Model Counting Modulo Theories

Quoc-Sang Phan

Queen Mary
University of London

A thesis submitted for the degree of

Doctor of Philosophy

London 2015

Statement of Originality

I, Quoc-Sang Phan, confirm that the research included within this thesis is my own work or that where it has been carried out in collaboration with, or supported by others, that this is duly acknowledged and my contribution indicated. Previously published material is also acknowledged in the next page.

I attest that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge break any UK law, infringe any third party's copyright or other Intellectual Property Right, or contain any confidential material.

I accept that the College has the right to use plagiarism detection software to check the electronic version of the thesis.

I confirm that this thesis has not been previously submitted for the award of a degree by this or any other university.

Quoc-Sang Phan
April 9, 2015

Publications

Parts of this thesis have been published; chapters 3, 4, 5, 6 and 7 have been published as the papers listed below, in reverse chronological order.

[1] **Quoc-Sang Phan** and Pasquale Malacaria. All-Solution Satisfiability Modulo Theories: applications, algorithms and benchmarks. In *Proceedings of the Tenth International Conference on Availability, Reliability and Security, ARES '15.*

[2] **Quoc-Sang Phan**, Pasquale Malacaria, and Corina S. Păsăreanu. Concurrent Bounded Model Checking. *SIGSOFT Software Engineering Notes*, 40(1):1–5, February 2015.

[3] **Quoc-Sang Phan**. Symbolic Execution as DPLL Modulo Theories. In *2014 Imperial College Computing Student Workshop*, volume 43 of *OpenAccess Series in Informatics (OASIcs)*, pages 58-65, Dagstuhl, Germany, 2014. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[4] **Quoc-Sang Phan**, Pasquale Malacaria, Corina S. Păsăreanu, and Marcelo d'Amorim. Quantifying Information Leaks Using Reliability Analysis. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 105-108, New York, NY, USA, 2014. ACM.

[5] **Quoc-Sang Phan** and Pasquale Malacaria. Abstract Model Counting: a novel approach for Quantification of Information Leaks. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '14, pages 283–292, New York, NY, USA, 2014. ACM.

[6] **Quoc-Sang Phan**. Self-composition by Symbolic Execution. In *2013 Imperial College Computing Student Workshop*, volume 35 of *OpenAccess Series in Informatics (OASIcs)*, pages 95-102, Dagstuhl, Germany, 2013. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[7] **Quoc-Sang Phan**, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Păsăreanu. Symbolic Quantitative Information Flow. *SIGSOFT Software Engineering Notes*, 37(6):1–5, November 2012.

Abstract

This thesis is concerned with the quantitative assessment of security in software. More specifically, it tackles the problem of efficient computation of *channel capacity*, the maximum amount of confidential information leaked by software, measured in Shannon entropy or Rényi's min-entropy.

Most approaches to computing channel capacity are either efficient and return only (possibly very loose) upper bounds, or alternatively are inefficient but precise; few target realistic programs. In this thesis, we present a novel approach to the problem by reducing it to a model counting problem on first-order logic, which we name *Model Counting Modulo Theories* or #SMT for brevity.

For quantitative security, our contribution is twofold. First, on the theoretical side we establish the connections between measuring confidentiality leaks and fundamental verification algorithms like Symbolic Execution, SMT solvers and DPLL. Second, exploiting these connections, we develop novel #SMT-based techniques to compute channel capacity, which achieve both accuracy and efficiency. These techniques are scalable to real-world programs, and illustrative case studies include C programs from Linux kernel, a Java program from a European project and anonymity protocols.

For formal verification, our contribution is also twofold. First, we introduce and study a new research problem, namely #SMT, which has other potential applications beyond computing channel capacity, such as returning multiple-counterexamples for Bounded Model Checking or automated test generation. Second, we propose an alternative approach for Bounded Model Checking using classical Symbolic Execution, which can be parallelised to leverage modern multi-core and distributed architecture.

For software engineering, our first contribution is to demonstrate the correspondence between the algorithm of Symbolic Execution and the DPLL($\mathcal{T}$) algorithm used in state-of-the-art SMT solvers. This correspondence could be leveraged to improve Symbolic Execution for automated test generation. Finally, we show the relation between computing channel capacity and reliability analysis in software.

To my parents Phan Quốc Kính and Nguyễn Thị Hương

Acknowledgements

I feel extremely lucky to have Pasquale Malacaria as my supervisor, and I would like to thank him for his guidance, encouragement and support for the last three years. Pasquale was always available for discussions and advice, which was important for the development of the ideas in this thesis. I also truly appreciate the opportunities he has given me to present my work, and network with the research community. I would also thank him for establishing the foundation of Quantitative Information Flow, the main topic of this thesis.

I am also deeply grateful to Corina Păsăreanu for her advice, encouragement and kindness. Indeed, Corina has been my *de facto* co-supervisor, and a large part of this thesis would not have been existed without the Symbolic PathFinder platform that she developed. I also want to thank Corina for sharing her office with me, and for escorting me almost every day during my internship at NASA Ames.

I wish to thank my mentors, Johann Schumann and Kristin Rozier, for giving me a chance to work at NASA, and for sharing with me parts of their vast knowledge and experience. Particular thanks to Johann for teaching me Unmanned Aerial Systems, and for allowing me to stay in his house during my internship.

I am grateful to Oksana Tkachuk and Marcelo d'Amorim for mentoring me during the Google Summer of Code programs in 2012 and 2013. I am also thankful to Nikos Tzevelekos, Michael Tautschnig and Dino Distefano for fruitful discussions and for their careful comments on early versions of my papers.

I would like to thank my family, their love has helped me overcome all difficulties. In particular, my wife Huyen Do has loved and cared for me throughout my PhD. My son Voi, born at the beginning of my third year of PhD, has brought me endless happiness.

I wish to thank my office-mate and fellow PhD student Nhat Anh Dang for his help and friendship. Although we work in totally different areas, this has not prevented us from having countless discussions, and Nhat Anh even managed to teach me several mathematical concepts. I am also thankful for his kind help in my personal life.

I am grateful for the scholarship of the School of Electronic Engineering and Computer Science, which covered my tuition fees and stipends to live in London. I also benefited greatly from the research environment of the school, in particular the weekly research seminars of the Theory group have helped me to broaden my knowledge. Thanks to Nikos Tzevelekos for having done an excellent job organizing these seminars.

Finally, I would like to thank my PhD examiners, David Clark of University College London and Cristian Cadar of Imperial College London, for their helpful, constructive comments which improved this thesis.

Contents

1 Introduction **11**

1.1 Motivation . . . 11

1.2 State of the art . . . 12

1.3 Contributions . . . 14

1.4 Thesis structure . . . 16

2 Preliminaries **17**

2.1 Quantitative Information Flow . . . 17

2.1.1 Information Flow and Non-interference . . . 18

2.1.2 Information Theoretical Measurement of Interference . . . 21

2.1.3 Problem Statement . . . 27

2.2 Logical Satisfiability Problems . . . 28

2.2.1 Propositional Satisfiability . . . 28

2.2.2 Model Counting . . . 29

2.2.3 Satisfiability Modulo Theories . . . 29

2.3 The programming language and the program . . . 31

2.4 Formal Methods . . . 32

2.4.1 Bounded Model Checking and CBMC . . . 33

2.4.2 Symbolic Execution and Symbolic PathFinder . . . 34

3 Model Counting Modulo Theories **35**

3.1 Illustrative Example . . . 35

3.2 Model Counting Modulo Theories . . . 37

3.3 Quantitative Information Flow as #SMT . . . 38

3.4 Symbolic Quantitative Information Flow . . . 40

3.5 Soundness and Completeness . . . 44

3.6 Evaluation . . . 45

3.6.1 CVE-2011-2208 . . . 46

3.6.2 CVE-2011-1078 47
3.6.3 Cyclic Redundancy Check 49
3.6.4 The Grade Protocol 50
3.6.5 The Dining cryptos protocol 52
3.7 Discussion of related work 53

4 Symbolic Execution as DPLL Modulo Theories 55

4.1 Introduction 55
4.2 Illustration of DPLL($\mathcal{T}$) 56
4.3 Symbolic Execution as DPLL($\mathcal{T}$) 58
4.4 SQIF by Symbolic Execution: 61
4.5 Soundness and Completeness 62
4.6 Evaluation 62
4.6.1 Tax Record 63
4.7 Discussion of related work 64

5 Concurrent Bounded Model Checking 65

5.1 Introduction 66
5.2 Illustrative example 67
5.3 Concurrent Bounded Model Checking 69
5.3.1 Bounded Model Checking by Symbolic Execution 70
5.3.2 Comparing our approach with BMC and SE 71
5.3.3 Sequential Verification 71
5.3.4 Concurrent Verification 73
5.4 Evaluation 75
5.4.1 Comparing with CBMC and SPF 76
5.4.2 Comparing with SPF 77
5.5 Discussion of related work 78

6 Quantifying Information Leaks using Reliability Analysis 81

6.1 Introduction 81
6.1.1 Qualitative analysis 82
6.1.2 Quantitative analysis 82
6.2 Preliminaries 83
6.2.1 Symbolic Execution 83
6.2.2 Reliability Analysis 85
6.3 Self-composition by Symbolic Execution 85

6.3.1 Self-composition as path-equivalence 86
6.3.2 Path-equivalence generation 86
6.3.3 Examples 87
6.3.4 Optimization 89
6.4 Quantifying Information Leaks using Reliability Analysis 90
6.4.1 Fine-grained self-composition 90
6.4.2 Model Counting for Symbolic Paths 91
6.5 Evaluation 92
6.5.1 Results and discussions 93
6.6 Discussion of related work 94

7 A solver for Model Counting Modulo Theories 96
7.1 Quantifying Information Leaks using a #SMT solver 97
7.1.1 Illustrative Example 97
7.1.2 Program transformation with CBMC 98
7.1.3 Formula instrumentation 99
7.2 All-Solution Satisfiability Modulo Theories 100
7.2.1 Multiple-counterexamples for BMC 100
7.2.2 Automated Test Generation 102
7.2.3 Reliability analysis 104
7.3 Algorithms for #SMT and All-SMT solver 104
7.3.1 Blocking clauses method 106
7.3.2 Depth-first search 107
7.3.3 Implementation 108
7.4 Evaluation 108
7.4.1 Evaluation of #SMT solvers 109
7.4.2 Evaluation of QIF analysers 110
7.5 Discussion of related work 111
7.5.1 Quantitative Information Flow 111
7.5.2 Multiple-counterexamples for BMC 111
7.5.3 Automated Test Generation 112
7.5.4 Reliability analysis 112
7.5.5 All-Solution SAT Modulo Theories 112

8 Conclusions 114
8.1 Summary 114
8.2 Future Research 115

Chapter 1

Introduction

1.1 Motivation

The year 2014 has witnessed several high-profile software security incidents, e.g. the disclosure of heartbleed bug [8], the leaks of celebrity photos in iCloud [9], the hack of Sony Pictures [10]. The damage caused by leaking confidential information varies, from personal embarrassment to the loss of dozens of millions of dollars.

These incidents show the importance of protecting confidential data from being leaked by software. Access control systems [11] can limit access to information, but cannot control internal information propagation once accessed. This motivates the research on *information flow security* [12], which aims to track the flows of information in software, and forbid illegal flows that leak information to public observers.

However, leakage of information is hardly avoidable in computer programs. Even "secure" programs do leak some information about the secret data being processed. A popular example is the password checking program, which can be considered as secure with a reasonably strong password. Every time it rejects an input string, it reveals to the adversary that the password is different from that string. This amount of information is small, but if the adversary is allowed to make enough attempts, i.e. brute force attack, the program will eventually leak all information to the attacker. As leakage of information is unavoidable, it is important to assess the leaks to decide if they are acceptable. This leads to the question *how much* information a program could leak to an adversary.

The research area of *Quantitative Information Flow* analysis (QIF [13, 14]) has been developed to provide a rigorous framework to answer the question above. Intuitively, after observing an execution of the program, the adversary gains more information and has less uncertainty about the confidential data. The difference of uncertainties before and after his observation is the amount of information leaked by the program. QIF has based its foundation on the entropy concept of Information Theory [15]: the uncertainties about the confidential data are measured in bits by, for example, Shannon entropy; leakage is then computed by numerical subtraction.

Manual computation of QIF, e.g. in [14], is tedious, expensive and infeasible for complex programs. In order to apply the theory of QIF to practice, it is crucial to have automated techniques that can efficiently quantify leakage in software systems. However, most approaches to automation of QIF are either efficient and return only (possibly very loose) upper bounds [16, 17, 18], or alternatively are inefficient but precise [19, 20, 21]; few target realistic programs. This thesis is a significant step towards efficient and accurate computation of QIF by means of formal methods.

1.2 State of the art

This section outlines some of the key advances that have led to the current state of the art for automation of QIF. Emphasis is given to work on deterministic programs.

The concept of information flow was first introduced in 1977 by the Dennings [22], who described a lattice model where variables are partitioned into security labels: H, standing for “high”, for variables containing sensitive data and L, standing for “low", for variables containing public information. The partial order L $\leq$ H in the lattice indicates that information flows from variables with label H to variables with label L are not allowed.

In 1982, Goguen and Meseguer [23] described *non-interference*, a security policy for a general automaton framework where there are a set of state changing commands and a set of users. A group of users G have non-interference on another group of users G’ if and only if any user in the group G cannot observe the effects of commands used by any user in the group G’.

In 1996, Volpano, Smith, and Irvine [24] achieved an impressive milestone by proving the soundness of the Dennings’ analysis using a type system, which coincides with the idea of non-interference. Since then, research in information flow has evolved in different directions. In 2003, Sabelfeld and Myers [12] published an excellent survey of the field up to that time, citing 147 papers.

The history of quantitative information flow is not much shorter than its qualitative counterpart. Again, Denning [25] was the first, in 1982, to propose the idea of using Information Theory [15] as a basic for a quantitative analysis of information flow. In 1987, Millen [26] established a connection between Information Theory and the general automaton framework of Goguen and Meseguer [23], in which non-interference was connected to the notation of mutual information.

Later work from the 1990's to the early 2000s were mostly theoretical, trying to define an information flow quantity in different settings, including Flow Model [27, 28], Communicating Sequential Processes [29], probabilistic program [30], process calculi [31], and so on.

The first complete quantitative analysis of information flow had been developed by Clark, Hunt and Malacaria through a series of papers [32, 33, 34, 13] from 2002 to 2007. The authors illustrated their analysis on a simple deterministic While language, and provided for each command of the language a lower bound and an upper bound on the amount of leakage. Their treatment for loops was very pessimistic, assuming that all information in the looping conditions will be leaked.

To address the problem above, Malacaria [14] introduced a more precise treatment for looping constructs by using the partition property of entropy. However, in that work the author manually determined if a loop terminated and the maximum number of iterations if it did terminate. For this reason, the analysis is labour intensive and impossible to automate.

In 2009, Smith [35] proposed to use Rényi's min-entropy as a metric for QIF, and showed that the *channel capacity* of information flow, i.e. the maximum leakage, measured by this metric is equal to the logarithm of the number of possible outputs. This result agrees with previous observations of Malacaria and Chen [36] that channel capacity measured by Shannon entropy also has the same tight upper bound.

Meanwhile, in the same year (2009) an important milestone was reached when the first automatic techniques for QIF analysis [37, 19] were introduced. Mu and Clark [37, 38] continued the line of previous research by Clark, Hunt and Malacaria, and introduced a technique and a tool to automate the analysis of the toy language While in [13]. Approximately at the same time, Backes, Köpf and Rybalchenko [19] presented an analysis using model checking [39] and the Barvinok algorithm [40] for model counting. This analysis is very expensive, and only applicable to a limited class of programs.

In 2010, Heusser and Malacaria [20] published a paper demonstrating QIF analysis for programs from the Linux kernel. Prior to this paper, all the work on QIF analysis were demonstrated with small “toy” examples. Heusser and Malacaria made assumptions that the program had `N` different possible outputs, and asserted that there did not exist an output different from those `N` outputs. Since these assumptions and assertion required the composition of `N+1` copies of the program, their analysis suffers severely from the state space explosion problem.

A year later, in 2011, Meng and Smith [18] described a fast approximation technique to compute an upper bound on channel capacity. They infer the relations between all pairs of bits of the output as a propositional formula, then use a #SAT solver [41] to count the model. The analysis was largely manual and was demonstrated with toy examples. The upper bound can be very loose if outputs are sparse and scattered.

In summary, the problem of an automated QIF analysis is still very challenging. A simpler problem, called bounding QIF, is considered in [42, 20]: deciding if a program P leaks less than a constant q. In previous work, Yasuoka and Terauchi have proved that bounding QIF is not a *k-safety* problem for any k [42]. Cerny et al. then proved that in the case of Shannon entropy, bounding QIF is PSPACE-complete [43]. Therefore, QIF and bounding QIF remain a huge challenge. Available techniques are either inefficient or imprecise. This thesis investigates techniques that improve both precision and efficiency for QIF analysis.

1.3 Contributions

This thesis is about solving a problem, QIF, using a set of tools referred to as Formal Methods. Through the process of solving the problem, we also gain the insights to improve the tools. As a result, this thesis makes contributions both to approaches for automated QIF analysis using Formal Methods as well as techniques for improving Formal Methods.

Contributions to Quantitative Information Flow

This thesis makes a theoretical advance by casting the QIF problem into a variant of the Satisfiability Modulo Theories (SMT) problem [44]. This results in a general algorithm for QIF analysis based on the DPLL($\mathcal{T}$) algorithm [45] for SMT. This theoretical advance leads to practical advance: compared to the work of Heusser and Malacaria [20], the technique proposed in this thesis dramatically reduces the time of analysing programs from Linux kernel, from some hours to a few seconds.

The second theoretical contribution of this thesis is to show that classical Symbolic Execution [46] can be understood as a variant of the DPLL($\mathcal{T}$) algorithm. In other words, Symbolic Executors are SMT solvers. This view enables us to develop the first QIF analysis tool for Java bytecode, built on top of the Symbolic PathFinder symbolic execution platform.

Our third contribution is to show the relation between Quantitative Information Flow and Reliability analysis [47], which are two totally separate research areas prior to this thesis. This relation leads to the development of an efficient QIF technique based on an available Reliability analyser.

The practical contribution of this thesis is the development of several QIF analysis tools for programs in both C and Java, and the experiments of these tools on real-world programs: C programs from the Linux kernel, a Java tax program from the European project HATS, and anonymity protocols.

Contributions to Formal Methods

The insights we have learned from solving the QIF problem lead us to investigate techniques that could improve Formal Methods. Our first contribution in this thesis is the introduction of the Model Counting Modulo Theories (or #SMT) problem, which has several potential applications beyond QIF.

Our second contribution is: from the view of Symbolic Executors as SMT solvers, we propose a new methodology for Bounded Model Checking based on Symbolic Execution. Our methodology is naturally parallelizable, thus it can exploit modern multi-core machines and distributed architecture. Experimental results show that it outperforms the state-of-the-art Bounded Model Checker CBMC [48] in several complex case studies.

The third contribution of this thesis is a light weight method for All-Solution SAT Modulo Theories (All-SMT). Unlike available approaches, our method can be used for any SMT problem with any ground theories, including bit vectors.

Another contribution of this thesis is two applications of All-SMT solvers: the first one is to find multiple-counter examples in Bounded Model Checking; the second one is to combine Bounded Model Checking and All-SMT solver for automated test vector generation.

1.4 Thesis structure

This thesis can be divided in two parts: the first part includes chapters 3, 6, and 7 focusing on automation of QIF using Formal Methods; the second part includes chapters 4 and 5 focusing on improving Formal Methods.

Chapter 2 provides necessary preliminaries on the two main concepts in this thesis: QIF and SMT.

Chapter 3 introduces the #SMT problem as a generalization of the SMT problem, and casts the QIF problem into #SMT. The result is a general algorithm for QIF analysis based on the DPLL($\mathcal{T}$) algorithm used in SMT solvers.

Chapter 4 shows how Symbolic Execution can be viewed as a variant of the DPLL($\mathcal{T}$) algorithm. This view enables one to turn a classical Symbolic Executor into a #SMT solver for QIF analysis with little effort.

Chapter 5 presents a new methodology for Concurrent Bounded Model Checking using classical Symbolic Execution. The effectiveness of the methodology is illustrated with several complex case studies in both C and Java.

Chapter 6 describes the relation between QIF and Reliability analysis. This relation is exploited to build an efficient QIF analysis tool for Java bytecode based on an available Reliability analysis engine.

Chapter 7 presents two lightweight algorithms for #SMT in a pure logic settings, and three applications of a #SMT/All-SMT solver apart from QIF, namely multiple-counterexample analysis for Bounded Model Checking, automated test generation and reliability analysis.

Chapter 8 concludes the thesis with a summary of contributions and discusses possible directions for future work.

Chapter 2

Preliminaries

In order to make the thesis self-contained, this chapter provides some basic notions and terminology about discrete probability theory, information theory and first-order theories. We also recall the elements and general concepts of formal methods.

The content of this chapter is synthesized from several sources, e.g. [12, 49, 35, 50, 15, 51, 52, 53]. *None of the results in this chapter were discovered by the author of this thesis*; a couple of proofs for well-known results, e.g. maximal entropy, are lifted almost verbatim from text books.

2.1 Quantitative Information Flow

Our attacker model is depicted in Figure 2.1. The program `P`, characterized by a function f, takes confidential input `H`, public input `L`, and produces output `O`. `L` may or may not be controlled by an adversary. The adversary tries to infer information from `H` by observing `L` and `O`.

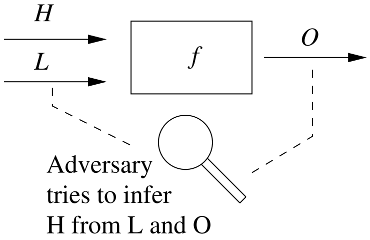

Figure 2.1: Attacker Model

2.1.1 Information Flow and Non-interference

Information flow is the (illegal) transmission of information from a variable `H` to a variable `O` in a given process. The simplest case of information flow is *explicit flow* (or direct flow) where the whole or partial value of `H` is copied directly to `O`, for example:

```
O = H + 3;
```

There are more subtle cases, which are categorized as *implicit flow* (or indirect flow). Consider, for example, the program below, which simulates a common password checking procedure:

```
if (H == L)
    O = true;
else
    O = false;
```

Figure 2.2: A password checking program

`H` is the password, i.e. the confidential data; `L` is the public input provided by the user; `O` is the observable output, "`O = true;`" means the password is accepted. Although `H` is not directly copied to `O`, there is still information flow leaked from `H` → `O`. This information is "small", but one can reveal all information about `H` if he is allowed to make enough attempts.

Obviously, information flow from confidential data to observable output is not desirable, which is the motivation of research in *secure information flow.* Dating back to the pioneering work of the Dennings in the 1970s [22], secure information flow analysis has been an active research topic for the last four decades.

A popular security policy that guarantees the absence of information flow leaks is non-interference [54, 23]. It was introduced by Goguen and Meseguer as a security policy for a general automaton framework, here we give a definition of non-interference in language-based settings:

Definition 1 (Non-interference) *Suppose a program `P` takes secret input `H`, public input `L` and produces public output `O`. `P` has the non-interference property if and only if: for all possible pairs of input vector $[H_1, L_1]$, and $[H_2, L_2]$:*

$$L_1 = L_2 \wedge O_1 = P([H_1, L_1]) \wedge O_2 = P([H_2, L_2]) \Rightarrow O_1 = O_2$$

where $\mathtt{O}_1 = P([\mathtt{H}_1, \mathtt{L}_1])$ denotes that $\mathtt{O}_1$ is the result of executing the program P with the input vector $[\mathtt{H}_1, \mathtt{L}_1]$.

We use the two terms "*information flow*" and "*interference*" interchangeably, since information flow from H to O means O is interfered by H. The non-interference policy guarantees the absence of interference or information flow.

Type system approach to non-interference

There has been a large body of work that has used type systems for validating non-interference, following the idea of Volpano, Irvine and Smith [24]. Type systems are fast and they support automated, compositional verification. Moreover, the analysis is *safe*, which means if a program is classified as "*secure*", then it is actually secure, there are no false negatives.

However, this approach is not extensible. Even a small modification in the information flow policy or in the programming language, e.g. adding a new feature, requires a non-trivial extension of the type system and its soundness proof. This approach also returns too many false positives, which means secure programs can be classified as "*insecure*". For example, consider again the two examples with a small modification to make them satisfy non-interference:

```
O = H - H + 3;
```

```
if (H == L)
    O = false;
else
    O = false;
```

Most typing rules would classify programs like the above as insecure. Another tricky case is of programs that leak information in the intermediate states, but sanitize information at the end, for example:

```
O = H + 3;
O = 3;
```

Given that the attacker can only observe the final value of the output O, the program is secure. However, it would be classified as insecure by most type systems.

Theorem proving approach to non-interference

Another prominent approach for secure information flow is to use theorem proving, in which non-interference is logically formulated by *self-composition* [55, 56], as non-interference itself is not a logical property.

We assume a similar setting as in the case of non-interference: given a program P that takes secret input H, public input L and producing public output O, we denote by $\mathtt{P}_1$ the same program as P, with all variables renamed: H as $\mathtt{H}_1$, L as $\mathtt{L}_1$ and O as $\mathtt{O}_1$. Self-composition is expressed in Hoare-style framework as [56]:

$$\{\mathtt{L} = \mathtt{L}_1\}\mathtt{P};\mathtt{P}_1\{\mathtt{O} = \mathtt{O}_1\} \tag{2.1}$$

The Hoare triple states that if the precondition $\mathtt{L} = \mathtt{L}_1$ holds, then after the execution of $P; P_1$, the postcondition $\mathtt{O} = \mathtt{O}_1$ also holds. The purpose of having the copy $\mathtt{P}_1$ with all variables renamed is to have *another* P to compare with P, so self-composition is logical formulation of non-interference.

For example, consider again the password checking program P in Figure 2.2, the composition of P and its copy $\mathtt{P}_1$ is shown in Figure 2.3. By choosing $\mathtt{H} = \mathtt{L} \wedge \mathtt{H}_1 \neq \mathtt{L}_1$, it is easy to find a counter-example for the Hoare triple in (2.1), such that $\mathtt{L} = \mathtt{L}_1$ holds and $\mathtt{O} = \mathtt{O}_1$ does not hold. Therefore, the password checking program violates non-interference.

```
if (H == L)
    O = true;
else
    O = false;
if (H1 == L1)
    O1 = true;
else
    O1 = false;
```

Figure 2.3: Self-composition of the password checking program

Compared to the type system approach, the theorem proving approach is much more precise, returning no false positives. However, it is impractical in reality, as elegantly put in [57] by Terauchi and Aiken:

“When we actually applied the self-composition approach, we found that not only are the existing automatic safety analysis tools not powerful enough to verify many realistic problem instances efficiently (or at all), but also that there are strong reasons to believe that it is unlikely to expect any future advance.”

Terauchi and Aiken also pointed out that the limitations of self-composition come from the symmetry and redundancy of the self-composed program, which lead to some partial-correctness conditions that hold between `P` and $\mathtt{P}_1$. To find these conditions is crucial for the effectiveness of the analysis, however, finding them is in general impractical. Moreover, the use of an interactive theorem prover requires considerable user interaction and verification expertise.

2.1.2 Information Theoretical Measurement of Interference

The fact that the password checking program, Figure 2.2, leaks information indicates that non-interference is over-pessimistic, and often unachievable. Moreover, there are situations that one needs to decide, between two programs, which one is more secure? Consider the following program:

```
if (H >= L)
    O = true;
else
    O = false;
```

Obviously this program is much less secure than the one in Figure 2.2, a fact that cannot be proved with non-interference. These examples show the need of a more quantitative assessment of information flow: instead of asking “*does the program leak information*?”, we would like to know “*how much does it leak*?” Information Theory [50] provides the tools to answer this question.

Discrete Probability Theory

We introduce the concepts that are to be used in the construction of an information theoretical analysis of information flow. A thorough background and description can be found in standard textbooks, e.g. [58].

Definition 2 (Sample space) *The sample space of an experiment, denoted by Ω, is a countable set of all possible outcomes of that experiment.*

Each outcome is a complete description of the state of the real world as a result of the experiment. It need not be a number, e.g. the outcome of tossing a coin is either "head" or "tail". Therefore, probability theory needs the following definition:

Definition 3 (Random variable) *A random variable is a function $X : \Omega \to N$ ($N \subseteq \mathbb{R}$) that associates a real number with each possible outcome in Ω.*

Each element $x \in N$ is assigned a "*probability*" value, denoted by $p(x)$ or $p(X = x)$, which is the measure of the likeliness that X takes the value x.

