Doctoral Dissertation

Towards Practically Applicable Quantitative Information Flow Analysis

Chu Bao Trung

January 23, 2020

Graduate School of Information Science
Nagoya University

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nagoya University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in INFORMATION SCIENCE

Chu Bao Trung

Thesis Committee:

Professor Seki Hiroyuki (Supervisor)
Professor Yuen Shoji
Professor Kaji Yuichi
Associate Professor Nishida Naoki

Abstract

Noninterference was coined by Goguen and Meseguer in 1982 as the property of a program such that any change on secret input values makes no change on publicly observable output values. However, this property is so strict that it disqualifies many useful yet safe programs and protocols such as password authentication programs, voting protocols and so forth. Addressing this strictness, Quantitative Information Flow (QIF) was introduced as a formal notion of the information leakage of a confidential input of a program that can be obtained by analyzing a publicly observable output. A program of which QIF is under a predefined threshold is considered to be safe even if the program does not have the noninterference property.

Intuitively, QIF of a program is defined as the difference between the *initial uncertainty* and the *remaining uncertainty*, which are the uncertainties about confidential input values before and after executing that program, respectively. There are several formal definitions which elaborate on that intuition. Whilst Shannon entropy and Min-entropy based notions use information theoretic entropies, Belief-based notion leverages Kullback-Leibler divergence to model the uncertainties. These QIF definitions and the traditional methods of computing QIF are reviewed in Chapter 2. However, all of the notions fail to represent reasonable quantitites of the leakage when there are some output values, which reveal much more information about the confidential input than others, because those definitions take the average leakage on all possible output values.

The contribution of this doctoral thesis includes:

- a framework consists of recognizable, algebraic and tree power series that helps answer: *Is there any method of counting models that directly analyzes variables with complex structures as what they are: strings, arrays, linked lists, trees and so on?*.

- an attempt to answer: *"How can we address reasonably and naturally the leakage of secure information when executing programs through a specific output value?"*, by proposing two new notions for dynamic leakage, the dynamic version of QIF.

- a method to accelerate the calculation of dynamic leakage in "divide and conquer" fashion by answering: *"How to calculate the dynamic leakage in executing a program P when the dynamic leakages of the sub-programs of P are known?"*.

Addressing the lacking of calculation model for the information leakage when observing a specific output value, in the first part of Chapter 3, we propose two notions called $\mathbf{QIF_1}$ and $\mathbf{QIF_2}$. There are cases in which QIF_2 gives more reasonable quantity than QIF_1 while calculating QIF_1 is generally easier than QIF_2. Both of QIF_1 and QIF_2 are notions of dynamic leakage metric while the ordinary QIF is a static one which is independent of a specific output value. We define the problems of computing QIF_1 and QIF_2 for Boolean programs and investigate the computational complexity of these problems. Even for *deterministic loop-free* programs, where QIF_1 coincides with QIF_2, the problem is proved to be $\sharp$P-hard, meaning that the problem is not easier than the $\sharp$SAT problem, which is the counting version of the well-known SAT problem. This implies that, though many off-the-shelf model counting tools are available, there is a need to speed up calculation of dynamic leakage. We conducted experiments on 14 toy programs of less than 100 lines of code when utilizing different model counting tools. Those programs are often used as benchmarks for evaluating a QIF computation method. All of the experiments were carried out under the assumption that the program under analysis (PUA) is deterministic and has uniformly distributed input. Those experiments are to reaffirm the model counting-based calculation, but the method does not scale up well.

In the second part of Chapter 3, we present an attempt to improve the performance of quantifying dynamic leakage. The idea is to decompose a given PUA to sub-programs so that leakage of the original program is calculated from the sub-programs. We propose two such types: *value domain-based composition* and *sequential composition*. The former is suitable for utilizing parallel computing to speed up calculation of dynamic leakage. *Value domain-based*

composition, however, does not help much when the PUA is too big to analyze as a whole, or is constituted from distributed parts, because the sizes and complexities of value domain-based decomposed sub-programs are almost the same to the PUA. In such cases, sequential composition helps by making much simpler sub-programs of smaller sizes than the original PUA. Thus, those two types of composition can also be combined to produce a better performance. For example, we first horizontally decompose the PUA into sub-programs using the sequential composition, and then vertically decompose the value domains of sub-programs. We experimented three simple benchmarks to confirm the feasibility of our methods. It is shown that parallel computing accelerates the model counting 25 times as fast as single-threaded computation.

Although dynamic leakage is more precise than the ordinary QIF in the sense that it is based on a specific output value, when we have to decide a program as safe or not before running it, dynamic leakage faces difficulties to analyze. Hence, there is a need to accelerate the calculation of the ordinary QIF. **For this problem, we attempt to speed up the model counting of string constraints which contributes to QIF analysis for programs with strings.** Existing string counting methods use finite automata to represent string constraints and the precision of approximated counting are low for recursive constraints. The proposed method uses Context-Free Grammars (CFGs) to represent string constraints. Similarly, constraints on the shape of trees can be represented using recognizable tree series. An experiment was conducted to compare the performance of the proposed method with ABC, the state-of-the-art automaton-based string counting tool, on 17,544 files of Kaluza benchmark. Our prototype provided the same answers to ABC on 17,183 files, different but exact or better answers on other 370 files, where the execution time of the prototype and ABC are almost the same on all the cases. These experimental results show the effectiveness of the proposed method of QIF calculation based on CFGs.

Keywords:

quantitative information flow, dynamic leakage, string constraint, model counting, compositional reasoning, formal language theory, information theory

Contents

List of Figures

List of Tables

List of Algorithms

List of Publications

Peer-reviewed Journal Papers

[1] Bao Trung Chu, Kenji Hashimoto, Hiroyuki Seki: *Counting algorithms for recognizable and algebraic series.* In: IEICE Transactions on Information and Systems, E101-D(6), June 2018, pp. 1479–1490. (Chapter 4)

[2] Bao Trung Chu, Kenji Hashimoto, Hiroyuki Seki: *Quantifying dynamic leakage: complexity analysis and model counting-based calculation.* In: IEICE Transactions on Information and Systems, E102-D(10), October 2019, pp. 1952–1965. (Sections 3.1-3.4)

Peer-reviewed International Conference

[1] Bao Trung Chu, Kenji Hashimoto, Hiroyuki Seki: *On the compositionality of dynamic leakage and its application to the quantification problem.* In: Proceedings of the 13th International Conference on Emerging Security Information, Systems and Technologies, SECURWARE 2019, pp. 1–8. (Section 3.5)

Acknowledgements

First of all, I greatly appreciate and would like to send my sincere thanks to my supervisor, professor Seki Hiroyuki. In 2011, after finishing my undergraduate study, I was selected by professor Seki after a round of interview. This chance opened my page in Japan. Upon my Masters graduation from Nara Institute of Science and Technology in early 2014, I moved to Tokyo to work at a company. Two years later, again thanks to Seki-sensei, I was at Nagoya University to pursue this doctoral course. Four more years have come and now I'm finishing my highest study degree. Most of my time in Japan until now has been supported by professor Seki. Not only is Seki-sensei my supervisor, but he is also as my father who figuratively gave birth to me here in Japan.

In almost the same duration with Seki-sensei, I used to work with Hashimoto-sensei, so we have worked as a team of three. I deeply appreciate and treasure Hashimoto-sensei's enormous support. While, usually, Seki-sensei gives me broad overviews and directions, Hashimoto-sensei gives me critical comments with his sharpen thinking. Normally, Hashimoto-sensei is not so talkative but when it comes to discuss about a research problem, he fully switches the mode. Without his enthusiastic guidance, I should not have reached this far in my academic career.

When I was considering to quit my job for continuing my study, what concerned me the most was money. I did not have any resource that allows me to study in several years without financial problems. When I was admitted to the Graduate Program for Real-World Data Circulation Leaders (RWDC), this concern was resolved. With RWDC's fully supported stipend, I had valuable full three years to concentrate on my research. Besides financial support, to be accepted as an RWDC member broadens my perspective with the new research field called Real-World Data Circulation. I took courses about data processing, analyzing and implementing, learned from top leaders in both academia and industry, interned at world-leading research centers: Denso ADAS and IBM Cyber Security Center of Excellence. I deeply recognize and thank all professors, staffs and students in RWDC for their invaluable guidance and support.

When I started writing this dissertation, I had no experience to write such a

long “book”. My first draft to send to the thesis committee was terrible with full of mistakes. I received a lot of indispensable comments from sensei-gata, so that I can write a much better dissertation both in terms of presentation and logical explanation. Therefore I would like to send my thanks with all sincerity to the professors in my thesis committee.

Last but not least, I would like to thank my grand father, parents, sisters, wife and son, who travel together with me and always unconditionally support my decisions in this long life. I believe my beloved father, who passed away before I came to Japan, will find proud of me from a far place. I’m deeply aware of my wife’s sacrifice when she agrees to go with me and support my academic career. My son came in my third year of the course when I struggled hard to fulfill the graduation requirements. He has been my spiritual “doping” in the difficult time.

1 Introduction

1.1 Motivations

After several decades of digitalization, much of real-world information was put into computers. This trend has made data management and processing become much more efficient and convenient while enormous risks, which can spread on a world-wide area at an unprecedented high speed, were brought up. In fact, data breaches are no longer only risks and they happen on a daily basis at different levels of damage. Back to the past years, you can easily find news about such serious incidents every year, which affected hundred millions, if not billions, of users all over the world. Just to name some, several months ago, Marriott, a prestigious hotel group, revealed that they have suffered a huge data breach, which affected up to 500 millions customers [1] ; In 2017, Equifax, one of the largest credit reporting agencies in the USA, announced that they suffered a data breach in which information of about 143 million Americans leaked [2]; In 2016, AdultFriendFinder breach [3] affected 412 million accounts; In 2015, Anthem breach [4] impacted 78.8 million people. And in 2017, Yahoo, a once giant tech company, reported about a record-breaking data breach which had happened in 2013 [5] and affected three billion accounts.

All of those data breaches were originated from vulnerabilities included in software. Without automated security testing, it is almost impossible to eliminate

[1] https://www.forbes.com/sites/kateoflahertyuk/2018/11/30/marriott-breach-what-happened-how-serious-is-it-and-who-is-impacted

[2] https://www.forbes.com/sites/leemathews/2017/09/07/equifax-data-breach-impacts-143-million-americans

[3] https://www.zdnet.com/article/adultfriendfinder-network-hack-exposes-secrets-of-412-million-users

[4] https://en.wikipedia.org/wiki/Anthem_medical_data_breach

[5] https://www.nytimes.com/2017/10/03/technology/yahoo-hack-3-billion-users.html

them from software products because (1) time for development is getting shorter to fulfill the requirement of marketing, while those systems become more sophisticated and complicated; (2) security vulnerabilities are not like functional bugs that can be detected in the testing phase by referring to the predefined specification, but the program under analysis (PUA) needs to be placed in specific attack scenarios. While functional bugs in most of the time, only give bad user experience, security vulnerabilities always result in losing customers' personal properties, for those vulnerabilities are always related to attacks controlled deliberately by someone aiming at some specific goals. Therefore, it is essential to automate the analysis and detection of those vulnerabilities in a systematic way.

In 1982, Goguen and Meseguer [45] introduced noninterference as a property that a program has, if and only if, any change in confidential input makes no change on observable output. If computer programs satisfy this property, they are considered to be safe. It was expected to be a criterion that can be used in automated software verification in a provable fashion. For many years, technologies that can enforce the noninterference property on practical systems have been developed. However, even if we can seamlessly apply this security criterion to real-world systems, its usefulness is still limited because the noninterference property is too strict. Many useful yet safe daily used programs and protocols such as password authentication programs, voting protocols and so on, are marked as "vulnerable".

Example 1.1.1. Password verification program:

if $input = secret$ then $output \leftarrow 1$
else $output \leftarrow 0$

Let us consider the Example 1.1.1 above. The source code is to check if the public *input* matches with the preset password *secret*. This is a basic and popular implementation of password verification. However, if we use the noninterference property as a criterion of its safety, the program will be judged as unsafe, because the output value changes between 0 and 1 depending on the input value. This violates noninterference. On the other hand, if we use Shannon entropy based Quantitative Information Flow (QIF), which is introduced in the next chapter, the amount of leakage is less than 10^{-9} bits if *input* is a 32-bit word

and uniformly distributed. This quantity would be acceptable when the popular password lengths nowadays are always no shorter than 8 characters, for example 64 bits. There are many other computer programs in the real-world that violate noninterference but leak insignificant amount of secret information. Therefore QIF explains the safety better than noninterference in software security.

For these ten years, much attention has been paid at QIF. Despite of many breakthroughs in QIF analysis, the current state-of-the-art technique is still far from being used for systems in the real-world. Most of PUAs appearing in the literature are toy examples to illustrate the proposed methods rather than to show the practical effectiveness of those methods. The challenges to use QIF in practice are divided into the following three categories.

- The first one includes problems related to *scalability*. For instance, speeding up model counting, parallelizing computation, approximating with guaranteed confidence on arbitrarily tight bounds and so on.
- The second one includes problems related to *feasibility*. For example, handling complex data like strings, linked-lists, trees in static analysis, proposing new notions to model the real-world problems more reasonably, targeting distributed systems and so forth.
- The last one includes efforts related to *applicability* of QIF, such as software verification during the development phase, access control, dynamic security monitoring and so on.

Those categories are not necessarily disjoint to each other. Based on the observation above mentioned we found the strong motivation to urgently pursue a practically applicable QIF analysis.

1.2 Goals and research questions

As seen in the previous section, we tackle three categories of research challenges in for QIF analysis. Naturally, the scalability and feasibility, which are the foundation for utilizing QIF, should be done at the early stage, and the applicability can be tackled at a later stage. This dissertation focuses on improving the scalability and feasibility. The applicability will be one of future work.

Up to now, the QIF is calculated in three approaches: The most popular is the one using the model counting method. These are based on SAT (Satisfiability) [53] or SMT (Satisfiability Modulo Theories) [75]. Another approach uses the statistical estimation such as Leak Watch [31]. The last one is modeling the QIF calculation as a network flow capacity computing problem [63]. For addressing the scalability we use model counting based approaches. Since counting models is the most heavy-weight phase in the calculation process of this approach, the improvement of this phase is the most effective for making QIF analysis more practical. Currently counting problems for complex data structures such as strings, linked lists and trees are reduced to counting on simpler basic data types. However, sometimes the reduction is either only an approximation or infeasible. Thus our first research question is: (Q1) *Is there any method of counting models that directly analyzes variables with complex structures as what they are: strings, arrays, linked lists, trees and so on?*.

For feasibility, the existing QIFs cannot give reasonable measure when there are some output values revealing much more information about secret input than others, because existing notions, both entropy-based like summarized in [80] and belief-based like presented in [36], treat QIF as the average on all possible output values. In addition, as discussed in [23], those existing notions fail to give non-negative leakage in some cases. This fact makes it unnatural to put those notions as a metric for QIF. Hence our second research question is: (Q2) *How can we address reasonably and naturally the leakage of secure information when executing programs through a specific output value?*.

As will be introduced later in Chapter 3, new notions called QIF_1 and QIF_2 are proposed for *dynamic leakage*, which is the amount of information about confidential input that can be inferred from a specific output value. In the thesis, the computation of dynamic leakage is shown to be intractable, since the problem of calculating dynamic leakage for loop-free deterministic Boolean program is already $\sharp$P-hard, similarly to the computation of the existing QIFs. One of the approaches to deal with the worst-case computational difficulty is to use the strategy called "divide and conquer", that we chose to work on the calculation of the dynamic leakage. Our third research question is: (Q3) *How to calculate the dynamic leakage in executing a program P when the dynamic leakages of the*

sub-programs of P are known?.

