + pow(p.j, 2));
        if (dist < min) {
            min = dist;
            closest = p;
        }
    }
    return closest;
}

int closest_pairs(struct Pair *pairs, int size) {
    int min = INFINITY;
    struct Pair closest;
    for (int i=0; i < size; i++) {
        struct Pair p1 = pairs[i];
        for (int j=i; j < size; j++) {
            struct Pair p2 = pairs[j];
            int dist = sqrt(pow(p1.i - p2.i, 2) + pow(p1.j - p2.j, 2));
            if (dist < min) {

```

```

        min = dist;
    }
}
}
return min;
}

int multiply(int *a, int *b, int size) {
    int result = 0;
    for (int i=0; i < size; i++) {
        result += a[i] * b[i];
    }
    return result;
}

/* Initialize a single-source shortest-paths computation. */
void initialize_single_source(double d[], int n) {
    for (int i = 1; i < n; ++i) {
        d[i] = 1000000000.0;
    }
    d[0] = 0.0;
}

double *bellman_ford(double **w, int n) {
    double *d = (double*) malloc(sizeof(double)*n);
    initialize_single_source(d, n);

    for (int i = 0; i < n-1; ++i) {
        for (int u = 0; u < n; ++u) {
            for (int v = 0; v < n; ++v) {
                if (d[v] > d[u] + w[u][v]) {
                    d[v] = d[u] + w[u][v];
                }
            }
        }
    }

    for (int u = 0; u < n; ++u) {
        for (int v = 0; v < n; ++v) {

```

```

        if (d[v] > d[u] + w[u][v])
            return NULL;
    }
}

return d;
}

int **matrix_mul(int **matA, int **matB, int n){
    int i, j, k, sum;
    int **result = (int**) malloc(n * sizeof(int*));
    for (i = 0; i < n; i++)
        result[i] = (int*) malloc(n * sizeof(int));

    for (i=0; i < n; i++) {
        for (j=0; j < n; j++) {
            sum = 0;
            for (k=0; k < n; k++) {
                sum += matA[i][k] * matB[k][j];
            }
            result[i][j] = sum;
        }
    }
    return result;
}

int **sum(int **matA, int **matB, int n) {
    if (!matA || !matB) return NULL;
    int **result = (int**) malloc(n * sizeof(int*));
    for (int i=0; i < n; i++) {
        result[i] = (int*) malloc(n * sizeof(int));
        for(int j=0; j < n; j++)
            result[i][j] = matA[i][j] + matB[i][j];
    }
    return result;
}

// ===== Polybench =====

```

```

void kernel_floyd_warshall(int **path, int n) {
    for (int k = 0; k < n; k++) {
        for(int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                path[i][j] = path[i][j] < path[i][k] + path[k][j] ?
                    path[i][j] : path[i][k] + path[k][j];
    }
}

void kernel_lu(int **A, int n) {
    for (int k = 0; k < n; k++) {
        for (int j = k + 1; j < n; j++)
            A[k][j] = A[k][j] / A[k][k];
        for(int i = k + 1; i < n; i++)
            for (int j = k + 1; j < n; j++)
                A[i][j] = A[i][j] + A[i][k] * A[k][j];
    }
}

void kernel_jacobi_2d_imper(int **A, int **B, int n, int tsteps) {
    for (int t = 0; t < tsteps; t++) {
        for (int i = 1; i < n - 1; i++)
            for (int j = 1; j < n - 1; j++)
                B[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][j+1]
                    + A[j+1][i] + A[j-1][i]);

        for (int i = 1; i < n-1; i++)
            for (int j = 1; j < n-1; j++)
                A[i][j] = B[i][j];
    }
}

void kernel_seidel_2d(int **A, int n, int tsteps) {
    for (int t = 0; t <= tsteps - 1; t++)
        for (int i = 1; i <= n - 2; i++)
            for (int j = 1; j <= n - 2; j++)
                A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
                    + A[i][j-1] + A[i][j] + A[i][j+1]
                    + A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])/9.0;
}

```

```

}

void kernel_2mm(int **tmp, int **A, int **B, int **C, int **D,
               int ni, int nj, int nk, int nl, int alpha, int beta) {
    /* D := alpha*A*B*C + beta*D */
    for (int i = 0; i < ni; i++)
        for (int j = 0; j < nj; j++) {
            tmp[i][j] = 0;
            for (int k = 0; k < nk; ++k)
                tmp[i][j] += alpha * A[i][k] * B[k][j];
        }

    for (int i = 0; i < ni; i++)
        for (int j = 0; j < nl; j++) {
            D[i][j] *= beta;
            for (int k = 0; k < nj; ++k)
                D[i][j] += tmp[i][k] * C[k][j];
        }
}

void kernel_fdt_d_2d(int **ex, int **ey, int **hz, int *_fict_,
                    int tmax, int nx, int ny) {
    for(int t = 0; t < tmax; t++) {
        for (int j = 0; j < ny; j++)
            ey[0][j] = _fict_[t];
        for (int i = 1; i < nx; i++)
            for (int j = 0; j < ny; j++)
                ey[i][j] = ey[i][j] - 0.5*(hz[i][j]-hz[i-1][j]);
        for (int i = 0; i < nx; i++)
            for (int j = 1; j < ny; j++)
                ex[i][j] = ex[i][j] - 0.5*(hz[i][j]-hz[i][j-1]);
        for (int i = 0; i < nx - 1; i++)
            for (int j = 0; j < ny - 1; j++)
                hz[i][j] = hz[i][j] - 0.7* (ex[i][j+1] - ex[i][j] +
                ey[i+1][j] - ey[i][j]);
    }
}

```