Definition 4 (Probability distribution) *The probability distribution of a random variable $X : \Omega \to N$ is a function $p : N \to [0, 1]$ such that:*

$$\sum_{x \in N} p(x) = 1$$

For discrete probability, the function p is also known as *probability mass function.* In this thesis, we are interested in the *uniform distribution*, in which all the elements of N have the same probability. This means every outcome of the sample space is equally likely to occur. We will prove later that under the uniform distribution of the confidential data, the information leakage of the program is maximal. Thus, by analyzing the program under this setting, we can provide an upper bound for the leakage over all possible distributions of the confidential data.

Any subset $E \subseteq \Omega$ is refereed to as an *event.* The probability of an event E is defined as follows:

$$p(E) = \sum_{\omega \in E} p(X(\omega))$$

It is straightforward from the definition of probability distribution that $p(\Omega) = 1$, which means Ω is an event that always occurs. At other extreme, it is easy to show that $p(\emptyset) = 0$, i.e. the empty set is an event that never occurs.

Definition 5 (Conditional Probability) *The probability of an event A given that another event B has occurred is defined as follows:*

$$p(A|B) = \frac{p(A \cap B)}{p(B)}$$

Definition 6 (Joint Probability) *The joint probability mass function of two discrete random variables X, Y is defined as:*

$$p(x, y) = p(X = x \text{ and } Y = y) = p(y|x)\ p(x) = p(x|y)\ p(y)$$

Information Theory

Information Theory [50, 15] provides the mathematical foundation to reason about "*information*" in a quantitative sense. Its main concept is "entropy", which measures the uncertainty about a random variable.

Definition 7 (Entropy) *Given a random variable $X : \Omega \rightarrow N$ with the probability mass function $p(x)$. Shannon entropy is defined as:*

$$F(X) = -\sum_{x \in N} p(x) \ \log p(x)$$

The logarithm is to the base 2 by convention, hence the units are *bits*. We also write $F(p)$ for the above quantity.

Lemma 1 *Given a random variable $X : \Omega \rightarrow N$ with the probability mass function $p(x)$, the entropy of X is bounded by:*

$$0 \leq F(X) \leq \log |N|$$

where $|N|$ is the cardinality of the set N.

Proof: By definition $0 \leq p(x) \leq 1$ for all $x \in N$, which leads to $p(x) \log p(x) \leq 0$. As a result, the first part of the lemma $F(X) \geq 0$ holds.

The second part of the lemma is proved by using Lagrange multiplier to find the maximal entropy. For simplicity, we define a system of indices for all elements in N as follows: $x_1, x_2, \ldots, x_n$ (which also means $|N| = n$).

We write p_i for $p(X = x_i)$, and maximize the function:

$$f(p_1, p_2, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log p_i$$

subject to the condition of probability:

$$g(p_1, p_2, \ldots, p_n) = \sum_{i=1}^{n} p_i = 1$$

Lagrange multipliers are used to find the maximum entropy $\vec{p}^* = \{p_1^*, p_2^*, \ldots, p_n^*\}$ across all probability distributions $\vec{p} = \{p_1, p_2, \ldots, p_n\}$ of X. It is required that:

$$\frac{\partial}{\partial \vec{p}}(f + \lambda(g - 1))\bigg|_{\vec{p}=\vec{p}^*} = 0$$

This equation is expanded into a system of n equations ($i = 1, 2, \ldots, n$):

$$\frac{\partial}{\partial p_i}\left\{-\left(\sum_{j=1}^{n} p_j \log_2 p_j\right) + \lambda\left(\sum_{j=1}^{n} p_j - 1\right)\right\}\Bigg|_{p_i = p_i^*} = 0$$

Carrying out the differentiation of these n equations results in:

$$-\left(\frac{1}{\ln 2} + \log_2 p_i^*\right) + \lambda = 0$$

Since the value of p_i^* only depends on λ, they are all equal:

$$p_1^* = p_2^* = \ldots = p_n^*$$

Moreover, by definition of probability distribution:

$$\sum_{i=1}^{n} p_i^* = 1$$

This leads to:

$$p_1^* = p_2^* = \ldots = p_n^* = \frac{1}{n}$$

The maximal entropy is:

$$F(p^*) = -\sum_{1}^{n} p_i \log p_i = -\sum_{1}^{n} \frac{1}{n} \log \frac{1}{n} = n = |N|$$

This proves the second part Lemma 1: $F(X) \leq \log |N|$, equality holds when X has uniform distribution.

Definition 8 (Joint Entropy) *Given two random variables $X : \Omega_X \to N_X$ and $Y : \Omega_Y \to N_Y$, the joint Shannon entropy is defined as follows:*

$$F(X,Y) = -\sum_{x \in N_X} \sum_{y \in N_Y} p(x,y) \log p(x,y)$$

where $p(x,y)$ is the joint probability mass function of X and Y.

The joint entropy measures how much uncertainty there is in the two random variables X and Y taken together.

Definition 9 (Conditional Entropy) *Given two random variables $X : \Omega_X \to N_X$ and $Y : \Omega_Y \to N_Y$, the conditional entropy of X given the knowledge of Y is defined as follows:*

$$F(Y|X) = \sum_{x \in N_X} p(x)\, F(Y|X = x)$$

The definition of conditional entropy can be expanded as:

$$\begin{aligned}
F(Y|X) &= -\sum_{x\in N_X} p(x) \sum_{y\in N_Y} p(y|x)\, \log p(y|x) \\
&= -\sum_{x\in N_X}\sum_{y\in N_Y} p(x,y)\, \log p(y|x) \\
&= -\sum_{x\in N_X}\sum_{y\in N_Y} p(x,y)\, \log\left(\frac{p(x,y)}{p(x)}\right) \\
&= -\sum_{x\in N_X}\sum_{y\in N_Y} p(x,y)\, \log p(x,y) + \sum_{x\in N_X}\sum_{y\in N_Y} p(x,y)\, \log p(x) \\
&= F(X,Y) + \sum_{x\in N_X} p(x)\, \log p(x) \\
&= F(X,Y) - F(X)
\end{aligned}$$

If X and Y are independent, then knowing the value of X does not change uncertainty about Y, which means $F(Y|X) = F(Y)$. At the other extreme, if the value of Y is completely determined by the value of X, then there is no uncertainty about Y when already knowing X, and $F(Y|X) = 0$ holds.

Lemma 2 *Given two random variables X and Y, the following inequalities hold:*

$$F(X,Y) \geq \max\{F(X), F(Y)\}$$

Proof: Without loss of generality, assume that $\max\{F(X), F(Y)\} = F(X)$, we need to prove:

$$F(X,Y) \geq F(X) \iff F(X) + F(Y|X) \geq F(X) \iff F(Y|X) \geq 0$$

By definition of conditional entropy, we have $F(Y|X) = \sum p(x)\, F(Y|X=x)$. Similar to the proof of the first part of Lemma 1, we can show that $F(Y|X=x) \geq 0$ holds. As a result, $F(Y|X) \geq 0$ also holds. This proves the inequality in Lemma 2. Equality holds if and only if $F(Y|X) = 0$ holds, which means the value of Y is completely determined by the value of X.

Definition 10 (Mutual Information) *Given two random variables $X : \Omega_X \to N_X$ and $Y : \Omega_Y \to N_Y$, the mutual information or mutual dependence between X and Y is defined as:*

$$I(X;Y) = \sum_{y\in N_Y}\sum_{x\in N_X} p(x,y) \log\left(\frac{p(x,y)}{p(x)\, p(y)}\right)$$

The mutual information can be rewritten as:

$$\begin{aligned}
I(X;Y) &= \sum_{y \in N_Y} \sum_{x \in N_X} p(x,y) \log\left(\frac{p(x|y)}{p(x)}\right) \\
&= -\sum_{y \in N_Y} \sum_{x \in N_X} p(x,y) \log p(x) + \sum_{y \in N_Y} \sum_{x \in N_X} p(x,y) \log p(x|y) \\
&= F(X) - F(X|Y)
\end{aligned}$$

Measurement of leakage

In the context of information flow analysis, we are interested in the value of variables `H`, `L`, and `O` as in our attacker model in Figure 2.1. For each variable, we denote its sample space and random variable as follows: $X_H : \Omega_H \to N_H$, $X_L : \Omega_L \to N_L$, and $X_O : \Omega_O \to N_O$.

The random variable X_H represents the *a priori* knowledge of the adversary about the secret data `H`. Consider, for example, the password checking program in Figure 2.2: if the adversary already knows that the password `H` is “abc”, this means $\Omega_H = \{\text{“abc”}\}$, and $p(X_H(\text{“abc”})) = 1$. Or suppose that Ω_H contains all possible strings of length up to 30, but the adversary knows in advance that the victim has the habit to use his daughter’s name, “Alice”, as password. In this case, in the probability distribution of X_H, the value of $p(X_H(\text{“Alice”}))$ is close to 1. The most general case is when the adversary does not have any information about the password, which means X_H has a uniform distribution.

The entropy $F(X_H)$ measures the initial uncertainty of the adversary about H. When the adversary already knows that the password is “abc”, there is no uncertainty at all. $p(X_H(\text{“abc”})) = 1$ leads to $F(X_H) = 0$. On the other hand, if the adversary has no information about the password in advance, his uncertainty is maximal.

After an execution of the program `P`, there is some information about `H` leaked via information flow from `H` to `O`. As a result, the adversary learns some information, and reduces his uncertainty about `H`. The difference in his uncertainties before and after the observation is the amount of leakage:

$$\texttt{leakage} = \texttt{initial uncertainty} - \texttt{remaining uncertainty} \tag{2.2}$$

The remaining uncertainty about `H` given the knowledge of O is, by definition, the conditional entropy $F(X_H|X_O)$. As a result, leakage is calculated as follows:

$$\Delta_F(X_H) = F(X_H) - F(X_H|X_O) = I(X_H; X_O)$$

In the case of non-interference, `O` is independent from `H`, or in other words `O` is not interfered by `H`, $F(X_H|X_O) = F(X_H)$ holds. As a result, the leakage is $\Delta_F(X_H) = 0$.

$\Delta_F(X_H) = I(X_H; X_O)$ is always positive. This property can be proved by using Jensen's inequality. The interested reader is referred to the textbook by Cover and Thomas [15] for a proof in details.

2.1.3 Problem Statement

Programs such as the one in our attacker model in Figure 2.1 can be viewed as a channel, in which (confidential) information flows from the input `H` to the output `O`. The *channel capacity* C is defined as the maximum amount of information can be transmitted in this channel. More formally, C is maximum mutual information of X_H and X_O over all possible input distributions p of X_H.

Theorem 1 (Channel Capacity) *Given a program `P` as a discrete channel from `H` to `O`, the channel capacity C is computed by:*

$$C = \log |N_O|$$

Proof: By definition of conditional entropy, the leakage can be rewritten as:

$$\begin{aligned}\Delta_F(X_H) &= F(X_H) - F(X_H|X_O) = F(X_H) - (F(X_H, X_O) - F(X_O)) \\ &= F(X_O) - (F(X_H) - F(X_H, X_O))\end{aligned}$$

From Lemma 2, we have $F(X_H, X_O) \geq F(X_H)$. This leads to:

$$\Delta_F(X_H) \leq F(X_O)$$

From Lemma 1: $F(X_O) \leq \log |N_O|$, which means $\Delta_F(X_H) \leq \log |N_O|$ holds. Equality holds when both of the equalities in Lemma 1 and Lemma 2 hold. Which means the value of `O` is completely determined by the value of `H`, and X_O has uniform distribution.

As such, counting the number of observables is the basis of state-of-the-art QIF analysis, e.g. [20, 18, 59, 60], and also the basis for this thesis. The channel capacity theorem also justifies the following:

Definition 11 (The QIF problem) *Given a program `P`, QIF is the problem of counting N, the number of possible outputs of `P`.*

2.2 Logical Satisfiability Problems

This section provides some logical concepts used throughout in this thesis. The interested readers are referred to [51] for more details.

2.2.1 Propositional Satisfiability

Definition 12 (Propositional atom) *A propositional atom or Boolean atom is a statement or an assertion that must be true or false.*

Examples of Boolean atoms are: "all humans are mortal" and "program `P` leaks k bits". Boolean atoms are the most basic building blocks of *propositional formulas*, each Boolean atom A_i is also a formula.

Propositional formulas are constructed from Boolean atoms using *logical connectives*: `not` ($\neg$), `and` ($\wedge$), `or` ($\vee$), and `imply` ($\rightarrow$). That means if φ_1 and φ_2 are formulas, then $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, and $\varphi_1 \rightarrow \varphi_2$ are also formulas. For example, $(\neg A_1 \wedge A_2) \rightarrow A_3$ is a propositional formula.

A Boolean atom A_i or its negation $\neg A_i$ is called a *literal*. We denote by $Atom(\varphi)$ the set $\{A_1, A_2, \ldots A_n\}$ of Boolean atoms that occur in φ. The truth of a propositional formula φ is a function of the truth values of the Boolean atoms it contains.

We denote by $\top$ and $\bot$ the truth values of true and false, respectively. The set $\mathbb{B} = \{\top, \bot\}$ is called *Boolean domain.*

Definition 13 *Given a propositional formula φ, a truth assignment μ of φ is defined as a function which assigns each Boolean atom of φ a truth value:*

$$\mu : Atom(\varphi) \rightarrow \{\top, \bot\}$$

A partial truth assignment of a formula φ is a function $\mu : \mathcal{A} \rightarrow \{\top, \bot\}$ where $\mathcal{A}$ is any subset of $Atom(\varphi)$. A (partial) truth assignment μ satisfies a propositional formula φ, denoted by $\mu \models \varphi$, if φ is evaluated to $\top$ under μ. For example $\mu : A_3 \mapsto \bot$ satisfies the formula $(\neg A_1 \wedge A_2) \rightarrow A_3$.

A formula φ is *satisfiable* if there exists a (partial) truth assignment such that $\mu \models \varphi$. If $\mu \models \varphi$ for every truth assignment μ, then φ is *valid.* Either a formula is valid or its negation is satisfiable.

Definition 14 *A propositional formula φ is in Conjunctive Normal Form (CNF) if and only if it is a conjunction of disjunctions of literals:*

$$\varphi = \bigwedge_{i=1}^{N} \bigvee_{j=1}^{M_i} l_{ij}$$

Any propositional formula can be converted to CNF by an algorithm with worst-case linear time [61, 62].

Definition 15 (The SAT problem) *Given a propositional formula φ in CNF, the Boolean Satisfiability Problem (SAT) is the problem of finding an assignment μ that satisfies φ.*

As the SAT problem is NP-complete, there is no algorithm that efficiently works on all instances of the problem. However, there are two main families of algorithms for state-of-the-art SAT solvers: DPLL [63, 64] and Stochastic Local Search [65]. This thesis focuses on the DPLL algorithm, which will be described in detail later in chapter 4.

2.2.2 Model Counting

Definition 16 (The #SAT problem) *Given a propositional formula φ, the Model Counting problem (#SAT) is the problem of counting all the solutions of the SAT problem.*

2.2.3 Satisfiability Modulo Theories

We assume countable sets of variables $\mathcal{V}$, function symbols $\mathcal{F}$ and predicate symbols $\mathcal{P}$. A first-order logic *signature* is defined as a partial function $\Sigma : \mathcal{F} \cup \mathcal{P} \to A$ ($A \subset \mathbb{N}$). Each $a \in A$ corresponds to an *arity* of an symbol. Obviously, a 0-ary predicate is a Boolean atom, and a 0-ary function symbol is called a *constant*.

A Σ-term τ is either a variable $x \in \mathcal{V}$ or it is built by applying function symbols in $\mathcal{F}$ to Σ-terms, e.g. $f(\tau_1, \ldots, \tau_n)$ where $f \in \mathcal{F}$ and $\Sigma(f) = n$. For example, $f(x, g(x))$ is a Σ-term if $\Sigma(f) = 2$ and $\Sigma(g) = 1$.

Definition 17 *If $\tau_1, \ldots, \tau_n$ are Σ-terms, and $p \in \mathcal{P}$ is a predicate symbol such that $\Sigma(p) = n$, then $p(\tau_1, \ldots, \tau_n)$ is a Σ-atom.*

A Σ-atom or its negation is called Σ-literal. We use the infix equality sign "$=$" as a shorthand for the equality predicate. If τ_1 and τ_2 are Σ-terms, then the Σ-atom $\tau_1 = \tau_2$ is called Σ-equality. $\neg(\tau_1 = \tau_2)$ or $\tau_1 \neq \tau_2$ is called Σ-disequality.

Σ-atoms are the most basic building blocks of Σ-formulas, each Σ-atom $p(\tau_1, \ldots, \tau_n)$ is also a Σ-formula. Similar to the construction of propositional formulas, Σ-formulas are constructed from Σ-terms which are glued together by `universal quantifiers` ($\forall$), `existential quantifiers` ($\exists$), and logical connectives. That means if φ_1 and φ_2 are Σ-formulas, then $\forall x : \varphi_1$, $\exists x : \varphi_1$, $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, and $\varphi_1 \rightarrow \varphi_2$ are also Σ-formulas.

A *quantifier-free* Σ-formula does not contain quantifiers; a *sentence* is a Σ-formula without free variables. A first-order theory is defined as follows:

Definition 18 (First-order theory) *A first-order theory $\mathcal{T}$ is a set of first-order sentences with signature Σ.*

A Σ-structure M is a triple $(|M|, \Sigma, \mathcal{I})$ consisting of a non-empty domain $|M|$, a signature Σ, and an interpretation $\mathcal{I}$. The interpretation $\mathcal{I}$ assigns meanings to symbols of Σ: for each function symbol $f \in \mathcal{F}$ such that $\Sigma(f) = n$, f is assigned a n-ary function $\mathcal{I}(f)$ on the domain $|M|$; for each predicate symbol $p \in \mathcal{P}$ such that $\Sigma(p) = n$, p is assigned a n-ary predicate $\mathcal{I}(p)$, represented by a subset of $|M|^n$. For each variable $x \in \mathcal{V}$, $\mathcal{I}(x) \in |M|$.

A Σ-structure M is a model of the Σ-theory $\mathcal{T}$ if it satisfies all sentences in $\mathcal{T}$. If a Σ-formula is satisfiable in a model of $\mathcal{T}$, then it is called *$\mathcal{T}$-satisfiable.*

Henceforth, for simplicity we will omit the prefix "$\Sigma-$" from term, atom, formula, etc. Instead, we will often use the prefix "$\mathcal{T}$-" to denote "in the theory $\mathcal{T}$".

We define a bijective function $\mathcal{BA}$ (*Boolean abstraction*) which maps Boolean atoms into themselves and $\mathcal{T}$-atoms into fresh Boolean atoms. The *Boolean refinement* function $\mathcal{BR}$ is then defined as the inverse of $\mathcal{BA}$, which means $\mathcal{BR} = \mathcal{BA}^{-1}$.

Definition 19 (The SMT problem) *Given a theory or a combination of theories $\mathcal{T}$ and a Σ-formula φ, the Satisfiability Modulo Theories problem (SMT) is the problem of deciding $\mathcal{T}$-satisfiability of φ.*

Most of the work on SMT focus on quantifier-free formulas, and *decidable* first-order theories. The SMT problem is NP-hard, as it subsumes the SAT problem.

Most state-of-the-art SMT solvers implement the DPLL($\mathcal{T}$) algorithm, which is the integration of two components: (i) an *enumerator* integrating a DPLL-based SAT solver enumerates truth assignments satisfying the Boolean abstraction of the input formula; (ii) $\mathcal{T}$-solvers validate the consistency w.r.t. theories $\mathcal{T}$ of the (partial) assignment produced by the SAT solver. The DPLL($\mathcal{T}$) algorithm will be described in more details later in chapter 4.

2.3 The programming language and the program

For simplicity, we illustrate our methodologies using the *guarded command* language [66] instead of C or Java. The grammar of the language is depicted in Figure 2.4.

$$\begin{array}{lcll}
\textit{program} & \equiv & \textit{stmt}* & \\
\textit{stmt } s & \equiv & \textbf{assume } e \mid \textbf{assert } e \mid \ v = e & \mid \textbf{if } e \textbf{ then goto } s \textbf{ else goto } s \\
v & \in & \textit{Var} & \textit{(variables)} \\
e & \in & \textit{Exp} & \textit{(expressions)}
\end{array}$$

Figure 2.4: The guarded command language

In this thesis, we focus on safety properties: note that the two commands $assume(c)$ and $assert(c)$ are powerful enough to encode expressive temporal properties [67], and also support assume-guarantee style compositional reasoning. Any program can be viewed as a system that transits between states. There are many ways to describe this system depending on how much detail of the program that needs to be captured. Apart from chapter 4, in this thesis a program P is modelled as a transition system as follows:

$$P = (S, I, F, T) \tag{2.3}$$

where S is the set of program states; $I \subseteq S$ is the set of initial states; $F \subseteq S$ is the set of final states; and $T \subseteq S \times S$ is the transition relation.

Under this setting, a trace of (a concrete) execution of the program P is represented by a sequence of states: $\rho = s_0 s_1 .. s_k$ such that $s_0 \in I, s_k \in F$ and $\langle s_i, s_{i+1} \rangle \in T$ for all $i \in \{0, .., k-1\}$.

We define two functions *init* and *fin* to get the initial state and final state of ρ: $init(\rho) = s_0$ and $fin(\rho) = s_k$ The semantics of P is then defined as the set $\mathcal{R}$ of all possible traces.

In the context of the information flow problem, we assume that each initial state $s \in I$ is a pair $\langle H, L \rangle$, i.e. $I = I_H \times I_L$, in which H is the confidential component to be protected and L is the public component that may be controlled by an attacker.

2.4 Formal Methods

Formal methods refer to a set of mathematical-based techniques used in Computer Science for the specification and verification of software and hardware systems. These techniques base their foundations on several conceptual frameworks: automata theory, logic calculi, formal languages, program semantics and so on.

The act of using formal methods to prove or to disprove the correctness of a system, with respect to a certain property, is called *formal verification*. Compared to testing, formal verification is much more expensive. However, it is crucial in the development of systems whose failure can cause huge financial lost or even cost human lives. History has witnessed several computer-related disasters that could have been prevented if formal verification had been used [39].

There are three main components involved in the formal verification of a hardware or software system:

- *A formal model* of the system. Models used in formal verification vary in the level of abstraction, from an automaton describing status changes of the system, to source code or machine code of the system.
- *A formal specification*, often described in a formal language. These formal languages also have different power of expressiveness.
- *A formal method*, implemented in a fully or partially automated tool, to prove or disprove the conformance of the formal model to the formal specification.

There are three possible cases for the result: the first case is the program conforms to the specification; the second case is the system violates the specification, in which a counterexample might be returned; the final case is the tool fails to prove or disprove within a period of time.

Naturally, there is a trade-off between the level of abstraction of the model and the expressiveness of the specification. Typically, formal methods-based tools can check complicated specification on highly abstract models, and simple specification on detailed models. In this thesis, the formal model of a program is C source code or Java bytecode. The formal specification is the reachability of some assertions in the source code or bytecode. The formal methods we used are Bounded Model Checking and Symbolic Execution.

2.4.1 Bounded Model Checking and CBMC

The aim of Bounded Model Checking [68] (BMC) is to find bugs or to prove their absence up to some bounded k number of transitions. Recall that a program P is modelled as a transition system $P = (S, I, F, T)$, and a trace is represented by a sequence of states $\rho = s_0 s_1 .. s_k$.

A state s_e is called an error state if the program reaches an error, e.g. buffer overflow, or it violates a specification at s_e. A trace σ_e that contains one or more error states is called an error trace.

A trace can be also seen in logical form: the set I and the relation T can be written as their characteristic functions: $s_0 \in I$ iff $I(s_0)$ holds; $\langle s_i, s_{i+1} \rangle \in T$ iff $T(s_i, s_{i+1})$ holds. In this way, a trace ρ is represented by the formula:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \tag{2.4}$$

Clearly the transition system P is a model for such a formula, i.e. P is a model for all formulas representing traces of the program. As BMC aims to find bugs or prove their absence up to some bounded k number of transitions, it explores all traces $\rho = s_0 s_1 .. s_k$ of the program P, in which s_k needs not to be in F.

Notice that because of the bound k there are only a finite number of traces to explore. Hence we can represent the bounded program as a formula $\mathcal{C}$ which is a conjunction of formulas, whose conjoints are possible traces. Notice formulas can also represent symbolic traces, for example if in a formula the value of a program variable is left unspecified then there can be several concrete traces satisfying that formula. Formulas satisfied by set of concrete traces can be referred to as symbolic traces.

CBMC is a verification tool implementing BMC for checking ANSI-C programs. It translates a C program into a logical formula $\mathcal{C}$ which is then used as a model for the property $\mathcal{P}$ to be verified. The property is verified by the C program iff $\mathcal{C} \wedge \mathcal{P}$ is valid. This can be checked by a satisfiability solver on $\mathcal{C} \wedge \neg\mathcal{P}$. In fact if $\mathcal{C} \wedge \neg\mathcal{P}$ is true in the model then one trace will satisfy $\neg\mathcal{P}$ hence the property is not valid. On the other hand if $\mathcal{C} \wedge \neg\mathcal{P}$ is false in the model then no trace will satisfy $\neg\mathcal{P}$ hence $\mathcal{P}$ is valid.

2.4.2 Symbolic Execution and Symbolic PathFinder

Symbolic Execution [46] (SE) is a programming analysis technique which executes programs on unspecified inputs, by using symbolic inputs instead of concrete data. For each executed program path, SE builds a *path condition* which is the condition on the inputs for the execution to follow that path, according to the branching conditions in the code.

A path condition pc is initialized as empty, and it doesn't change when executing non-branching instructions. For an **if** statement with condition c, there are three possible cases: (**i**) $pc \vdash c$: SE chooses the **then** path; (**ii**) $pc \vdash \neg c$: SE chooses the **else** path; (**iii**) $(pc \nvdash c) \wedge (pc \nvdash \neg c)$: SE executes both paths: in the **then** path, it updates the path condition $pc_1 = pc \wedge c$, in the **else** path it updates the path condition $pc_2 = pc \wedge \neg c$.

In classical SE, the satisfiability of the path condition is checked at every branching point, using off-the-shelf constraint solvers. In this way only feasible program paths are explored. A symbolic execution tree characterizes the execution paths followed during the symbolic execution of a program. The nodes represent program (symbolic) states and the arcs represent transitions between states.

Symbolic PathFinder (SPF) is a SE framework built on top of the Java PathFinder (JPF) model checking tool-set for Java bytecode analysis. It implements a bytecode interpreter that replaces the standard, concrete execution semantics of bytecodes with a non-standard symbolic execution.

SPF implements classical SE, in its default mode SPF only explores feasible symbolic paths. However, it also has an option to run without constraint solving, which means for an **if** statement with condition c, both $pc \nvdash c$ and $pc \nvdash \neg c$ are assumed to be true. As a result of this option, SPF will explore all the possible paths (feasible and infeasible) through the program, up to the given bound. This particular option will be used later in chapter 5 for the design of a concurrent bounded model checker.

Chapter 3

Model Counting Modulo Theories

This chapter introduces the #SMT problem and a #SMT-based technique for QIF analysis, which provides a dramatic improvement on state-of-the-art implementations of QIF analysis. On the theoretical side, this work establishes a connection between fundamental verification algorithms and QIF. This connection is exploited to mitigate the state explosion problem by developing a novel approach for QIF based on SMT. More specific contributions are:

1. Introduction of a new research problem, Model Counting Modulo Theories or #SMT, and its applications to QIF.
2. A framework, called #DPLL($\mathcal{T}$), to build a solver for #SMT-based QIF.
3. A prototyping tool for QIF analysis: **sqifc** built on top of CBMC [48].
4. Analysis of complex code, including recent vulnerabilities from the National Vulnerability Database of the US government [69] and anonymity protocols.

3.1 Illustrative Example

To illustrate our approach, consider the data sanitization program `P` from [17, 18], shown in Figure 3.1. All variables are unsigned integers. It is straightforward to show that only integer values from 8 to 23 are possible outputs of this program. An attacker has hence available 16 possible output observations: observing outputs 9 .. 23 will know the secret `H` is 1 .. 15 and observing 8 will know the secret is 0 or greater than 15. Assuming the attacker has no prior knowledge of the secret `H` apart that is a 32 bits variable his *a-priori* probability of guessing the value of `H` in one try is

```
L = 8;
if (H < 16)
  O = H + L;
else
  O = L;
```

Figure 3.1: Data sanitization

$\frac{1}{2^{32}}$, and the expected probability of guessing the secret in one try after observing the outputs is:

$$\frac{15}{2^{32}} + \frac{2^{32}-15}{2^{32}} \frac{1}{2^{32}-15} = \frac{15}{2^{32}} + \frac{1}{2^{32}} = \frac{16}{2^{32}}$$

We can measure the leakage of the program as the difference (of the $-\log$ base 2) between the probability of guessing the secret before and after observing the outputs of the program; in this case:

$$-\log(\frac{1}{2^{32}}) - (-\log\frac{16}{2^{32}}) = \log(16) = 4$$

The result of this measurement, $\log(16) = \log$ (number of output observations), is an alternative explanation for theorem 1, i.e. theorem of channel capacity, that we have proved in the previous chapter. Our goal is to develop an efficient automated technique to compute this number of output observations.