1.3 Overview of results

The main purpose and contributions of this doctoral research are the attempts to tackle on the above research questions (Q1), (Q2) and (Q3). As for the first question about the existence of a counting method that does not transform complex data structure to basic types like bit-vectors, we propose a novel framework of power series including recognizable and algebraic series. While the former can be considered as the generalization of the state-of-the-art automata-based string counting method [12], the latter derives a new approach for string counting based on Context-Free Grammar (CFG) [33].

For answering (Q2), we propose two new notions called QIF_1 and QIF_2 that are the metrics for information leakage, which in this case we call dynamic leakage, of a program by observing a specific output value. We need both of the notions because there are cases where QIF_2 gives more reasonable result than QIF_1 does, but in general, it is simpler to calculate QIF_1 than QIF_2. Moreover, when considering deterministic programs, the notions coincide to each other. Besides, we give computational complexity bounds in calculating those dynamic leakage as well as prototype a model counting-based analyzer [34].

The question (Q3) comes naturally after answering (Q2). We propose two approaches: *sequential composition* and *value domain-based composition*. As for the former, we introduce a method to calculate lower and upper bounds of the dynamic leakage of the original program from the dynamic leakage of the constituted sub-programs. As for the latter, we give a parallel computing method to speed up the calculation. In practice, the two approaches can be combined to break big programs into smaller ones [35].

1.4 Related work

Feasibility: For *definitions of QIF*: Smith [80] gives a comprehensive summary on entropy-based QIF, such as Shannon entropy, guessing entropy and min-entropy, as well as compares them in various scenarios. Clarkson et al. [36], on the

other hand, include attacker's belief into their model. Alvim et al. [10] introduce gain function to generalize information leakage by separating the probability distribution and the impact of individual information. Another perspective on QIF was proposed by McCamant et al. [63], which is about dynamic leakage in terms of targeting binaries using taint analysis technique. They model the calculation of QIF as calculating capacity of a network flow. As there are multiple definitions for QIF, Alvim et al. [9] build a system of axioms that reasonable definitions of QIF should satisfy. Bielova [23] discusses the need to have new notion for dealing with leakage by different values of observable output. For *programming languages*: Phan et al. [76] propose an analysis using symbolic execution along with a prototype which works for Java language. Biondi et al. [27] introduce their own imperative language and implement QIF analysis for it. Besides, other CBMC-based [6] approaches are handling C language. For *complex data structure*: the problems with complex data structure are usually related to static analysis engines which are utilized during calculating QIF. Pasareanu et al. [71] develop Symbolic Path Finder (SPF), a symbolic executors. This is created as a plugin of Java Path Finder (JPF) [7], an extensible open source Java Model Checker, developed by NASA. Rosner et al. [78] improved SPF with the technique called lazy initialization so that the executor can handle efficiently dynamic structures such as: heap, linked list and so forth. Pham et al. [73] present an in-progress report about using separation logic to reason about variables with complex data structures. For *others*: Ying et al. [92] transfer the notion of noninterference from the model of classical computers to that of quantum computers. Anjaria and Mishra [11] introduce a fresh idea on making some hardware which can enforce QIF.

Scalability: For *model counting*: Klebanov et al. [53] propose a SAT-based calculation method of QIF which uses CBMC to transform a program into a Boolean formula, then count models projected on the set of variables corresponding to the output of the original program. Phan et al. [75] present a similar approach but using SMT instead of SAT. Nakashima et al. [70] make some improvements on the SMT-based method using blocking-clause, model cache and

[6] `https://www.cprover.org/cbmc`
[7] `https://ti.arc.nasa.gov/tech/rse/vandv/jpf`

static analysis. Suzuki et al. [82] attempt to speed up projected mode counting using decomposition. Chakraborty et al. [30] build an approximated model counter called ApproxMC2, based on Markov Chain Monte Carlo. More specifically, ApproxMC2 provides an approximation on the number of models of a Boolean formula represented in Conjunctive Normal Form (CNF) with adjustable precision and confidence. It uses hashing technique to divide the solution space into smaller buckets with almost equal number of elements, then count models only for one bucket and multiply it by the number of buckets. Biondi et al. [25] leverage ApproxMC2 to calculate QIF. Recently, Val et al. [86] reported a SAT-based method that can scale to programs of 10,000 lines of C code. For *string counting*: Luu et al. [61] use generating functions to count strings satisfying a set of constraints. This approach gives an approximated result. Aydin et al. [12], on the other hand, utilizing automata to represent the string constraints and achieve good performance by using matrix multiplication with the counter called ABC. The counting provides exact result except for non-regular constraints which are over-approximated to regular ones. In [13], the authors improve ABC by not limiting the domain of automata, but using some parameters, consequently so that the result can be applied for arbitrarily large bound. For *statistical estimation*: Chothia et al. [31] build "Leak Watch" to give approximation with provable confidence using statistical estimation by executing a program multiple times. Its descendant called HyLeak [26] combines the randomization strategy with static analysis. Köpf and Rybalchenko propose an approximation approach to calculate both upper and lower bounds of QIF. For *computational complexity*: as a foundation to speed up QIF calculation, the hardness of computing QIF was investigated by Yasuoka and Terauchi [90]. They proved that even the problem of comparing QIF of two programs, which is obviously not more difficult than calculating QIF, is not a k-safety property for any k. Consequently, self-composition, a successful technique to verify noninterference property, is not applicable to the comparison problem. Their subsequent work [91] proves a similar result for bounding QIF, as well as the PP-hardness of precisely quantifying QIF in all entropy-based definitions for loop-free Boolean programs. Extending those result, Chadha and Ummels [29] show that the QIF bounding problem of loop-free Boolean programs, when the number of output variables is logarithmic in the size of the program, lies

in the fourth level of the counting hierarchy, P^{CH_3}, and that of Boolean programs with loops is PSPACE-complete. Then they prove that, for recursive Boolean programs, both the problem of checking noninterference and of QIF bounding are EXPTIME-complete. Eventually, they extend their findings to the problem of bounding leakage through timing side-channel of recursive Boolean programs on uniformly distributed inputs, which turns out to be also EXPTIME-complete.

Applicability: Though calculating QIF is still not efficient enough to use for real-world systems, there are some researches about its applications. For *side-channel attacks*: Köpf and Basin [55] derive quantitative bounds of adaptive side-channel attacks. Köpf et al. [56] compute upper bound of the information that a malicious attacker can learn by observing CPU's cache behavior during running cryptographic algorithms. On the other hand, Huang et al. [49] build a tool to quantify side-channel leakage of web applications. Malacaria et al. [62] introduce additional inputs and use parameterized model counter to account for noise and randomness present when observing side-channels. For *differential privacy*: Alvim et al. [7] reason bound that differential privacy induces on QIF. The same group of authors in [8] show the close relation between differential privacy and QIF. Barthe and Köpf [19] derive a more comprehensive bounds on the leakage of an ε-differentially private mechanism in terms of ε and the size of the mechanism's input domain. Pettai and Laud [72] utilize the quantification of differential privacy of components to derive a bound for the whole workflow. Wang et al. [89] investigate relationship between three notions: identifiability, differential privacy and mutual information privacy. They also show that there exists a mechanism which optimizes both identifiability level and mutual-information privacy at the same time.

1.5 On perspective of Real-World Data Circulation (RWDC)

RWDC is a new research field that includes three phases arranged as one cycle (hence "circulation"): Acquisition, Analysis and Implementation [8]. In *Acquisi-*

[8] `http://www.rwdc.is.nagoya-u.ac.jp/eng/Pamphlet-En.pdf`

tion, real-world data, which is generated through the interactions taken in our daily lives, is collected in digital format by directly using recorders, readers, cameras and so forth, or indirectly via some information systems such as social networks, online services and so on. Normally raw data collected from real-world needs to be preprocessed such as by normalization, augmentation, sampling and conversion. Then the collected data is learned in *Analysis*, where predictions about wider pictures can be made and/or answers for unsolved problems can be found. Those values mined in the previous phase is transferred to the next phase, *Implementation*, in which they are used to build new (or change old) products and systems.

Applying the methodology of RWDC into QIF analysis, we have two targets that are changed according to the change of real-world data. The first one is the information leakage threshold, which is used to decide whether a program is safe or not. Namely, if the QIF of a program is bigger than the threshold, the program is not safe enough, and vice versa. So, if the threshold is too big, many vulnerable programs will be considered to be safe. On the other hand, if the threshold, however, is too small, many useful programs will be considered to be unsafe and stopped. The reasonableness of the threshold changes according to the real-world. For instance, when there are vulnerability reports CVEs of the MITRE [9] that help malicious attackers break a system with smaller amount of leaked information, the information leakage threshold needs to be decreased.

The second target is the confidential data that needs to be protected. As being showed in Chapter 3, dynamic leakage is the amount of information being leaked after each run of a program. Hence, the information leakage about a confidential data will be accumulated run after run. For instance, in case of side channel attacks, the attackers gather small amount of secret information after each execution to, eventually, reveal the whole information. Thus confidential data is gradually learned, and at one point, is no longer confidential. That is when the confidential data, such as passwords, need to be updated. Further discussion on those two data circulation system is developed in Chapter 5.

[9] `https://cve.mitre.org/`

Structure of the remaining parts

- Chapter 2 *Preliminaries*: this chapter presents foundational knowledge in the field of QIF that is necessary to comprehend the later coming chapters.
- Chapter 3 *Dynamic Leakage*: this chapter consists of every thing related to dynamic leakage, which is one of my main contributions in this doctoral research. It will elaborate on the new notions of dynamic leakage, the compositionality of those notions.
- Chapter 4 *QIF of String Manipulating Programs*: this chapter is about treating strings in analyzing QIF of programs. At first, a general framework to reason about counting complex data structures like strings and trees is introduced. After that, the fitting of the state-of-the-art automaton-based string counting technique is showed. Lastly, a novel counting method using CFGs is presented. The framework is built hierarchically according to the increasing expressive strengths of formal structures: recognizable, algebraic, and tree formal power series.
- Chapter 5 *Toward an RWDC System*: this chapter discusses the building of an RWDC system including QIF analysis as well as social values the system may produce.
- Chapter 6 *Conclusion*: the dissertation is concluded in this chapter by summarizing the proposed methods and results, discussing the extensible directions and predicting the future of QIF.

2 Preliminaries

Although the basic idea of taking the difference between the uncertainties about secret inputs before and after observing public outputs of a program is fixed as the core of QIF, there are various definitions. For instance, Min entropy based, Shannon entropy based, Guessing entropy based [80], generalized gain function based [10] and Belief based [36]. This chapter will only elaborate on Shannon entropy based and Min entropy based definitions with the aim at making this dissertation self-contained.

2.1 Quantitative information flow

In this section we will review the definitions of QIF based on Shannon entropy and Min entropy. The content is hugely based on [80].

2.1.1 Scenario and framework

QIF is usually placed under a scenario where there is some malicious attacker against a program who can observe public results when executing a program to deduce a secret input to that program. The source code of that program is fully exposed to the attacker. It might not be easy or impossible for an attacker to have the full knowledge of the source code of the system under attack. At the same time, it is not easy to assure that the source code is securely protected because normally a system is developed by many people, the development phase can last quite long and some public library could be integrated. Moreover, reverse-engineering and disassemble techniques are evolving rapidly. Therefore, it is reasonable to assume the worst case that the source code is publicly known.

In the remaining parts of this chapter we use the following notations. Let P be the program under analysis of QIF (and dynamic leakage introduced in Chapter

3), $\mathcal{S}$ be the finite set of input values and $\mathcal{O}$ be the finite set of output values of P. Without loss of generality, we assume that P has one secret input variable $s \in \mathcal{S}$ and one publicly observable output variable $o \in \mathcal{O}$. Random variables S and O represent the events in which s takes a value from $\mathcal{S}$ and o takes a value from $\mathcal{O}$, respectively. The definitions and calculation of QIF, which are presented in the next sections, can be extended naturally when P has multiple input variables, including both public and secret ones, and multiple output variables.

2.1.2 Entropy-based QIF

Given random variables X and Y ranging over finite sets $\mathcal{X}$ and $\mathcal{Y}$ respectively, Shannon entropy over X is defined as

$$H(X) = \sum_{x \in \mathcal{X}} p(X = x) \log \frac{1}{p(X = x)} \tag{2.1}$$

and the conditional entropy of X, provided that an event associated with the random variable Y happened, is

$$H(X|Y) = \sum_{y \in \mathcal{Y}} p(Y = y) \sum_{x \in \mathcal{X}} p(X = x|Y = y) \log \frac{1}{p(X = x|Y = y)} \tag{2.2}$$

where $p(X = x)$ denotes the probability that X has value x, and $p(X = x|Y = y)$ denotes the probability that X has value x when Y has value y. Intuitively, Shannon entropy expresses the average uncertainty about the result of a random event. Informally, QIF is defined by the equation:

$$QIF = Initial\ Uncertainty - Remaining\ Uncertainty. \tag{2.3}$$

In (2.3), *Initial Uncertainty* and *Remaining Uncertainty* are the degree of surprise about the value of s **before** and **after** executing P, respectively. Under Shannon entropy, the former is $H(S)$ while the latter is $H(S|O)$. Therefore we have the Shannon entropy-based QIF of P defined as

$$QIF^{Shannon}(P) = H(S) - H(S|O) \tag{2.4}$$

$H(S) - H(S|O)$ coincides with $I(S;O)$, *Mutual Information* between S and O. *Mutual Information* is symmetric in the sense that $H(S) - H(S|O) = I(S;O) =$

$I(O;S) = H(O) - H(O|S)$ holds. In addition, when P is deterministic, s determines o, so $H(O|S) = 0$. Therefore, for deterministic programs, the information leakage is simplified as $H(O)$.

Example 2.1.1. Let us consider the following example borrowing from [80].

$$o := s \ \& \ 0\text{x}037$$

where s is an unsigned uniformly-distributed 32-bit integer, and hence $0 \leq s \leq 2^{32}$. We have $H(S) = 2^{32}\frac{1}{2^{32}}\log 2^{32} = 32$ bits, $H(S|O) = 2^5\frac{1}{2^5}(2^{27}\frac{1}{2^{27}}\log 2^{27}) = 27$ bits. By (2.4), $QIF^{Shannon}(P) = 5$ bits. However Smith criticizes in [80] that Shannon entropy-based QIF does not always give smaller QIFs to safer programs. The following two examples are used in the literature to support that statement.

Example 2.1.2. The whole value of s is revealed totally whenever it is divisible by 8.

$$\text{if } s \ mod \ 8 = 0 \text{ then } o := s \text{ else } o := 1 \ (*)$$

Example 2.1.3. The first $7k - 1$ bits of s is masked.

$$o := s \ \& \ \underbrace{0...0}_{7k-1}\underbrace{1...1}_{k+1} \ (**)$$

Assume that, in both (*) and (**), s is a uniformly distributed $8k$-bit integer and $0 \leq s \leq 2^{8k}$. In the first example, the probability of going to the if-branch is just 1/8, while the probability for the program to proceed to the else-branch is 7/8. In other words, in 2^{8k-3} cases the program return $o = s$, and in the other 7/8 of the cases $o = 1$. Because (*) is deterministic, we have $QIF^{Shannon}(P) = H(O) = \frac{7}{8}\log\frac{8}{7} + 2^{8k-3}\frac{1}{2^{8k}}\log 2^{8k} \approx k + 0.169$. In the second example, because o is the copy of $k + 1$ right-most bits of s, we have $H(O) = k + 1$ bits. Also, (**) is deterministic, so that $QIF^{Shannon}(P) = H(O) = k + 1$ bits. According to the calculated QIFs, (*) leaks less than (**) does. However, while (*) leaks totally about s in 1/8 of the cases, in (**) the probability to exactly guess about s is only $\frac{1}{2^{7k-1}}$ that is almost 0 when k gets bigger. That is, even though the second example looks much safer than the first one, Shannon entropy-based QIF results in a bigger leakage for the second example, hence creates the counter-intuitive.