Notice that the output `O` is stored in the computer memory as a string of 32 bits $b_1 b_2 \ldots b_{32}$, which can be represented by a set of Boolean variables $V_I = \{p_1, p_2, \ldots p_{32}\}$ such that $p_i = \top$ if and only if b_i is 1, and $p_i = \bot$ if and only if $b_i = 0$. Thus, each possible value of `O` corresponds to a truth assignment for V_I. For example, `O` = `1000b` corresponds to: $p_1 \mapsto \top, p_2 \mapsto \bot, p_3 \mapsto \bot, p_4 \mapsto \top, p_5 \mapsto \bot, \ldots, p_{32} \mapsto \bot$.

Since there are 16 possible values of `O` from 8 to 23, there are 16 possible truth assignments for V_I. We can view these truth assignments as partial models of a logical satisfiability problem on a logical formula φ_P. Obviously, this formula φ_P characterizes the behaviour of the program `P` because of the correspondence between a possible output of `P` and a partial model of φ_P.

Our goal is to count all possible values of `O`, which corresponds to counting all models of φ_P with respect to the set V_I. In other words, we cast the problem of measuring information leaks of `P` to a model counting problem on φ_P. In the next sections, we will give a formal definition for this problem, which we name Model Counting Modulo Theories, and develop an algorithm for QIF analysis based on it.

3.2 Model Counting Modulo Theories

In the previous chapter, we have recalled three logical satisfiability problems that have been studied extensively in recent years, namely SAT, #SAT, and SMT. Their relations with each other are depicted in Figure 3.2. Since propositional logic is a special case of first-order theories whose signature contains only 0-ary predicates, the SAT problem is a simple case of the SMT problem. Moreover, the #SAT problem is a generalization of the SAT problem from finding one model to counting all models. What is lacking in this big picture is a satisfiability problem that is a generalization of #SAT to first-order theories, and is a generalization of SMT from finding one model to counting all models.

$$\begin{tikzcd} \text{SAT} \arrow[r, "\text{generalize to first-order theories}"] \arrow[d, "\text{generalize to counting models}"] & \text{SMT} \arrow[d, "\text{generalize to counting models}"'] \\ \#\text{SAT} \arrow[r, "\text{generalize to first-order theories}"] & ? \end{tikzcd}$$

Figure 3.2: Logical satisfiability problems

Note that it is not always possible to count models in SMT as most SMT theories permit an infinite number of models. For example, there are uncountably infinite models for the formula: $\varphi = A \wedge (x > 1)$, where the background theory $\mathcal{T}$ is $\mathcal{LA}(\mathbb{Q})$, the theory of linear arithmetic over the rationals, which means $x \in \mathbb{Q}$.

Here we restrict the problem to counting all models with respect to a set of Boolean variables. This restriction guarantees that there are always finite number of models regardless of the background theories.

Definition 20 (The #SMT problem) *Given a theory or a combination of theories $\mathcal{T}$ and a Σ-formula φ, Model Counting Modulo Theories (#SMT) is the problem of counting all models M of $\mathcal{T}$ with respect to a set of Boolean variables V_I such that φ is $\mathcal{T}$-satisfiable in M.*

With this new #SMT problem, our picture of logical satisfiability problems in Figure 3.3 is now complete. The #SMT problem is a generalization of both the #SAT problem and the SMT problem.

$$\begin{tikzcd}
\text{SAT} \arrow[r, "\text{generalize to first-order theories}"] \arrow[d, "\begin{array}{l}\text{generalize to}\\ \text{counting models}\end{array}"'] & \text{SMT} \arrow[d, "\begin{array}{r}\text{generalize to}\\ \text{counting models}\end{array}"] \\
\#\text{SAT} \arrow[r, "\text{generalize to first-order theories}"] & \#\text{SMT}
\end{tikzcd}$$

Figure 3.3: Logical satisfiability problems

Recall that most state-of-the-art SMT solvers are the integration of two components: (i) an *enumerator* integrating a SAT solver enumerates truth assignments satisfying the Boolean abstraction of the input formula; (ii) $\mathcal{T}$-solvers validate the consistency w.r.t. theories $\mathcal{T}$ of the (partial) assignment produced by the SAT solver.

Naturally, an SMT solver can be extended into a #SMT solver by replacing the SAT solver with a #SAT solver that can explicitly enumerate all models. Early algorithms for #SAT, e.g. [70], were extensions of the DPLL algorithm [63, 64] that could enumerate all models. However, modern #SAT solvers, e.g. RelSat [71] and c2d [72], are more efficient thanks to smarter approaches that exploit the structure of the clauses and avoid explicitly enumerating models. In the context of #SMT, however, explicit enumeration of all models is necessary since we need to check the consistency of each model w.r.t. the background theory $\mathcal{T}$.

3.3 Quantitative Information Flow as #SMT

We assume the setting in our attacker model in Figure 2.1: a program `P` that takes secret input `H`, public input `L` and producing public output `O`. As per definition 11, the quantitative information flow problem is to count the number N of possible values of the output `O`, which is an M-bit data $b_1 b_2 \dots b_M$.

Assuming that we have a (first-order) formula φ_P with the following properties: (i) φ_P contains a set of Boolean variables $V_I := \{p_1, p_2, .., p_M\}$; (ii) $p_i = \top$ if and only if

b_i is 1, and $p_i = \perp$ if and only if $b_i = 0$. Under these settings, the QIF problem of counting N can be viewed as a #SMT problem on the formula φ_P and the set V_I of Boolean variables.

Under this view, on one hand we have a program P to perform QIF analysis, and a tool box of formal methods. On the other hand, we have a logical formula φ_P to compute #SMT and the DPLL($\mathcal{T}$) algorithm. Hence, there are two possible approaches to our QIF problem: the first one is to construct such a formula φ_P from the program P, then solving it with an #SMT solver; the second one is to use formal methods in a way that mimics the DPLL($\mathcal{T}$) algorithm.

$$\begin{tikzcd}
\mathtt{P} \arrow[r, leftrightarrow] & \varphi_P \\
\text{QIF} & \text{\#SMT} \\
\text{Formal methods} & \text{DPLL}(\mathcal{T})
\end{tikzcd}$$

The first approach is more intuitive. However, it is also more complicated, since there is no available off-the-shelf #SMT solver. Therefore, we will leave it for chapter 7. In this chapter, we will explore the second approach, which is much simpler yet powerful enough to analyse real-world programs, and to outperform dramatically the state-of-the-art technique.

An extremely naive technique for QIF using formal methods is to check each number one by one if it can be a value of the output O. This can be done as follows:

```
$N = 0$
**for all** $v$ from 0 to $2^M$ **do**
    **if** (**assert** O != $v$ is violated)
        $N \leftarrow N + 1$
**return** $N$
```

We make an assertion that the output O is always different from v, then checking the validity of this assertion using formal methods. If the assertion is valid, then v is not a possible value of O. On the contrary, if the assertion is violated, then O can take the value v, and we increase the counter.

Assuming that we have a very powerful tool that can verify each assertion in one second, the procedure would take around 2^{32} seconds, which is approximately 136 years. Therefore, this technique is impractical. The reason is that it checks one concrete value at a time, and thus it is vulnerable to the state-space explosion problem.

Although this technique is naive, it inspires us to develop a procedure to process multiple values at a time.

Consider again the set of Boolean variables $V_I := \{p_1, p_2, .., p_M\}$, for example the partial truth assignment $\mu = \{p_1 \mapsto \top, p_2 \mapsto \bot\}$ represents 2^{M-2} concrete values: all the bit configurations over M bits where the first bit is 1 and the second bit is 0. Thus, if we can verify that φ_P cannot be satisfiable in μ, we can ignore those 2^{M-2} values. Although for this approach we do not construct the formula φ_P, checking that φ_P cannot be satisfiable in μ can be done by using formal methods to check the assertion that $\neg(b_1 = 1 \;\&\&\; b_2 = 0)$ is valid.

So a partial truth assignments of V_I is a symbolic representation for a set of values of the output `O`. Based on this, we develop a technique, called *Symbolic Quantitative Information Flow* that mimics the DPLL($\mathcal{T}$) algorithm and explores the state space. Recall that DPLL($\mathcal{T}$) algorithm consists of a DPLL-based SAT solver to enumerate (partial) truth assignments, and a $\mathcal{T}$-solver to check the consistency of these truth assignment. As we use formal methods, in particular model checking, to check if the formula φ_P can be satisfiable in a partial truth assignment μ, the model checker plays the role of the $\mathcal{T}$-solver.

$$\begin{tikzcd}
\mathtt{P} \arrow[rr, leftrightarrow] & & \varphi_P \\
\text{Model Checker} & & \mathcal{T}\text{-solver}
\end{tikzcd}$$

3.4 Symbolic Quantitative Information Flow

Our first step is to construct the set of Boolean variables V_I that we have described. In a language that supports bitwise operators such as C/C++ and Java, this can be done by instrumenting the program `P` as follows:

```
for all $i$ from 1 to $M$ do
    $b_i$ = (O >> ($i$ - 1)) & 1
    if ($b_i$ == 1)
        $p_i \leftarrow \top$
    else
        $p_i \leftarrow \bot$
```

Figure 3.4: Program instrumentation

```
function SymbolicQIF($V_I, \varphi_P$)
    $\varPsi = \epsilon$, $pc = \epsilon$, $N = 0$, $i = 1$
    EarlyPrunning($V_I$)
    SymCount($V_I, \varPsi, \varphi_P, N, pc, i$)
return $\varPsi$, $\log_2(N)$
```

Figure 3.5: Symbolic QIF analysis

```
function SymCount($V_I, \varPsi, \varphi_P, N, pc, i$)
    if ($N \geq 2^k$)
        return $Insecure$
    Extract $p_i$ from $V_I$
    $pc_1 \leftarrow pc \wedge p_i$
    if ($\mathcal{T}$-solver($\varphi_P, pc_1$))
        if ($i == M$)
            $\varPsi \leftarrow \varPsi \cup \{pc_1\}$
            $N \leftarrow N + 1$
        else
            SymCount($V_I, \varPsi, \varphi_P, N, pc_1, i + 1$)
    $pc_2 \leftarrow pc \wedge \neg p_i$
    if ($\mathcal{T}$-solver($\varphi_P, pc_2$))
        if ($i == M$)
            $\varPsi \leftarrow \varPsi \cup \{pc_2\}$
            $N \leftarrow N + 1$
        else
            SymCount($V_I, \varPsi, \varphi_P, N, pc_2, i + 1$)
```

Figure 3.6: Symbolic counting for QIF

This instrumentation guarantees that the variable p_i corresponds to the i^{th} bit of the output `O`. A high level framework to explore the state-space and quantify the leaks of confidential data is described by the procedures `SymbolicQIF` and `SymCount` in Figure 3.5 and 3.6.

V_I, Ψ, φ_P and N are passed by reference, while pc and i are passed by value. V_I is the symbolic representation of the output described in the previous section, φ_P is the formula representing the program P and Ψ is the set of models of φ_P. N is the cardinality of Ψ, and the procedure `SymbolicQIF` returns $\log_2(N)$ as the channel capacity.

M is the size of the output data type, e.g. $M = 32$ if O is a 32-bit integer, and i is the depth of the recursive call. The parameter pc is a partial assignment of V_I, it is incrementally updated when the search progresses. In `SymCount`, $\mathcal{T}$-solver(φ_P, pc) means the $\mathcal{T}$-solver is called to check if there is a model of φ_P where pc is (assigned to) $\top$.

We illustrate the algorithm `SymCount` by running it on a simple example (we ignore temporarily lines 2 and 3 that will be clarified in section 3.5). Consider again the case study of the data sanitization program in Figure 3.1. Only integer values from 8 to 23 are possible outputs of this program, which means the number of possible outputs is $N = 16$.

At the beginning, all variables are initialised in the procedure `SymbolicQIF` as in Figure 3.5, the method `EarlyPrunning` employs a heuristic that will be discussed later in this section. The method `SymCount` is then called to count the number of possible models of φ_P.

When a variable $p_i \in V_I$ is selected, we systematically explore in the same way for both p_i and $\neg p_i$. Hence, the block of code from line 5 to line 11, and the one from line 12 to line 18 in Figure 3.6 are symmetric: we only explain the first one.

A partial run of `SymCount` on the illustrative example is depicted in Figure 3.7. At the first call of `SymCount`: $i = 1$, the variable p_1 is in consideration and it is added to pc in line 5. Since pc is initialised to be empty, $pc_1 = p_1$. The $\mathcal{T}$-solver is called to check if there is a model of φ_P where p_1 is (assigned to) $\top$. This can be done by using assertion to check the validity of $\neg p_1$ in a program as follows:

assert $!p_1$;

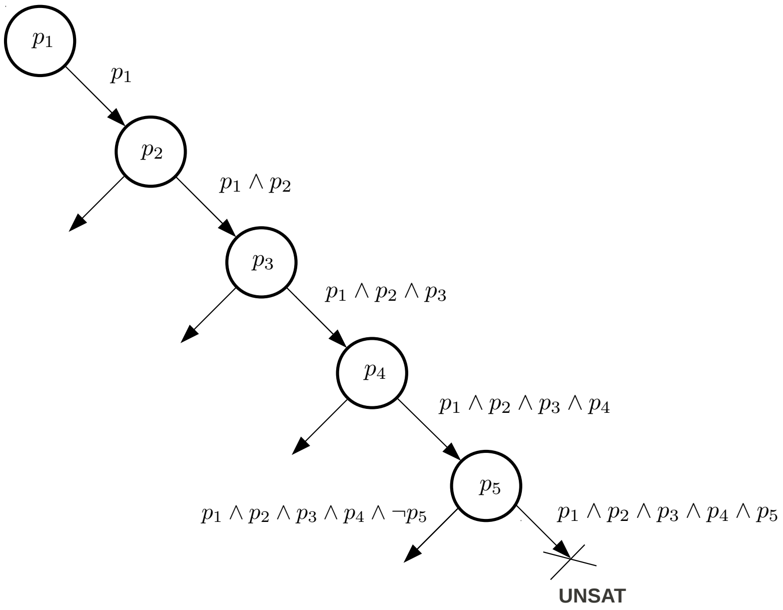
p_1
p_1
p_2
$p_1 \wedge p_2$
p_3
$p_1 \wedge p_2 \wedge p_3$
p_4
$p_1 \wedge p_2 \wedge p_3 \wedge p_4$
p_5
$p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge \neg p_5$
$p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5$
UNSAT

Figure 3.7: Partial exploration path of SQIF for the program from Figure 3.1.

A model checking tool like JPF or CBMC can be used as a $\mathcal{T}$-solver to verify this assertion and it will return $\top$ if the assertion fails, and $False$ otherwise. In this example, the $\mathcal{T}$-solver would return $\top$ since p_1 stands for "first bit is 1" and all odd values from 9 to 23 are possible outputs satisfying the condition p_1. Hence, SQIF proceeds by calling `SymCount` with $i = 2$. Similarly, the procedure progresses until calling `SymCount` with $i = 5$, which means it needs to verify:

$$\textbf{assert}\ !(p_1\ \&\&\ p_2\ \&\&\ p_3\ \&\&\ p_4\ \&\&\ p_5);$$

This time the $\mathcal{T}$-solver would return $False$, since $p_1 \wedge p_2 .. \wedge p_5$ represents a set of outputs of which each element is at least $2^0 + 2^1 + .. + 2^4 = 31$, while the possible range of O is only from 8 to 23. For a program with an output of 32-bits, by using `EarlyPrunning`, SQIF trims a set of 2^{27} concrete values represented by the family of sets:

$$\{V_I := \{p_1, p_2, .., p_{32}\} : p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5\}$$

This is how the state-space explosion problem is mitigated.

At the depth $i = 5$ as above, if SQIF takes the path of $\neg p_5$ from line 12, then the $\mathcal{T}$-solver returns $\top$ ($O = 15$ is one of the models). Hence, the procedure continues with $i = 6$, and from this point until $i = 32$, only the path of $\neg p_i$ is SAT. At $i = 32$, SQIF finds a full path 00..01111 which represents an output $O = 15$. This path is added to Ψ, and SQIF increases N. Finally, at the end of the method `SymbolicQIF`, we have $\Psi = \{8, 9, .., 23\}$ and $N = 16$, thus we can conclude that the data sanitization program in Figure 3.1 leaks at most 4 bits.

The method `EarlyPrunning` implements the idea that if p_1 is unsatisfiable, then $p_1 \wedge C$ is also unsatisfiable for any C. Therefore, at the beginning of the `SymbolicQIF`, all p_i are checked for satisfiability, and the results are stored for later use. We note that `EarlyPrunning` speeds up `SymbolicQIF` dramatically when the number of possible models (outputs of the program) is small.

We have developed a prototyping tool for QIF analysis of C programs, **sqifc** built on top of CBMC.

3.5 Soundness and Completeness

By soundness of the SQIF approach we mean that given Ψ, $\log_2(N)$ returned by `SymbolicQIF`(V_I, φ_P), each element of Ψ is a model of φ_P i.e. corresponds to a possible value of the output of the program P. By completeness of SQIF, we mean that Ψ is the set of all models of φ_P i.e. all values of the output of P.

Theorem 2 *Given a sound (resp. complete) $\mathcal{T}$-solver the SQIF approach is sound (resp. complete) i.e.* `SymCount` *solves the QIF problem (Definition 11).*

Proof sketch 1 *The SQIF algorithm as described in Figure 3.6 is based on DPLL which itself is a depth-first search procedure. As the search space is a binary tree with bounded depth M, the number of bits of the output, the depth-first search procedure is complete. The soundness of SQIF is guaranteed by the soundness of the $\mathcal{T}$-solver, i.e. model checker.*

In reality $\mathcal{T}$-solvers are only complete in particular domains. Moreover, even with sound and complete $\mathcal{T}$-solvers, a large leak requires an exponential number of calls to the $\mathcal{T}$-solver and so in practice SQIF is complete only for programs with *small* leaks.

Since our tools are based on bounded model checker, we choose to analyse only bounded programs. Notice however that Theorem 2 holds for general $\mathcal{T}$-solvers.

Because of these practical issues about completeness, it has been proposed to shift the focus from the question *"How much does it leak?"* to the simpler quantitative question *"Does it leak more than k?"* [20, 42]. This approach not only makes the problem easier to analysis, but it is also more intuitive in term of security, because the user policy, i.e. *threshold k*, is encoded in the analysis. The ultimate goal of security analysis is to determine whether a program is *secure* or *insecure*. As discussed in the previous section, the goal of QIF is to relax security policy from *non-interference* to an acceptable threshold *k bits of interference*, so that we can tolerate "*small*" leak, and accept more programs as secure. The SQIF approach can also be used in the same way: with a user policy k, if SQIF finds out more than $k = 2^k$ possible outputs, we can stop the procedure and conclude that the program is *insecure*. This is the meaning of lines from 2 to 3 of function `SymCount` in Figure 3.6.

A straightforward consequence of the Theorem 2 is that, assuming a sound $\mathcal{T}$-solver, given a user policy k, `SymCount` never returns *secure* for a program leaking more than k bits. This can be formally expressed as:

Corollary 1 *SQIF is sound w.r.t a user policy k.*

3.6 Evaluation

Only few papers present QIF static code analysis of real-world applications: examples are [20], [59] and the more recent [60]. Of these three approaches, [59] uses a different attacker model, namely cache side-channels and so is not directly comparable with our approach. The other two, [20] and [60], use the same attacker model as we do but are at the moment both restricted to C programs, and hence only comparable to **sqifc**. We will concentrate on [20], to which we refer as **selfcomp**, because it is based on the well-known concept of self-composition [56]. For the analysis of anonymity protocols, we compare **sqifc** against **QUAIL** [73, 74], a state-of-the-art quantitative analyser for probabilistic programs. The case studies broadly fall in three categories:

- the first category, consisting of case studies from the National Vulnerability Database of the US government [69], is aimed to demonstrate how our analysis is able to deal with complex C-code,
- the CRC case study shows the applicability to quantifying leakage in programs which leak by design,
- the case studies Grade and Dining cryptos protocols show how our technique, even if it is unable to analyse probabilistic programs, is able to compute channel capacity for anonymity protocols.

The experiments are conducted on a desktop machine with Intel Core i5 3.3GHz and 8GB of memory.

3.6.1 CVE-2011-2208

This case study is an example of a program that leaks information when the attacker can control the public input. It is taken from the National Vulnerability Database (NVD) of the US government [69], and it is released on 13/06/2012.

```
 1 int osf_getdomainname(char __user *name, int namelen)
 2 {
 3   unsigned len;
 4   int i, error;
 5 
 6   error = verify_area(VERIFY_WRITE, name, namelen);
 7   if (error)
 8     goto out;
 9 
10   len = namelen;
11   if (namelen > 32)
12     len = 32;
13 
14   down_read(&uts_sem);
15   for (i = 0; i < len; ++i) {
16     __put_user(system_utsname.domainname[i],
17         name + i);
18     if (system_utsname.domainname[i] == '\0')
19       break;
20   }
21   up_read(&uts_sem);
22  out:
23    return error;
24 }
```

Figure 3.8: arch/alpha/kernel/osf_sys.c

The system call `osf_getdomainname`, depicted in Figure 3.8, in the Linux kernel before 2.6.39.4 leaks sensitive information from kernel memory. This is caused by an integer signedness error: the signed parameter `namelen` is assigned to the unsigned variable `len` in line 10, so a negative value can be transformed into a big positive one. Therefore, although the condition in line 11 restricts `namelen` to 32, the number of characters returned to the user via the structure `name` may be much greater including bytes from kernel memory.

The kernel memory is the secret, modelled as an array of data, of which elements are assigned non-deterministic values. This is done by using the built-in non-deterministic functions of CBMC. The observable output is the array `*name` returned to the user. The domain name to be copied to the user is fixed, thus in the case there is no leakage, there will be only one possible value for the output. On the other hand, if there is leakage from kernel memory, the non-deterministic data will cause discrepancies in the output.

In order to quantify the information leakage caused by this vulnerability, we chose the thresholds of security policy $k = 64$ and $k = 256$, which means the program is secure if it leaks less than 6 and 8 bits respectively. After the times in Figure 3.11 , **sqifc** and **selfcomp** conclude that the program is insecure. We then apply the patch provided for this vulnerability, and run **sqifc** again. This time, **sqifc** found only one possible value for `name`, which means a leak of zero bit. Hence, we prove that the patch fixed the leak.

3.6.2 CVE-2011-1078

This case study is also taken from NVD, and it is released on 21/06/2012. The function `sco_sock_getsockopt_old` in the Linux kernel before 2.6.39, depicted in Figure 3.9, leaks sensitive information from kernel memory.

As in line 24, `cinfo` is copied to the user. Although its total size is 5 bytes, and all bytes are correctly assigned, when compiled it includes an additional padding byte for alignment purposes. This padding byte is not zeroed out, and hence it contains kernel memory, and is leaked to the user.

Similar to the previous case study, the kernel memory is the secret, modelled as a non-deterministic array of data. The array `*optval` returned to the user is the observable output. CBMC can precisely model the program after being compiled, including the padding bytes, which enabled us to analyse the vulnerability of this case study. Results of the analysis for $k = 8$ and $k = 64$, are shown in Figure 3.11.

```
static int sco_sock_getsockopt_old(
    struct socket *sock, int optname,
    char __user *optval, int __user *optlen)
{
  struct sock *sk = sock->sk;
  struct sco_conninfo cinfo;
  int len, err = 0;
  ...

  lock_sock(sk);

  switch (optname) {
    case SCO_OPTIONS:
      ...

    case SCO_CONNINFO:
      ...

      cinfo.hci_handle = sco_pi(sk)->conn->hcon->handle;
      memcpy(cinfo.dev_class,
        sco_pi(sk)->conn->hcon->dev_class, 3);

      len = min_t(unsigned int, len, sizeof(cinfo));
      if (copy_to_user(optval, (char *)&cinfo, len))
        err = -EFAULT;
        break;
        ...
      }

      release_sock(sk);
      return err;
}
```

Figure 3.9: net/bluetooth/sco.c

3.6.3 Cyclic Redundancy Check

The program in Figure 3.10 performs Cyclic Redundancy Check[1] (CRC) and shifts right the result `sft` bits. We also have a Java version of the program to test with **jpf-qif**.

```
unsigned char GetCRC8(
        unsigned char check , unsigned char ch)
{
  int i, sft ;
  for ( i = 0 ; i < 8 ; i++ ) {
    if ( check & 0x80 ) {
      check<<=1;
      if ( ch & 0x80 ) { check = check | 0x01;}
      else { check =check & 0xfe; }
      check = check ^ 0x85;
    } else {
      check <<=1;
      if ( ch & 0x80 ) { check = check | 0x01; }
      else { check = check & 0xfe; }
    }
    ch<<=1;
  }
  check >>= sft;
  return check;
}
```

Figure 3.10: Cyclic Redundancy Check

We quantify the amount of information of the confidential input `ch` revealed by observing the output of function `GetCRC8`. We analyse this program with **sqifc** and **selfcomp** for `sft` values of 3 and 5 giving a maximum leakage for this program of 5 (**selfcomp** times out on this case) and 3 bits respectively which is consistent with the design of the program. Results of the analysis are shown in Figure 3.11. In the case value of `sft` is 5, i.e. $k = 8$, **selfcomp** is faster as the state-space is still small enough, and **selfcomp** requires only one call to CBMC. When `sft` $= 3$, i.e. $k = 32$, the state-space explosion makes **selfcomp** fail to solve. SQIF requires several calls to the solver, but it is less vulnerable to state-space explosion.

[1] http://en.wikipedia.org/wiki/Cyclic_redundancy_check

Case Study	**Policy**	**LoC**	**sqifc** time	**selfcomp** time
Data Sanitization	-	< 10	11.898	timed out
CVE-2011-2208	64	> 200	22.759	119.117
CVE-2011-2208	256		88.196	timed out
CVE-2011-1078	8	> 200	10.380	13.853
CVE-2011-1078	64		37.899	timed out
CRC	8	< 30	1.209	0.498
CRC	32		8.657	timed out

Figure 3.11: Comparing sqifc against selfcomp. Times are in seconds, timeout is 30 minutes. In the first case study, "-" means the policy is not specified.

3.6.4 The Grade Protocol

This case study was used to illustrate protocol analysis in [75, 74]. This anonymity protocol is designed to enable a group of students to compute the sum of their grades (e.g., to compute the average) without revealing individual grades. We denote $S_1, ..., S_k$ be the k students arranged in a ring, each one is given a secret grade g_i between 0 and $m-1$. To compute the sum of g_i without disclosing them, the students produce k random numbers between 0 and $n = (m-1)k + 1$ such that the number r_i is known only to the students S_i and $S_{(i+1)\%k}$. Each student s_i then outputs a number $d_i = g_i + r_i - r_{(i+1)\%k}$ and the sum of all grades is equivalent to the sum of the outputs modulo n.

This protocol is implemented as a probabilistic program in both [75] and [74]. Here we implement it in standard ANSI C with the built-in non-deterministic functions of CBMC. The source code, shown on Figure 3.12, is based on the one provided in [74]. The array `h[S]` stores the grades of all students, i.e. the secret. The attacker can observe `sum % n`.

To compare **QUAIL** with our tool, **sqifc**, we repeat the experiment of the authors for the grade protocol with the tool and examples provided in [73]. However, **QUAIL** timed out after 1 hours for most of the cases, as shown in Figure 3.14 (we had the same results of leakage with the authors in the cases the tool did not time out). Therefore, we take the result in Figure 3.13 directly from the paper [74]. Comparing the results in Figure 3.13, it is easy to realise that the bounds on the leaks, measured by **sqifc**, do not exceed the real leaks, measured by **QUAIL**, by more than 1 bit, while **sqifc** required no more than 1 minute in all cases as shown in Figure 3.14.

```
int func(){
  size_t S = 5, G = 5, i = 0, j = 0;
  size_t n = ((G-1)*S)+1, sum = 0;
  size_t numbers[S], announcements[S], h[S];

  for (i = 0; i < S; i++) h[i] = nondet_int() % G;

  for (i = 0; i < S; i++)
        numbers[i] = nondet_int() % n;

  while (i<S) {
    j=0;
    while (j<G) {
      if (h[i]==j)
        announcements[i] =
          (j+numbers[i]-numbers[(i+1)%S])%n;
      j=j+1;
    }
    i=i+1;
  }

  for (i = 0; i < S; i++)
    sum += announcements[i];

  return sum % n;
}
```

Figure 3.12: The Grade protocol

Tool		**QUAIL**				**sqifc**			
Students		**2**	**3**	**4**	**5**	**2**	**3**	**4**	**5**
Grades	**2**	1.500	1.811	2.030	2.198	1.585	2.000	2.322	2.585
	3	2.197	2.525	2.745	2.910	2.322	2.807	3.170	3.459
	4	2.655	2.984	3.201	3.365	2.807	3.322	3.700	4.000
	5	2.999	3.325	3.541	timed out	3.170	3.700	4.087	4.392

Figure 3.13: Leakage measured by **QUAIL** and **sqifc**

Tool		**QUAIL**				**sqifc**			
Students		**2**	**3**	**4**	**5**	**2**	**3**	**4**	**5**
Grades	**2**	1.306	241.483	-	-	5.657	7.029	10.767	9.469
	3	28.613	-	-	-	9.145	11.597	17.987	20.930
	4	508.313	-	-	-	10.095	16.872	21.869	18.579
	5	-	-	-	-	14.639	20.666	33.298	40.399

Figure 3.14: Elapsed time in seconds of **QUAIL** and **sqifc**

3.6.5 The Dining cryptos protocol

This case study is a variation of the dining cryptographers protocol of Chaum [76], one of the most popular problems in anonymity protocol. There is a group of cryptographers gathering around a table for dinner. After the meal, they are informed that the bill has been paid by someone, who could be one of them or the National Security Agency (NSA). Even though the cryptographers respect each other's right to make an anonymous payment, they want to find out whether the NSA paid.

To determine this, they use a protocol as follows: each pair of adjacent cryptographers tosses a coin hidden from everybody else, so that each cryptographer only knows the values of the coin shared with the one on his left and with the one on his right; then each cryptographer declares aloud the exclusive OR of the two coins he sees, i.e. 0 if they have the same value and 1 otherwise. However if the payer is one of the cryptographers, he declares the opposite. In the end, if the sum of all declared values is even, then it is concluded that the NSA paid the bill. On the other hand, the sum is odd means one of the cryptographers did it.

```
size_t func(){

  size_t N = 5, output = 0, i = 0;
  size_t coin[N], obscoin[2], decl[N];
  size_t h;

  h = nondet_uchar() % (N+1);

  for (i = 0; i < N; i++){
    coin[i] = nondet_uchar() % 2;
  }

  for (i = 0; i < N; i++){
    decl[i] = coin[i] ^ coin[(i+1)%N];
    if (h==i+1){
      decl[i] = !decl[i];
    }
    i = i+1;
  }

  for (i = 0; i < N; i++){
    output = output + decl[c];
  }

  return output;
}
```

Figure 3.15: The Dining cryptos protocol

We are interested in knowing how much information about the payer can be leaked by the sum of all declared values (in the dining cryptographers the observation are the declared values instead). The input code for the protocol is depicted in Figure 3.15. `h` is the identity of the payer, i.e. the secret, `output` is the observable. The coin toss is modelled by a built-in non-deterministic function in line 10. This model is less precise than implementation in probabilistic programs where it is possible to select random values from a specific distribution. By modelling with non-deterministic function and computing channel capacity, we can only compute the maximum leakage in all possible distributions. Figure 3.16 shows the channel capacity computed by **sqifc**, and the time to compute them.

Cryptos	3	4	5	6	100	300
Channel capacity	2	2.32	2.59	2.81	6.658	8.234
Time in seconds	2.145	3.496	3.632	18.634	158.517	3326.915

Figure 3.16: The dining cryptos protocol analysed by **sqifc**

3.7 Discussion of related work

Meng and Smith introduce an *approximate* technique to calculate an upper bound on channel capacity in [18]. The authors' implementation of the method is largely manual, and we proposed an automation for it in [7]. While the work of Meng and Smith is very inspiring, the technique can be very imprecise, for example when the leaks are sparse in the state space. Moreover, the user policy is not encoded in the analysis which makes it infeasible when the leaks are not small. Take an example of a program that leaks all M bits of integral confidential data, this technique needs to make a total of $2M^2$ calls to the bit vector solver to calculate the leaks.

The first automated method for QIF was proposed by Backes et al. [19]. The method can be divided into two stages: first, it employs model checking to compute an equivalence relation $\mathcal{R}$ on the set of confidential inputs w.r.t. observable outputs; secondly, if this relation $\mathcal{R}$ can be represented by a system of *linear integer inequalities* $A\bar{x} \geqslant \bar{b}$, which means it is a bounded integer *polytopes*, then a variant of Barvinok's algorithm [40] can be used to count the number of integer solutions of $\mathcal{R}$. While this work is important as the first effort on automation of QIF analysis, it is not clear however how this approach can be applied to real-world programs because of, for example, bit-wise operators in the CRC case study or non-linear relations and so on.

Closer to our work is the paper of **selfcomp** [20] discussed in the previous section. However, as already outlined their approach to address the question *"Does it leak more than k?"* is quite different from ours. Köpf et al. [59] also apply QIF to real-world applications, i.e. leakage of cache side-channels; their technique is based on abstract interpretation and hence not based on bounded models. Because of this however they over-approximate channel capacity.

A preliminary version of this chapter has been presented first in a workshop [7], and then in a conference [5]. However our definition for #SMT was a little bit different from the one in this chapter: we required that each of the Boolean variables in the set V_I is a Boolean abstraction of some $\mathcal{T}$-atom, hence we named the problem *Propositional Abstract Model Counting.* This requirement makes the definition more complicated and less general.

A recent paper [60] explores QIF in a pure logical framework. The approach is powerful and elegant, however it is more limited when compared to our approach as it relies on the solver to generate models whereas our approach can use any solver instead. For example we can analyse Java by using JPF as a solver for bytecode even if JPF doesn't generate a model in the sense of [60].

McCamant and Ernst released FlowCheck [16], a tool for security testing based on dynamic taint analysis. What FlowCheck measures is the number of tainted bits, not an information-theoretic bound, so it is significantly different from our approach. Another tool is described in [17], it is able to analyse large programs using the notion of channel capacity in the context of dynamic taint analysis, while our approach is based on verification techniques. In this sense, our work comes with stronger theoretical guarantees.

Chapter 4

Symbolic Execution as DPLL Modulo Theories

The previous chapter has introduced the Symbolic Quantitative Information Flow (SQIF) approach, which mimics the DPLL($\mathcal{T}$) algorithm. SQIF was implemented on top of the Bounded Model Checker CBMC, used as a sub-routine, and can analyse programs written in C/C++. This chapter presents an alternative implementation for the SQIF approach, using Symbolic Execution.

The implementation is based on a key observation that Symbolic Execution can be viewed as a variant of the DPLL($\mathcal{T}$) algorithm, or in other words, Symbolic Executors are SMT solvers.

This view enables us to modify Symbolic PathFinder [77], a Symbolic Executor for Java bytecode, into a QIF analysis tool, **jpf-qif**, with little effort. The work in this chapter is the first to use Symbolic Execution for QIF analysis, and jpf-qif is the first QIF analysis tool for Java.

4.1 Introduction

Symbolic Execution (SE) [46] is now popular. It is increasingly used not only in academic settings but also in industry, such as in Microsoft, NASA, IBM and Fujitsu [78]. In the success of SE, the efficiency of SMT solvers [44] is a key factor. In fact, while SE was introduced more than three decades ago, it had not been made practical until research in SMT made significant advances [78].

Recall that most state-of-the-art SMT solvers, e.g. [79, 80], implement the DPLL($\mathcal{T}$) algorithm [45] which is an integration of two components as the following. The first component is a DPLL-based SAT solver, to search on the *Boolean skeleton* of the formula. The second component is a $\mathcal{T}$-solver to check the consistency w.r.t. the theory $\mathcal{T}$ of conjunctions of literals. The path conditions generated by a Symbolic Executor, e.g. Symbolic PathFinder (SPF) [81], are also conjunctions of literals. Therefore, when an SMT solver checks such a path condition, only the $\mathcal{T}$-solver works on it, and the SAT component is not used[1].

On the other hand, a *classical* Symbolic Executor [46] can also be divided into two components. The first component, called *Boolean Executor* hereafter, executes the instructions, and updates the path condition. The second component is a $\mathcal{T}$-solver (since the SAT solver is not used) to validate the consistency of the path condition. This thesis shows that a Boolean Executor does the same work as DPLL algorithm. Thus, SE is a variant of DPLL($\mathcal{T}$). This view is important since it connects two communities and can give an insight for future research.

4.2 Illustration of DPLL($\mathcal{T}$)

A complete formal description of the DPLL($\mathcal{T}$) algorithm can be found in, e.g., [45]. Here we briefly recall some background via a running example as follows.

$$\begin{aligned}\varphi := {} & (\neg(x_0 > 5) \vee T_1) \wedge ((x_0 > 5) \vee T_2) \wedge (\neg(x_0 > 5) \vee (x_1 = x_0 + 1)) \wedge \\ & (\neg(x_1 < 3) \vee T_3) \wedge (\neg(x_1 < 3) \vee (x_2 = x_1 - 1)) \wedge \\ & ((x_1 < 3) \vee T_4) \wedge ((x_1 < 3) \vee (y_1 = x_1 + 1))\end{aligned} \tag{4.1}$$

φ is a Linear Arithmetic formula. Boolean variables, $T_1 \dots T_4$, are called Boolean atoms, and atomic formulas, e.g. $(x_0 > 5)$, are called theory atoms or $\mathcal{T}$-atoms. Any first-order formula φ can be abstracted into a Boolean skeleton by replacing each $\mathcal{T}$-atom in φ with its Boolean abstraction. For the example above, we define new Boolean variables G_1, G_2, A_1, A_2, A_3 for the Boolean abstraction of $\mathcal{T}$-atoms, and the abstraction can be expressed as:

$$\begin{aligned}\mathcal{BA} := {} & G_1 = (x_0 > 5) \wedge G_2 = (x_1 < 3) \wedge \\ & \quad A_1 = (x_1 = x_0 + 1) \wedge A_2 = (x_2 = x_1 - 1) \wedge A_3 = (y_1 = x_1 + 1)\end{aligned} \tag{4.2}$$

[1] This claim is not for the decision procedure STP [82], which converts Bit Vector formulas into propositional formulas and solves them with a SAT solver.

As the result, we obtain a formula φ^P (^{P} stands for propositional) as the Boolean skeleton of φ. Obviously, φ is logically equivalent to $\varphi^P \wedge \mathcal{BA}$.

$$\begin{aligned}\varphi^P := {} & (\neg G_1 \vee T_1) \wedge (G_1 \vee T_2) \wedge (\neg G_1 \vee A_1) \wedge \\ & (\neg G_2 \vee T_3) \wedge (\neg G_2 \vee A_2) \wedge \\ & (G_2 \vee T_4) \wedge (G_2 \vee A_3)\end{aligned} \tag{4.3}$$

```
function DPLL(BooleanFormula $\varphi$){
    $\mu$ = True; status = propagate($\varphi, \mu$);
    if (status == Sat)  return Sat;
    else if (status == UnSat)  return UnSat;
    while (True) {
        $l$ = decide($\varphi$);
        $\mu = \mu \wedge l$;
        status = propagate($\varphi, \mu$);
        if (status == Sat)  return Sat;
        else if (status == UnSat)
            if (allStatesAreExplored())
                return UnSat;
            else backtrack($\varphi, \mu$);
} }
```

Figure 4.1: DPLL algorithm

The DPLL($\mathcal{T}$) algorithm is the integration of the DPLL algorithm with a $\mathcal{T}$-solver. The DPLL algorithm searches on φ^P, returning a conjunction of Boolean literal μ^P. Replacing all the new Boolean atoms, G_i and A_i, in μ^P with their corresponding $\mathcal{T}$-atoms, we obtain the conjunction μ in $\mathcal{T}$. The $\mathcal{T}$-solver then checks whether μ is consistent with the theory $\mathcal{T}$. Below is the illustration of DPLL($\mathcal{T}$) on φ (to fit the status of DPLL($\mathcal{T}$) in one row, only decision literals are shown in μ^P):

0. $\mu^P = \texttt{True}$ $\quad \varphi^P$
1. $\mu^P = G_1$ $\quad \varphi^P = (\neg G_2 \vee T_3) \wedge (\neg G_2 \vee A_2) \wedge (G_2 \vee T_4) \wedge (G_2 \vee A_3)$
2. $\mu^P = G_1 \wedge G_2$ $\quad \varphi^P = \texttt{True}$; $\mathcal{T}$-solver$(\mu) = \texttt{Inconsistent}$
3. $\mu^P = G_1$ $\quad \varphi^P = (\neg G_2 \vee T_3) \wedge (\neg G_2 \vee A_2) \wedge (G_2 \vee T_4) \wedge (G_2 \vee A_3)$
4. $\mu^P = G_1 \wedge \neg G_2$ $\quad \varphi^P = \texttt{True}$; $\mathcal{T}$-solver$(\mu) = \texttt{Consistent}$

The DPLL algorithm tries to build a model using three main operations: `decide`, `propagate`, and `backtrack` [44]. The operation `decide` heuristically chooses a literal l (which is an unassigned Boolean atom or its negation) for branching. The operation `propagate` then removes all the clauses containing l, and deletes all occurrences of $\neg l$ in the formula; this procedure is also called *Boolean Constraint Propagation* (BCP). If after deleting a literal from a clause, the clause only has only one literal left (*unit clause*), BCP assigns this literal to `True`. If deleting a literal from a clause results in an empty clause, this is called a conflict. In this case, the DPLL procedure must `backtrack` and try a different branch value.

At step 1, G_1 is decided to be the branching literal, and the $\mathcal{T}$-solver validates that $(x_0 > 5)$ is consistent. BCP removes the clause $(G_1 \vee T_2)$, and deletes all occurrences of $\neg G_1$. This results in two unit clauses T_1 and A_1, so they are assigned to `True`, which means $\mu^P = G_1 \wedge T_1 \wedge A_1$. Similarly, at step 2 G_2 is chosen, i.e. $\mu^P = G_1 \wedge T_1 \wedge A_1 \wedge G_2$. The $\mathcal{T}$-solver checks the conjunction: $\mu = (x_0 > 5) \wedge T_1 \wedge (x_1 < 3) \wedge (x_1 = x_0 + 1)$. This is obviously inconsistent, thus DPLL($\mathcal{T}$) backtracks and tries $\neg G_2$, which leads to a consistent model.

Note that DPLL($\mathcal{T}$) refers to various procedures integrating DPLL and a $\mathcal{T}$-solver. There are DPLL($\mathcal{T}$) procedures with integration schemas different from what we have described here. The interested reader is pointed to [83] for further references.

4.3 Symbolic Execution as DPLL($\mathcal{T}$)

Intuitively, a program can be encoded into a (first-order) formula whose models correspond to program traces. Symbolic Executors explore all program traces w.r.t. the set of program conditions, therefore they can be viewed as SMT solvers that return all (partial) models w.r.t. a set of Boolean atoms.

In this thesis we only consider bounded programs, since this is the class of programs that SE can analyse. This means every loop can be unwound into a sequence of `if` statements. In order to encode a program into a formula, all program variables are renamed in the manner of Static Single Assignment form [84]: each variable is assigned exactly once, and it is renamed into a new variable when being reassigned. In this way, assignments such as $x = x + 1$ will not be encoded into an unsatisfiable atomic formula. Under these settings, a program P can be modelled by a *Symbolic Transition System* (STS) as follows:

$$P \equiv (S, I, G, A, T)$$

S is the set of program states, $I \subseteq S$ is the set of initial states; each state in the STS models the computer memory at a program point. G is the set of guards and A is the set of actions; guards and actions are first-order formulas.

An action models the effect of an instruction on the computer memory. Actions that do not update the computer memory (e.g. conditional jumps) are Boolean atoms, the others are $\mathcal{T}$-atoms. $T \subseteq S \times G \times A \times S$ is the transition function, $t_{ij} = \langle s_i, g_{ij}, a_{ij}, s_j \rangle \in T$ models a transition from state s_i to state s_j by taking action a_{ij} under the guard g_{ij}. After a transition $t_{ij} : s_i \rightarrow s_j$, the state s_j is exactly as the state s_i apart from the variable updated by the action a_{ij}.

Note that this STS models the program in more detail than the transition system in section 2.3 that will be used in other chapters.

One way to encode a transition t_{ij} into a first-order formula is to present it in the form: $t_{ij} \equiv g_{ij} \rightarrow a_{ij}$, or equally $t_{ij} \equiv \neg g_{ij} \vee a_{ij}$. This encoding expresses that satisfying the guard g_{ij} implies that the action a_{ij} is performed. In this way, a program trace is defined as a sequence of transitions:

$$t_{01} \wedge t_{12} \wedge \cdots \wedge t_{(k-1)k} = (\neg g_{01} \vee a_{01}) \wedge (\neg g_{12} \vee a_{12}) \cdots \wedge (\neg g_{(k-1)k} \vee a_{(k-1)k})$$

The semantics of the program is then defined as the set of all possible traces, or equally the set of all possible transitions, which can be represented as the following formula:

$$\varphi = \bigwedge_{t_{ij} \in T} t_{ij} = \bigwedge_{t_{ij} \in T} (\neg g_{ij} \vee a_{ij}) \tag{4.4}$$

```
void test(int x, int y){
  if(x > 5){
    x++;
    if (x < 3)
      x--;
    else
      y = x + 1;
  }
}
```

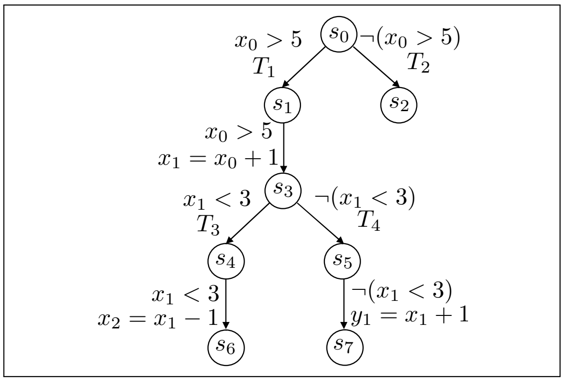

Figure 4.2: A simple program and its associated STS. "`if`$(x > 5)$" is modelled by two transitions $\langle s_0, (x_0 > 5), T_1, s_1 \rangle$ and $\langle s_0, \neg(x_0 > 5), T_2, s_2 \rangle$; then "`x++`" is modelled by $\langle s_1, (x_0 > 5), x_1 = x_0 + 1, s_3 \rangle$; similarly for the rest of the program.

Figure 4.2 depicts a simple example program and its associated STS. Encoding this STS following (4.4) results in the formula (4.1) that we have illustrated with DPLL($\mathcal{T}$) in the previous section. We now illustrate this example with SE.

Similar to an SMT solver, a Symbolic Executor at a high level can be viewed as the integration of two components: a Boolean Executor (BE) to execute the instructions and a $\mathcal{T}$-solver to check the feasibility of path conditions. For example, SPF has a parameter `symbolic.dp` to customize which decision procedure to use. If we set this parameter with the option `no_solver` then SPF solely works on the BE.

```
function EXECUTOR(Program $P$){
    PathCondition pc = TRUE;
    InstructionPointer i = NULL;
    update($P, i$);
    if ($i$ == RETURN)  return;
    while (TRUE) {
        $l$ = decide($i$);
        $pc = pc \wedge l$;
        update($P, i$);
        if ($i$ == RETURN)
            if (allStatesAreExplored())
                return;
            else backtrack($pc, i$);
} }
```

Figure 4.3: A simplified Boolean Executor

Figure 4.3 depicts a simplified procedure of a BE. This procedure can be described as trying to build *all* path conditions using three main operations: `decide`, `update` and `backtrack`. The operation `decide` chooses a literal l, a condition (or its negation) of an `if` statement, for branching, adding it to the path condition. The operation `update` then symbolically executes a block of statement, i.e. no branching statement present, updating the computer memory. When the BE reaches the end of a symbolic path, it backtracks to explore other paths. A Symbolic Executor, which is the integration of a BE and a $\mathcal{T}$-solver, backtracks if the path condition is not satisfied.

Both DPLL and BE rely on Depth-First Search, they are similar in the way they `decide` and `backtrack`[2]. After choosing a literal, e.g. $(x_0 > 5)$, BE executes the block it guards, i.e. T_1 and $x_1 = x_0 + 1$. This is exactly the same as in DPLL: after choosing g_{ij}, for all the clauses $(\neg g_{ij} \vee a_{ij})$, BCP deletes $\neg g_{ij}$, assigning a_{ij} to `True`.

[2] We consider DPLL in its simplest form, without non-chronological backtracking.

Therefore, the operation `update` does the same work as BCP, we can view a BE as implementing the DPLL algorithm, and SE as DPLL($\mathcal{T}$).

4.4 SQIF by Symbolic Execution:

With the view of SE as #DPLL($\mathcal{T}$), we are able to make a Symbolic Executor work as `SymbolicQIF` with little effort. The key idea here is to enumerate all concrete values from symbolic executions.

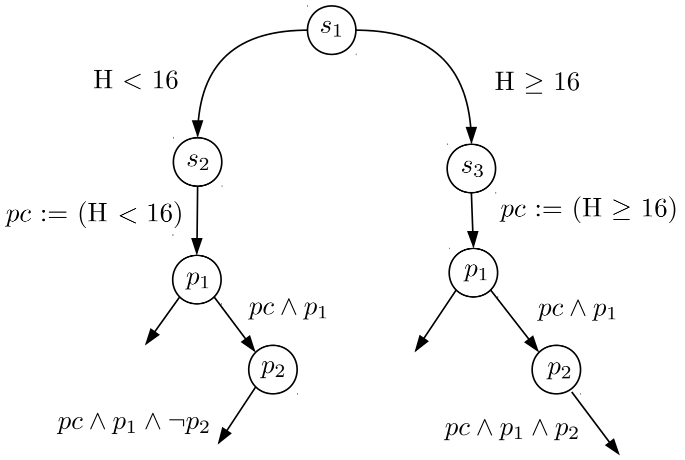

Figure 4.4: Partial exploration path of SQIF-SE for the data sanitization program from Figure 3.1.

For a program P that takes symbolic inputs $i_1, i_2, ..i_\alpha$, and produces an output O, the result of running SE on P is as follows:

$$O = \left\{ \begin{array}{ll} f_1(i_1, i_2.., i_\alpha) & \text{if } pc_1 \\ f_2(i_1, i_2.., i_\alpha) & \text{if } pc_2 \\ \ldots & \ldots \\ f_\beta(i_1, i_2.., i_\alpha) & \text{if } pc_\beta \end{array} \right\}$$

where $f_1, f_2, ..f_\beta$ are formulas over symbolic inputs $i_1, i_2, ..i_\alpha$. $pc_1, pc_2, ..pc_\beta$ are the path conditions. Notice f_i expresses a symbolic final value for O, i.e. in terms of `SymEx` instead of $f_i(i_1, i_2.., i_\alpha)$ we could write $\sigma_i(O)$ for $\sigma_i \in \Sigma$. The following proposition was proved by King [46]:

Proposition 1

$$\forall i, j \in [1, \beta] \wedge i \neq j, \; pc_i \wedge pc_j = \bot$$

which means that path conditions are mutually exclusive.

Definition 21 *For a path condition pc_i obtained from SE, the concretization set of pc_i, denoted $\mathcal{CS}(pc_i)$, is the set of all concrete values of output O that can be reached by executing the program following pc_i.*

Consider again the illustrative example in Figure 3.1, in which there are two path conditions: $pc_1 = H < 16$ and $pc_2 = H \geq 16$. The corresponding concretization sets of these path conditions are: $\mathcal{CS}(pc_1) = [8..23]$ and $\mathcal{CS}(pc_2) = [24..2^{32}]$. The set of all possible values of output O is formed by the union of concretization sets of all paths, and thus:

$$N = \left| \bigcup_{i=1}^{\beta} \mathcal{CS}(pc_i) \right|$$

The set $\mathcal{CS}(pc_i)$ can be computed by inserting the code in Figure 3.4 at the end of the program and run SE: we add M conditions, each one tests whether bit b_i of the output O is 0 or 1. These M conditions test all the bits of the output O. Exploring all possible combinations of these conditions leads to enumerating all possible values of O. We denote by SQIF-SE the implementation of SQIF using SE. A partial exploration path of SQIF-SE is described as in Figure 4.4. SE as implemented by Symbolic PathFinder (SPF) returns a concrete values for each possible path. The number of distinct concrete values is the N that we need to count.

SQIF-SE is implemented into a prototyping tool **jpf-qif** built on top of SPF. The tool works on Java programs.

4.5 Soundness and Completeness

SQIF-SE relies on a Symbolic Executor, and hence it is complete in programs with a bounded model of runtime behaviour, which means programs have no recursion or unbounded loops. These are well-known issues in SE and handling them is orthogonal to our work. SQIF-SE is also sound given a sound Symbolic Executor.

4.6 Evaluation

To evaluate **jpf-qif**, we compare it against our previous implementation **sqifc** and the **selfcomp** technique. For case studies, we revisit the data sanitization program and the CRC programs in the previous chapter. Moreover, we consider the Tax program as follows.

4.6.1 Tax Record

Balliu et al. [85] provided a very interesting case study of information flow security in Java programs derived from the EU-funded FP7-project HATS. The program contains 8 classes/interfaces and 267 LoC. In our analysis we assume the year 2011-2012 basic tax rate in UK, which is applied for a person whose income does not exceed 35 thousand pounds per annum, is $F = 20\%$[3]. Thus, the tax is less than 7 thousands pound per year. We assume that donations to charities is below the amount of tax to be paid. Obviously, one cannot pay more than what one earns. Following [85] we are interested in leaks of a taxpayer's income and donations to a Tax checker.

QIF vs. Declassification

Balliu et al. considered two cases of declassification: the first one, called taxChecker1, is associated with the policy "$\mathit{income} \times F\% + \mathit{donation} > \mathit{payment}$", and the second one, called taxChecker2, is associated with the policy "$\mathit{income} \times F\% + \mathit{donation} - \mathit{payment}$". They claimed that: *"The value declassified in the taxChecker1 case, resp. taxChecker2 case, is a lower bound, resp. upper bound, of the value revealed to the tax checker in the fixed tax rate variant."*

We notice in this sentence the use of terms like *"value revealed"* and *"bound"*: the authors were trying to describe *quantitative* concepts. From the result of **jpf-qif**, we can give hence quantitative answers to these questions. In the case of *taxChecker1*, the observable is whether the payment is greater or smaller than the sum of the tax and donations, which means there are 2 possible outputs. This corresponds to the leak of 1 bit. In other words, the user policy or threshold $k = 1$. Regarding the taxChecker2 case, under the assumptions listed above, the leakage is upper bounded by 4.86 bits obtained in 24.988 seconds.

Case Study	**LoC**	**sqifc** time	**jpf-qif** time	**selfcomp** time
Data Sanitization	< 10	11.898	20.695	timed out
CRC (8)	< 30	1.209	8.386	0.498
CRC (32)		8.657	9.357	timed out
Tax Record	267	-	24.988	-

Figure 4.5: Times in seconds, timeout is 30 minutes. "-" means inapplicable.

[3] Since SPF can only handle conditions with integer values, we simplify the code by replacing $(\mathit{income} \times 20)/100$ with *tax* as an integer. The simplification we made does not change the secrecy, i.e. entropy, of *income*, so it will not affect the result of our analysis.

Results of the analysis are shown in Figure 4.5. In general, **jpf-qif** is slower than **sqifc**, although they are different implementation of the same algorithm. It is not surprising, as Java is always considered to be slower than C. Moreover, SPF is a virtual machine running on top of the Java Virtual Machine. Hence, there are more overheads in the implementation using SPF.

4.7 Discussion of related work

The correspondence between Symbolic Execution and the DPLL($\mathcal{T}$) algorithm was first briefly mentioned in our previous work [7]. There, we described a preliminary version of the DPLL-based algorithm in the previous chapter (Figure 3.6). However, at that time the tool sqifc was not yet available. Instead, we presented a quick implementation of the algorithm with SPF.

A year later, Brain et al. [86] published an excellent paper showing the correspondence between DPLL($\mathcal{T}$) and Abstract Interpretation. Although it is believed that Symbolic Execution is a case of Abstract Interpretation[4], we are not aware of any paper to discuss rigorously this relation.

Note that the correspondence between SE and DPLL($\mathcal{T}$) we have showed in this thesis is only valid for Symbolic Executors using Depth-First Search. In practice, Breadth-First and other heuristic searches have also been used for SE. For example, SPF has a parameter that allows users to select either Depth-First Search or Breadth-First Search or Heuristic Search. However, Depth-First Search is still the dominant and default search strategy in most of the tools.