Smith [80] proposes a different notion called Min entropy-based QIF to address the above counter-intuitive. Firstly, he defines the vulnerability over a random variable X as

$$V(X) = \max_{x \in \mathcal{X}} p(X = x) \tag{2.5}$$

and the conditional vulnerability of X given Y is

$$V(X|Y) = \sum_{y \in \mathcal{Y}} p(Y = y) \max_{x \in \mathcal{X}} p(X = x|Y = y) \tag{2.6}$$

The vulnerability is exactly the probability of being guessed in the worst-case. Based on (2.5), Min entropy over X is defined as

$$H_\infty(X) = \log \frac{1}{V(X)} \tag{2.7}$$

and the conditional Min entropy of X given Y is

$$H_\infty(X|Y) = \log \frac{1}{V(X|Y)}. \tag{2.8}$$

Thus by (2.3), Min entropy-based QIF of P is defined as following

$$QIF^{min}(P) = H_\infty(S) - H_\infty(S|O) \tag{2.9}$$

Especially when P is deterministic and s is uniformly distributed, (2.9) is simplified to

$$QIF^{min}(P) = \log |\mathcal{S}| - \log \frac{|\mathcal{S}|}{|\mathcal{O}|} = \log |\mathcal{O}| \tag{2.10}$$

where $|\mathcal{S}|$ and $|\mathcal{O}|$ denote the cardinalities of $\mathcal{S}$ and $\mathcal{O}$, respectively.

2.2 Model counting-based calculation of QIF

This subsection is dedicated to the introduction of two popular methods to calculate QIF by model counting. As required by model counters, let us assume the program under analysis P be deterministic and terminating. We assume a program is non-deterministic and non-terminating. Another assumption is that the input value set $\mathcal{S}$ is uniformly distributed. However, this assumption technically can be removed by using the result in [15].

2.2.1 Satisfiability (SAT)-based counting

Definition 2.2.1 (Truth Assignment)**.** Given a propositional (or Boolean) formula φ and $var(\varphi)$, the set of Boolean variables of φ, a truth assignment μ is a total function $\mu : var(\varphi) \rightarrow \{True, False\}$.

Definition 2.2.2 (SAT)**.** Given a propositional formula φ, SAT problem is the problem of finding a truth assignment μ that makes φ true. If such an assignment μ exists, φ is said to be *Satisfiable* and μ is called a model of φ, otherwise φ is *Unsatisfiable*.

Definition 2.2.3 (#SAT)**.** Given a propositional formula φ, #SAT is the problem of finding all models of φ. #SAT is also referred to as All-SAT problem.

When $\mathcal{S}$ is uniformly distributed, we have $QIF^{min}(P) = \log |\mathcal{O}|$ and $QIF^{Shannon}(P) = H(O)$. It suffices to count the distinct output values to calculate the former, while the distribution on $\mathcal{O}$ is needed to calculate the latter. Let $\mathcal{O} = \{o_1, o_2, ...o_n\}$ such that $o_i \neq o_j \ \forall i \neq j$. For each $o_i \in \mathcal{O}$, let $C_{o_i} = \{s' \in \mathcal{S} \mid P(s') = o_i\}$. Because P is deterministic, we have $C_{o_i} \cap C_{o_j} = \emptyset \ \forall i \neq j$. C_{o_i} is called the preimage set of o_i by program P. The probability distribution on $\mathcal{O}$ is $p(o = o_1) = \frac{|C_{o_1}|}{|\mathcal{S}|}, p(o = o_2) = \frac{|C_{o_2}|}{|\mathcal{S}|}, ..., p(o = o_n) = \frac{|C_{o_n}|}{|\mathcal{S}|}$. Thus,

$$QIF^{Shannon}(P) = H(O) = \sum_{o_i \in \mathcal{O}} \frac{|C_{o_i}|}{|\mathcal{S}|} \log \frac{|\mathcal{S}|}{|C_{o_i}|}. \tag{2.11}$$

The very first step is to transform a given program P into an equivalent propositional formula that represents P's behavior. By using the model checker CBMC the propositional formula is obtained in the following way. At first, all function calls are unfolded by the function bodies, and then all loops are unwound by a predefined number that can be chosen as a setting option. Secondly, the program is transformed into Single-Static-Assignment (SSA) form in which each statement is converted to an equation of bit vectors. As the result, original variables are represented by vectors of propositional variables. Finally, the equations are simplified and transformed to Conjunctive Normal Form (CNF).

Let us use the example 2.2.1 to illustrate the transformation of a program into a Boolean formula. For the purpose of illustration, we will set the unwinding

Example 2.2.1. Electronic purse program from [16].

```
1    int purse(int l, int h){
2        int output = 0;
3        while (h ≥ l){
4            h = h − l;
5            output = output + 1;
6        }
7        return output;
8    }
```

bound is two. The program after being unwound is as in Figure 2.1. At line 2 of the original program, *output* becomes $output_0$, the very first value of this variable. After that, whenever value of *output* is updated, the subscription is increased so that all values are stored in a distinct version of *output*. Similarly, variables h and l are added subscriptions. The **while** loop from line 3 to 6 of the original program is unwound by a depth of two into a nested branching statements from line 3 to line 15 in Figure 2.1. There are three possible execution paths of the unwound program. **The first one** is when the condition $(h_0 \geq l_0)$ in line 3 is $false$, so $output_3 = output_0$. **The second one** is when the condition $(h_0 \geq l_0)$ in line 3 is $true$, and $(h_1 \geq l_0)$ in line 6 is $false$ hence $output_2 = output_1$, $output_1 = output_0 + 1$ and $output_3 = output_2$. **The last one** is when both of the conditions: $(h_0 \geq l_0)$ in line 3 and $(h_1 \geq l_0)$ in line 6, are $true$, thus $output_1 = output_0 + 1$, $output_2 = output_1 + 1$ and $output_3 = output_2$. Aggregating those three execution paths, we get the propositional formula (2.12) that represents the original program in example 2.2.1 with the unwinding depth of two. In the formula, **result** stands for the return value of the function. Finally, replacing

```
1    int purse(int l, int h){
2        int output_0 = 0;
3        if (h_0 ≥ l_0){
4            h_1 = h_0 − l_0;
5            output_1 = output_0 + 1;
6            if (h_1 ≥ l_0){
7                h_2 = h_1 − l_0;
8                output_2 = output_1 + 1;
9                return output_2;
10           } else return output_1;
11       } else return output_0;
12   }
```

Figure 2.1: The program after being unwound

variables by vectors of bits yields a Boolean formula that can be normalized

$$\begin{aligned}
&\Big(\neg(h_0 \geq l_0) \wedge (\mathbf{result} = output_0)\Big) \vee \\
&\Big((h_0 \geq l_0) \wedge (h_1 = h_0 - l_0) \wedge \neg(h_1 \geq l_0) \wedge (output_1 = output_0 + 1) \wedge \\
&\qquad\qquad (\mathbf{result} = output_1) \Big) \vee \\
&\Big((h_0 \geq l_0) \wedge (h_1 = h_0 - l_0) \wedge (h_1 \geq l_0) \wedge (h_2 = h_1 - l_0) \wedge \\
&\quad (output_1 = output_0 + 1) \wedge (output_2 = output_1 + 1) \wedge (\mathbf{result} = output_2)\Big)
\end{aligned} \tag{2.12}$$

Let $\langle P \rangle$ denote the propositional formula in CNF of P, $(b_1 b_2 ... b_k)$ and $(r_1 r_2 ... r_t)$ be the propositional representation of s and o, respectively. Without loss of generality, we assume that $\langle P \rangle$ has one input and one output. Basically, besides the propositional variables corresponding to the input and the output, $\langle P \rangle$ includes many not-of-interest variables. It is necessary, therefore, to project models onto the set of Boolean variables that represent the input and the output, to count the distinct output values. Let Δ be the set of projection variables. Projecting a model m on to Δ is to remove from the model all the truth values of propositional variables that are not in Δ. Algorithm 1 illustrates the counting method.

Algorithm 1 SATBasedProjectionCount($\langle P \rangle$, Δ) [53]

1: $M \leftarrow \emptyset$
2: $m \leftarrow \text{model}(\langle P \rangle)$
3: **while** $m \neq \bot$ **do**
4: $\quad M \leftarrow M \cap m|_{\Delta}$
5: $\quad \langle P \rangle \leftarrow \langle P \rangle \wedge (\Delta \not\equiv m|_{\Delta})$
6: $\quad m \leftarrow \text{model}(\langle P \rangle)$
7: **return** M

model($\langle P \rangle$) (line 2 and 6) invokes a SAT solver and it returns a model of $\langle P \rangle$. In the right hand side of line 5, $\Delta \not\equiv m|_{\Delta}$ expresses the constraint that all the variables in Δ are different from their counterparts in the projection of model m on Δ. This constraint assures that all models in M are distinct.

For Min entropy-based QIF, it suffices to count the distinct outputs. This is achieved by letting $\Delta = \{r_1, r_2..., r_t\}$. By running Algorithm 1, we have $|\mathcal{O}| = |M|$, and hence $QIF^{min}(P) = \log |\mathcal{O}|$.

For Shannon entropy-based QIF, let us set $\Delta = \{b_1, b_2, ..., b_k, r_1, r_2, ..., r_t\}$. After running Algorithm 1, each model $m \in M$ is a mapping from Δ to $\{True, False\}$. Let us denote m_j for the M's j-th model and s_j for the mapping of m_j from $\{b_1, b_2, ..., b_k\}$ to $\{True, False\}$. We have $s_u \neq s_v \; \forall u \neq v$ because P is deterministic, and therefore $|\mathcal{S}| = |M|$. For each $o_i \in \mathcal{O}$, $|C_{o_i}|$ is calculated by iterating through all $m \in M$ to pick up the models that $(r_1 r_2 ... r_t)$ maps to the corresponding propositional representation of o_i. Finally, $QIF^{Shannon}(P)$ is calculated based on (2.11).

2.2.2 Satisfiability Modulo Theory (SMT)-based counting

An SMT-based counting method is proposed by Phan et al. [74, 75]. Assume $\mathcal{V}$, $\mathcal{F}$ and $\mathcal{P}$ be countable sets of variables, function symbols and predicate symbols, respectively.

Definition 2.2.4 (First-Order Logic Signature)**.** A first-order logic signature is defined as a partial function $\Sigma : \mathcal{F} \cup \mathcal{P} \rightarrow \mathbb{N}$.

If $\Sigma(p) = 0$ for a predicate symbol p, then p is a Boolean variable. If $\Sigma(f) = 0$

for a function symbol f, then f is a constant.

Definition 2.2.5 (Σ-term)**.** A variable $x \in \mathcal{V}$ is a Σ-term. If $\tau_1, \ldots, \tau_n$ are Σ-terms, and $f \in \mathcal{F}$ is a function symbol such that $\Sigma(f) = n$, then $f(\tau_1, \ldots, \tau_n)$ is a Σ-term.

Definition 2.2.6 (Σ-atom)**.** If $\tau_1, ..., \tau_n$ are Σ-terms, and $p \in \mathcal{P}$ is a predicate symbol such that $\Sigma(p) = n$, then $p(\tau_1, ..., \tau_n)$ is a Σ-atom.

Definition 2.2.7 (Σ-formula)**.** A Σ-formula is either a Σ-atom or an expression constructed from other Σ-formulas by using logic connectives $(\neg, \wedge, \vee, \rightarrow)$, universal quantifier $(\forall)$ and existential quantifier $(\exists)$.

Definition 2.2.8 (Σ-model)**.** A Σ-model $\mathcal{A}$ is a pair consisting of a non-empty set A, the universe of the model, and a mapping $(\_)^{\mathcal{A}}$ that assigns

$$\begin{cases} \text{to } a \text{ an element } a^{\mathcal{A}} \in A & \text{if } a \text{ is a constant symbol,} \\ \text{to } f \text{ a total function } f^{\mathcal{A}} : A^n \to A & \text{if } f \text{ is a function symbol} \\ \quad \text{and } \Sigma(f) = n, & \\ \text{to } B \text{ an element } B^{\mathcal{A}} \in \{True, False\} & \text{if } B \text{ is a Boolean variable,} \\ \text{to } p \text{ a total function } p^{\mathcal{A}} : A^n \to \{True, False\} & \text{if } p \text{ is a predicate symbol} \\ \quad \text{and } \Sigma(p) = n. & \end{cases}$$

Definition 2.2.9 (Assignment)**.** An assignment to variables is a mapping $\theta : \mathcal{V} \to A$.

A Σ-model and an assignment uniquely determines a homomorphic extension that maps each Σ-term t to an element $t^{\mathcal{A},\theta} \in A$, and each Σ-formula φ to an element $\varphi^{\mathcal{A},\theta} \in \{True, False\}$.

Definition 2.2.10 (First-Order Theory)**.** A first-order theory $\mathcal{T}$ is a set of Σ-models.

Definition 2.2.11 ($\mathcal{T}$-satisfiability)**.** A Σ-model $\mathcal{A}$ and an assignment θ satisfy a Σ-formula φ iff $\varphi^{\mathcal{A},\theta}$ is $True$. A Σ-formula φ is satisfiable in a theory $\mathcal{T}$, or $\mathcal{T}$-satisfiable, iff there exists a Σ-model $\in \mathcal{T}$ and an assignment that satisfies φ.

Definition 2.2.12 (SMT [74])**.** Given a theory or a combination of theories $\mathcal{T}$ and a Σ-formula φ, the problem SMT is the problem of deciding $\mathcal{T}$-satisfiability of φ.

Definition 2.2.13 (#SMT [74])**.** Given a theory or a combination of theories $\mathcal{T}$ and a Σ-formula φ, the Model Counting Modulo Theories problem (#SMT) is the problem of finding all models M of $\mathcal{T}$ w.r.t a set of Boolean variables V_I such that φ is $\mathcal{T}$-satisfiable in M.

Because there are theories $\mathcal{T}$s that allow infinite range of variables, the set V_I of *important* variables is added to assure the number of models w.r.t V_I to be finite. The basic difference between #SAT and #SMT is that the former works on Boolean variables after transforming original constraints into this basic unit of propositional logic, while the latter directly works on original constraints. A program P is transformed into a Σ-formula in a similar way that was illustrated in the example 2.2.1 previously. In that example, the formula (2.12) is exactly the Σ-formula (modulo the theory of integers) that represents the original program P.