[4] `http://en.wikipedia.org/wiki/Symbolic_execution`

Chapter 5

Concurrent Bounded Model Checking

In chapters 3 and 4, we have introduced two different implementations of the Symbolic Quantitative Information Flow approach: the first one is **sqifc**, which employs the Bounded Model Checker CBMC; and the second one is **jpf-qif**, built on top of the Symbolic PathFinder symbolic execution platform. A natural research question would be whether there is a relation between the two symbolic techniques: Bounded Model Checking and Symbolic Execution.

This chapter studies this relation and introduces a methodology, based on Symbolic Execution, for Concurrent Bounded Model Checking. In our approach, we translate a program into a formula in a disjunctive form, and this design enables concurrent verification: a main thread running a symbolic executor, without constraint solving, to build sub-formulas, and a set of worker threads running a decision procedure for satisfiability checks.

We have implemented this methodology in a tool called JCBMC, the first bounded model checker for Java. JCBMC is built as an extension of Java PathFinder, an open-source verification platform developed by NASA. JCBMC uses Symbolic PathFinder (SPF) for the symbolic execution, Z3 as the solver and implements concurrency with multi-threading.

For evaluation, we compare JCBMC against SPF and CBMC. The results of the experiments show that we can achieve significant advantages of performance over these two state-of-the-art tools.

5.1 Introduction

Model checking techniques are often classified in two categories: explicit-state or symbolic, depending on how they process the states of the system. While explicit-state model checking enumerates all possible states of the system explicitly, possibly on-the-fly [87, 88], symbolic model checking represents sets of states symbolically, and hence more efficiently, by using Binary Decision Diagrams [89] or Boolean formulae [68]. SAT or SMT-based Bounded Model Checking (BMC) [68] unwinds the transition relation of a program for a fixed number of steps k and checks whether a property violation can occur in k or fewer steps. This bounded verification is reduced to a satisfiability check performed by a SAT or SMT solver. BMC is widely used in the hardware industry.

For software, the application of BMC for ANSI-C is embodied in CBMC [67, 48], which has been successfully used for many practical applications. In CBMC a C program containing assertions is translated into a formula (in Static Single Assignment form) which is then fed to a SAT or SMT solver to check its satisfiability. A satisfying assignment indicates that an error was found.

Bounded model checking has not been explored so far for many other languages, including Java. However, explicit-state model checking tools such as Java PathFinder (JPF) [87] have been successfully used for the verification of many Java applications. Furthermore, there has been an explosion of symbolic execution [46] tools that have been used successfully for test case generation and error detection in the context of many high level languages, for example [90, 91, 92, 93]. In particular, relevant for the work reported here, Symbolic PathFinder (SPF) [81] is a symbolic execution tool built as an extension of JPF, that provides a symbolic analysis for Java programs involving multi-threading and complex data structures.

In this thesis, we describe an alternative methodology for BMC which is based on "classical" symbolic execution (SE) in the sense of King [46]. Note that the way CBMC transforms a program into Static Single Assignment form can also be viewed as executing the program symbolically. However, this encoding is different from the SE of King and we evaluate the two encodings as part of the work reported here. Our methodology is not language specific and only relies on a symbolic executor for that language and an SMT solver.

By using symbolic execution we obtain a translation of a program and assertions into a disjunctive formula encoding the path conditions for each bounded (complete) path explored in the code. This suggests a simple concurrent verification strategy that relies on the observations that any subset of disjuncts in a disjunction $F = F_1 \vee \cdots \vee F_n$ can be separately checked for satisfiability and whenever a subset is found to be satisfiable the satisfiability task can be stopped. Hence the verification of disjunctive formulas is naturally parallelizable.

We have implemented this methodology in a tool called JCBMC, which stands for **J**ava **C**oncurrent **B**ounded **M**odel **C**hecker. The tool is built on top of SPF, and uses it to generate the disjunctive formula from the code (constraint solving is turned off in SPF itself) and while generating the formula it sends sub-formulas to multiple worker threads for satisfiability checking. JCBMC handles programs with multi-threading and recursive input data structures and relies on a standard SMT solver, namely Z3 [94] for solving the constraints. Other solvers can easily be incorporated. One can even use different solvers for solving different path constraints in parallel.

Although JCBMC is only a prototype, and concurrency is implemented by multi-threading but not parallelized yet, its performance, compared with existing tools, i.e. Symbolic PathFinder and CBMC, is remarkable. We summarize our contributions as follows:

- A methodology for concurrent bounded model checking that is based on "classical" symbolic execution and it is naturally parallelizable.
- The methodology is language independent and supports assume-guarantee reasoning.
- A tool JCBMC, a concurrent bounded model checker for Java.
- Experiments to show effectiveness of the tool for verification of programs with multi-threading and data structures.
- Comparisons with bounded model checking and "classical" symbolic execution, as embodied by CBMC and SPF respectively.

5.2 Illustrative example

We illustrate our approach using the simple example program in Figure 5.1. We want to check if the assertion in line 9 is valid for all possible inputs. Note that an analysis using an explicit state model checker such as JPF would not be feasible, as this would involve enumerating all the possible inputs to the program.

```
 1  void test(int x, int y){
 2    if(x > 5){
 3      x++;
 4      if (x < 3)
 5        x--;
 6      else
 7        y = x;
 8    }
 9    assert (x < 10 );
10  }
```

Figure 5.1: A simple example

1. $pc = \top;$ $\quad\sigma = \{x \mapsto x_0, y \mapsto y_0\}$
2. $pc = (x_0 > 5);$ $\quad\sigma = \{x \mapsto x_0, y \mapsto y_0\}$
3. $pc = (x_0 > 5);$ $\quad\sigma = \{x \mapsto x_0 + 1, y \mapsto y_0\}$
4. $pc = (x_0 > 5);$ $\quad\sigma = \{x \mapsto x_0 + 1, y \mapsto y_0\}$
6. $pc = (x_0 > 5);$ $\quad\sigma = \{x \mapsto x_0 + 1, y \mapsto y_0\}$
7. $pc = (x_0 > 5);$ $\quad\sigma = \{x \mapsto x_0 + 1, y \mapsto x_0 + 1\}$

Figure 5.2: SE for the path: (1,2,3,4,6,7)

Figure 5.2 illustrates a part of classical SE with constraints solving on the program following the path $(1, 2, 3, 4, 6, 7)$. This analysis could be performed using e.g. SPF. Instead of concrete values, SE takes the symbols x_0 and y_0 as inputs and executes them just like concrete values. It also keeps track of the path condition pc which consists of the conditions true along that path and the symbolic environment σ which maps variables into expressions over the input symbols x_0, y_0. Typically, whenever the pc is updated, SE checks the satisfiability of pc using an off-the-shelf solver.

Initially, pc is true, and σ maps inputs to theirs symbols. When SE reaches line 2, it updates pc as $(x_0 > 5)$ since this is the condition to reach line 3 where σ becomes $x \mapsto x_0 + 1$. In line 4, the condition $(x < 3)$ is translated in σ to $c \equiv (x_0 + 1 < 3)$. At this point SE calls an SMT solver, and detects that $pc \vdash \neg c$ because $(x_0 > 5) \vdash \neg(x_0 + 1 < 3)$, therefore it jumps to line 6 with pc unchanged.

In our approach for BMC, SE plays the role of generating the formula which encodes the program behaviour and the property to be checked. The satisfiability of the resulting formula will be checked separately by an SMT solver. Therefore, we execute SE *without invoking constraint solving* whenever pc is updated, and we postpone checking the pc until the end of the execution path. In this way, we can save the execution time of calling the solver, but the trade-off is that infeasible paths are also included. However, this will not affect the soundness of the analysis, since constraint solving is performed later. When SE reaches line 9 following the path $\{1, 2, 3, 4, 6, 7\}$, we have:

$$pc = (x_0 > 5) \wedge \neg(x_0 + 1 < 3)$$

Here we reach the property $\mathcal{P}$ to verify, which is $(x < 10)$. We denote by $\mathcal{P}|_\sigma$ the evaluation of $\mathcal{P}$ in the symbolic environment σ. At this point, σ maps x to $x_0 + 1$, which leads to the following:

$$\mathcal{P}|_\sigma = (x_0 + 1 < 10)$$

The property $\mathcal{P}$ is violated in this path if we can find a model for

$$pc \wedge \neg\mathcal{P}|_\sigma = (x_0 > 5) \wedge \neg(x_0 + 1 < 3) \wedge \neg(x_0 + 1 < 10)$$

Setting $x_0 = 11$ will provide such a model. The whole formula generated by our approach for the code in Figure 5.1 is:

$$\begin{aligned} ((x_0 > 5) \wedge \neg(x_0 + 1 < 3) \wedge \neg(x_0 + 1 < 10)) \quad \vee \\ ((x_0 > 5) \wedge (x_0 + 1 < 3) \wedge \neg(x_0 < 10)) \quad \vee \\ (\neg(x_0 > 5) \wedge \neg(x_0 < 10)) \end{aligned}$$

In general, we use SE to explore all possible symbolic paths up to a certain length, and then encode the program together with the property to check into a formula of the form:

$$\bigvee_{i=0}^{M} (pc_i \wedge \neg\mathcal{P}|_{\sigma_i})$$

where N is the number of paths that may trigger the error. This form allows us to divide the formula into blocks of D disjunctions:

$$\bigvee_{i=0}^{D-1} (pc_i \wedge \neg\mathcal{P}|_{\sigma_i}) \vee \bigvee_{i=D}^{2D-1} (pc_i \wedge \neg\mathcal{P}|_{\sigma_i}) \cdots \vee \bigvee_{i=(k-1)D}^{kD-1} (pc_i \wedge \neg\mathcal{P}|_{\sigma_i}) \vee \bigvee_{i=kD}^{M} (pc_i \wedge \neg\mathcal{P}|_{\sigma_i})$$

In this way, we can solve the formula concurrently using several threads, each one solving a single block. A model of a single block is also a model of the formula, therefore the procedure stops when any of the threads find out a model. In JCBMC, after the main thread generates a sub-formula and passes it to a worker thread, it moves on to generate the next sub-formula, while the worker thread solves the given sub-formula concurrently.

5.3 Concurrent Bounded Model Checking

Our method for concurrent bounded model checking is illustrated in Figure 5.3. The inputs to the method are: a program under test, a property to verify and three parameters – B is the search bound, N is the number of workers and D is the number of disjuncts to give to one worker. The goal is to check if the property holds in the program, up to exploration bound B.

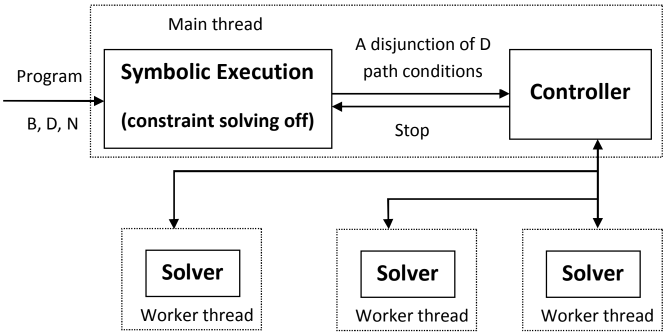

Figure 5.3: Concurrent Bounded Model Checking architecture

5.3.1 Bounded Model Checking by Symbolic Execution

The program under test is analysed using "classical" bounded symbolic execution with constraint solving turned off. This means that whenever a path condition is updated, we do not check its satisfiability, but rather continue the exploration. As a result, symbolic execution may explore infeasible paths, which will be checked later using constraint solving. Our approach can be used for the bounded verification of safety properties, which we assume have been reduced to checking assertions embedded in the code. Furthermore, our method supports both *assume* and *assert* statements to enable *assume-guarantee style* verification. The assumed conditions are simply added to the path conditions during the symbolic execution.

The result of SE is a disjunction of path conditions, encoding constraints on the inputs to follow those paths, up to the pre-specified search bound. From among these paths, only the ones that may lead to assert violations are selected for solving. This is achieved by the *controller* which collects sets of D violating path conditions and sends them for solving to parallel worker threads, using off-the-shelf solvers. The workers start solving as soon as they receive the disjunctive formulas, which may happen while the symbolic execution is still exploring the program. The verification terminates as soon as one of the threads finds a satisfying assignment, in which case an error is reported, or when all the disjunctions are found to be un-satisfiable, in which case the assertion holds (no error) up to the given bound. Note that if the symbolic execution discovers no potentially violating paths (i.e. the error is unreachable), then no solving will be performed.

5.3.2 Comparing our approach with BMC and SE

Compared with classical BMC we use an explicit enumeration of paths, while BMC uses an implicit enumeration of paths. Although at first glance the implicit encoding should be better our experiments, even with sequential JCBMC, show that this is not the case. Furthermore, the explicit enumeration is easily parallelizable, with simple and natural load balancing for different threads. Crucially our approach stops as soon as a path leading to an error is found to be satisfiable, while with classical BMC, all the program needs to be explored.

Compared with classical symbolic execution: we solve only in the end. So obviously the price to be paid is the exploration of infeasible paths. On the other hand, again, it is naturally parallelizable and constraint solving, which is one of the bottlenecks in SE, can be done in parallel, even with different solvers, with little coordination, if any, needed.

In the following we describe in more detail our method. We start with a description of a sequential approach, to clarify how we use symbolic execution to built a disjunctive formula of the path conditions. Solving this formula happens after the symbolic execution, sequentially.

5.3.3 Sequential Verification

Our approach of employing SE for BMC is based on a simple observation. Suppose s_k is an error state in the transition system P. To determine the reachability of s_k from the initial state s_0, BMC builds a series of transitions $s_0 \rightarrow s_1 \rightarrow .. \rightarrow s_k$, resulting in the formula in (2.4) of Section 2.4.1. On the other hand, SE builds the path condition *pc* which is a first order formula also characterising reachability of s_k from s_0. Therefore, both condition (2.4) and *pc* characterize all the inputs that reach the error on that path.

Suppose we want to prove formally that the program P satisfies, within given bounds, a property $\mathcal{P}$ represented by a set of assertions. This means whenever the program reaches an assertion point, the assertion needs to be valid, which means: $\bigwedge(pc_i \rightarrow \mathcal{P}|_{\sigma_i})$. This can be verified by checking the satisfiability of $\bigvee(pc_i \wedge \neg\mathcal{P}|_{\sigma_i})$.

```
function SymExBMC($\sigma$, $pc$, $l$, $i$)
    if ($i > B$)
        return
    Extract statement $s$ at $l$
    while ($s$ is not an if-statement $\wedge$ $l \neq EOF$)
        if ($isSAT = \top$)
            return
        if ($s$ is ‘assume c’)
            $pc \leftarrow pc \wedge c|_\sigma$
        else if ($s$ is ‘assert c’)
            Process($pc \wedge \neg c|_\sigma$)
        else
            Execute the assignment $s$, update $\sigma$
        $l \leftarrow$ next($l$)
        Extract statement $s$ at $l$
    if ($l = EOF$)
        return
    Extract $\{c, l_\top, l_\bot\}$ from if-statement
    $i \leftarrow i + 1$
    $pc_1 \leftarrow pc \wedge c|_\sigma$
    SymExBMC($\sigma$, $pc_1$, $l_\top$, $i$)
    $pc_2 \leftarrow pc \wedge \neg c|_\sigma$
    SymExBMC($\sigma$, $pc_2$, $l_\bot$, $i$)
```

Figure 5.4: Formula generation for BMC

```
function Process($\gamma$)
    $\Gamma \leftarrow \Gamma \vee \gamma$
```

Figure 5.5: Process error paths

Proof 1 *First notice one formula is the negation of the other one, i.e.*

$$\bigwedge(pc_i \to \mathcal{P}|_{\sigma_i}) = \bigwedge(\neg pc_i \vee \mathcal{P}|_{\sigma_i}) = \bigwedge\neg(pc_i \wedge \neg\mathcal{P}|_{\sigma_i}) = \neg\bigvee(pc_i \wedge \neg\mathcal{P}|_{\sigma_i})$$

hence if $\bigvee(pc_i \wedge \neg\mathcal{P}|_{\sigma_i})$ is satisfiable there exists an i such that $pc_i \wedge \neg\mathcal{P}|_{\sigma_i}$ is satisfiable i.e. an initial state leading to the path pc_i and not satisfying the property $\mathcal{P}|_{\sigma_i}$. On the other hand if $\bigvee(pc_i \wedge \neg\mathcal{P}|_{\sigma_i})$ if not satisfiable then no disjunct $pc_i \wedge \neg\mathcal{P}|_{\sigma_i}$ is satisfiable hence there exists no initial state leading to an execution path satisfying the assertion.

The algorithm in Figure 5.4 shows how to build the formula $\Gamma = \bigvee(pc_i \wedge \neg\mathcal{P}|_{\sigma_i})$ for the GC language described in the previous section. In essence, the algorithm runs classical SE with no constraint solving for checking *pc* satisfiability. A statement of GC is determined by its location l, and the function `next`(l) returns the location of the next statement. An if-statement consists of a condition c, the location $l_\top$ if c is true, and the location $l_\bot$ if c is false. In the recursive procedure `SymExBMC` as well as the function `Process`, all parameters are passed by value. In `SymExBMC`, the checking from line 6 to line 7 is only used in concurrent mode. In sequential mode, $isSAT$ is set to false, and it is not changed in the whole procedure. Γ is declared as a global variable, initialised to false (denoted by $\perp$). k is global constant, defining the bound of BMC.

Similar to standard SE, at the beginning the symbolic environment σ maps program inputs to symbols, the path condition *pc* is initialised to true (denoted by $\top$). The depth i of the recursion is initialised to 0. The procedure `SymExBMC` starts by ensuring the search depth i not to reach the bound B (line 2 to line 3). As from line 4 to line 15, it symbolically executes a basic block, i.e. without branching statement, of the program. The basic block ends by an if-statement or when the location l reaches the end of the source file ($l = EOF$). Assumptions and assertions are evaluated by the current symbolic environment (line 9, 11). When there is an assignment, the symbolic environment updates the mapping for the variable in the left hand side by the evaluation of the right hand side. At the end of the block, if there is an if-statement at the next location, `SymExBMC` is recursively called for both the *then* path and the *else* path. The condition is added to the path conditions without any checking of path feasibility.

5.3.4 Concurrent Verification

The simple algorithm presented in the previous sections will essentially enumerate all the possible feasible and infeasible paths through a program (up to a given bound), collect the path conditions for each path into a formula in disjunctive form and then invoke a constraint solver to check the satisfiability, all at once. The disadvantage of the approach is that it needs to enumerate all the possible paths through the program, which can quickly become expensive, especially if multi-threading is also considered. We therefore propose a concurrent algorithm that parallelises `SymExBMC` by delegating the satisfiability check of the disjuncts in Γ to worker threads. The concurrency of the algorithm relies on two parameters: the number of concurrent workers available and the number of disjuncts sent to each worker: the optimal choice is architecture and SMT solver dependent. In our experiments we found 200 disjuncts to be a reasonable choice.

The main function is in Figure 5.6. It initialises Γ, *pc*, σ exactly the same as in sequential mode of `SymExBMC`. Here, $isSAT$ is a boolean variable shared between the threads, and can be modified (set to true) by them. d is also a global variable of

```
$\Gamma$, $isSAT \leftarrow \bot$; $pc \leftarrow \top$; $d, i \leftarrow 0$
$l \leftarrow$ first statement
SymExBMC($\sigma$, $pc$, $l$, $i$)
if ($i > 0 \land isSAT = \bot$)
    Execute Run($\Gamma$, $isSAT$) in worker thread
if ($isSAT = \bot$)
    Return Verification successful
else
    Return Verification failed
```

Figure 5.6: Main thread

```
function Run($\Gamma$, $isSAT$)
    Run SMT-Solver on $\Gamma$
    if ($\Gamma$ has a model)
        $isSAT = \top$
```

Figure 5.7: Worker thread

```
function Process($\gamma$)
    $\Gamma \leftarrow \Gamma \lor \gamma$
    $d \leftarrow d + 1$
    if ($d \geq D$)
        Execute Run($\Gamma$, $isSAT$) in worker thread
        $\Gamma \leftarrow \bot$; $d \leftarrow 0$
```

Figure 5.8: Process error paths for Concurrent BMC

the main thread only to keep the number of current disjuncts. After initializing, the function `SymExBMC` is called. The main difference between sequential and concurrent mode is the function `Process` in Figure 5.8 and the checking from line 6 to line 7 in `SymExBMC`. In sequential mode, *isSAT* is always false, so `SymExBMC` keeps building the formula until it reaches EOF. In concurrent mode, when the number of disjuncts in Γ reaches a bound B, `Process` sends Γ for satisfiability check to a worker thread. Crucially this worker thread can run in parallel to any other running thread as they run completely independent tasks.

Whenever a worker thread finds a model for its own Γ it sets the shared variable *isSAT* to true which will return control to the main thread and end the computation with `Verification failed`. If no thread sets *isSAT* to true then `SymExBMC` will eventually terminate and the remaining disjuncts (whose number is hence less than D) are sent for satisfiability check to a final thread. `Verification successful` is returned only if no thread has set *isSAT* to true during the computation.

Implementation

Our prototype tool, JCBMC, has been implemented following the Observer Design Pattern [95]. SPF executes symbolically the Java bytecode program and acts as the subject. The Controller acts as the observer, it waits for SPF to have generated a sub-formula Γ with D disjuncts and then sends it to an available worker thread. D is a user chosen parameter of JCBMC. The worker thread executes `Run`(Γ, $isSAT$) which first writes Γ into a file in SMT2 format [96], then calls the SMT solver Z3 to check for satisfiability. JCBMC creates a thread pool of N workers; current architecture doesn't support parallelism but only multi-threading.

JCBMC is built on top of SPF and as an extension of the JPF platform, therefore it inherits all the power of JPF and SPF.

5.4 Evaluation

Our evaluation comprises cases studies to compare JCBMC with SPF (with default configuration) and case studies to compare JCBMC with CBMC[1]. To compare with CBMC we have considered C code whose Java translation is almost literal.

An important reminder is that the current implementation of JCBMC is multi-threaded but not yet parallel. In our experiments in a machine with 16 cores, we submitted up to 200 constraint solving jobs to a Java thread pool with 16 worker threads. Contrary to our expectation that each worker thread would concurrently run on a machine core, the `top` command of Linux showed that only 3 worker threads were scheduled to run concurrently. As far as we are aware, it is impossible for a pure Java program to specify the machine core on which a thread is executed. We expect a parallel implementation to have a significant advantage over the current one.

By JSBMC we denote the sequential implementation of JCBMC where a single thread is used. Experiments are run on a machine equipped with dual Xeon(R) E5-2670 CPUs. The results are shown in Tables 5.9 and 5.10. Unless otherwise specified times are in seconds, xmy means x minutes and y seconds, "timed out" is one hour

[1] To compare both tools with the same solver in the experiments CBMC will be called with option –smt2, and we will use Z3 for satisfiability checks.

	SPF	**JSBMC**	**CBMC**	**JCBMC** (10)	**JCBMC** (200)
Array size	Bubble sort with assertion negated				
5	4.517	2.174	0.460	1.097	1.338
6	5.622	12.604	0.817	1.160	1.389
15	36.194	x	56m34.033	1.195	1.948
30	4m32.790	x	timed out	1.387	2.905
100	timed out	x	timed out	4.944	34.697
	Verification of bubble sort				
5	6m19.222	3.712	7.171	4.193	3.622
6	timed out	26.293	37.816	29.512	21.834
7	x	x	5m22.641	x	x
8	x	x	timed out	x	x
Sum of array					
unsafe	1.403	12.671	1m5.738	1.576	2.479
safe	failed	12.030	2.252	9.466	10.614

Figure 5.9: Performance of all tools. JCBMC(10) and JCBMC(200) mean JCBMC is run with the parameter D as 10 and 200 respectively. "failed" refers to SPF failing to solve the constraints using the integrated solver.

and x denotes that the memory limit was exceeded[2]. The source code for the examples can be found at: `https://github.com/qsphan/jpf-bmc`.

5.4.1 Comparing with CBMC and SPF

Bubble Sort

We consider the classical bubble sort algorithm, which has already been studied in the BMC community [97, 98]. Here, differently from [98], we consider the more challenging symbolic version where the values of the array are non-deterministically chosen. We consider both the verification of the assertion "the elements of the array are ordered after bubble sort" and its negation "the elements of the array are not ordered after bubble sort". We analyse a program implementing bubble sort. It will hence contain no bugs for the positive assertion and will be buggy for the negation. Results are shown in Fig 5.9. We notice that while CBMC is better for the positive assertion, JCBMC outperforms the other tools for the negative assertion and is capable of find a counterexample for array sizes of a higher order of magnitude.

[2] This memory problem can be addressed by a different memory manager in JPF or with a direct implementation of `SymExBMC`.

Sum of array

We consider the array case studies taken from [97], in particular *sum_ array_ safe.c* for verification and *sum_ array_ unsafe.c* for refutation. Both of the programs compute the sum of two arrays, *sum_ array_ safe.c* then makes assertions that its computations are correct, while *sum_ array_ unsafe.c* negates these assertions. We increase the sizes of all arrays to 1000. Results show both SPF and JCBMC outperform CBMC for the unsafe version, while CBMC has a slight advantage for the safe version.

5.4.2 Comparing with SPF (Java code)

The following examples consist of substantial Java code which is not naturally translatable in C; we hence compare JCBMC only with SPF. Notice JCBMC and SPF are both extensions of JPF: in the case all inputs are concrete they both reduce to JPF-core hence their performance is identical. Hence we only consider programs with symbolic inputs.

Flap controller

This case study is shipped with the distribution of SPF. It is a multi-threaded program modelling a simplified flap controller on an aircraft. It contains 3 classes, and 80 lines of code. This example demonstrates handling of multi-threading.

Red Black Tree

This is another example from the SPF distributions (3474 LOC in one class). We check for consistency of the tree after performing `put`, `remove`, `get` and `firstKey` symbolically. Results show that both JSBMC and JCBMC significantly outperform SPF.

MER Arbiter

The MER Arbiter models a component of the flight software for NASA JPL's Mars Exploration Rovers (MER). The MER Arbiter has been modelled in Simulink/Stateflow and it was automatically translated into Java using the Polyglot framework and analyzed with SPF [99]. The configuration for our analysis involved two users and five

Tool	**SPF**	**JSBMC**	**JCBMC** (10)	**JCBMC** (200)
Flap controller (unsafe)	1.141	2.899	0.948	1.370
Red-black tree (safe)	53.602	3.942	3.267	2.774
MER Arbiter (unsafe)	5.275	8.111	7.479	7.579
MER Arbiter (safe)	47.065	59.145	57.740	58.886

Figure 5.10: Performance on Flap controller, Red-black tree and MER Arbiter. JCBMC(10) and JCBMC(200) mean JCBMC is run with the parameter D as 10 and 200 respectively.

resources. The example has 268 classes, 553 methods, 4697 lines of code (including the Java Polyglot execution framework) but only approx. 50 classes are relevant. We analyse the code with and without the error (see [99]). The performances of SPF, JSBMC and JCBMC are comparable, with SPF slightly better.

Discussion of experiments

Comparing with CBMC, JCBMC scores better in finding counterexamples than in verifying their absence; this is consistent with its design because a counterexample corresponds to a worker thread finding a model of the formula. Compared with SPF, JCBMC can be much better (see bubble sort or red black tree) but can also be comparable or slightly worse (see MER Arbiter and Flap Controller results). The reason for the latter is that the cost of generating path conditions dominates the cost of solving them. Similarly, SPF failed to generate formulas for bubble sort for sizes 7 and higher. Furthermore, an error path (e.g. in MER) may occur at the beginning of the SE exploration, and it is therefore discovered quickly by SPF, while JCBMC still needs to generate the pre-specified number D of error paths before solving them. The results suggest one direction for future work, namely to investigate improving the cost of SE-based path generation (see last section).

5.5 Discussion of related work

Related approaches on parallelising BMC [100, 101] address parallel solving of the conjunctive formula that is built for BMC and aim at performing solving at different bounds, where some clauses are shared to enable more efficient SAT solving. In contrast we aim to solve the formulas generated with SE for the same bound, which are naturally disjoint resulting in a simpler parallel algorithm. Furthermore our work aims at verifying programs written in high-level languages such as Java and it is not clear how the previous work, performed in the context of finite state automata, would be applicable.

PKIND [102] is a parallel model checker for Lustre that uses k-induction. PKIND runs in parallel the different tasks involved in performing the induction, e.g. the base step, the induction step and also the generation of auxiliary invariants used for verification. Therefore PKIND performs the parallel work at a higher level of granularity than JCBMC. It would be interesting to investigate if we can replace the parallel tasks in PKIND with our own version of SE-based bounded verification, which in turn is parallelized at the level of granularity of symbolic paths.

Parallel model checking has been investigated in the context of explicit-state [103, 104, 105, 106, 107, 108] and symbolic [109, 110] exploration. The latter were done in the context of verification using Binary Decision Diagrams, and hence are very different from ours. These approaches concentrate on partitioning the state space to be explored in parallel and on dealing with the communication overhead between parallel workers. In contrast, in our approach the workers perform the solving independently, with no communication between them.

In a previous paper [111], Staats and Păsăreanu described a framework for performing parallel SE in SPF. The authors used a set of pre-conditions to partition the symbolic execution tree to distribute its processing. These pre-conditions were computed *a priori*, using a "shallow" symbolic execution up to a small exploration bound, to *statically* compute the different partitions of the input space with no communication overhead.

Previous approaches to parallel SE [112, 113, 114] operate primarily by *dynamically* partitioning the symbolic execution tree for load balancing, which may result in better use of computational resources but also in more communication overhead. All these approaches were done in the context of "classical" [111, 112] or dynamic [113, 114] symbolic execution, using constraint solving during path generation. In contrast the approach we advocate here has a clear separation between path generation and constraint solving, allowing us to easily achieve load balancing between workers, with little communication overhead. It would be interesting to compare experimentally and to also combine the techniques in JCBMC and the parallel version of SPF and we plan to do that in future work.

Also related is the work on parallel SAT and SMT solving [115, 116, 117], which can be seen complementing the work presented here, in the sense that we can use e.g. the parallel version of Z3 [117] in each of the workers to further speed up our proposed approach.

Chapter 6

Quantifying Information Leaks using Reliability Analysis

In chapter 4, we have presented the first technique to use Symbolic Execution for Quantitative Information Flow analysis. There, Symbolic Execution has been used for both exploring the program and counting the models. In this chapter, we explore an alternative approach which uses Symbolic Execution only for exploring the program, leaving the task of counting models for a reliability analysis tool.

This chapter can be divided in two parts. The first part studies the *qualitative* aspect of information flow analysis. Its contribution is a novel and practical approach for self-composition using Symbolic Execution.

In the second part of the chapter, we show the relation between reliability analysis and Quantitative Information Flow. Exploiting this relation, we combine our new self-composition technique with a Symbolic Execution-based reliability analysis tool to quantify information leaks in Java bytecode.

6.1 Introduction

Recall that in section 2.1.1, we have introduced the theorem proving approach to non-interference, in which non-interference is logically formulated by *self-composition*. As far as we are aware, this has been the only approach for qualitative analysis that returns neither false positives nor false negatives. We also quoted the claim of Terauchi and Aiken [57] that self-composition was impractical and that it would be unlikely to expect any future advance. The main limitations of self-composition, as pointed out by Terauchi and Aiken, come from the symmetry and redundancy of the self-composed program, which lead to some partial-correctness conditions that hold between `P` and $\texttt{P}_1$. To find these conditions is crucial for the effectiveness of the analysis, however, finding them is in general impractical. Here we present a *practical* implementation for self-composition.

6.1.1 Qualitative analysis

The idea of self-composition is to have a copy $\texttt{P}_1$ of the program `P` to compare with itself. The approach can be divided into two steps: the first step is to compose the program with a copy of itself; the second one is to perform analysis on the self-composed program. Our approach is to delay self-composing to the second step: first, we perform analysis on the original program with Symbolic Execution; second, we self-compose the result of the analysis to get the formula of self-composition.

We expand the idea of comparing the program `P` with it copy $\texttt{P}_1$ into comparing all pairs of executions ρ of `P` and ρ_1 of $\texttt{P}_1$. Since it is impossible to enumerate all possible executions, we use Symbolic Execution to synthesize the symbolic paths that represents a set of concrete executions, and perform comparison on these symbolic paths, which we formulate as *path-equivalence.*

The delay of self-composing after performing the analysis is the main novelty of our approach. In this way, we could avoid the symmetry and redundancy of the self-composed program. Moreover, the symbolic paths synthesized by Symbolic Execution are presented by first-order theories, just as the generated formula of self-composition. The validity of this formula can be automatically and efficiently checked by powerful SMT solvers.

6.1.2 Quantitative analysis

Traditional self-composition techniques can only tell if a program leaks information. On the contrary, we can refine our Symbolic Execution-based self-composition into a more fine-grained analysis that can decide if a path of the program leaks information. We then quantify the leaks for each symbolic path using a reliability analysis tool.

Our approach is implemented into an automated tool, called QILURA (which stands for **Q**uantify **I**nformation **L**eaks **U**sing **R**eliability **A**nalysis). Given a program, and inputs labeled as *high* and *low*, QILURA computes an upper bound on the maximum number of bits that the program can leak to a public observer. Our implementation is done in the context of Java bytecode programs and the SPF [81] symbolic execution engine, extended for reliability analysis [118]. However, the work is general and can be applied in the context of any programming language for which a symbolic execution tool exists.

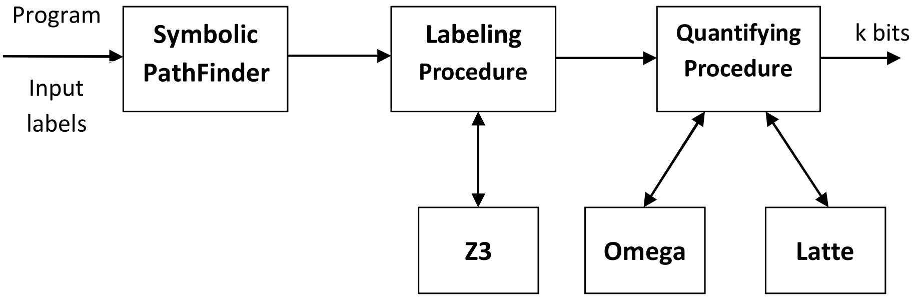

Figure 6.1: Architecture of QILURA

At a high level, the architecture of QILURA is depicted in Figure 6.1. The user labels the inputs of the program with *high* and *low*. The program is then passed to SPF to collect all possible symbolic paths. The *Labeling Procedure*, using a fine-grained self-composition [56], classifies all the paths into three categories: `clean`, `direct` and `indirect`. The procedure uses Z3 [79] for satisfiability checking of self-composition condition.

Finally, the *Quantifying Procedure* uses Barvinok model counting techniques [119] over the symbolic constraints (simplified using Omega [120]) collected by SPF to count the number of inputs that follow paths labeled with "direct" and provides an upper bound of k bit on the leaks.

6.2 Preliminaries

This section reformulates Symbolic Execution and provides some background on the new Symbolic Execution-based reliability analysis framework of Filieri et al. [118].

6.2.1 Symbolic Execution

In chapter 4, we described Symbolic Execution on a Symbolic Transition System which modelled in detail the transition of a program with guards and actions. These details are not necessary in this chapter and are not captured in the transition system in section 2.3, that we use to describe concrete execution. Therefore, we reformulate Symbolic Execution in a more coarse-grained transition system that only takes in to account the source and target states of a transition. A program `P` is modelled as follows:

$$P = (S^*, I^*, F^*, T^*)$$

where S^* is the set of symbolic states; each $s^* \in S^*$ represents a set of concrete states $s \in S$. $I^* \subseteq S^*$ is the set of initial symbolic states; $F^* \subseteq S^*$ is the set of final symbolic states; and $T^* \subseteq S^* \times S^*$ is the transition function. A symbolic path (symbolic trace) of the program `P` is represented by a sequence of symbolic states:

$$\rho^* = s_0^* s_1^* .. s_k^*$$

such that $s_0^* \in I^*, s_k^* \in F^*$ and $\langle s_i^*, s_{i+1}^* \rangle \in T^*$ for all $i \in \{0, \ldots, k-1\}$. The symbolic semantics of `P` is then defined as the set of all symbolic paths $\mathcal{R}^*$, which is also called as the *symbolic execution tree*. Likewise, each $\rho^* \in \mathcal{R}^*$ represents a set of traces $\rho \in \mathcal{R}$.

We denote by $X|_y$ the value of the variable X at the state y. After symbolically executing the program `P` with initial input symbols $H = \alpha, L = \beta$, for each $s_i^* \in F^*$, i.e. each leaf of the symbolic execution tree, we have a symbolic formula for the value of the output `O` in the symbolic environment:

$$O|_{s_i^*} = f_i(\alpha, \beta)$$

Another product of SE is the path condition $pc_i \equiv c_i(\alpha, \beta)$ for s_i^* to be reachable. Each pc_i corresponds to a symbolic path ρ_i^*. The following theorem was also proved by King [46]:

Theorem 3

$$\forall i, j \in [1, n] \wedge i \neq j . pc_i \wedge pc_j = \bot$$

We define the function *path* such that:

$$path(\rho_i^*) = pc_i$$

The output `O` can be considered as a result of the following function:

$$O = \left\{ \begin{array}{ll} f_1(\alpha, \beta) & \text{if } c_1(\alpha, \beta) \\ f_2(\alpha, \beta) & \text{if } c_2(\alpha, \beta) \\ \ldots & \ldots \\ f_n(\alpha, \beta) & \text{if } c_n(\alpha, \beta) \end{array} \right\} \tag{6.1}$$

Or the following always holds:

Corollary 2

$$\forall i \in [1, n].c_i(\alpha, \beta) \rightarrow O = f_i(\alpha, \beta)$$

f_i and c_i are in general combination of first-order theories, e.g. *linear arithmetic*, *bit vector* and so on. SE tools make use of off-the-shelf SMT solvers to check the satisfiability of c_i, and eliminate unreachable paths (which may appear in the control flow graph).

6.2.2 Reliability Analysis

Reliability analysis [47] aims to compute the probability that a program successfully accomplishes its task without errors. Most previous work performs reliability analysis at early stages of design, on an architectural abstraction of the program, and thus they are not applicable to source code.

In [118], Filieri et al. introduced the first approach that can compute the reliability of program from Java bytecode. Their approach is to use SE to enumerate each of the symbolic paths (and its path condition pc_i). The symbolic path is then labelled as: (i)**T** if the program accomplishes the task; (ii) **F** if the program reaches an error state; (iii) **G** if we cannot decide because the path is not fully explored (G stands for grey).

From the path condition pc_i, Filieri et al. use the tool Latte [119] to compute efficiently the number of inputs $\#(pc_i)$ that satisfies the path condition pc. The reliability of the program, i.e. the probability that the program accomplishes its task, is then computed as:

$$\mathcal{R} = \frac{\Sigma\#(pc^T)}{\Sigma\#(pc^T) + \Sigma\#(pc^F) + \Sigma\#(pc^G)}$$

For QILURA we do not compute probabilities but use directly the counts over the computed symbolic constraints.

6.3 Self-composition by Symbolic Execution

To avoid the limitation of the theorem proving approach, we need to reformulate the self-composition formula into a simpler logic which does not contain the program `P`. This is made possible by using the trace semantics of programs.

6.3.1 Self-composition as path-equivalence

Given a program P that takes secret input H, public input L and producing public output O; $\mathtt{P}_1$ is the same program as P, with all variables renamed: H as $\mathtt{H}_1$, L as $\mathtt{L}_1$ and O as $\mathtt{O}_1$. The trace semantics of P and $\mathtt{P}_1$ are $\mathcal{R}$ and $\mathcal{R}_1$ respectively.

Definition 22 (trace-equivalence) *The program P satisfies non-interference if:*

$$\forall \rho \in \mathcal{R}, \rho_1 \in \mathcal{R}_1 . L|_{init(\rho)} = L_1|_{init(\rho_1)} \rightarrow O|_{fin(\rho)} = O_1|_{fin(\rho_1)} \tag{6.2}$$

It is stated similarly to the Hoare triple in (2.1): for all possible pairs of traces ρ of P, and ρ_1 of $\mathtt{P}_1$: if $L = L_1$ at the initial states, then $O = O_1$ at the final states. At this point, we have a formulation of self-composition that does not involve the programs P and $\mathtt{P}_1$.

However, even with simple programs, it is impossible to compute all the traces. Our solution is to use trace-equivalence with SE. Recall that each symbolic path represents a set of traces, and it is possible to build a complete symbolic execution tree (here we only consider bounded programs). Following Corollary 2, trace-equivalence in the context of SE is redefined as follows:

Definition 23 (path-equivalence) *The program P satisfies non-interference if and only if for all $\rho^* \in \mathcal{R}^*$ and for all $\rho_1^* \in \mathcal{R}_1^*$, the following equation holds:*

$$(L|_{init(\rho^*)} = L_1|_{init(\rho_1^*)}) \wedge path(\rho^*) \wedge path(\rho_1^*) \rightarrow (O|_{fin(\rho^*)} = O_1|_{fin(\rho_1^*)}) \tag{6.3}$$

In this way, we have an SMT formula, i.e. a combination of first-order theories. This is the key novelty of our approach, since the formulation of self-composition in first-order theories enables us to solve it efficiently using off-the-shelf SMT solvers.

6.3.2 Path-equivalence generation

Suppose P is symbolically executed with $H = \alpha, L = \beta$. To simplify the formula, we choose the input symbols for $\mathtt{P}_1$ as $H_1 = \alpha_1, L_1 = \beta$ so that $L|_{init(\rho^*)} = L_1|_{init(\rho_1^*)}$ is automatically satisfied. That means:

$$(H|_{init(\rho^*)} = \alpha) \wedge (L|_{init(\rho^*)} = \beta) \wedge (H_1|_{init(\rho_1^*)} = \alpha_1) \wedge (L_1|_{init(\rho_1^*)} = \beta)$$

Given the result of SE is a function of the output O as in (6.1), the path-equivalence in (6.3) can be rewritten as:

$$PE \equiv DF \wedge IF$$

where:

$$DF \equiv \bigwedge_{i=1}^{n} c_i(\alpha, \beta) \wedge c_i(\alpha_1, \beta) \rightarrow (f_i(\alpha, \beta) = f_i(\alpha_1, \beta))$$

$$IF \equiv \bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} c_i(\alpha, \beta) \wedge c_j(\alpha_1, \beta) \rightarrow (f_i(\alpha, \beta) = f_j(\alpha_1, \beta))$$

DF checks the path-equivalence when both P and $\mathtt{P}_1$ follow the same symbolic path, and thus it guarantees the absence of direct flows. On the other hand, *IF* checks the path-equivalence when P and $\mathtt{P}_1$ follow different symbolic paths, and it guarantees the absence of implicit flows.

6.3.3 Examples

We illustrate the approach with some toy examples. Here we assume the same setting as above: a program P with confidential input H, public input L, and output O. SE executes P with input symbols $H = \alpha$ and $L = \beta$.

Implicit flow

Consider the password checking program: By SE, we have:

```
if (H == L)
     O = true;
else
     O = false;
```

$$O = \left\{ \begin{array}{ll} true & \text{if } \alpha = \beta \\ false & \text{if } \alpha \neq \beta \end{array} \right\}$$

DF and *IF* are generated as follows:

$$DF \equiv (\alpha = \beta \wedge \alpha_1 = \beta \rightarrow true = true) \wedge (\alpha \neq \beta \wedge \alpha_1 \neq \beta \rightarrow false = false)$$

$$IF \equiv \alpha = \beta \wedge \alpha_1 \neq \beta \rightarrow true = false$$

It is trivial to prove that *DF* is valid and *IF* is invalid, and thus the program violates non-interference and leaks information via implicit flows.

No flow

Consider the modified version of the password checking procedure in Listing 2.1.1.

```
if (H == L)
    O = false;
else
    O = false;
```

By SE, we have:

$$O = \left\{ \begin{array}{ll} false & \text{if } \alpha = \beta \\ false & \text{if } \alpha \neq \beta \end{array} \right\}$$

DF and *IF* are generated as follows:

$$\begin{aligned} DF &\equiv (\alpha = \beta \land \alpha_1 = \beta \rightarrow false = false) \land (\alpha \neq \beta \land \alpha_1 \neq \beta \rightarrow false = false) \\ IF &\equiv \alpha = \beta \land \alpha_1 \neq \beta \rightarrow false = false \end{aligned}$$

It is trivial to prove that both *DF* and *IF* are valid, and thus the program satisfies non-interference.

No confidential data involved.

Consider the password checking program, with a small modification to exclude the confidential data in its computation, i.e. to make it secure.

```
if (L == 3)
    O = true;
else
    O = false;
```

Similarly we have:

$$O = \left\{ \begin{array}{ll} true & \text{if } \beta = 3 \\ false & \text{if } \neg(\beta = 3) \end{array} \right\}$$

DF and *IF* are derived as:

$$\begin{aligned} DF &\equiv (\beta = 3 \land \beta = 3 \rightarrow true = true) \land (\neg(\beta = 3) \land \neg(\beta = 3) \rightarrow false = false) \\ IF &\equiv \beta = 3 \land \neg(\beta = 3) \rightarrow true = false \end{aligned}$$

Both *DF* and *IF* are valid, which confirms the intuition that the program is secure.

Both implicit and explicit flows

Consider again the data sanitization program:

```
if (H < 16)
    O = H + L;
else
    O = L;
```

The summaries and path conditions returned by SE are as follows:

$$O = \left\{ \begin{array}{ll} \alpha + \beta & \text{if } \alpha < 16 \\ \beta & \text{if } \neg(\alpha < 16) \end{array} \right\}$$

DF and *IF* are generated similarly:

$$DF \equiv (\alpha < 16 \wedge \alpha_1 < 16 \rightarrow \alpha + \beta = \alpha_1 + \beta) \wedge (\neg(\alpha < 16) \wedge \neg(\alpha_1 < 16) \rightarrow \beta = \beta)$$
$$IF \equiv \alpha < 16 \wedge \neg(\alpha_1 < 16) \rightarrow \alpha + \beta = \beta$$

It is easy to find counterexamples to make *DF* and *IF* invalid, for example: $(\alpha = 1; \alpha_1 = 2)$ for *DF* and $(\alpha = 1; \alpha_1 = 17)$ for *IF*. So the program leaks via both implicit and explicit flows.

6.3.4 Optimization

After SE collects the symbolic paths and path conditions as in (6.1), there are three possible cases for a $\rho^* \in \mathcal{R}^*$:

- $O|_{\mathit{fin}(\rho^*)}$ and $\mathit{path}(\rho^*)$ does not contain α: ρ_s can be classified as "secure", and is excluded in computing *DF* and *IF*. With this optimization, programs like the one in Figure 2.1.1 can be classified as secure without computing anything.
- $O|_{\mathit{fin}(\rho^*)}$ does not contain α; $\mathit{path}(\rho^*)$ contains α: in this case, computing *DF* for this path will result a formula $C(\beta) \rightarrow \mathit{true}$ which is valid for any C. Therefore, ρ^* is excluded in computing *DF*.
- $O|_{\mathit{fin}(\rho^*)}$ contains α: we can conclude that the program is insecure, leaking information via direct flow. This is very similar to taint analysis, however SE is more precise, since it can detect cases such as $H \times 0$ or $H - H$ and so on.

These three optimizations are sufficient to eliminate the computation of *DF*, and simplify the formula to be validated.

In a previous paper [57], Terauchi and Aiken presented an interesting program that computes Fibonacci numbers, and containing confidential data. Applying the self-composition technique for this program turned out to be very tricky because of the symmetry and redundancy of the self-composed program, and the state-of-the-art safety analysis tool BLAST failed to terminate. With our Symbolic Execution-based approach and these optimizations, the case study becomes trivial, and can be checked quickly without computation of *DF* and *IF*.

6.4 Quantifying Information Leaks by combining Self-composition and Reliability Analysis

At a high level, QILURA performs a two-step analysis. First, SE is run to collect all symbolic paths of the program (up to a user-specified depth), then each path is assigned a label: (i) **clean**: if it leaks no information, (ii) **direct**: if it leaks information via direct flow, and (iii) **indirect**: if it leaks information via indirect flow. Secondly, a model counter for symbolic paths from [118] is used to count the number of possible inputs that go to "direct" paths, and compute an upper bound on the leakage. QILURA is available at: `https://github.com/qif/jpf-qilura`.

6.4.1 Fine-grained self-composition

Checking the satisfiability of the path-equivalence condition $PE \equiv DF \wedge IF$ in section 6.3.2 can only decide whether a program leaks information. However, we can refine it to a path-level analysis of leakage.

We assume the same settings as in section 6.3.2: we symbolically execute the program $\mathtt{P}$ with the input symbols: $\mathtt{H} = \alpha, \mathtt{L} = \beta$, and the program $\mathtt{P}_1$ with the input symbols $\mathtt{H}_1 = \alpha_1, \mathtt{L}_1 = \beta$. Based on the path-equivalence condition for direct flow and indirect flow, we define the self-composition condition for each symbolic path as follows.

Definition 24 *Given a path ρ_i^* such that $path(\rho_i^*) = c_i(\alpha, \beta)$ and $O|_{fin(\rho_i^*)} = f_i(\alpha, \beta)$, ρ_i^* does not leak information via direct flow if and only if the following condition holds:*

$$c_i(\alpha, \beta) \wedge c_i(\alpha_1, \beta) \rightarrow (f_i(\alpha, \beta) = f_i(\alpha_1, \beta))$$

Applying the *material implication* rule on the condition above results in:

$$\neg(c_i(\alpha,\beta) \land c_i(\alpha_1,\beta)) \lor (f_i(\alpha,\beta) = f_i(\alpha_1,\beta))$$

In case ρ_i^* leaks information via direct flow, the condition above is violated, which means its negation is satisfiable:

$$\neg(\neg(c_i(\alpha,\beta) \land c_i(\alpha_1,\beta)) \lor (f_i(\alpha,\beta) = f_i(\alpha_1,\beta)))$$

This formula can be simplified using De Morgan's law as:

$$c_i(\alpha,\beta) \land c_i(\alpha_1,\beta) \land \neg(f_i(\alpha,\beta) = f_i(\alpha_1,\beta)) \tag{6.4}$$

Definition 25 *Given two symbolic paths ρ_i^* and ρ_j^* such that $path(\rho_i^*) = c_i(\alpha,\beta)$, $O|_{fin(\rho_i^*)} = f_i(\alpha,\beta)$, $path(\rho_j^*) = c_j(\alpha,\beta)$ and $O|_{fin(\rho_j^*)} = f_j(\alpha,\beta)$, ρ_i^* and ρ_j^* do not leak information via indirect flow if and only if the following condition holds:*

$$c_i(\alpha,\beta) \land c_j(\alpha_1,\beta) \rightarrow (f_i(\alpha,\beta) = f_j(\alpha_1,\beta))$$

Similar to the derivation above, in case ρ_i^* and ρ_j^* leaks information via indirect flow, the following formula is satisfiable.

$$c_i(\alpha,\beta) \land c_j(\alpha_1,\beta) \land \neg(f_i(\alpha,\beta) = f_j(\alpha_1,\beta)) \tag{6.5}$$

Based on the conditions in (6.4) and (6.5), we implement a procedure to label all symbolic paths as in Figure 6.2. The function `isSAT` is implemented by calling the SMT solver Z3 [79].

The algorithm of this procedure is straightforward. At the beginning, all paths are labeled as being `clean`. The procedure then searches for direct flow by checking the condition (6.4) for all paths. It then searches for indirect flow by checking the condition (6.5) on all possible pairs of symbolic paths.

6.4.2 Model Counting for Symbolic Paths

For a symbolic path ρ, let $\#in(\rho)$ and $\#out(\rho)$ denote the number of concrete inputs and outputs of ρ respectively. Obviously $\#in(\rho_i)$ is $\#(pc_i)$ computed in [118]. After being labeled, all paths are classified into three categories: clean, direct and indirect. So the channel capacity is bounded by:

$$CC(P) \leq \log_2(S\#out(\rho_c) + S\#out(\rho_i) + S\#out(\rho_d))$$

where ρ_c is the clean path, ρ_i is the indirect path, and ρ_d is the indirect path.

```
for all $\rho_i$ do {
    label[i] ← clean
    $\varphi \leftarrow c_i(\alpha, \beta) \wedge c_i(\alpha_1, \beta) \wedge \neg(f_i(\alpha, \beta) = f_i(\alpha_1, \beta))$
    if (isSAT($\varphi$)) label[i] ← direct
}
for $i = 1$ to $n - 1$ do
    for $j = i + 1$ to $n$ do {
        $\varphi \leftarrow c_i(\alpha, \beta) \wedge c_j(\alpha_1, \beta) \wedge \neg(f_i(\alpha, \beta) = f_j(\alpha_1, \beta))$
        if (isSAT($\varphi$)) {
            if (label[i] = clean) label[i] ← indirect
            if (label[j] = clean) label[j] ← indirect
        }
    }
```

Figure 6.2: Fine-grained self-composition

- Since clean paths are not interfered by the confidential input we can replace $S\#out(\rho_c)$ with 1.
- An indirect path only reveals that the program follows that path, its output is not interfered, and each path has one output. Thus, $S\#out(\rho_i)$ is just the number of indirect paths.
- We hence only need to compute $S\#out(\rho_d)$.

A deterministic program can be viewed as a function that maps each input to exactly one output (denotational semantics). Therefore, the number of inputs is always greater than or equal to the number of possible outputs. This means $\#in(\rho) \geq \#out(\rho)$, and $S\#in(\rho_d) \geq S\#out(\rho_d)$.

By using the model counting engine for symbolic paths in [118], we can compute $S\#in(\rho_d)$, and hence compute an upper bound of channel capacity $CC(P)$.

6.5 Evaluation

Automated QIF analysis is notoriously hard. To the best of our knowledge, the only tool for QIF analysis of Java bytecode is our own work jpf-qif [7] which uses SE for QIF analysis, but no model counting. Instead jpf-qif adds the conditions for testing each bit of the output at the end of the program, hence exploring all these conditions using SPF. We compare jpf-qif with QILURA below.

We also compare with BitPattern [18], which computes an upper bound on channel capacity by exploring the relations between every pair of bits of the output. In more recent work [121], BitPattern was improved using new heuristics. We compare QILURA with (the improved) BitPattern on several case studies taken from [18, 121].

```
if (H > 999){
  O = -1;
}
O = H;
O = O - H;
```

Figure 6.3: No flow

Moreover, we consider a special case in Figure 6.3 when the program does not leak any information to assess the effectiveness and precision of our technique in such a corner case. The program does not leak information because the output `O` is always 0 regardless of the value of the secret `H`.

6.5.1 Results and discussions

Figure 6.4 summaries our experiment, we take the time from the faster version of BitPattern in [121]. Note that in both [18] and [121], the authors manually transform the programs into bit vector predicates, so there will be extra time if they automate this process.

Case Study	**jpf-qif**		**QILURA**		**BitPattern**	
	Capacity	Time	Bound	Time	Bound	Time
No Flow	0	2.304	0	0.790	-	-
Sanity check, base =`0x00001000`	4	45.324	4.09	1.066	4	0.036
Sanity check, base =`0x7ffffffa`	4	35.346	4.09	1.049	4.59	0.203
Implicit Flow	2.81	0.897	3	0.796	3	0.011
Electronic Purse	2	1.169	2.32	0.854	2	0.157
Ten random outputs	3.32	1.050	3.32	0.814	18.645	0.224

Figure 6.4: Capacity and bounds are in bits, times are in seconds. "-" means "not reported".

Comparing with jpf-qif, QILURA has both advantages and disadvantages. As shown in Figure 6.4, thanks to the model counting tool Latte, QILURA is much faster than jpf-qif while the upper bounds it computed only deviate to a small extent from the exact channel capacities.

This increase in performance comes with a price, Latte can only count models of a system of linear integer inequalities $A\bar{x} \geqslant \bar{b}$. For this reason, QILURA cannot analyse the case studies of CRC and Tax Record in chapter 4, which have complicated constraints. The limitation due to using Latte is shared with previous work [19, 21] which were also demonstrated with toy examples.

The BitPattern technique can also compute rather tight upper bounds in most of the cases. However, by analysing the relations of pairs of bits, the technique is vulnerable when possible values of the output are not in a specific range, as shown in the last case study.

6.6 Discussion of related work

Self-composition was first introduced by Darvas et al. [55] who expressed it in a dynamic logic and proved information flow properties for Java CARD programs. Their approach is not automated, requiring users to provide loop invariants, induction hypotheses and so on. Barthe et al. [56] then coined the term "self-composition" and investigated its theoretical aspects, extending the problem to non-deterministic and termination-sensitive cases.

Terauchi and Aiken [57] found that self-composition was problematic, since the self-composed programs contains symmetry and redundancy. They proposed a type-directed transformation for a simple imperative language to deal with the problem. Milushev et al. [122] implemented this type-directed transformation and used *Dynamic Symbolic Execution* (also known as *concolic testing*) as a program analysis tool for non-interference.