Algorithm 2 All-BC(φ, V_I, V_R) [74]

```
1:  N ← 0; Ψ ← ε;
2:  Assert(φ);
3:  while Check() = SAT do
4:      N ← N + 1;
5:      m ← Model(φ);
6:      m_ir ← filter(m, V_I, V_R);
7:      Ψ ← Ψ ∪ {m_ir};
8:      block ← FALSE;
9:      for all p_i ∈ V_I do block ← block ∨ (p_i ≠ Eval(p_i));
10:     Assert(block);
11: return N, Ψ;
```

Besides φ and V_I, V_R is another argument that specifies *relevant* variables, namely the assignments of the variables of interest when a model is found w.r.t V_I. V_R is designed for the automated test generation, but not directly related to calculating

QIF. In Algorithm 2: N and Ψ are the number of models and the enumeration of the models; Assert in lines 2 and 10 is to set the Σ-formula φ in the problem #SMT, by line 10 φ is reassigned to $\varphi\wedge$ block; Check() in line 3 is to verify if φ is $\mathcal{T}$-satisfiable or not; Model(φ) in line 5 is to retrieve one model in the case φ is $\mathcal{T}$-satisfiable; filter(m, V_I, V_R) in line 6 is to extract from model m the counterparts of V_I and V_R; Lines 8 and 9 are used to add the blocking clause to the Σ-formula so that the same model will not be retrieved again. The idea is similar to line 5 in Algorithm 1.

For implementation, Σ-formula φ is constructed from a program P by CBMC in SMT-LIB v2 format. Then, Algorithm 2 is run using SMT solvers such as Z3. For calculating QIF, V_R can be omitted. For Min entropy-based QIF, as shown in the previous section, we count the distinct output values. This can be done by letting V_I be the set of bits in the bit-vector representation of output o. The step is completed by augmenting the following assertion into φ. The first bit of O_3 is assigned to $\#b1$, and p_1 is set to the truth value of $\#b1$.

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

For Shannon entropy-based QIF, let us set V_I to be the set of bits in the bit-vector representation of output o **and input s**. By applying Algorithm 2, similar to SAT-based counting, we get a set of models including both output-related and input-related truth assignments. Thus, it is straight forward to count $|\mathcal{S}|$, $|C_{o_i}|$ for every $o_i \in \mathcal{O}$ and hence $QIF^{Shannon}(P)$ can be calculated by (2.11).

3 Dynamic leakage

This chapter is dedicated to represent dynamic leakage that is the quantitative information flow when observing concrete output value. In the first half, two new notions called QIF_1 and QIF_2 are introduced following by discussions on computational complexity and a model counting-based leakage quantification method. Naturally, the question of how to speed up the calculation comes and is answered in the second half.

3.1 Background

Researchers have realized the importance of knowing where confidential information reaches by the execution of a program to verify whether the program is safe. The *noninterference* property, namely, any change of confidential input does not affect public output, was coined in 1982 by Goguen and Meseguer [45] as a criterion for the safety. This property, however, is too strict in many practical cases, such as password verification, voting protocol and averaging scores. A more elaborated notion called quantitative information flow (QIF) [80] has been getting much attention of the community. QIF is defined as the amount of information leakage from secret input to observable output. The program can be considered to be safe (resp. vulnerable) if this quantity is negligible (resp. large). QIF analysis is not easier than verifying noninterference property because if we can calculate QIF of a program, we can decide whether it satisfies noninterference or not. QIF calculation is normally approached in an information-theoretic fashion to consider a program as a communication channel with input as source, and output as destination. The quantification is based on entropy notions including Shannon entropy, min-entropy and guessing entropy [80]. QIF (or the information leakage) is defined as the remaining uncertainty about secret input after observ-

ing public output, i.e., the mutual information between source and destination of the channel. Another quantification proposed by Clarkson, et al. [36], is the difference between 'distances' (Kullback-Leibler divergence) from the probability distribution on secret input that an attacker believes in to the real distribution, before and after observing the output values.

While QIF is about the average amount of leaked information over all observable outputs, *dynamic leakage* is about the amount of information leaked by observing a *particular* output. Hence, QIF is aimed to verify the safety of a program in a static scenario in compile time, and dynamic leakage is aimed to verify the safety of a specific running of a program. So which of them should be used as a metric to evaluate a system depends on in what scenario the software is being considered.

Example 3.1.1.

if $source < 16$ then $output \leftarrow 8 + source$
else $output \leftarrow 8$

In Example 3.1.1 above, assume *source* to be a positive integer, then there are 16 possible values of *output*, from 8 to 23. While an observable value between 9 and 23 reveals *everything* about the secret variable, i.e., there is only one possible value of *source* to produce such *output*, a value of 8 gives almost nothing, i.e., there are so many possible values of *source* which produce 8 as output. Taking the average of leakages on all possible execution paths results in a relatively small value, which misleads us into regarding that the vulnerability of this program is small. Therefore, it is crucial to differentiate risky execution paths from safe ones by calculating dynamic leakage, i.e., the amount of information that can be learned from observing the output which is produced by a specific execution path. Practical application of quantifying dynamic leakage includes monitoring dynamically side channels of systems, dynamic access control at run-time, and so forth.

But, as discussed in [23], any of existing QIF models (either entropy based or belief tracking based) does not always seem reasonable to quantify dynamic leakage. For example, entropy-based measures give sometimes negative leakage.

Usually, we consider that the larger the value of the measure is, the more information is leaked, and in particular, no information is leaked when the value is 0. In the interpretation, it is not clear how we should interpret a negative value as a leakage metric. Actually, [23] claims that the non-negativeness is a requirement for a measure of dynamic QIF. Also, *MONO*, one of the axioms for QIF in [9] turns out to be identical to this non-negative requirement. Belief-based one always give non-negative leakage for deterministic programs but it may become negative for probabilistic programs. In addition, the measure using belief model depends on secret values. This would imply (1) even if a same output value is observed, the QIF may become different depending on which value is assumed to be secret, which is unnatural, and (2) a side-channel may exist when further processing is added by system managers after getting quantification result. Hence, as suggested in [23], it is better to introduce a new notion for quantifying dynamic leakage caused by observing a specific output value.
The contributions of this research are three-fold.

- We present our criteria for an appropriate definition of dynamic leakage and propose two notions that satisfy those criteria. We propose two notions because there is a trade-off between the easiness of calculation and the preciseness (see Section 3.2).

- Complexity of computing the proposed dynamic leakages is analyzed for three classes of Boolean programs.

- By applying model counting of logical formulae, a prototype was implemented and feasibility of computing those leakages is discussed based on experimental results.

According to [23], we arrange three criteria that a 'good' definition of dynamic leakage should satisfy, namely, the measure should be (R1) non-negative, (R2) independent of a secret value to prevent a side channel and (R3) compatible with existing notions to keep the consistency within QIF as a whole (both dynamic leakage and normal QIF). Based on those criteria, we come up with two notions of dynamic leakage QIF1 and QIF2, where both of them satisfy all (R1), (R2) and (R3). QIF1, motivated by entropy-based approach, takes the difference between the initial and remaining self-information of the secret before and after

observing output as dynamic leakage. On the other hand, QIF2 models that of the joint probability between secret and output. Because both of them are useful in different scenarios, we studied these two models in parallel in the theoretical part of the paper. We call the problems of computing QIF1 and QIF2 for Boolean programs CompQIF1 and CompQIF2, respectively. For example, we show that even for deterministic loop-free programs with uniformly distributed input, both CompQIF1 and CompQIF2 are $\sharp P$-hard. Next, we assume that secret inputs of a program are uniformly distributed and consider the following method of computing QIF1 and QIF2 (only for deterministic programs for QIF2 by the technical reason mentioned in Section 4): (1) translate a program into a Boolean formula that represents relationship among values of variables during a program execution, (2) augment additional constraints that assign observed output values to the corresponding variables in the formula, (3) count models of the augmented Boolean formula projected on secret variables, and (4) calculate the necessary probability and dynamic leakage using the counting result. Based on this method, we conducted experiments using our prototype tool with benchmarks taken from QIF related literatures, in which programs are deterministic, to examine the feasibility of automatic calculation. We also give discussion, in subsection 5.3, on difficulties and possibilities to deal with more general cases, such as, of probabilistic programs. In step (3), we can flexibly use any off-the-shelf model counter. To investigate the scalability of this method, we used four state-of-the-art counters, SharpCDCL [53] and GPMC [82] for SAT-based counting, an improved version of aZ3 [75] for SMT-based counting, and DSharp-p [68] for SAT-based counting in d-DNNF fashion. Finally, we discuss the feasibility of automatic calculation of the leakage in general case.

Related work The very early work on *computational complexity* of QIF is that of Yasuoka and Terauchi. They proved that even the problem of comparing QIF of two programs, which is obviously not more difficult than calculating QIF, is not a k-safety property for any k [90]. Consequently, self-composition, a successful technique to verify noninterference property, is not applicable to the comparison problem. Their subsequent work [91] proves a similar result for bounding QIF, as well as the PP-hardness of precisely quantifying QIF in all entropy-based definitions for loop-free Boolean programs. Chadha and Ummels [29] show that the

QIF bounding problem of recursive programs is not harder than checking reachability for those programs. Despite given those evidences about the hardness of calculating QIF, for this decade, *precise QIF analysis* gathers much attention of the researchers. In [53], Klebanov et al. reduce QIF calculation to ♯SAT problem projected on a specific set of variables as a very first attempt to tackle with automating QIF calculation. On the other hand, Phan et al. reduce QIF calculation to ♯SMT problem for utilizing existing SMT (satisfiability modulo theory) solver. Recently, Val et al. [86] reported a method that can scale to programs of 10,000 lines of code but still based on SAT solver and symbolic execution. However, there is still a gap between such improvements and practical use, and researchers also work on *approximating QIF*. Köpf and Rybalchenko [57] propose approximated QIF computation by sandwiching the precise QIF by lower and upper bounds using randomization and abstraction, respectively with a provable confidence. LeakWatch of Chothia et al. [31], also give approximation with provable confidence by executing a program multiple times. Its descendant called HyLeak [26] combines the randomization strategy of its ancestor with precise analysis. Also using randomization but in Markov Chain Monte Carlo (MCMC) manner, Biondi et al. [25] utilize ApproxMC2, an existing model counter created by some of the co-authors. ApproxMC2 provides approximation on the number of models of a Boolean formula in CNF with adjustable precision and confidence. ApproxMC2 uses hashing technique to divide the solution space into smaller buckets with almost equal number of elements, then count the models for only one bucket and multiply it by the number of buckets. As for *dynamic leakage*, McCamant et al. [63] consider QIF as network flow through programs and propose a dynamic analysis method that can work with executable files. Though this model can scale to very large programs, its precision is relatively not high. Alvim et al. [9] give some axioms for a reasonable definition of QIF to satisfy and discuss whether some definitions of QIF satisfy the axioms. Note that these axioms are for *static* QIF measures, which differ from dynamic leakage. However, given a similarity between static and dynamic notions, we investigated how our new dynamic notions fit in the lens of the axioms (refer to Section 3.2).

Dynamic information flow analysis (or taint analysis) is a bit confusing term that does not mean an analysis of dynamic leakage, but a runtime analysis of

information flow. Dynamic analysis can abort a program as soon as an unsafe information flow is detected. Also, hybrid analysis has been proposed for improving dynamic analysis that may abort a program too early or unnecessarily. In hybrid analysis, the unexecuted branches of a program is statically analyzed in parallel with the executed branch. Among them, Bielova et al. [22] define the knowledge $\kappa(z)$ of a program variable z as the information on secret that can be inferred from z (technically, $\kappa(z)^{-1}(v)$ is the same of the pre-image of an observed value v of z, defined in Section 2). In words, hybrid analysis updates the 'dynamic leakage' under the assumption that the program may terminate at each moment. Our method is close to [22] in the sense that the knowledge $\kappa(z)^{-1}(v)$ is computed. The difference is that we conduct the analysis after the a program is terminated and v is given. We think this is not a disadvantage compared with hybrid analysis because the amount of dynamic leakage of a program is not determined until a program terminates in general.

3.2 New notions

This section introduces two new notions of dynamic leakage as well as discusses why they can be considered to be reasonable under a theoretically firmly backed-up framework of axioms. As far as we know, these are first notions of dynamic leakage that have ever been introduced.

3.2.1 $\mathbf{QIF}_1$ and $\mathbf{QIF}_2$

The standard notion for static quantitative information flow (QIF) is defined as the mutual information between random variables S for secret input and O for observable output:

$$\mathrm{QIF} = H(S) - H(S|O) \tag{3.1}$$

where $H(S)$ is the entropy of S and $H(S|O)$ is the expected value of $H(S|o)$, which is the conditional entropy of S when observing an output o. Shannon entropy and min-entropy are often used as the definition of entropy, and in either case, $H(S) - H(S|O) \geq 0$ always holds by the definition.

In [23], the author discusses the appropriateness of the existing measures for dynamic QIF and points out their drawbacks, especially, each of these measures may become negative. Hereafter, let $\mathcal{S}$ and $\mathcal{O}$ denote the finite sets of input values and output values, respectively. Since $H(S|O) = \sum_{o\in\mathcal{O}} p(o)H(S|o)$, [23] assumes the following measure obtained by replacing $H(S|O)$ with $H(S|o)$ in (3.1) for dynamic QIF:

$$\mathrm{QIF}^{dyn}(o) = H(S) - H(S|o). \tag{3.2}$$

However, $\mathrm{QIF}^{dyn}(o)$ may become negative even if a program is deterministic (see Example 3.2.1). Another definition of dynamic QIF is proposed in [36] as

$$\mathrm{QIF}^{belief}(\dot{s}, o) = D_{KL}(p_{\dot{s}}||p_S) - D_{KL}(p_{\dot{s}}||p_{S|o}) \tag{3.3}$$

where D_{KL} is KL-divergence defined as $D_{KL}(p||q) = \sum_{s\in\mathcal{S}} p(s)\log\frac{p(s)}{q(s)}$, and $p_{\dot{s}}(s) = 1$ if $s = \dot{s}$ and $p_{\dot{s}}(s) = 0$ otherwise. Intuitively, $\mathrm{QIF}^{belief}(\dot{s}, o)$ represents how closer the belief of an attacker approaches to the secret $\dot{s}$ by observing o. For deterministic programs, $\mathrm{QIF}^{belief}(\dot{s}, o) = -\log p(o) \geq 0$ [23]. However, QIF^{belief} may still become negative if a program is probabilistic (see Example 3.2.2).

Let P be a program with secret input variable S and observable output variable O. For notational convenience, we identify the names of program variables with the corresponding random variables. Throughout the paper, we assume that a program always terminates. The syntax and semantics of programs assumed in this section will be given in the next section. For $s \in \mathcal{S}$ and $o \in \mathcal{O}$, let $p_{SO}(s, o)$, $p_{O|S}(o|s)$, $p_{S|O}(s|o)$, $p_S(s)$, $p_O(o)$ denote the joint probability of $s \in \mathcal{S}$ and $o \in \mathcal{O}$, the conditional probability of $o \in \mathcal{O}$ given $s \in \mathcal{S}$ (the likelihood), the conditional probability of $s \in \mathcal{S}$ given $o \in \mathcal{O}$ (the posterior probability), the marginal probability of $s \in \mathcal{S}$ (the prior probability) and the marginal probability of $o \in \mathcal{O}$, respectively. We often omit the subscripts as $p(s, o)$ and $p(o|s)$ if they are clear from the context. By definition,

$$\begin{aligned} p(s, o) &= p(s|o)p(o) = p(o|s)p(s), && (3.4)\\ p(o) &= \sum_{s\in\mathcal{S}} p(s, o), && (3.5)\\ p(s) &= \sum_{o\in\mathcal{O}} p(s, o). && (3.6) \end{aligned}$$

We assume that (the source code of) P and the prior probability $p(s)$ ($s \in \mathcal{S}$) are known to an attacker. For $o \in \mathcal{O}$, let $\mathrm{pre}_P(o) = \{s \in \mathcal{S} \mid p(s|o) > 0\}$, which is called the pre-image of o (by the program P).