To our knowledge, our technique is unique in that it only performs analysis on the original program, rather than the self-composed program, the idea of self-composition is shown in the way we rename the symbolic formula, not in the analysis stage.

Backes et al. [19] describe how to use the model checker ARMC and Latte for QIF analysis. Their technique is very precise but also extremely expensive: it involves input counting to compute the pre-image of the observables; in contrast our input counting is used for counting the observable. The work of Backes et al. is extended in [21], which uses the interactive theorem prover KeY instead of ARMC, and requiring significant user effort. Moreover, this work is based on "*classical*" self-composition, and as Terauchi and Aiken [57] have pointed out, it is unlikely to be practical. Of course, both [19] and [21] were demonstrated with toy examples.

The only technique that can precisely determine if a program leaks information is self-composition [56]. QILURA also uses self-composition with the key difference that it is able to determine if a single symbolic path leaks information.

Chapter 7

A solver for Model Counting Modulo Theories

In Section 3.3, we have cast the problem of QIF analysis into the #SMT problem and proposed two #SMT-based approaches to QIF, which can be summarized as the following.

$$\begin{tikzcd}
\texttt{P} \arrow[rr, leftrightarrow] & & \varphi_P \\
\text{QIF} & & \#\text{SMT} \\
\text{Formal methods} & & \text{DPLL}(\mathcal{T})
\end{tikzcd}$$

So far we have investigated the approach on the left-hand side: we directly analyse the program `P`, and use formal methods, specifically Bounded Model Checking and Symbolic Execution, in a way that mimics a #SMT solver for QIF analysis.

This chapter investigates our second approach on the right-hand side: it demonstrates how to build from the program `P` a formula φ_P with the two properties described in section 3.3, and how to build a #SMT solver to count the models of φ_P.

Moreover, we study a variant of the #SMT problem: the *All-Solution Satisfiability Modulo Theories* (All-SMT) problem. All-SMT is only different from #SMT in that instead of counting the number of models, it asks for the enumeration of each of the models. We show that our algorithms for #SMT can also be used for All-SMT, and we propose the use of an All-SMT solver in new application domains: Bounded Model Checking, automated test generation and reliability analysis.

7.1 Quantifying Information Leaks using a #SMT solver

We assume the setting in our attacker model in Figure 2.1: a program `P` that takes secret input `H`, public input `L` and producing public output `O`. Our analysis consists of two steps as the following.

- The first step is to build a first-order formula from the program `P` a formula φ_P with the following properties: (i) φ_P contains a set of Boolean variables $V_I := \{p_1, p_2, .., p_M\}$; (ii) $p_i = \top$ if and only if b_i is 1, and $p_i = \bot$ if and only if $b_i = 0$.
- The second step is to use a #SMT solver to count the number of models of φ_P with respect to the set V_I.

We will show how to build a #SMT solver later in the next section. At the moment, we assume that there is such a solver for our QIF analysis.

7.1.1 Illustrative Example

To demonstrate the approach, let us consider again the previously used illustrative example in Figure 3.1, which can be encoded into a first-order formula as in Figure 7.1.

```
L = 8;
if (H < 16)
  O = H + L;
else
  O = L;
```

$$\begin{array}{ll} (L_1 = 8) & \wedge \\ (G_0 = H_0 < 16) & \wedge \\ (O_1 = H_0 + L_1) & \wedge \\ (O_2 = L_1) & \wedge \\ (O_3 = G_0 ? O_1 : \mathtt{O}_3) & \end{array}$$

Figure 7.1: A simple program encoded into a first-order formula

The program is transformed into Static Single Assignment (SSA) form [84]: variables are renamed when they are reassigned. At the beginning, assuming that the variables `H`, `L` and `O` take the value H_0, L_0 and O_0 respectively after being declared. Then, L_1 is the value of the variable `L` after being reassigned (as 8), and similarly for the rest of the program.

We then need to build the set of boolean variable $V_I := \{p_1, p_2, .., p_{32}\}$. In chapter 3, we directly analysed the program, and thus hence the set V_I by instrumenting the source code of the program `P` with the block of code in Figure 3.4. This block of code used bitwise operators to extract each bit of the output `O`. Here, we analyse the formula (encoded from the program), and hence build the set V_I by instrumenting the formula. Moreover, the formula needs to be in a first-order theory $\mathcal{T}$ that supports bitwise operators. Fortunately, the model checker CBMC can automatedly transform a C program into a formula in the theory of bit vector QF_AUFBV [96] which satisfies this requirement.

The formula in Figure 7.1 can be easily expressed in QF_AUFBV with O_3 declared as a 32-bit vector. We instrument the formula by adding a set of Boolean variables $V_I = \{p_1, p_1, \dots p_{32}\}$, each one tests the value of a bit of O_3. For example:

$$\texttt{(assert (= (= \#b1 ((\_ extract 0 0) } O_3\texttt{)) } p_1\texttt{))}$$

This statement in SMT-LIB v2 format [96] extracts the first bit of O_3 (all bits from position 0 to position 0), then comparing if this bit is equal to 1 (#b1). The Boolean variable p_1 is asserted to be the truth value of this comparison. Similar settings are applied for the rest of Boolean variables $p_2, p_3, \dots, p_{32}$.

At this point we have built a formula φ_P that characterizes the behaviour of the program `P`, and contains a set V_I of Boolean variables, each one represents a bit of the output `O` of the program `P`. By using a #SMT solver to count the number of models of φ_P with respect to the set V_I, which is also the number of possible values of the output `O`, we can conclude the maximum leakage of the program `P` as per definition 11.

7.1.2 Program transformation with CBMC

In our two-step analysis, the second step is automated with a #SMT solver, we only need to automate the first step: building a formula φ_P from the program `P`.

$$\boxed{\begin{array}{l}
\mathcal{C}(``\texttt{if(c)}\ I_1\ \texttt{else}\ I_2", g) := \mathcal{C}(I_1, g \wedge \rho(c)) \wedge \mathcal{C}(I_2, g \wedge \neg\rho(c)) \\
\mathcal{P}(``\texttt{if(c)}\ I_1\ \texttt{else}\ I_2", g) := \mathcal{P}(I_1, g \wedge \rho(c)) \wedge \mathcal{P}(I_2, g \wedge \neg\rho(c)) \\
\mathcal{C}(``I_1; I_2", g) := \mathcal{C}(I_1, g) \wedge \mathcal{C}(I_2, g) \\
\mathcal{P}(``I_1; I_2", g) := \mathcal{P}(I_1, g) \wedge \mathcal{P}(I_2, g) \\
\mathcal{P}(``\texttt{assert(a)}", g) := g \rightarrow \rho(a) \\
\mathcal{C}(``\texttt{v = e}", g) := (v_\alpha = (g?\rho(e) : v_{\alpha-1}))
\end{array}}$$

Figure 7.2: The two functions $\mathcal{C}(\texttt{P}, g)$ and $\mathcal{P}(\texttt{P}, g)$ for program transformation. $\rho(c)$ is expression c after being renamed as per SSA form; $v_{\alpha-1}$ and v_α are value of v before and after the assignment respectively.

The model checker CBMC transforms a program `P` and a guard g into a logical formula using two functions: $\mathcal{C}(\texttt{P}, g)$ transforms the program constraints, and $\mathcal{P}(\texttt{P}, g)$ transforms the program specification, namely assertions. At the beginning, the guard g is initialised to $\top$. Both functions are defined by induction on the syntax of program as in Figure 7.2 (interested readers are pointed to [67] for full details).

In its default settings, CBMC transforms the program and its specification into a propositional formula. However, it also has the option $--$`smt2` (still experimental) to transform the program and specification into a QF_AUFBV formula in SMT-LIB v2 format. Hence, we use leverage this option to build the formula φ_P.

Recall (section 2.4.1) that the formula generated by CBMC is in the form $\mathcal{C} \wedge \neg\mathcal{P}$, where $\mathcal{C}$ is the program constraints, and $\mathcal{P}$ is the program specification, i.e. assertions. Since we only assess the security of a program when it is free from errors, we only need the program constraints $\mathcal{C}$. However, without a specification, the aggressive program slicing in CBMC can decide immediately that the program does not violate any specification, and do not generate any formula. This is actually a strong feature of CBMC, however it prevents us from getting a formula $\mathcal{C}$.

A simple solution for the problem above is to add a fake error, "`assert(0);`", at the end of the program. Hence, the generated formula is $\mathcal{C} \wedge \neg\bot$, or simply $\mathcal{C}$.

7.1.3 Formula instrumentation

As we have demonstrated with the example, we need to instrument the formula generated by CBMC with a set of Boolean variables V_I.

In the SSA form, a variable is renamed when it is reassigned. For example, the output `O` is named O_0 when initialised. When it is assigned, it is renamed to O_1, and so on. The index is incremented, and `O` keeps the final value after final assignment. Therefore, we build a simple parser to locate the variable O_k renamed from the output `O`, which has the maximal index k.

The declaration of the set of Boolean variables V_I, and their binding with the bits of O_k can be appended to the end of the formula. The whole formula instrumentation procedure is implemented in a simple Java program.

7.2 All-Solution Satisfiability Modulo Theories

The SMT solver MathSAT, from version 4 [80], provides a functionality, called All-SMT, that given a formula φ and a set V_I of *important* Boolean variables, MathSAT in All-SMT mode computes all models of φ with respect to the set V_I.

In this thesis, we extend the All-SMT problem with a set V_R of *relevant*, possibly non-Boolean, variables. The extended All-SMT(φ, V_I, V_R) problem is to compute all models of φ with respect to the set V_I and the models include value assignments for variables in V_R. We show how this All-SMT problem can be used to analyse the correctness, reliability and security of programs:

- *Bounded Model Checking* [68]: SMT-based Bounded Model Checking can only return a single error trace, the user has to fix the error, then runs the model checker again for other error traces. This is because SMT solvers can return only one model. Combining Bounded Model Checking with an All-SMT solver, we can compute multiple counterexamples in one run of the model checker.
- *Automated Test Generation*: an All-SMT solver can be combined with either a Symbolic Executor or a Bounded Model Checker for test input generation. Although traditional Symbolic Execution with an SMT solver is already capable of generating test inputs, it needs to make hundreds or thousands of calls to the SMT solver. In our approach, the Symbolic Execution tool needs to make only one call to the All-SMT solver for any program.
- *Reliability analysis*: we can build a reliability analysis tool by combining an All-SMT solver with a Symbolic Executor to enumerate all path conditions of the program, then use the Barvinok model counting technique [119] to compute the number of inputs that go into each symbolic path. In this way, we can compute the reliability of the program, i.e. the probability that the program successfully accomplishes its task without errors.

7.2.1 Multiple-counterexamples for BMC

Recall that (section 2.4.1), Bounded Model Checking (BMC) transforms the program and its specification into a formula, then solving the resulting formula with a SAT or SMT solver.

Since a SAT or SMT solver can only return a single model, state-of-the-art SAT-based or SMT-based Bounded Model Checker can only return a single error trace per run. The user has to fix the error and run the model checker again to find more error traces. On the other hand, All-SMT solver can return all models w.r.t. a set of Boolean variables, it can be exploited to find multiple counterexamples for BMC.

```
x=x+y;
if(x!=1){
  x=2;
  if(z) x++;
  assert(y > 1);
}
assert(x ≤ 3);
```

→

```
x_1=x_0+y_0;
if(x_1!=1){
  x_2=2;
  if(z_0) x_3=x_2+1;
  assert(y_0 > 1);
}
assert(x_3 ≤ 3);
```

→

$$\begin{aligned} \mathcal{C} := {} & x_1 = x_0 + y_0 \land \\ & \quad x_2 = ((x_1 \neq 1)?2 : x_1) \land \\ & \quad x_3 = ((x_1 \neq 1 \land z_0)?x_2 + 1 : x_2) \\ \mathcal{P} := {} & (x_1 \neq 1 \rightarrow y_0 > 1) \land (x_3 \leq 3) \end{aligned}$$

Figure 7.3: Example, modified from [67]: the program is transformed into Static Single Assignment form, and then encoded into a logical formula

To illustrate our approach, we reconsider the example in Figure 7.3 from [67], which was used to illustrate CBMC [48]. We made a small modification by adding the assertion `(y > 1)`, so that the program P contains more than one error. At the first step, the program is transformed into SSA form.

As shown in Figure 7.3, applying $\mathcal{C}(\texttt{P}, g)$ and $\mathcal{P}(\texttt{P}, g)$ on the SSA program results in the set of guards $(x_1 \neq 1)$ and $(z_0 \neq 0)$. We denote Boolean variable g_1 and g_2 such that $g_1 := \mathcal{BA}(x_1 \neq 1)$ and $g_2 := \mathcal{BA}(z_0 \neq 0)$.

A model of the formula $\mathcal{C} \land \neg\mathcal{P}$ will correspond to a trace of the program that violates the specification $\mathcal{P}$. By asking an All-SMT solver to return all models of $\varphi = \mathcal{C} \land \neg\mathcal{P}$ with respect to the set of Boolean abstractions of variables in the guards, i.e. $V_I = \{g_1, g_2\}$, we can get a set of all models, each one corresponds to an error trace.

Since a single trace represents a set of concrete executions, to get the inputs for just one representative concrete execution that triggers the error, we set them as the relevant variables to be included in the models, which means $V_R = \{x_0, y_0, z_0\}$.

In this case hence V_I and V_R represents the guards and the inputs of the program respectively.

7.2.2 Automated Test Generation

This section shows how an All-SMT solver can be used in two different approaches for Automated Test Generation (ATG), namely Bounded Model Checking and Symbolic Execution.

ATG using Bounded Model Checking

We use the same trick as in the previous section. The goal is to compute all models of a formula, each one corresponds to a program trace in the program. Take an example as in Figure 7.4. Different from the one in Figure 7.3, the program contains no error, thus it will go through CBMC without any solver being called. CBMC will not generate a formula either.

```
void foo(int x, int y){
  if(x > 5){
    x++;
    if (x < 3)
      x--;
    else
      y = x + 1;
  }
  return;
}
```

$$\begin{array}{ll}
(g_1 = x_1 > 5) & \wedge \\
(x_2 = 1 + x_1) & \wedge \\
(g_2 = x_2 < 3) & \wedge \\
(x_3 = -1 + x_2) & \wedge \\
(x_4 = x_2) & \wedge \\
(y_2 = 1 + x_4) & \wedge \\
(x_5 = g_2 ? x_3 : x_4) & \wedge \\
(y_3 = g_2 ? y_1 : y_2) & \wedge \\
(x_6 = \neg g_1 ? x_1 : x_5) & \wedge \\
(y_4 = \neg g_1 ? y_1 : y_3) &
\end{array}$$

Figure 7.4: A simple program encoded into a formula

In order to generate test inputs that cover all program paths (to a given bound), similar to the previous section, we append "`assert (0);`" as a fake error at the end of the program. Since this error is reachable by all program paths, CBMC will include all the paths into the formula.

The box in the right in Figure 7.4 shows the formula encoded by CBMC. We run the All-SMT solver on the formula with $V_I = \{g_1, g_2\}$, $V_R = \{x_1, y_1\}$. The All-SMT solver will return a set of solutions, each one contains value assignments for x_1 and y_1, which can be used as test inputs for the function `foo`.

ATG using Symbolic Execution

Recall that Symbolic Execution (SE) executes programs on unspecified inputs, by using symbolic inputs instead of concrete data. For each executed program path, a path condition *pc* is built which represents the condition on the inputs for the execution to follow that path, according to the branching conditions in the code. In classical SE, the satisfiability of the path condition is checked at every branching point, using off-the-shelf solvers. In this way only feasible program paths are explored. Test generation is performed by solving the path conditions.

Here we propose another approach for SE using an All-SMT solver. We use SE with the *constraint solver turned off*, to compute the set of all possible program paths: $pc_1, pc_2, \ldots pc_M$. Since there is no constraint solving, a pc_i can be infeasible. Hence, the program under test can be viewed as corresponding to the following formula:

$$\varphi := pc_1 \vee pc_2 \cdots \vee pc_M$$

To illustrate, let us consider again the program in Figure 7.4. The program can be viewed as corresponding to the following formula:

$$\begin{aligned} \varphi \;:=\; & ((x > 5) \wedge (x + 1 < 3)) \quad \vee \\ & ((x > 5) \wedge \neg(x + 1 < 3)) \quad \vee \\ & \neg(x > 5) \end{aligned} \tag{7.1}$$

Notice that the path $(x > 5) \wedge (x + 1 < 3)$ is infeasible, but it is still included in the formula, since we do not check the constraint at each branching point. Applying Boolean abstraction on φ leads to:

$$\mathcal{BA} := ((C_1 = (x > 5)) \wedge (C_2 = (x + 1 < 3)))$$

We use an All-SMT solver on $\varphi \wedge \mathcal{BA}$ with $V_I = \{C_1, C_2\}$ and $V_R = \{x, y\}$. The set of models returned by the All-SMT solver is the set of feasible paths, and the value assignments for relevant variables in V_R can be used as test inputs for the program.

For ATG, in either the case of using Symbolic Execution or using Bounded Model Checking, V_I represents the guards of the program and V_R holds the test vector to be generated.

7.2.3 Reliability analysis

This section introduces an alternative implementation for the approach in [118] by using our All-SMT-based SE instead of classical SE. The improvement is that we only need to make one call to the All-SMT solver to explore all feasible paths.

```
void foo(int x, y){
  if(x > 5){
    x++;
    if (x < 3)
      x--;
    else {
      y = x + 1;
      assert false;
    }
  }
  return;
}
```

Let us consider again the previous example with only one difference: we add an error for the path $x \geq 3$. Similar to the previous section, we use SE with the constraint solver turned off, to encode the program into a logical formula φ as in (7.1). Moreover, the two paths $((x > 5) \wedge (x+1 < 3))$ and $\neg(x > 5)$ are labelled with **T** as the program finishes normally in these two paths. On the other hand, the path $((x > 5) \wedge \neg(x+1 < 3))$ is labelled with **F** since an error is reachable in this path.

Similar to the previous section, using an All-SMT solver on $\varphi \wedge \mathcal{BA}$ with $V_I = \{C_1, C_2\}$ and $V_R = \{x, y\}$ will eliminate the infeasible path $((x > 5) \wedge (x + 1 < 3))$. We then can use the Latte tool to count the models for each paths, and compute the reliability of the program.

7.3 Algorithms for #SMT and All-SMT solver

Obviously, there is no off-the-shelf solver for our new problem #SMT. The closet to a #SMT solver is the functionality All-SMT of MathSAT. However, MathSAT does not support model generation for relevant variables in All-SMT mode. Moreover, when using MathSAT for our analysis, MathSAT returns incorrect number of models in several benchmarks (we will show later in section 7.4). For these reasons, we have developed a lightweight approach to implement an All-SMT/#SMT front-end for

SMT solvers. We build our algorithms from a number of API functions provided by the SMT solver, which we list below.

API function	Description
Assert(φ)	Assert formula φ into the solver.
Check()	Check consistency of all assertions.
Model()	Get model of the last **Check**.
Eval(`t`)	Evaluate expression `t` in current model.
Push()	Create a backtracking point.
Pop(`n`$=1$)	Backtracks `n` backtracking points.

A key feature of SMT solvers for our algorithms is that of being *incremental* and *backtrackable*. The following example shows a sequence of API calls and their effects to the solver.

Assert(φ_1); **Check**();	φ_1	$\Rightarrow$ `SAT`
Push();	φ_1	
Assert(φ_2); **Check**();	$\varphi_1 \wedge \varphi_2$	$\Rightarrow$ `SAT`
Push();	$\varphi_1 \wedge \varphi_2$	
Assert(φ_3); **Check**();	$\varphi_1 \wedge \varphi_2 \wedge \varphi_3$	$\Rightarrow$ `UNSAT`
Pop(2);	φ_1	
Assert(φ_4); **Check**();	$\varphi_1 \wedge \varphi_4$	$\Rightarrow$ `SAT`

It is possible for an incremental SMT solver to add additional assertions to the original formula. Moreover, when **Check** is being called several times, the solver can remember its computation from one call to the other. Thus, when being called to check $\varphi_1 \wedge \varphi_2$ after checking φ_1, it avoids restarting the computation from scratch by restarting the computation from the previous status. Backtrackable means that the solver is able to undo steps, using **Push** and **Pop**, and returns to a previous status on the stack in an efficient manner.

Both Z3 [79] and MathSAT [80] provide similar API functions to interact with the solver in incremental mode. Beside these functions, we develop a function `filter`(`m`, V_I, V_R) such that: given a model of the formula φ, a set of important Boolean variable V_I, and the set of relevant variables V_R, the function will return a subset $\mathtt{m}_{ir}$ of `m` that only contains literals from V_I and V_R. This function will be used in both algorithms.

7.3.1 Blocking clauses method

A straightforward approach for #SMT is to add clauses that prevent the solver from finding the same solution again.

```
function ALL-BC($\varphi, V_I, V_R$) {
    N $\leftarrow$ 0; $\Psi \leftarrow \epsilon$;
    Assert($\varphi$);
    while (Check() = SAT) {
        N $\leftarrow$ N + 1;
        m $\leftarrow$ Model($\varphi$);
        m$_{ir}$ $\leftarrow$ filter(m, $V_I, V_R$);
        $\Psi \leftarrow \Psi \cup \{$m$_{ir}\}$;
        block $\leftarrow$ FALSE;
        for all $p_i \in V_I$ do {
            block $\leftarrow$ block $\vee$ ($p_i \neq$ Eval($p_i$));
        }
        Assert (block);
    }
    return N, $\Psi$;
}
```

Figure 7.5: Blocking clauses #SMT

The pseudo-code for the blocking clauses method is shown in Figure 7.5. Every time the solver discovers a solution m of φ such that m $= l_0 \wedge l_1 \wedge \cdots \wedge l_n \wedge \ldots$, in which only $l_0, l_1 \ldots l_n$ are literals of $p_0, p_1 \ldots p_n$ in V_I. The negation of $l_1 \wedge l_2 \wedge \cdots \wedge l_n$ would be, by De Morgan's law, as follows:

$$\texttt{block} = \neg l_0 \vee \neg l_1 \vee \cdots \vee \neg l_n$$

A literal l_i in the model can be viewed as a mapping p_i to {TRUE, FALSE}, thus the negation $\neg l_i$ is $p_i \neq$ **Eval**(p_i). By adding this clause to the formula, by **Assert**(block), a solution with $l_0 \wedge l_1 \wedge \cdots \wedge l_n$ will not be discovered again. This procedure repeats until no other solution is found. At that point, we have enumerated all the solutions of φ with respect to V_I. All solutions are stored in Ψ, and N $= |\Psi|$ is the result for the corresponding #SMT problem.

The blocking clauses method is straightforward and simple to implement. However, adding a large number of blocking clauses will require a large amount of memory. Moreover, increasing the number of clauses also means that the *Boolean Constraint Propagation* procedure is slowed down. Despite these inefficiencies, this method can be used as a test *oracle* to determine the correctness of other techniques.

7.3.2 Depth-first search

To address the inefficiencies caused by adding a large number of clauses, we introduce an alternative method which avoids re-discovering solutions using depth-first search (DFS).

We divide the set of variables of φ into two sets: V_I is the set of important Boolean variables, and V_U is the set of unimportant, possibly non-Boolean, variables ($V_R \subseteq V_U$). Hence, with some abuse of notation the formula φ can be viewed as the function $f_\varphi(V_I, V_U)$ with codomain $\mathbb{B}$.

Our #SMT procedure is the integration of two components: the first component is a simple SAT solver to enumerate all possible partial truth assignments μ_I of V_I; the second component is the SMT solver to check the consistency of $\varphi \wedge \mu_I$.

```
function ALL-DFS($\varphi, V_I, V_R$) {
    N $\leftarrow$ 0; $\Psi \leftarrow \epsilon$;
    Assert($\varphi$);
    if (Check() $\neq$ SAT)  return N, $\Psi$;
    depth $\leftarrow$ 0; finished $\leftarrow$ FALSE;
    while (finished = FALSE) {
        $l \leftarrow$ choose_literal($V_I$);
        Push();
        Assert($l$); depth $\leftarrow$ depth + 1;
        if (Check() = SAT)  {
            if (depth = $|V_I|$)  {
                N $\leftarrow$ N + 1;
                m $\leftarrow$ Model($\varphi$);
                m$_{ir}$ $\leftarrow$ filter(m, $V_I, V_R$);
                $\Psi \leftarrow \Psi \cup \{$m$_{ir}\}$;
                backtrack();
            } }
        else backtrack();
    }
    return N, $\Psi$;
}
```

Figure 7.6: Depth-first search #SMT

The pseudo-code for DFS-based #SMT is depicted in Figure 7.6. The method `choose_literal` chooses the next state to explore from V_I in a DFS manner, and the variable `depth` keeps the number of important variables that have been chosen. That means, `choose_literal` will select a literal V_I[`depth`] or $\neg V_I$[`depth`]. This literal is then "*pushed*" to the formula as a unit clause. Recall that the plain DPLL algorithm [64] is a depth-first search combining with the BCP procedure. Here we do not perform BCP, however by adding all literals of μ_I as unit clauses to φ, we force the SMT solver to perform BCP on those literals.

When all important variables has been assigned a truth value, i.e. `depth` $= |V_I|$, and the formula in the solver is consistent, then the search has found a model. It then *backtracks* to find another one. The method `backtrack` implements a simple chronological backtracking, it "*pops*" the unit clauses and sets the variable `finished` to `TRUE` when all states are explored. It is also called when the formula in the solver is inconsistent.

Compared to the blocking clauses method, the DFS-based method is much more efficient in term of memory usage. The reason is that the blocking clauses method needs to add N blocking clauses to find all models while the DFS adds maximum $|V_I|$ of unit clauses. This memory efficiency leads to timing efficiency when there are a large number of models.

7.3.3 Implementation

We have implemented both of the algorithms discussed above in a prototype tool, called **aZ3**. The tool is built in Java, using the API functions provided by the SMT solver Z3 [79]. aZ3 supports standard SMT-LIB v2 with two additional commands: the first one is `check-allsat`, also supported by MathSAT, to specify the list of important variables; and the second one is `allsat-relevant` to specify the list of relevant variables.

We have also implemented a QIF analyzer, called **sqifc++**, which uses CBMC 4.7 to encode a program into a formula, then invoking aZ3 to compute channel capacity.

7.4 Evaluation

We conduct two experiments: the first one is to compare our #SMT solver aZ3 against MathSAT 5.2.11 modified for #SMT; the second one is to compare the QIF approach using a #SMT solver with the one in chapter 3, which uses formal methods.

The benchmarks, the aZ3 solver, and the wrapper of MathSAT for #SMT can be found at: `https://github.com/qsphan/aZ3`.

7.4.1 Evaluation of #SMT solvers

Benchmark		Expected	**MathSAT 5**		**aZ3**	
		N	N	Time	**BC** time	**DFS** time
`QF_LIA`	Example in Figure 7.3	2	2	0.007	0.021	0.013
	Example in Figure 7.4	3	3	0.005	0.008	0.007
	Flap controller [118]	5	5	0.031	0.020	0.012
	Red-black tree [123]	31	31	0.016	0.054	0.073
	Bubble sort [97]	541	541	0.136	1.850	2.069
	Array false [97]	1370	1370	0.037	3.008	2.650
	Sum array false [97]	1024	1024	0.026	0.899	0.792
	Linear search false [97]	1024	1024	0.028	0.899	0.604
`QF_AUFBV`	Data sanitization [18]	16	16	0.008	0.035	0.086
	Implicit flow [18]	7	7	0.012	0.029	0.049
	Population count [18]	33	**71**	0.012	0.074	0.398
	Mix and duplicate [18]	65536	**162087**	4.648	-	136.947
	Masked copy [18]	65536	65536	1.319	-	18.630
	Sum query [18]	28	**64**	0.010	0.055	0.133
	Ten random outputs [18]	10	10	0.014	0.038	0.093
	CRC (8) [5]	8	**12**	0.018	0.041	0.099
	CRC (32) [5]	32	**36**	0.019	0.075	0.325

Figure 7.7: Comparing aZ3 against MathSAT. N is the number of models. **BC** time and **DFS** time are the time of aZ3 using the blocking clauses method and depth-first search-based method respectively. Times are in seconds. "-" means "timed out in 1 hour". Notice that for both aZ3 implementations the number of models is Expected N.

In order to evaluate aZ3 and MathSAT, we create two sets of benchmarks. The first group of benchmarks are formulas in QF_LIA (integer linear arithmetic) [96]. These benchmarks are used to evaluate All-SMT solvers in the context of test input generation. The formulas are generated using Symbolic PathFinder [81] (SPF). SPF has a parameter, `symbolic.dp`, to set the constraint solver for it. If this parameter set to `no_solver`, the tool will run without constraint solving.

The architecture of SPF enables us to attach a "listener" to it. When SPF executes a program, the listener collects the path conditions, and outputs them to a QF_LIA formula. The models of the integer variables can be used as test inputs for the original programs.

The second group of benchmarks that we considered are formulas in QF_AUFBV [96] (bit vector with array). The source code of these benchmarks are case studies in the QIF literature, mostly collected in [18]. We use CBMC with the option −−`smt2` to transform the programs into QF_AUFBV formulas. These resulting formulas are then instrumented into #SMT problems. There are no relevant variables in these benchmarks.

Discussion of evaluation

Figure 7.7 summaries our experiments with the two solvers aZ3 and MathSAT 5.2.11 on the benchmarks. In order to compare with MathSAT, we commented out the relevant variables in the QF_LIA benchmarks. As shown in the figure MathSAT is faster than aZ3. This is not surprising, since we built the tool from the front-end, while the All-SMT functionality of MathSAT is built from the back-end, making use of the internal data structure.

However, MathSAT returns incorrect models in several benchmarks[1]. Especially, in the benchmark "Mix and duplicate" MathSAT is significantly faster than aZ3, but it is also extremely imprecise at the same time. Note that benchmarks in QF_AUFBV are derived from the QIF literature, and their numbers of models were already reported in other papers. For example "Mix and duplicate" was reported in [17] and [18] to have 2^{16} models.

The blocking clauses method is comparable, or even faster than the DFS-based method when the number of models is small. However, for the benchmarks with 2^{16} models, adding 2^{16} blocking clauses is obviously not efficient in both time and memory. As a result, the method failed to provide the answer for such benchmarks. On the other hand, the DFS-based method was still able to provide the answer in a reasonable time.

7.4.2 Evaluation of QIF analysers

Figure 7.8 compares the performance of sqifc++ against the tool sqifc in chapter 3. The results show that sqifc++ is much more efficient. The reason is that, sqifc makes several calls to CBMC, and for each call CBMC has to transform the program into a formula, then calling a SAT/SMT solver to check the formula. On the other hand, sqifc++ transforms the program only once, and its search is also more efficient.

[1] We reported these errors to MathSAT developers, and in the latest version of MathSAT the All-SMT functionality is disabled if the input is a bit-vector formula.

Benchmark	Leaks	**sqifc** time	**sqifc**++ time **CBMC** time		
			CBMC time	**aZ3** time	Total
Data sanitization [18]	4	11.898	0.165	0.086	0.251
Implicit flow [18]	2.81	5.033	0.169	0.049	0.218
Population count [18]	5.04	17.278	0.162	0.398	0.560
Mix and duplicate [18]	16	-	0.154	136.947	137.101
Masked copy [18]	16	-	0.175	18.630	18.805
Sum query [18]	4.81	64.557	0.162	0.133	0.295
Ten random outputs [18]	3.32	64.202	0.160	0.093	0.253
CRC (8) [5]	3	2.551	0.184	0.099	0.283
CRC (32) [5]	5	7.755	0.193	0.325	0.518

Figure 7.8: Comparing the new approach with the sqifc tool in chapter 3. Leaks are in bits. aZ3 runs with the DFS-based algorithm. Times are in seconds, "-" means timeout in one hour. Total time of sqifc++ is the sum of CBMC time and aZ3 time.

However, sqifc++ relies on CBMC for program transformation, and this functionality of CBMC (−−`smt2`) is still experimental. We found that CBMC generates incorrect formulas for several case studies in chapter 3, therefore we could not use sqifc++ to analyse, for example, the dining cryptos case study.

7.5 Discussion of related work

7.5.1 Quantitative Information Flow

We have just compared the two prototype tools sqifc++ and sqifc in the previous section. Moreover, in chapter 3, we already compared our #SMT-based approach with other work in QIF literature, so we do not repeat the discussion here.

7.5.2 Multiple-counterexamples for BMC

The most relevant work to ours is that of Bhargavan et al. [124] embodied in the Verisim testing tool for network protocols. When an error trace is found to violate the specification, which is an extended LTL formula ϕ, Verisim uses a technique, called *tuning*, to replace ϕ with φ that ignores the violation. Tuning is not fully automatic.

Another technique introduced by Ball et al. [125] is embodied in the SLAM tool-kit. The algorithm uses a model checker as a sub-routine. When the model checker finds an error trace, SLAM localizes the error cause, modifying the source code with a `halt` statement at the error cause. The model checker is then invoked again, and the `halt` statements instruct the model checker to stop exploring paths at the previously found error causes. This procedure is very expensive, it requires comparing the error trace with all correct traces to localize the error, and requires to run the model checker several times. Our work is much simpler, and faster but there is a possibility that multiple error traces come from one cause.

7.5.3 Automated Test Generation

The closest to our work is FShell [126], which also uses CBMC for automated test generation. FShell transforms the program under test into a CNF formula, and solves it using an incremental SAT solver. Every time the SAT solver finds a solution representing a symbolic path, FShell adds a blocking clause to prevent that path from being explored again. As our experiments have shown, the blocking clauses method is suffered from rapid space growth.

Classical Symbolic Execution also uses SMT solvers to check the satisfiability of path condition. In this approach, the SMT solver is called whenever a conditional statement is executed, hence it may be called hundreds or thousands of times. In our approach, the symbolic executor makes only one call to the All-SMT solver.

7.5.4 Reliability analysis

Our approach to reliability analysis is based on the paper of Filieri et al. [118] that uses classical Symbolic Execution and Barvinok model counting tool. We extend the approach using our new All-SMT-based Symbolic Execution instead of classical Symbolic Execution. The main difference is the same as in the case of test generation, our approach only makes one call to the All-SMT solver in the whole analysis.

7.5.5 All-Solution SAT Modulo Theories

As we have discussed throughout the chapter, MathSAT is the only SMT solver that supports All-SMT. Its algorithm is briefly described in [127], and it was used to compute predicate abstraction in [128] and [129]. Our experimental results show that MathSAT is imprecise in several benchmarks. However, we also notice that the imprecisions seem to be limited in QF_AUFBV benchmarks, while the authors performed experiments with QF_LRA formulas in [128].

Also in the context of predicate abstraction, Lahiri et al. [130] have proposed several techniques, which use the SMT solver Barcelogic to generate the set of all satisfying assignments over a set of predicates. However, we are not able to include Barcelogic in our experiments, since the solver provided to us by the author does not support All-SMT.

A principal difference between the work mentioned above and the one in this chapter is that we implemented from the front-end of an SMT solver. For this reason, our implementation is slower than MathSAT. On the good side, our approach is applicable to implement even in closed-source SMT solvers that do not support All-SMT but provide similar API functions.

Chapter 8

Conclusions

8.1 Summary

This thesis introduces a new research problem, Model Counting Modulo Theories or #SMT, and presents a #SMT-based approach to quantification of information leaks. Although our implementations are far from being optimised, they drastically outperform the existing technique based on self-composition, e.g. reducing the time of analysing some programs from Linux kernel from some hours to a few seconds. Our approach is applicable to programs with difficult data structures including pointers, and to Java bytecode.

On the theoretical side, this thesis makes the original contributions by discovering the relations among different research areas: (i) it casts the QIF problem into the #SMT problem; (ii) it shows the correspondence between Symbolic Execution and the DPLL($\mathcal{T}$) algorithm; (iii) it explores the relation between Symbolic Execution and Bounded Model Checking; (iv) finally, it exploits the connection between QIF analysis and Reliability analysis.

On the application side, this thesis is the first to use Symbolic Execution to quantify information leaks. It is also the first to use classical Symbolic Execution for Bounded Model Checking. Beside, it proposes the use of an All-SMT solver for multiple-counterexamples in Bounded Model Checking, for automated test generation, and for reliability analysis.

On the practical side, this thesis has developed several tools in both C/C++ and Java: sqifc, jpf-qif and QILURA, for the quantification of information leaks in software. It also demonstrated the use of these tools to analyse vulnerabilities from the National Vulnerability Database of the US government, and anonymity protocols.

For the verification community, important contributions of this thesis include the development of JCBMC, a Concurrent Bounded Model Checker for Java, and the All-SMT solver aZ3.

8.2 Future Research

In the previous chapter, we have proposed the use of an All-SMT solver for multiple-counterexamples in Bounded Model Checking, for automated test generation and for reliability analysis. An immediate direction would be to implement these ideas into automated tools, and to perform experiments on standard benchmarks. Some other possible directions for investigation are the following.

Fault localization

Another interesting avenue of further research would be to the All-SMT solver with CBMC to localize error causes using similar idea in [125]. Models of $\varphi_1 = \mathcal{C} \wedge \mathcal{P}$ correspond to correct traces that satisfy the specification, and models of $\varphi_2 = \mathcal{C} \wedge \neg\mathcal{P}$ correspond to error traces that violate the specification. Using an All-SMT solver, we can compute the sets of all models of φ_1 and φ_2 with respect to the set of guards. Comparing the two sets of models, we can localize the transitions that only appear in error traces.

Concurrent Bounded Model Checking

A first improvement on this direction would be to upgrade its concurrency from single CPU multi-threading to true parallelism and to perform obvious optimisations. Another improvement would be to replace Symbolic PathFinder with a lighter weight tool, or a parallel version, to reduce the cost of generating path conditions. It will also be interesting to implement the methodology for other languages (C, Python) and investigate how to use Symbolic Execution for IC3 style verification.

Statistical analysis for QIF

The QILURA tool is still just a prototype. A possible improvement for it would be to use approximate exploration techniques to replace the exact, complete exploration presented in this thesis. In this way, for the tool can be used with increased scalability, but with formal statistical guarantees on the results.

References

[1] Q.-S. Phan and P. Malacaria, “All-Solution Satisfiability Modulo Theories: applications, algorithms and benchmarks,” in *Proceedings of the 2015 Tenth International Conference on Availability, Reliability and Security*, ARES ’15, (Washington, DC, USA), IEEE Computer Society, 2015.

[2] Q.-S. Phan, P. Malacaria, and C. S. Păsăreanu, “Concurrent Bounded Model Checking,” *SIGSOFT Softw. Eng. Notes*, vol. 40, pp. 1–5, Feb. 2015.

[3] Q.-S. Phan, “Symbolic Execution as DPLL Modulo Theories,” in *2014 Imperial College Computing Student Workshop*, vol. 43 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 58–65, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

[4] Q.-S. Phan, P. Malacaria, C. S. Păsăreanu, and M. d’Amorim, “Quantifying Information Leaks Using Reliability Analysis,” in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, (New York, NY, USA), pp. 105–108, ACM, 2014.

[5] Q.-S. Phan and P. Malacaria, “Abstract Model Counting: A Novel Approach for Quantification of Information Leaks,” in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS ’14, (New York, NY, USA), pp. 283–292, ACM, 2014.

[6] Q.-S. Phan, “Self-composition by Symbolic Execution,” in *2013 Imperial College Computing Student Workshop*, vol. 35 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 95–102, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.

[7] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu, “Symbolic Quantitative Information Flow,” *SIGSOFT Softw. Eng. Notes*, vol. 37, pp. 1–5, Nov. 2012.

[8] "Heartbleed bug." `http://heartbleed.com/`.

[9] "August 2014 Celebrity Photo Leaks." `http://en.wikipedia.org/wiki/2014_celebrity_photo_leaks`.

[10] "Sony Pictures Entertainment hack." `http://en.wikipedia.org/wiki/Sony_Pictures_Entertainment_hack`.

[11] R. Sandhu and P. Samarati, "Access control: principle and practice," *Communications Magazine, IEEE*, vol. 32, pp. 40–48, Sept 1994.

[12] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[13] D. Clark, S. Hunt, and P. Malacaria, "A static analysis for quantifying information flow in a simple imperative language," *J. Comput. Secur.*, vol. 15, pp. 321–371, Aug. 2007.

[14] P. Malacaria, "Assessing security threats of looping constructs," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, (New York, NY, USA), pp. 225–235, ACM, 2007.

[15] T. M. Cover and J. A. Thomas, *Elements of information theory.* New York, NY, USA: Wiley-Interscience, 1991.

[16] S. McCamant and M. D. Ernst, "Quantitative information flow as network flow capacity," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, (New York, NY, USA), pp. 193–205, ACM, 2008.

[17] J. Newsome, S. McCamant, and D. Song, "Measuring channel capacity to distinguish undue influence," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, (New York, NY, USA), pp. 73–85, ACM, 2009.

[18] Z. Meng and G. Smith, "Calculating bounds on information leakage using two-bit patterns," in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11, (New York, NY, USA), pp. 1:1–1:12, ACM, 2011.

[19] M. Backes, B. Kopf, and A. Rybalchenko, “Automatic Discovery and Quantification of Information Leaks,” in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP ’09, (Washington, DC, USA), pp. 141–153, IEEE Computer Society, 2009.

[20] J. Heusser and P. Malacaria, “Quantifying information leaks in software,” in *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC ’10, (New York, NY, USA), pp. 261–269, ACM, 2010.

[21] V. Klebanov, “Precise Quantitative Information Flow Analysis Using Symbolic Model Counting,” in *Proceedings, International Workshop on Quantitative Aspects in Security Assurance (QASA)* (F. Martinelli and F. Nielson, eds.), 2012.

[22] D. E. Denning and P. J. Denning, “Certification of programs for secure information flow,” *Commun. ACM*, vol. 20, pp. 504–513, July 1977.

[23] J. A. Goguen and J. Meseguer, “Security Policies and Security Models,” in *IEEE Symposium on Security and Privacy*, pp. 11–20, 1982.

[24] D. Volpano, C. Irvine, and G. Smith, “A sound type system for secure flow analysis,” *J. Comput. Secur.*, vol. 4, pp. 167–187, Jan. 1996.

[25] D. E. Robling Denning, *Cryptography and Data Security.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1982.

[26] J. K. Millen, “Covert channel capacity,” in *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 27-29, 1987*, pp. 60–66, IEEE Computer Society, 1987.

[27] J. McLean, “Security models and information flow,” in *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pp. 180–187, May 1990.

[28] I. Gray, J.W., “Toward a mathematical foundation for information flow security,” in *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on*, pp. 21–34, May 1991.

[29] G. Lowe, “Defining information flow quantity,” *J. Comput. Secur.*, vol. 12, pp. 619–653, May 2004.

[30] A. Di Pierro, C. Hankin, and H. Wiklicky, “Approximate non-interference,” in *Computer Security Foundations Workshop, 2002. Proceedings. 15th IEEE*, pp. 3–17, 2002.

[31] M. Boreale, "Quantifying information leakage in process calculi," *Inf. Comput.*, vol. 207, pp. 699–725, June 2009.

[32] D. Clark, S. Hunt, and P. Malacaria, "Quantitative Analysis of the Leakage of Confidential Data ," *Electronic Notes in Theoretical Computer Science*, vol. 59, no. 3, pp. 238 – 251, 2002. QAPL'01, Quantitative Aspects of Programming Laguages (Satellite Event of {PLI} 2001).

[33] D. Clark, S. Hunt, and P. Malacaria, "Quantified Interference for a While Language," *Electron. Notes Theor. Comput. Sci.*, vol. 112, pp. 149–166, Jan. 2005.

[34] D. Clark, S. Hunt, and P. Malacaria, "Quantitative Information Flow, Relations and Polymorphic Types," *J. Log. and Comput.*, vol. 15, pp. 181–199, Apr. 2005.

[35] G. Smith, "On the Foundations of Quantitative Information Flow," in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS '09, (Berlin, Heidelberg), pp. 288–302, Springer-Verlag, 2009.

[36] P. Malacaria and H. Chen, "Lagrange multipliers and maximum information leakage in different observational models," in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, (New York, NY, USA), pp. 135–146, ACM, 2008.

[37] C. Mu and D. Clark, "Quantitative analysis of secure information flow via probabilistic semantics," in *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, pp. 49–57, March 2009.

[38] C. Mu and D. Clark, "An abstraction quantifying information flow over probabilistic semantics," in *Workshop on Quantitative Aspects of Programming Languages (QAPL)*, pp. 119–141, 2009.

[39] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking.* MIT press, 1999.

[40] A. I. Barvinok, "A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed," *Math. Oper. Res.*, vol. 19, pp. 769–779, Nov. 1994.

[41] "Wolfram Mathematica." `http://www.wolfram.com/mathematica/`.

[42] H. Yasuoka and T. Terauchi, "On bounding problems of quantitative information flow," *Journal of Computer Security*, vol. 19, no. 6, pp. 1029–1082, 2011.

[43] P. Cerny, K. Chatterjee, and T. A. Henzinger, "The Complexity of Quantitative Information Flow Problems," in *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, (Washington, DC, USA), pp. 205–217, IEEE Computer Society, 2011.

[44] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, pp. 69–77, Sept. 2011.

[45] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *J. ACM*, vol. 53, pp. 937–977, Nov. 2006.

[46] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[47] R. C. Cheung, "A User-Oriented Software Reliability Model," *IEEE Trans. Softw. Eng.*, vol. 6, pp. 118–125, Mar. 1980.

[48] E. Clarke, D. Kroening, and F. Lerda, " A Tool for Checking ANSI-C Programs ," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, vol. 2988 of *Lecture Notes in Computer Science*, pp. 168–176, Springer, 2004.

[49] P. Malacaria, "Algebraic foundations for quantitative information flow," *Mathematical Structures in Computer Science*, vol. 25, pp. 404–428, 2 2015.

[50] C. E. Shannon, "A Mathematical Theory of Communication," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, pp. 3–55, Jan. 2001.

[51] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009.

[52] L. de Moura, B. Dutertre, and N. Shankar, "A Tutorial on Satisfiability Modulo Theories," in *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, (Berlin, Heidelberg), pp. 20–36, Springer-Verlag, 2007.

[53] H. B. Enderton, *A mathematical introduction to logic*. Academic Press, 1972.

[54] E. S. Cohen, "Information Transmission in Sequential Programs," in *Foundations of Secure Computation* (R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, eds.), pp. 297–335, Academic Press, 1978.

[55] A. Darvas, R. Hähnle, and D. Sands, "A theorem proving approach to analysis of secure information flow," in *Proceedings of the Second international conference on Security in Pervasive Computing*, SPC'05, (Berlin, Heidelberg), pp. 193–209, Springer-Verlag, 2005.

[56] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure Information Flow by Self-Composition," in *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, CSFW '04, (Washington, DC, USA), pp. 100–, IEEE Computer Society, 2004.

[57] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *Proceedings of the 12th international conference on Static Analysis*, SAS'05, (Berlin, Heidelberg), pp. 352–367, Springer-Verlag, 2005.

[58] W. Feller, *An Introduction to Probability Theory and Its Applications*, vol. 1. Wiley, January 1968.

[59] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, (Berlin, Heidelberg), pp. 564–580, Springer-Verlag, 2012.

[60] V. Klebanov, N. Manthey, and C. Muise, "SAT-Based Analysis and Quantification of Information Flow in Programs," in *Quantitative Evaluation of Systems*, vol. 8054 of *Lecture Notes in Computer Science*, pp. 177–192, Springer Berlin Heidelberg, 2013.

[61] D. A. Plaisted and S. Greenbaum, "A Structure-preserving Clause Form Translation," *J. Symb. Comput.*, vol. 2, pp. 293–304, Sept. 1986.

[62] T. Boy de la Tour, "An Optimality Result for Clause Form Translation," *J. Symb. Comput.*, vol. 14, pp. 283–301, Oct. 1992.

[63] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM*, vol. 7, pp. 201–215, July 1960.

[64] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.

[65] H. Hoos and T. Sttzle, *Stochastic Local Search: Foundations & Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[66] C. Flanagan and J. B. Saxe, "Avoiding exponential explosion: generating compact verification conditions," POPL '01, pp. 193–205, ACM, 2001.

[67] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of C and verilog programs using bounded model checking," in *Proceedings of the 40th annual Design Automation Conference*, DAC '03, (New York, NY, USA), pp. 368–371, ACM, 2003.

[68] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, (London, UK, UK), pp. 193–207, Springer-Verlag, 1999.

[69] "National Vulnerability Database." `http://nvd.nist.gov/`.

[70] E. Birnbaum and E. L. Lozinskii, "The good old Davis-Putnam procedure helps counting models," *J. Artif. Int. Res.*, vol. 10, pp. 457–477, June 1999.

[71] R. J. Bayardo, "RelSat: A Propositional Satisfiability Solver and Model Counter." `http://code.google.com/p/relsat/`.

[72] A. Darwiche, "The c2d Compiler." `http://reasoning.cs.ucla.edu/c2d/`.

[73] "QUAIL." `https://project.inria.fr/quail/`.

[74] F. Biondi, A. Legay, L.-M. Traonouez, and A. Wasowski, "QUAIL: A Quantitative Security Analyzer for Imperative Code," in *Proceedings of the 25th international conference on Computer Aided Verification*, CAV'13, (Berlin, Heidelberg), Springer-Verlag, 2013.

[75] S. Kiefer, A. S. Murawski, J. Ouaknine, B. Wachter, and J. Worrell, "APEX: an analyzer for open probabilistic programs," in *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, (Berlin, Heidelberg), pp. 693–698, Springer-Verlag, 2012.

[76] D. Chaum, "The dining cryptographers problem: unconditional sender and recipient untraceability," *J. Cryptol.*, vol. 1, pp. 65–75, Mar. 1988.

[77] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis," *Automated Software Engineering*, pp. 1–35, 2013.

[78] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Commun. ACM*, vol. 56, pp. 82–90, Feb. 2013.

[79] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[80] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4 SMT Solver," in *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, (Berlin, Heidelberg), pp. 299–303, Springer-Verlag, 2008.

[81] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, (New York, NY, USA), pp. 179–180, ACM, 2010.

[82] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, (Berlin, Heidelberg), pp. 519–531, Springer-Verlag, 2007.

[83] R. Sebastiani, "Lazy Satisfiability Modulo Theories," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 3, pp. 141–224, 2007.

[84] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "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, (New York, NY, USA), pp. 25–35, ACM, 1989.

[85] M. Balliu, M. Dam, and G. L. Guernic, "ENCoVer: Symbolic Exploration for Information Flow Security," in *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, (Washington, DC, USA), pp. 30–44, IEEE Computer Society, 2012.

[86] M. Brain, V. D'Silva, L. Haller, A. Griggio, and D. Kroening, "An Abstract Interpretation of DPLL(T)," in *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pp. 455–475, 2013.

[87] "Java PathFinder." `http://babelfish.arc.nasa.gov/trac/jpf/`.

[88] "Spin." `http://spinroot.com`.

[89] K. L. McMillan, *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[90] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," PLDI '05, pp. 213–223, ACM, 2005.

[91] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated Whitebox Fuzz Testing," in *NDSS*, 2008.

[92] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," ESEC/FSE-13, pp. 263–272, ACM, 2005.

[93] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," OSDI'08, pp. 209–224.

[94] "Z3." `http://z3.codeplex.com/`.

[95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[96] C. Barrett, A. Stump, and C. Tinelli, "The smt-lib standard: Version 2.0," in *SMT Workshop*, 2010.

[97] "Benchmarks of loops in the Software Verification competition 2014." `https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp14/loops/`.

[98] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *STTT*, vol. 11, pp. 69–83, Jan. 2009.

[99] D. Balasubramanian, C. S. Păsăreanu, G. Karsai, and M. R. Lowry, "Polyglot: systematic analysis for multiple statechart formalisms," TACAS'13, pp. 523–529.

[100] E. Ábrahám, T. Schubert, B. Becker, M. Fränzle, and C. Herde, "Parallel SAT solving in bounded model checking," FMICS'06/PDMC'06, pp. 301–315.

[101] S. Wieringa, M. Niemenmaa, and K. Heljanko, "Tarmo: A Framework for Parallelized Bounded Model Checking," in *PDMC*, pp. 62–76, 2009.

[102] T. Kahsai and C. Tinelli, “PKind: A parallel k-induction based model checker,” in *PDMC* (J. Barnat and K. Heljanko, eds.), vol. 72 of *EPTCS*, pp. 55–62, 2011.

[103] J. Barnat, L. Brim, M. Češka, and P. Ročkai, “DiVinE: Parallel Distributed Model Checker (Tool paper),” in *HiBi/PDMC 2010*, pp. 4–7, IEEE, 2010.

[104] R. Palmer and G. Gopalakrishnan, “Partial order reduction assisted parallel modelchecking (full version,” tech. rep., PDMC’2002, 2002.

[105] E. Burns and R. Zhou, “Parallel model checking using abstraction,” SPIN’12, (Berlin, Heidelberg), pp. 172–190, Springer-Verlag, 2012.

[106] G. J. Holzmann and D. Bosnacki, “The Design of a Multicore Extension of the SPIN Model Checker,” *IEEE Trans. Softw. Eng.*, vol. 33, pp. 659–674, Oct. 2007.

[107] U. Stern and D. L. Dill, “Parallelizing the Murphi Verifier,” CAV ’97, (London, UK, UK), pp. 256–278, Springer-Verlag, 1997.

[108] S. Jabbar and S. Edelkamp, “Parallel external directed model checking with linear i/o,” VMCAI’06, (Berlin, Heidelberg), pp. 237–251, Springer-Verlag, 2006.

[109] P. K. Nalla, J. Weiss, J. Ruf, T. Kropf, and W. Rosenstiel, “Parallel Bounded Property Checking with SymC.”

[110] M. Z. Kwiatkowska, A. Lomuscio, and H. Qu, “Parallel Model Checking for Temporal Epistemic Logic,” in *ECAI*, pp. 543–548, 2010.

[111] M. Staats and C. Păsăreanu, “Parallel symbolic execution for structural test generation,” ISSTA ’10, (New York, NY, USA), pp. 183–194, ACM, 2010.

[112] A. King, “Distributed parallel symbolic execution,” in *Master Thesis, Kansas State University*, 2009.

[113] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, “Cloud9: a software testing service,” *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 5–10, Jan. 2010.

[114] J. Siddiqui and S. Khurshid, “ParSym: Parallel symbolic execution,” in *ICSTE*, vol. 1, pp. V1–405–V1–409, 2010.

[115] C. Sinz, W. Blochinger, and W. KÕìchlin, “PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications,” in *SAT*, 2001.

[116] T. Schubert, M. Lewis, and B. Becker, "PaMira - A Parallel SAT Solver with Knowledge Sharing," in *MTV '05*, pp. 29–36, 2005.

[117] C. M. Wintersteiger, Y. Hamadi, and L. Moura, "A Concurrent Portfolio Approach to SMT Solving," in *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, (Berlin, Heidelberg), pp. 715–720, Springer-Verlag, 2009.

[118] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analysis in symbolic pathfinder," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 622–631, IEEE Press, 2013.

[119] "LattE." `http://www.math.ucdavis.edu/~latte/`.

[120] "Omega." `http://www.cs.umd.edu/projects/omega/`.

[121] Z. Meng and G. Smith, "Faster Two-Bit Pattern Analysis of Leakage," in *Proceedings of the 2nd International Workshop on Quantitative Aspects in Security Assurance*, QASA '13, 2013.

[122] D. Milushev, W. Beck, and D. Clarke, "Noninterference via symbolic execution," in *Proceedings of the 14th joint IFIP WG 6.1 international conference and Proceedings of the 32nd IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, (Berlin, Heidelberg), pp. 152–168, Springer-Verlag, 2012.

[123] "Symbolic PathFinder's repository." `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc`.

[124] K. Bhargavan, C. A. Gunter, I. Lee, O. Sokolsky, M. Kim, D. Obradovic, and M. Viswanathan, "Verisim: Formal Analysis of Network Simulations," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 129–145, Feb. 2002.

[125] T. Ball, M. Naik, and S. K. Rajamani, "From Symptom to Cause: Localizing Errors in Counterexample Traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, (New York, NY, USA), pp. 97–105, ACM, 2003.

[126] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement," in *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, (Berlin, Heidelberg), pp. 209–213, Springer-Verlag, 2008.

[127] C. Papadopoulos, A. Cavallo, A. Cimatti, and M. Bozzano, “Installation optimisation.” `http://www.google.com/patents/US20130076767`, Mar. 28 2013. US Patent App. 13/623,977.

[128] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar, “Computing Predicate Abstractions by Integrating BDDs and SMT Solvers,” in *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD ’07, (Washington, DC, USA), pp. 69–76, IEEE Computer Society, 2007.

[129] A. Cimatti, J. Dubrovin, T. A. Junttila, and M. Roveri, “Structure-aware computation of predicate abstraction,” in *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design*, FMCAD ’09, pp. 9–16, IEEE, 2009.

[130] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras, “SMT Techniques for Fast Predicate Abstraction,” in *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV’06, (Berlin, Heidelberg), pp. 424–437, Springer-Verlag, 2006.