Considering the discussions in the literature, we aim to define new notions for dynamic QIF that satisfy the following requirements:

(R1) Dynamic QIF should be always non-negative because an attacker obtains some information (although sometimes very small or even zero) when he observes an output of the program.

(R2) It is desirable that dynamic QIF is independent of a secret input $s \in \mathcal{S}$. Otherwise, the controller of the system may change the behavior for protection based on the estimated amount of the leakage that depends on s, which may be a side channel for an attacker.

(R3) The new notion should be compatible with the existing notions when we restrict ourselves to special cases such as deterministic programs, uniformly distributed inputs, and taking the expected value.

The first proposed notion is the self-information of the secret inputs consistent with an observed output $o \in \mathcal{O}$. Equivalently, the attacker can narrow down the possible secret inputs after observing o to the pre-image of o by the program. We consider the self-information of $s \in \mathcal{S}$ after the observation as the logarithm of the probability of s divided by the sum of the probabilities of the inputs in the pre-image of o (see the upper part of Figure 4.2).

$$\mathrm{QIF}_1(o) \;=\; -\log\Big(\sum_{s' \in \mathrm{pre}_P(o)} p(s')\Big). \tag{3.7}$$

The second notion is the self-information of the joint events $s' \in \mathcal{S}$ and an observed output $o \in \mathcal{O}$ (see the lower part of Figure 4.2). This is equal to the self-information of o.

$$\mathrm{QIF}_2(o) \;=\; -\log\Big(\sum_{s' \in \mathcal{S}} p(s', o)\Big) \tag{3.8}$$

$$= -\log p(o) \;=\; -\log p(s, o) + \log p(s|o). \tag{3.9}$$

Theorem 3.2.1. If a program P is deterministic, for every $o \in \mathcal{O}$ and $s \in \mathcal{S}$,

$$\mathrm{QIF}^{belief}(s, o) = \mathrm{QIF}_1(o) = \mathrm{QIF}_2(o) = -\log p(o),$$

and if input values are uniformly distributed, for every $o \in \mathcal{O}$,

$$\mathrm{QIF}_1(o) = \log \frac{|\mathcal{S}|}{|\mathrm{pre}_P(o)|}$$

□

Both notions are defined by considering how much possible secret input values are reduced by observing an output. We propose two notions because there is a trade-off between the easiness of calculation and the appropriateness. As illustrated in Example 3.2.2, QIF_2 can represent the dynamic leakage more appropriately than QIF_1 in some cases. On the other hand, the calculation of QIF_1 is easier than QIF_2 as discussed in Section 4.3.1. Both notions are independent of the secret input $s \in \mathcal{S}$ (Requirement (R2)).

$$0 \leq \mathrm{QIF}_1(o) \leq \mathrm{QIF}_2(o). \tag{3.10}$$

If we assume Shannon entropy,

$$\begin{aligned} \mathrm{QIF} &= -\sum_{s \in \mathcal{S}} p(s) \log p(s) \\ &\quad + \sum_{o \in \mathcal{O}} p(o) \sum_{s \in \mathcal{S}} p(s|o) \log p(s|o) \qquad (3.11) \\ &= -\sum_{s \in \mathcal{S}} p(s) \log p(s) \\ &\quad + \sum_{s \in \mathcal{S}, o \in \mathcal{O}} p(s, o) \log p(s|o). \qquad (3.12) \end{aligned}$$

If a program is deterministic, for each $s \in \mathcal{S}$, there is exactly one $o_s \in \mathcal{O}$ such that $p(s, o_s) = p(s)$ and $p(s, o) = 0$ for $o \neq o_s$, and therefore

$$\mathrm{QIF} = \sum_{s \in \mathcal{S}, o \in \mathcal{O}} p(s, o)(-\log p(s, o) + \log p(s|o)). \tag{3.13}$$

Comparing (3.9) and (3.13), we see that QIF is the expected value of QIF_2, which suggests the compatibility of QIF_2 with QIF (Requirement (R3)) when a program is deterministic. Also, if a program is deterministic, $\mathrm{QIF}^{belief}(\dot{s}, o) = -\log p(o)$,

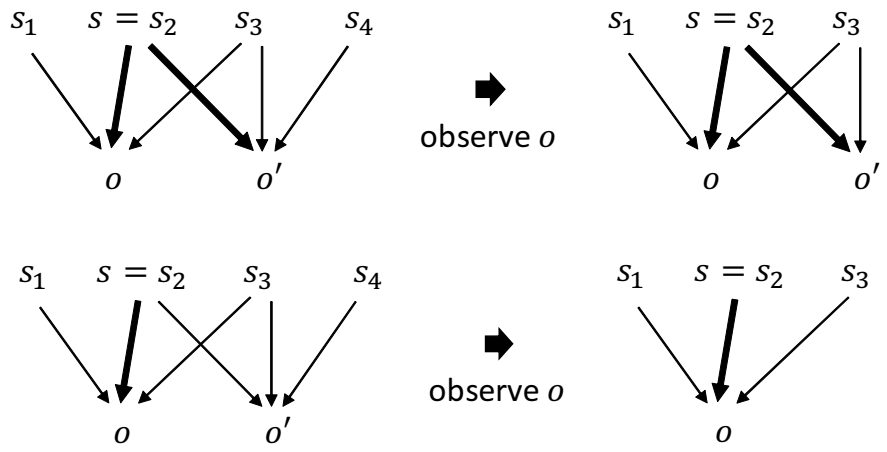

Figure 3.1: QIF_1 (the upper) and QIF_2 (the lower)

which coincides with $\mathrm{QIF}_2(o)$ (Requirement (R3)). By (3.10), Requirement (R1) is satisfied. Also in (3.10), $\mathrm{QIF}_1(o) = \mathrm{QIF}_2(o)$ holds for every $o \in \mathcal{O}$ if and only if the program is deterministic.

We omit the trivial proof for Theorem 3.2.1. Let us get back to the Example 3.1.1 in the previous section to see how new notions convey the intuitive meaning of dynamic leakage. We assume: both *source* and *output* are 8-bit numbers of which values are in 0..255, *source* is uniformly distributed over this range. Then, because the program in this example is deterministic, as mentioned above QIF_1 coincides with QIF_2. We have $\mathrm{QIF}_1(output = 8) = -\log \frac{241}{256} = 0.087 bits$ while $\mathrm{QIF}_1(output = o) = -\log \frac{1}{256} = 8 bits$ for every o between 9 and 23. This result addresses well the problem of failing to differentiate vulnerable output from safe ones of QIF.

Example 3.2.1. Consider the following program taken from Example 1 of [23]:

if $S = s_1$ then $O \leftarrow a$ else $O \leftarrow b$

Assume that the probabilities of inputs are $p(s_1) = 0.875$, $p(s_2) = 0.0625$ and $p(s_3) = 0.0625$. Then, we have the following output and posterior probabilities:

$p(a) = 0.875, p(b) = 0.125$
$p(s_1|a) = 1, p(s_2|a) = p(s_3|a) = 0$

$$p(s_1|b) = 0, p(s_2|b) = p(s_3|b) = 0.5$$

If we use Shannon entropy, $H(S) = 0.67$, $H(S|a) = 0$ and $H(S|b) = 1$. Thus, $\mathrm{QIF}^{dyn}(b) = -0.33$, which is negative as pointed out in [23]. Also, $\mathrm{QIF}_2(a) = -\log p(a) = -\log 0.875 = 0.19$ and $\mathrm{QIF}_2(b) = -\log p(b) = -\log 0.125 = 3$. $\mathrm{QIF}_2(a) < \mathrm{QIF}_2(b)$ reflects the fact that the difference of the posterior and the prior of each input when observing b is larger ($s_1 : 0.875 \to 0$, $s_2, s_3 : 0.0625 \to 0.5$) than observing a ($s_1 : 0.875 \to 1$, $s_2, s_3 : 0.0625 \to 0$).

Since the program is deterministic, $\mathrm{QIF}^{belief}(s, o) = \mathrm{QIF}_1(o) = \mathrm{QIF}_2(o)$.

o	a	b
$\mathrm{QIF}^{dyn}(o)$	0.67	-0.33
$\mathrm{QIF}_2(o)$	0.19	3

Example 3.2.2. The next program is quoted from Example 2 of [23] where $c_1 \ {}_r[]_{1-r} \ c_2$ means that the program chooses c_1 with probability r and c_2 with probability $1 - r$.

if $S = s_1$ then $O \leftarrow a \ {}_{0.81}[]_{0.19} O \leftarrow b$
else $O \leftarrow a \ {}_{0.09}[]_{0.91} O \leftarrow b$

Assume that the probabilities of inputs are $p(s_1) = 0.25$ and $p(s_2) = 0.75$. $(p(a), p(b)) = (0.25, 0.75)\begin{pmatrix} 0.81 & 0.19 \\ 0.09 & 0.91 \end{pmatrix} = (0.27, 0.73)$ and the posterior probabilities are calculated by (3.4) as:

$$p(s_1|a) = 0.75, p(s_2|a) = 0.25$$
$$p(s_1|b) = 0.065, p(s_2|b) = 0.935$$

Let us use Shannon entropy for QIF^{dyn}. As $H(S) = H(S|a) = -0.25 \log 0.25 - 0.75 \log 0.75$, $\mathrm{QIF}^{dyn}(a) = H(S) - H(S|a) = 0$. As already discussed in [23], $\mathrm{QIF}^{dyn}(a) = 0$ though an attacker may think that $S = s_1$ is more probable by observing $O = a$. For each $o \in \{a, b\}$, $\mathrm{QIF}^{belief}(s, o)$ takes different values (one of them is negative) depending on whether $s = s_1$ or s_2 is the secret input. $\mathrm{QIF}_2(a) = -\log p(a) = -\log 0.27 = 1.89$ and $\mathrm{QIF}_2(b) = -\log p(b) = -\log 0.73 = 0.45$. $\mathrm{QIF}_1(a) = \mathrm{QIF}_1(b) = 0$ because the set of possible input values does not shrink whichever a or b is observed. Similarly to Example 3.2.1,

$\mathrm{QIF}_2(a) > \mathrm{QIF}_2(b)$ reflects the fact that the probability of each input when observing a varies more largely ($s_1 : 0.25 \to 0.75$, $s_2 : 0.75 \to 0.25$) than when observing b ($s_1 : 0.25 \to 0.065$, $s_2 : 0.75 \to 0.935$). In this example, the number $|\mathcal{S}|$ of input values is just two, but in general, $|\mathcal{S}|$ is larger and we can expect $|\mathrm{pre}_P(o)|$ is much smaller than $|\mathcal{S}|$ and QIF_1 serves a better measure for dynamic QIF.

o	a	b
$\mathrm{QIF}^{dyn}(o)$	0	0.46
$\mathrm{QIF}^{belief}(s_1, o)$	1.58	-1.94
$\mathrm{QIF}^{belief}(s_2, o)$	-1.58	0.32
$\mathrm{QIF}_1(o)$	0	0
$\mathrm{QIF}_2(o)$	1.89	0.45

A program is *noninterferent* if for every $o \in \mathcal{O}$ such that $p(o) > 0$ and for every $s \in \mathcal{S}$, $p(o) = p(o|s)$. Assume a program P is noninterferent. By (3.4), $p(s) = p(s|o)$ for every $o \in \mathcal{O}$ ($p(o) > 0$) and $s \in \mathcal{S}$, then QIF $= 0$ by (3.11). If P is deterministic in addition, $p(o) = p(o|s) = 1$ for $o \in \mathcal{O}$ ($p(o) > 0$) and $s \in \mathcal{S}$. That is, if a program is deterministic and noninterferent, it has exactly one possible output value.

Relationship to the hybrid monitor Let us see how our notions relate to the knowledge tracking hybrid monitor proposed by Bielova et al. [22].

Example 3.2.3. Consider the following program taken from Program 5 of [22]:

if h then $z \leftarrow x + y$
else $z \leftarrow y$ - x;
output z

where h is a secret input, x and y are public inputs and z is a public output.

In [22], the knowledge about secret input h carried by public output z is $\kappa(z) = \lambda\rho.if([\![h]\!]_\rho, [\![x + y]\!]_\rho, [\![y - x]\!]_\rho)$ where ρ is an initial environment (an assignment of values to h, x and y) and $[\![e]\!]_\rho$ is the evaluation of e in ρ. If $h = 1$, $x = 0$ and $y = 1$, then $z = 1$. In [22], to verify whether this value of z reveals any information about h in this setting of public inputs (i.e., $x = 0$, $y = 1$), they take

$\kappa(z)^{-1}(1) = \{\rho | if([\![h]\!]_\rho, [\![x+y]\!]_\rho, [\![y-x]\!]_\rho) = 1\} = \{\rho | if([\![h]\!]_\rho, 1, 1) = 1\}$. Because $if([\![h]\!]_\rho, 1, 1) = 1$ for every ρ, [22] concluded that $z = 1$ in that setting leaks no information.

On the other hand, with that settings of $x = 0$ and $y = 1$, given $z = 1$ as the observed output, h can be either $true$ or $false$. For the program is deterministic, $\mathrm{QIF}_1(z = 1) = \mathrm{QIF}_2(z = 1) = -\log \sum_{p(s'|o)>0} p(s') = -\log(p(h = true) + p(h = false)) = -\log 1 = 0$, which is consistent with that of [22] though the approach looks different. Actually, the function $\kappa(z)$ encodes all information revealed from a value of z about secret input. By applying $\kappa(z)^{-1}$ for a specific value o of z, we get the pre-image of o. In other words, $\kappa(z)^{-1}(o)$ is exactly what we are getting toward quantifying our notions of dynamic leakage. The monitor proposed in [22] tracks the knowledge about secret input carried by all variables along an execution of a program according to the inlined operational semantics. It seems, however, impractical to store all the knowledge during an execution, and furthermore, it would take time to compute the inverse of the knowledge when an observed output is fixed.

3.2.2 An axiomatic perspective

The three requirements (R1), (R2) and (R3) we presented summarize the intuitions about dynamic leakage following the spirit of [23]. However, those requirements lack a firm back-up theory, whilst in [9] Alvim et al. provide a set of axioms for QIF. Despite there is difference between QIF and dynamic leakage, we investigate in this subsection how well our notions fit in the lens of those axioms to confirm their feasibility to be used as a metric.

A. Preliminaries

This subsection briefly summarizes the background theory of [9] to make this paper self-contained. Following the notation in section 3.2.1, let P be a program with secret input variable S and observable output variable O and let $\mathcal{S}$ and $\mathcal{O}$ denote the sets of input values and output values, respectively. We denote a prior distribution over $\mathcal{S}$ by π just for readability in this subsection. Let $\mathbb{D}\mathcal{X}$ denote the set of all probability distributions over a finite set $\mathcal{X}$. The *prior vulnerability*

$\mathbb{V} : \mathbb{D}\mathcal{S} \to \mathbb{R}$ ($\mathbb{R}$ is the set of reals) is defined based on the type of threat which is considered in the context. For instance, we may define *Bayes* prior vulnerability $\mathbb{V}_b(\pi) = \max_{s \in \mathcal{S}} \pi(s)$.

A hyper-distribution (abbrev. hyper) over a finite set $\mathcal{X}$ is a distribution on distributions on $\mathcal{X}$. Thus, the set of all hypers over $\mathcal{X}$ is $\mathbb{D}(\mathbb{D}\mathcal{X})$, which is abbreviated as $\mathbb{D}^2\mathcal{X}$. A program P transforms a prior distribution $\pi \in \mathbb{D}\mathcal{S}$ to a collection of posterior distributions $p(s|o)$ (as a function that takes $s \in \mathcal{S}$ as an argument) with probability $p(o)$. Hence, P can be regarded as a mapping from $\mathbb{D}\mathcal{S}$ to $\mathbb{D}^2\mathcal{S}$. Then, the *posterior vulnerability* $\widehat{\mathbb{V}} : \mathbb{D}^2\mathcal{S} \to \mathbb{R}$ is defined either as the expected value ($Exp_\Delta \mathbb{V}$) or as the maximum value ($\max_{\lceil \Delta \rceil} \mathbb{V}$) of the prior vulnerability over a hyper $\Delta \in \mathbb{D}^2\mathcal{S}$, in which $\lceil \Delta \rceil$ denotes the set of posterior distribution with non-zero probability. We use $[\pi]$ to denote the point-hyper assigning probability 1 to π, and $[\pi, P]$ to denote the hyper obtained by the action of P on π.

In [9], three axioms for prior vulnerability and three axioms for posterior vulnerability are proposed. For prior vulnerability, Continuity ($CNTY$): $\forall \pi.\mathbb{V}$ is a continuous function of π; Convexity (CVX): $\forall \sum_i a_i\pi^i.\mathbb{V}(\sum_i a_i\pi^i) \leq \sum_i a_i\mathbb{V}(\pi^i)$; and Quasiconvexity ($Q$-$CVX$): $\forall \sum_i a_i\pi^i.\mathbb{V}(\sum_i a_i\pi^i) \leq \max_i \mathbb{V}(\pi^i)$, provided a_i are non-negative real numbers adding up to 1. For posterior vulnerability, Noninterference (NI): $\forall \pi.\widehat{\mathbb{V}}[\pi] = \mathbb{V}(\pi)$; Data-processing inequality (DPI): $\forall \pi, P, Q.\widehat{\mathbb{V}}[\pi, P] \geq \widehat{\mathbb{V}}[\pi, PQ]$; and Monotonicity ($MONO$): $\forall \pi, P : \widehat{\mathbb{V}}[\pi, P] \geq \mathbb{V}(\pi)$, provided P, Q are programs, and PQ denotes the sequential composition of P and Q in this order.

B. Fitting in the lens of axioms

Information leakage, either static or dynamic, is basically defined as the difference between prior vulnerability and posterior vulnerability. Recall QIF_1 and QIF_2 are defined as the reducing amount of self-information of the secret input and the joint event (between secret input and output) respectively. Hence, given secret s, output o, we can regard the prior vulnerabilities for QIF_1 as $\mathbb{V}_1(\pi) = \pi(s)$, for QIF_2 as $\mathbb{V}_2(\pi) = p(s, o)$, and the posterior vulnerabilities in that order $\widehat{\mathbb{V}}_1[\pi, P] = \frac{\pi(s)}{\sum_{s' \in \text{pre}_{P(o)}} \pi(s')}$, $\widehat{\mathbb{V}}_2[\pi, P] = p(s, o|o) = p(s|o)$. In both of the cases, neither are posterior vulnerabilities the maximum value ($\max_{\lceil \Delta \rceil}$) nor the expected value

($Exp_{\Delta}\mathbb{V}$) of prior vulnerabilities over some Δ, which is different from those in [9]. This difference in definition is unavoidable to consider QIF_1 and QIF_2 from the axiomatic perspective because these are dynamic notions.

1. For prior vulnerability, both QIF_1 and QIF_2 satisfy *CNTY*, *CVX* and *Q-CVX*.

2. For posterior vulnerability, we found that QIF_1 satisfies all the three axioms: *NI*, *MONO* and *DPI* whilst QIF_2 satisfies only the first two axioms but the last one, *DPI*. In fact, QIF_2 still aligns well to *DPI* in cases for deterministic programs, and only misses for probabilistic ones. Note that in deterministic cases, $\text{QIF}_1 \equiv \text{QIF}_2$ by Theorem 3.2.1. Hence, for deterministic programs, QIF_2 satisfies *DPI* because QIF_1 does. For it is quite trivial and the space is limited, we will omit the proof of those satisfaction. Instead, we will give a counterexample to show that QIF_2 does not satisfy *DPI* when programs are probabilistic. Let $P_1 : \{s_1, s_2\} \to \{u_1, u_2\}$ and $P_2 : \{u_1, u_2\} \to \{v_1\}$ in which P_2 is a post-process of P_1. Also assume the following probabilities: $\pi_S : p(s_1) = p(s_2) = 0.5; p(u_1|s_1) = 0.1, p(u_2|s_1) = 0.9, p(u_1|s_2) = 0.3, p(u_2|s_2) = 0.7$ and $p(v_1|u_1) = p(v_1|u_2) = 1$, in which s_1, s_2, u_1, u_2, v_1 annotate events that the corresponding variables have those values. Given these settings, in the cases that u_1 and v_1 are the output of P_1 and P_1P_2 respectively, we have $\widehat{\mathbb{V}}_2[\pi_S, P_1] = p(s_1|u_1) = \frac{0.5\times0.1}{0.5\times0.1+0.5\times0.3} = 0.25$, and $\widehat{\mathbb{V}}_2[\pi_S, P_1P_2] = p(s_1|v_1) = \frac{0.5\times0.1+0.5\times0.9}{0.5\times0.1+0.5\times0.9+0.5\times0.3+0.5\times0.7} = 0.5$. In other words, $\widehat{\mathbb{V}}_2[\pi_S, P_1P_2] > \widehat{\mathbb{V}}_2[\pi_S, P_1]$, which is against *DPI*.

It turns out that, provided some unavoidable differences in definition, our proposed notions satisfy all the axioms except *DPI*. We came to the conclusion that *DPI* is not a suitable criterion to verify if a dynamic leakage notion is reasonable. It is because dynamic leakage is about a specific execution path, in which the inequality of *DPI* does no longer make sense, rather than the average on all possible execution paths. Therefore, it is not problematic that QIF_2 does not satisfy *DPI* for probabilistic programs while QIF_2 for deterministic programs and QIF_1 satisfy *DPI*.

3.3 Program model

We assume probabilistic programs where every variable stores a natural number and the syntactical constructs are assignment to a variable, conditional, probabilistic choice, while loop and concatenation:

$$\begin{aligned} b &::= \bot \mid \top \mid \neg b \mid b \vee b \mid e < e \\ e &::= X \mid n \mid e + e \\ c &::= \text{skip} \mid X \leftarrow e \mid \text{if } b \text{ then } c \text{ else } c \text{ end} \\ &\quad \mid c \ {}_r[]_{1-r} \ c \mid \text{while } b \text{ do } c \text{ end} \mid c; c \end{aligned}$$

where $<, X, n, +$ stand for a binary relation on natural numbers, a program variable, a constant natural number and a binary operation on natural numbers, respectively, and r is a constant rational number representing the branching probability for a choice command where $0 \leq r \leq 1$. In the above BNFs, objects derived from the syntactical categories b, e and c are called conditions, expressions and commands, respectively. A command $X \leftarrow e$ assigns the value of expression e to variable X. A command $c_1 \ {}_r[]_{1-r} \ c_2$ means that the program chooses c_1 with probability r and c_2 with probability $1-r$. Note that this is the only probabilistic command. The semantics of the other constructs are defined in the usual way.

A program P has the following syntax:

$$P \ ::= \ \text{in } \vec{S};\ \text{out } \vec{O};\ \text{local } \vec{Z}; c \mid P; P$$

where $\vec{S}, \vec{O}, \vec{Z}$ are sequences of variables which are disjoint from one another. A program is required to satisfy the following constraints on variables. We first define $In(P), Out(P), Local(P)$ for a program P as follows.

- If $P = \text{in } \vec{S};\ \text{out } \vec{O};\ \text{local } \vec{Z}; c$, we define $In(P) = \{V \mid V \text{ appears in } \vec{S}\}$, $Out(P) = \{V \mid V \text{ appears in } \vec{O}\}$ and $Local(P) = \{V \mid V \text{ appears in } \vec{Z}\}$. In this case, we say P is a simple program. We require that no varible in $In(P)$ appears in the left-hand side of an assignment command in P, i.e., any input variable is not updated.

- If $P = P_1; P_2$, we define $In(P) = In(P_1)$, $Out(P) = Out(P_2)$ where we require that $In(P_2) = Out(P_1)$ holds. We also define $Local(P) = Local(P_1) \cup Local(P_2) \cup Out(P_1)$.

A program P is also written as $P(S, O)$ where S and O are enumerations of $In(P)$ and $Out(P)$, respectively. A program $P_1; P_2$ represents the sequential composition of P_1 and P_2. Note that the semantics of $P_1; P_2$ is defined in the same way as that of the concatenation of commands $c_1; c_2$ except that the input and output variables are not always shared by P_1 and P_2 in the sequential composition.

3.4 Quantifying dynamic leakage

Similar to QIF, to calculate dynamic leakage is hard. In this section, we give formal bounds on the hardness of quantifying dynamic leakage. After that, we show a model-counting based calculation followed by some experimental results.

3.4.1 Complexity results

A. Assumption and overview

We define the problems CompQIF1 and CompQIF2 as follows.

- Inputs: a probabilistic Boolean program P, an observed output value $o \in \mathcal{O}$, and a natural number j (in unary) specifying the error bound.
- Problem: Compute $\mathrm{QIF}_1(o)$ (resp. $\mathrm{QIF}_2(o)$) for P and o.

General assumption

(A1) The answer to the problem CompQIF1 (resp. CompQIF2) should be given as a rational number (two integer values representing the numerator and denominator) representing the probability $\sum_{s' \in \mathrm{pre}_P(o)} p(s')$ (resp. $p(o)$).

(A2) If a program is deterministic or non-recursive, the answer should be exact. Otherwise, the answer should be within j bits of precision, i.e., $|$ (the answer)$-$ $\sum_{s' \in \mathrm{pre}_P(o)} p(s')$(resp. $p(o)$) $| \leq 2^{-j}$.

If we assume (A1), we only need to perform additions and multiplications the number of times determined by an analysis of a given program, avoiding the computational difficulty of calculating the exact logarithm. The reason for assuming (A2) is that the exact reachability probability of a recursive program is

not always a rational number even if all the transition probabilities are rational [42, Theorem 3.2].

When we discuss lower-bounds, we consider the corresponding decision problem by adding a candidate answer of the original problem as a part of an input. The results on the complexity of CompQIF1 and CompQIF2 are summarized in Table 3.1. As mentioned above, if a program is deterministic, $\mathrm{QIF}_1 = \mathrm{QIF}_2$. Some proofs for the results on lower bounds showed in Table 3.1 mention about uniformly distributed input. But the results automatically apply for input with arbitrary distribution because the problem for uniformly distributed input is not harder than for arbitrarily distributed input.

Table 3.1: Complexity results

programs	**deterministic**	**probabilistic**	
		CompQIF1	**CompQIF2**
loop-free	PSPACE $\sharp P$-hard (Proposition 3.4.1)	PSPACE (Theorem 3.4.1) $\sharp P$-hard	PSPACE (Theorem 3.4.1) $\sharp P$-hard
while	PSPACE-comp (Proposition 3.4.2)	PSPACE-comp (Theorem 3.4.2)	EXPTIME (Theorem 3.4.3) PSPACE-hard
recursive	EXPTIME-comp (Proposition 3.4.3)	EXPSPACE (Theorem 3.4.4) EXPTIME-hard	EXPSPACE (Theorem 3.4.4) EXPTIME-hard

Recursive Markov chain (abbreviated as RMC) is defined in [42] by assigning a probability to each transition in recursive state machine (abbreviated as RSM) [6]. Probabilistic recursive program in this section is similar to RMC except that there is no program variable in RMC. If we translate a recursive program into an RMC, the number of states of the RMC may become exponential to the number of Boolean variables in the recursive program. In the same sense, deterministic recursive program corresponds to RSM, or equivalently, pushdown systems (PDS) as mentioned and used in [29]. Also, probabilistic while program corresponds to Markov chain. We will review the definition of RMC in Section 3.4.1.

B. Deterministic case

We first show lower bounds for deterministic loop-free, while and recursive programs. For deterministic recursive programs, we give EXPTIME upper bound as a corollary of Theorem 3.4.4.

Proposition 3.4.1. CompQIF1(= CompQIF2) is $\sharp P$-hard for deterministic loop-free programs even if the input values are uniformly distributed. [1]
(Proof) $\sharp$SAT is a $\sharp P$-hard problem. We show that $\sharp$SAT can be reduced to CompQIF1 where the input values are uniformly distributed. It is necessary and sufficient for CompQIF1 to compute the number of inputs $\vec{s}$ such that $p(\vec{s}|\vec{o}) > 0$ because $\sum_{p(\vec{s}|\vec{o})>0} p(\vec{s}) = |\{\vec{s} \in \vec{\mathcal{S}} \mid p(\vec{s}|\vec{o}) > 0\}|/|\vec{\mathcal{S}}|$. For a given propositional logic formula ϕ with Boolean variables $\vec{S}$, we just construct a loop-free program P with input variables $\vec{S}$ and an output variable O such that the value of ϕ for $\vec{S}$ is stored to O. Then, the result of CompQIF1 with P and $o = \top$ coincides with the number of models of ϕ. □

Proposition 3.4.2. CompQIF1(= CompQIF2) is PSPACE-hard for deterministic while programs. [2]
(Proof) Quantified Boolean formula problem (QBF) is PSPACE-complete. The proposition can be shown in the same way as the proof of PSPACE-hardness of the noninterference problem for deterministic while programs by a reduction from QBF validity problem given in [29] as follows. For a given QBF φ, we construct a deterministic while program P having one output variable such that P is noninterferent if and only if φ is valid as in the proof of Proposition 19 of [29]. The deterministic program is noninterferent if and only if the output of the program is always $\top$, i.e., $p(\top) = 1$. Thus, we can decide if ϕ is valid by checking whether $p(\top) = 1$ or not for the deterministic program, the output value $\top$, and the probability 1. □

Proposition 3.4.3. CompQIF1(= CompQIF2) is EXPTIME-complete for deterministic recursive programs.

[1] The upper bound for deterministic loop-free programs is inferred as the upper bound for probabilistic loop-free programs.

[2] The upper bound for deterministic while programs is inferred as the upper bound for probabilistic while programs.

(Proof) EXPTIME upper bound can be shown by translating a given program to a pushdown system (PDS). Assume we are given a deterministic recursive program P and an output value $o \in \mathcal{O}$. We apply to P the translation to a recursive Markov chain (RMC) A in the proof of Theorem 3.4.4. The size of A is exponential to the size of P. Because P is deterministic, A is also deterministic; A is just a recursive state machine (RSM) or equivalently, a PDS. It is well-known [28] that the pre-image of a configuration c of a PDS A $\mathrm{pre}_A(c) = \{c' \mid c' \text{ is reachable to } c \text{ in } A\}$ can be computed in polynomial time by so-called P-automaton construction. Hence, by specifying configurations outputting o as c, we can compute $\mathrm{pre}_P(o) = \mathrm{pre}_A(c)$ in exponential time.

The lower bound can be shown in the same way as the EXPTIME-hardness proof of the noninterference problem for deterministic recursive programs by a reduction from the membership problem for polynomial space-bounded alternating Turing machines (ATM) given in the proof of Theorem 7 of [29]. From a given polynomial space-bounded ATM M and an input word w to M, we construct a deterministic recursive program P having one output variable such that P is noninterferent if and only if M accepts w as in [29]. As in the proof of Proposition 3.4.2, we can reduce to CompQIF1 instead of reducing to the noninterference problem. □

C. Probabilistic case

C-1. Loop-free programs

We show upper bounds for loop-free programs. For CompQIF2, the basic idea is similar to the one in [29], but we have to compute the conditional probability $p(\vec{o}|\vec{s})$. For CompQIF1, $\sharp P^{NP}$ upper bound can be obtained by a similar result on model counting if the input values are uniformly distributed.

Theorem 3.4.1. CompQIF1 and CompQIF2 are solvable in PSPACE for probabilistic loop-free programs. CompQIF1 is solvable in $\sharp P^{NP}$ if the input values are uniformly distributed.

(Proof) We first show that CompQIF2 is solvable in PSPACE for probabilistic loop-free programs. If a program is loop-free, we can compute $p(\vec{o}|\vec{s})$ for every $\vec{s}$ in the same way as in [29], multiply it by $p(\vec{s})$ and sum up in PSPACE. Note

that in [29], it is assumed that a program is deterministic and input values are uniformly distributed, and hence it suffices to count the input values $\vec{s}$ such that $p(\vec{o}|\vec{s}) = 1$, which can be done in P^{CH3}. In contrast, we have to compute the sum of the probabilities of $p(\vec{s})p(\vec{o}|\vec{s})$ for all $\vec{s} \in \vec{\mathcal{S}}$. We can easily see that CompQIF1 is solvable in PSPACE for probabilistic loop-free programs in almost the same way as CompQIF2. Instead of summing up $p(\vec{s})p(\vec{o}|\vec{s})$ for all $\vec{s} \in \vec{\mathcal{S}}$, we just have to sum up $p(\vec{s})$ for all $\vec{s} \in \vec{\mathcal{S}}$ such that $p(\vec{o}|\vec{s}) > 0$ (if and only if $p(\vec{s}|\vec{o}) > 0$).

Next, we show that CompQIF1 is solvable in $\sharp P^{NP}$ if the input values are uniformly distributed. As stated in the proof of Proposition 3.4.1, in this case, CompQIF1 can be solved by computing the number of inputs s such that $p(\vec{s}|\vec{o}) > 0$. Deciding $p(\vec{s}|\vec{o}) > 0$ for a given probabilistic loop-free program P can be reduced to the satisfiability problem of a propositional logic formula. Note that for any probabilistic choice like $X \leftarrow c_1\ {}_r[]_{1-r}\ c_2$ with $0 < r < 1$, we just have to treat it as a non-deterministic choice like $X = c_1$ *or* $X = c_2$ because all we need to know is whether $p(\vec{s}|\vec{o}) > 0$. We construct from P a formula ϕ with Boolean variable corresponding to input and output variables of P and intermediate variables. Here, we abuse the symbols $\vec{S}$ and $\vec{O}$, which are used for the variables of P, also as the Boolean variables corresponding to them, respectively. The formula ϕ is constructed such that $\phi \wedge \vec{S} = \vec{s} \wedge \vec{O} = \vec{o}$ is satisfiable if and only if $p(\vec{s}|\vec{o}) > 0$ for $\vec{s}$ and $\vec{o}$. Thus, the number of inputs $\vec{s}$ such that $p(\vec{s}|\vec{o}) > 0$ is the number of truth assignments for $\vec{S}$ such that $\phi \wedge \vec{O} = \vec{o}$ is satisfiable, i.e., the number of projected models on $\vec{S}$. This counting can be done in $\sharp P^{NP}$ because projected model counting is in $\sharp P^{NP}$ [14]. □

C-2. While programs

We show upper bounds for while programs. For CompQIF1, we reduce the problem to the reachability problem of a graph representing the state reachability relation. An upper bound for CompQIF2 will be obtained as a corollary of Theorem 3.4.4.

Theorem 3.4.2. CompQIF1 is PSPACE-complete for probabilistic while programs.

(Proof) It suffices to show that QIF_1 is solvable in PSPACE for probabilistic while programs. QIF1 for probabilistic while programs is reduced to the reachability

problem of graphs that represents the reachability among states of P. We construct a directed graph G from a given program P as follows. Each node (l, σ) on G uniquely corresponds to a location l on P and an assignment σ for all variables in P. An edge from (l, σ) to (l', σ') represents that if the program is running at l with σ then, with probability greater than 0, it can transit to l' with σ' by executing the command at l. Deciding the reachability from a node to another node can be done in nondeterministic log space of the size of the graph. The size of the graph is exponential to the size of P due to exponentially many assignments for variables. We see that $p(\vec{s}|\vec{o}) > 0$ if and only if there are two nodes (l_s, σ_s) and (l_o, ρ_o) such that l_s is the initial location, l_o is an end location, $\sigma_s(\vec{S}) = \vec{s}$, $\sigma_o(\vec{O}) = \vec{o}$, and (l_o, ρ_o) is reachable from (l_s, σ_s) in G. Thus, $p(\vec{s}|\vec{o}) > 0$ can be decided in PSPACE, and also $\sum_{p(\vec{s}|\vec{o})>0} p(\vec{s})$ can be computed in PSPACE. □

Theorem 3.4.3. CompQIF2 is solvable in EXPTIME for probabilistic while programs.
(Proof) We postpone the proof until we show the result on recursive programs.

C-3. Recursive programs

As noticed in the end of Section 3.3, we will use recursive Markov chain (RMC) to give upper bounds of the complexity of CompQIF1 and CompQIF2 for recursive programs because RMC has both probability and recursion and the complexity of the reachability probability problem for RMC was already investigated in [42].

Recursive Markov chains

A *recursive Markov chain* (RMC) [42] is a tuple $A = (A_1, \ldots, A_k)$ where each $A_i = (N_i, B_i, Y_i, En_i, Ex_i, \delta_i)$ $(1 \leq i \leq k)$ is a *component graph* (or simply, component) consisting of:

- a finite set N_i of *nodes*,
- a set $En_i \subseteq N_i$ of *entry nodes*, and a set $Ex_i \subseteq N_i$ of *exit nodes*,
- a set B_i of *boxes*, and a mapping $Y_i : B_i \to \{1, \ldots, k\}$ from boxes to (the indices of) components. To each box $b \in B_i$, a set of *call sites* $Call_b =$

$\{(b, en) \mid en \in En_{Y_i(b)}\}$ and a set of *return sites* $Ret_b = \{(b, ex) \mid ex \in Ex_{Y_i(b)}\}$ are associated.

- δ_i is a finite set of *transitions* of the form $(u, p_{u,v}, v)$ where
  - the source u is either a non-exit node $u \in N_i \backslash Ex_i$ or a return site.
  - the destination v is either a non-entry node $v \in N_i \backslash En_i$ or a call site.
  - $p_{u,v} \in \mathbb{Q}$ is a rational number between 0 and 1 representing the transition probability from u to v. We require for each source u, $\sum_{\{v' \mid (u, p_{u,v'}, v') \in \delta_i\}} p_{u,v'} = 1$. We write $u \stackrel{p_{u,v}}{\rightarrow} v$ instead of $(u, p_{u,v}, v)$ for readability. Also we abbreviate $u \stackrel{1}{\rightarrow} v$ as $u \rightarrow v$.

Intuitively, a box b with $Y_i(b) = j$ denotes an invocation of component j from component i. There may be more than one entry node and exit node in a component. A call site (b, en) specifies the entry node from which the execution starts when called from the box b. A return site has a similar role to specify the exit node.

Let $Q_i = N_i \cup \bigcup_{b \in B_i} (Call_b \cup Ret_b)$, which is called the set of *locations* of A_i. We also let $N = \bigcup_{1 \leq i \leq k} N_i$, $B = \bigcup_{1 \leq i \leq k} B_i$, $Y = \bigcup_{1 \leq i \leq k} Y_i$ where $Y : B \rightarrow \{1, \ldots, k\}$, $\delta = \bigcup_{1 \leq i \leq k} \delta_i$ and $Q = \bigcup_{1 \leq i \leq k} Q_i$.

The probability $p_{u,v}$ of a transition $u \stackrel{p_{u,v}}{\rightarrow} v$ is a rational number represented by a pair of non-negative integers, the numerator and denominator. The size of $p_{u,v}$ is the sum of the numbers of bits of these two integers, which is called the *bit complexity* of $p_{u,v}$.

The semantics of an RMC A is given by the global (infinite state) Markov chain $M_A = (V, \Delta)$ induced from A where $V = B^* \times Q$ is the set of global states and Δ is the smallest set of transitions satisfying the following conditions:

(1) For every $u \in Q$, $(\varepsilon, u) \in V$ where ε is the empty string.

(2) If $(\alpha, u) \in V$ and $u \stackrel{p_{u,v}}{\rightarrow} v \in \delta$, then $(\alpha, v) \in V$ and $(\alpha, u) \stackrel{p_{u,v}}{\rightarrow} (\alpha, v) \in \Delta$.

(3) If $(\alpha, (b, en)) \in V$ with $(b, en) \in Call_b$, then $(\alpha b, en) \in V$ and $(\alpha, (b, en)) \rightarrow (\alpha b, en) \in \Delta$.

(4) If $(\alpha b, ex) \in V$ with $(b, ex) \in Ret_b$, then $(\alpha, (b, ex)) \in V$ and $(\alpha b, ex) \rightarrow (\alpha, (b, ex)) \in \Delta$.

Intuitively, (α, u) is the global state where u is a current location and α is a pushdown stack, which is a sequence of box names where the right-end is the stack top. (2) defines a transition within a component. (3) defines a procedure call from a call site (b, en); the box name b is pushed to the current stack α and the location is changed to en. (4) defines a return from a procedure; the box name b at the stack top is popped and the location becomes the return site (b, ex). For a location $u \in Q_i$ and an exit node $ex \in Ex_i$ in the same component A_i, let $q^*_{(u,ex)}$ denote the probability of reaching (ε, ex) starting from (ε, u) [3]. Also, let $q^*_u = \sum_{ex \in Ex_i} q^*_{(u,ex)}$. The reachability probability problem for RMCs is the one to compute $q^*_{(u,ex)}$ within j bits of precision for a given RMC A, a location u and an exit node ex in the same component of A and a natural number j in unary.

The following property is shown in [42].

Proposition 3.4.4. The reachability probability problem for RMCs can be solved in PSPACE. Actually, $q^*_{(u,ex)}$ can be computed for every pair of u and ex simultaneously in PSPACE by calculating the least fixpoint of the nonlinear polynomial equations induced from a given RMC. □

Results

Theorem 3.4.4. CompQIF1 and CompQIF2 are solvable in EXPSPACE for probabilistic recursive programs.

(Proof) We will prove the theorem by translating a given program P into a recursive Markov chain (RMC) whose size is exponential to the size of P. By Proposition 3.4.4, we obtain EXPSPACE upper bound. Because an RMC has no program variable, we expand Boolean variables in P to all (reachable) truth-value assignments to them. A while command is translated into two transitions; one for exit and the other for while-body. A procedure call is translated into a box and transitions connecting to/from the box. For the other commands, the translation is straightforward.

Let $P = (\pi_1, \ldots, \pi_k)$ be a given program. For $1 \leq i \leq k$, let $Val(\pi_i)$ be the set of truth value assignments to $Var(\pi_i)$. We will use the same notation $Val(e)$ and $Val(c)$ for an expression e and a command c. For an expression e

[3] Though we usually want to know $q^*_{(en,ex)}$ for an entry node en, the reachability probability is defined in a slightly more general way.

and an assignment $\theta \in Val(e)$, we write $e\theta$ to denote the truth value obtained by evaluating e under the assignment θ. For an assignment θ and a truth value c, let $\theta[X \leftarrow c]$ denote the assignment identical to θ except $\theta[X \leftarrow c](X) = c$. We use the same notation for sequences of variables $\vec{X}$ and truth values $\vec{c}$ as $\theta[\vec{X} \leftarrow \vec{c}]$.

We construct the RMC $A = (A_1, \ldots, A_k)$ from P where each component graph $A_i = (N_i, B_i, Y_i, En_i, Ex_i, \delta_i)$ $(1 \leq i \leq k)$ is constructed from $\pi_i =$ in $\vec{X}$; out $\vec{Y}$; local $\vec{Z}$; c_i as follows.

- $En_i = \{(c_i, \theta) \mid \theta \in Val(\pi_i)$ where $\theta(W)$ is arbitrary for $W \in \vec{X}$ and $\theta(W) = \bot$ for $W \in \vec{Y} \cup \vec{Z}\}$.
- $Ex_i = \{\sigma \mid \sigma$ is an assignment to $\vec{Y}\}$.
- N_i, B_i, Y_i and δ_i are constructed as follows.

(1) $N_i \leftarrow En_i$, $B_i \leftarrow \emptyset$, $Y_i \leftarrow$ the function undefined everywhere, $\delta_i \leftarrow \{(\text{skip}, \theta) \rightarrow \theta|_{\vec{Y}} \mid \theta \in Val(\pi_i)\}$ where $\theta|_{\vec{Y}}$ is the restriction of θ to $\vec{Y}$. Note that $\theta|_{\vec{Y}} \in Ex_i$.

(2) Repeat the following construction until all the elements in N_i are marked: Choose an unmarked (c, θ) from N_i, mark it and do one of the followings according to the syntax of c.

(i) $c = X \leftarrow e; c'$. Add $(c', \theta[X \leftarrow e\theta])$ to N_i and add $(c, \theta) \rightarrow (c', \theta[X \leftarrow e\theta])$ to δ_i.

(ii) $c =$ if e then c_1 else c_2 end; c'. Add $(c_1; c', \theta)$ to N_i and add $(c, \theta) \rightarrow (c_1; c', \theta)$ to δ_i if $e\theta = \top$. Add $(c_2; c', \theta)$ to N_i and add $(c, \theta) \rightarrow (c_2; c', \theta)$ to δ_i if $e\theta = \bot$.

(iii) $c = c_1 \ {}_r[]_{1-r} \ c_2; c'$. Add $(c_1; c', \theta)$ and $(c_2; c', \theta)$ to N_i. Add $(c, \theta) \xrightarrow{r} (c_1; c', \theta)$ and $(c, \theta) \xrightarrow{1-r} (c_2; c', \theta)$ to δ_i.

(iv) $c =$ while e do c_1 end; c'. Add (c', θ) to N_i and add $(c, \theta) \rightarrow (c', \theta)$ to δ_i if $e\theta = \bot$. Add $(c_1; c, \theta)$ to N_i and add $(c, \theta) \rightarrow (c_1; c, \theta)$ to δ_i if $e\theta = \top$.

(v) $c = \pi_j(\vec{e'}; \vec{X'}); c'$ where $\pi_j =$ in $\vec{X''}$; out $\vec{Y''}$; local $\vec{Z''}$; c_j. Add a new box b to B_i. Define $Y_i(b) = j$.

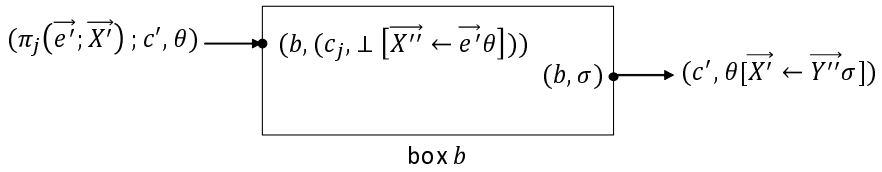

Figure 3.2: Construction of an RMC from a recursive program

Add $(c, \theta) \to (b, (c_j, \perp[\vec{X''} \leftarrow \vec{e'}\theta]))$ to δ_i where the assignment $\perp$ denotes the one that assigns $\perp$ to every variable. For every $\sigma \in Ex_j$,

add $(c', \theta[\vec{X'} \leftarrow \vec{Y''}\sigma])$ to N_i and add $(b, \sigma) \to (c', \theta[\vec{X'} \leftarrow \vec{Y''}\sigma])$ to δ_i (see Figure 3.2).

The number $|Q|$ of locations of the constructed RMC A is exponential to the size of P. More precisely, $|Q|$ is in the order of the number of commands in P multiplied by 2^N where N is the maximum number of variables appearing in a procedure of P because we construct locations of A by expanding each variable to two truth values. Recall that both $\mathrm{QIF}_1(o)$ and $\mathrm{QIF}_2(o)$ can be computed by calculating $p(o|s')$ for each $s' \in \mathcal{S}$, i.e., the reachability probability from s' to o. By Proposition 3.4.4, the reachability probability problem for RMCs are in PSPACE, and hence CompQIF1 and CompQIF2 are solvable in EXPSPACE. □

Proof of Theorem 3.4.3

Let P be a given probabilistic while program and $o \in \mathcal{O}$ is an output value. Our algorithm works as follows.

1. Compute $\mathrm{pre}_P(o)$.
2. Calculate $\sum_{s' \in \mathcal{S}} p(s')p(o|s')$ (see (3.9)).

In the proof of Theorem 3.4.4, a given program P is translated into a recursive Markov chain A whose size is exponential to the size of P. If a given program P is a while program, A is an ordinary (non-recursive) Markov chain. The constraint on the stationary distribution vector of A is represented by a system of linear equations whose size is polynomial of the size of A (see [66] for example) and

the system of equations can be solved in polynomial time. Hence CompQIF2 is solvable in EXPTIME. □

3.4.2 Model counting-based computation of dynamic leakage

In the previous section, we show that the problems of calculating dynamic leakage, i.e., CompQIF1 and CompQIF2, are computationally hard. We still, however, propose a solution to these problems by reducing them to model counting problems.

Reduction to model counting Model counting is a well-known and powerful technique in quantitative software analysis and verification including QIF analysis. In existing studies, QIF calculation has been reduced to model counting of a logical formula using SAT solver [53] or SMT solver [75]. Similarly, we are showing that it is possible to reduce CompQIF1 and CompQIF2 to model counting in some reasonable assumptions. Let us consider what is needed to compute based on their definitions (3.7) and (3.9), i.e., $\mathrm{QIF}_1 = -\log(\sum_{p(s'|o)>0} p(s'))$ and $\mathrm{QIF}_2 = -\log(\sum_{s'\in S} p(s')p(o|s'))$.

For calculating QIF_1 for a given output value o, it suffices (1) to enumerate input values s' that satisfy $p(s'|o) > 0$ (i.e., possible to produce o), and (2) to sum the prior probabilities over the enumerated input values s'. (2) can be computed from the prior probability distribution of input values, which is reasonable to assume. When input values are uniformly distributed, only step (1) is needed because QIF_1 is simplified to $\log \frac{|\mathcal{S}|}{|\mathrm{pre}_P(o)|}$ by Theorem 3.2.1.

Let us consider QIF_2. For *deterministic* programs, $\mathrm{QIF}_1 = \mathrm{QIF}_2$ holds (Theorem 3.2.1). For *probabilistic* programs, we need to compute the conditional probability $p(o|s')$ for each s', meaning that we have to examine all possible execution paths. We would leave CompQIF2 for probabilistic programs as future work.

Given a program P together with its prior probability distribution on input, and an observed output o, all we need for CompQIF1 and CompQIF2 (deterministic case for the latter) is the enumeration of $\mathrm{pre}_P(o)$, the input values consistent with o. Also, we can forget the probability of a choice command and regard it just as a

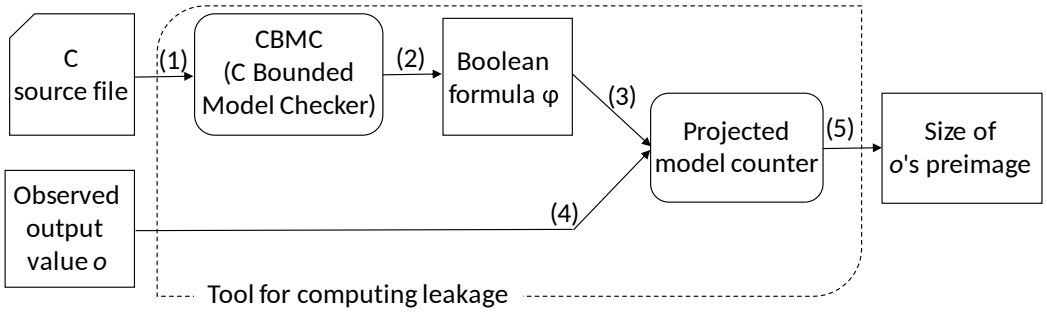

Figure 3.3: Reduction of computing dynamic leakage to model counting

nondeterministic choice. Especially when input values are uniformly distributed, only the number of elements of $\text{pre}_P(o)$ is needed.

In the remainder of Section 4, we assume input values are uniformly distributed for simplicity. Figure 3.3 illustrates the calculation flow using model counting. The basic idea is similar to other existing QIF analysis tools based on model counting, namely, (1) feeding a target C program into CBMC; (2) getting a Boolean formula φ equivalent to the source program in terms of constraints among variables in the program; (3) feeding φ into a projected model counter that can count the models with respect to projection on variables of interest; and (5) getting the result. The only difference of this framework from existing ones is (4), augmenting information about an observed output value o into the Boolean formula φ so that each model corresponds to an input value which produces o. The set of the obtained models is exactly the pre-image of o.

Pluggability There are several parts in the framework above that can be flexibly changed to utilize the strength of different tools and/or approaches. Firstly, the *projected model counter* at (5) could be either a projected ♯SAT solver (e.g., SharpCDCL) or a ♯SMT solver (e.g., aZ3). Consequently, a formula at (3) could be either a SAT constraints (e.g., a Boolean formula in DIMACS format) or a SMT constraints (e.g., a formula in SMTLIB format) generated by CBMC. Moreover, this framework can be extended to different programming languages other than C, such as Java, having JPF [88] and KEY [39] as two well-known counterparts of CBMC. In the next section, we are showing experimental results in

which we tried several set-ups of tools in this framework to observe the differences.

3.4.3 Experiments

We conducted some experiments to investigate the flexibility of the framework to reduce computing dynamic leakage to model counting introduced in the previous section, as well as the scalability of this method. For the simplicity to achieve this purpose, we restricted to calculate dynamic leakage for *deterministic programs with uniformly distributed input*. Toward the analysis in more general cases and possibilities on performance improvement, we give some discussion and leave it as one of future work.

A. Overview

All experiments were done in a same PC with the following specification: core i7-6500U, CPU@2.5GHz x 4, 8GB RAM, Ubuntu 18.04 64 bits. We set one hour as time-out and interrupted execution whenever the running time exceeds this duration. The model counters we used are described below.

- *aZ3*: a ♯SMT solver developed by Phan et al. [75], which is built on top of the state-of-the-art SMT solver Z3. We used an improved version of aZ3 which is developed by Nakashima et al. [70]. It allows specifying variables of interest, which is equivalent to projection in SAT-based model counter.

- *SharpCDCL*: a ♯SAT solver with capability of projected counting based on Conflict-Driven Clause Learning (CDCL) [4]. The tool finds a new projected model and then adds a clause blocking to find the same model again. It enumerates all projected models by repeating that.

- *DSharp-p*: another ♯SAT solver based on d-DNNF format [68]. The tool first translates a given formula into d-DNNF format. It is known that, once given a d-DNNF format of constraints, it takes only linear time to the size of the formula to count models of those constraints. We used an extended version with the capability of projected counting which is added by Klebanov et al. [53].

[4] `http://tools.computational-logic.org/content/sharpCDCL.php`

- *GPMC*: a projected model counter built on top of the SAT solver glucose [5], in which component analysis and caching used in the model counter SharpSAT are implemented [82].

The benchmarks are taken from previous researches about QIF analysis with most of them are taken from benchmarks of aZ3 [75], except *bin_search32.c* which is taken from [64]. The difference between the ordinary QIF analysis and dynamic leakage quantification is that the former is not interested in an observed output value, but the latter is. Therefore, for the purpose of these experiments, we augmented the original benchmarks with additional information about concrete values of public output (public input also if there is some). Because we assume deterministic programs with uniformly distributed input, $\mathrm{QIF}_1(o) = \mathrm{QIF}_2(o) = \log \frac{|\mathcal{S}|}{|\mathrm{pre}_P(o)|}$ by Theorem 3.2.1. Hence, without loss of precision in comparison, we consider counting $\mathrm{pre}_P(o)$ as the final goal of these experiments.

B. Results

Table 3.2 shows execution time of model counting based on the four different model counters, in which *t/o* indicates that the experiment was interrupted because of time-out and - means the counter gave a wrong answer (i.e., only DSharp-p miscounted for UNSAT cases, probably because the tool does assume input formula to be satisfiable). By eliminating parsing time from the comparison, we measured only time needed to count models.

According to the experimental results, aZ3-based model counting did not win the fastest for any benchmark, and moreover its execution time is always at least ten times slower than the best. On the other hand, DSharp-p seems to take much time to translate formulas into d-DNNF format for *dining6/50.c* and *grade.c*, and gave wrong answers for *mix_duplicate.c* and *sanity_check.c*, the number of models of which are 0, i.e., unsatisfiable. By and large, aZ3 and DSharp-p can hardly take advantage to the other tools, SharpCDCL and GPMC, in dynamic leakage quantification. Though SharpCDCL won 8 out of 14 benchmarks, the difference between the tools in those cases are not significant, yet the execution times are too short that it can be fluctuated by insignificant parameters. Therefore, it is

[5] `https://www.labri.fr/perso/lsimon/glucose`

better to look at long run benchmarks, *grade.c* and *masked_copy.c*. In both cases, GPMC won by 9.7 times and 287.9 times respectively. The execution times as well as the difference in those two cases are significant. We also noticed that, those two cases have 65,536 and 65 models, which are the two biggest counts among the benchmarks. The more the number of the models is, the bigger is the difference between execution times of GPMC and SharpCDCL. Hence, we can empirically conclude that GPMC-based works much better than SharpCDCL-based in cases the number of models is large, while not so worse in other cases.

By implementing the prototype, we reaffirmed the possibility of automatically computing QIF_1 and QIF_2. Speaking of scalability, despite of small LOC (Lines of Code), there is still the case of *grade.c* (48 lines) for which all settings take longer than one minute, a very long time from the viewpoint of runtime analysis, to count models. There are several directions to improve the current performance which we leave as one of future work. First, because dynamic leakage should be calculated repeatedly for different observed outputs but a same program, we can leverage such an advantage of d-DNNF that while transforming to a d-DNNF format takes time, the model counting can be done in linear time once a d-DNNF format is obtained. That is, we generate merely once in advance a d-DNNF format of the constraints representing the program under analysis, then each time an observed output value is given, we make only small modification and count models in linear time to the size of the constraints. The difficulty of this direction lies in how to augment the information of observed output to the generated d-DNNF without breaking its d-DNNF structure. Another direction is to loose the required precision to accept approximate count. This could be done by counting on existing approximate model counters.

C. Toward general cases

In order to calculate QIF_1 and QIF_2 for a probabilistic program with a non-uniform input distribution, we must identify projected models of the Boolean formula, rather than the number of the models, to obtain the probabilities determined by them in general. GPMC, specialized for model counting, does not compute the whole part of each model explicitly. Hence, GPMC is not appro-

Table 3.2: Counting result and execution time (ms) of different settings

Benchmark	Count	aZ3	SharpCDCL	DSharp-p	GPMC
bin_search32	1	781	37	52	**9**
crc8	32	303	**11**	31	36
crc32	8	294	**8**	32	32
dining6	6	1,305	**44**	*t/o*	49
dining50	50	*t/o*	199	*t/o*	**193**
electronic_purse	5	525	**137**	9,909	223
grade	65	2,705,934	910,655	*t/o*	**93,445**
implicit_flow	1	253	**15**	31	33
masked_copy	65,536	*t/o*	9,214	**30**	32
mix_duplicate	0	241	12	-	**4**
population_count	32	477	**19**	37	34
sanity_check	0	247	13	-	**8**
sum_query	3	310	**20**	31	35
ten_random_outputs	1	249	**18**	32	34

priate for a calculation of the probability depending on the concrete models. On the other hand, sharpCDCL basically enumerates all projected models, and thus we think we can extend it as follows to compute QIF_1 and QIF_2 in general cases.

To calculate QIF_1 for a probabilistic program with a non-uniform input distribution, we can replace each probabilistic choice in a give program with a non-deterministic choice as stated in Section 4, and then enumerate projected models with respect to the input variables, summing up the probabilities of the corresponding input values.

As for QIF_2, we have to calculate not only the probabilities of possible input values but also those of possible execution paths reachable to the observed output. To achieve this, in addition to the replacement of probabilistic choices with non-deterministic choices, we may insert variables to remember which branch is chosen at each of the non-deterministic choices. Then, given a projected model of the Boolean formula generated from the modified source code with respect to the

input variables and the additional choice variables, we can get to know a possible input value and an execution path from the projected model. For a possible input value s, $p(o|s)$ is the sum of the probabilities of all possible execution paths from s to the observed o.

3.5 On the compositionality of dynamic leakage

Aiming at speeding up the model counting-based calculation of dynamic leakage, we proposed a "divide and conquer" fashioned method. The dynamic leakage of the original program will be calculated from the leakage of constituted subprograms, sequentially or in parallel or a combination of both.

3.5.1 Sequential composition

This section proposes a method of computing both exact and approximated dynamic leakage by using sequential composition. For making the idea behind the proposed method understandable, we first assume the programs under analysis are deterministic with uniformly distributed input, so that the problem of quantifying dynamic leakage is reduced to model counting. Then, in **"C. Extensibility to Probabilistic Programs"** that appears later in this section, we will discuss the extensibility of the proposed method to probabilistic programs with input of an arbitrary distribution when the distribution is known.

A. Exact calculation

For a program $P(S, O)$, an input value $s \in \mathcal{S}$ and a subset $\mathcal{S}'$ of input values, let

$$\begin{aligned} \mathrm{post}_P(s) &= \{o \mid p(o|s) > 0\}, \\ \mathrm{post}_P(\mathcal{S}') &= \bigcup_{s \in \mathcal{S}'} \mathrm{post}_P(s). \end{aligned}$$

If P is deterministic and $\mathrm{post}_P(s) = \{o\}$, we write $\mathrm{post}_P(s) = o$.

Let $P = P_1; P_2$ be a program. We assume that $In(P_1), Out(P_1), In(P_2), Out(P_2)$ are all singleton sets for simplicity. This assumption does not lose generality; for example, if $In(P_1)$ contains more than one variables, we instead introduce a new input variable that stores the tuple consisting of a value of each

variable in $In(P_1)$. Let $In(P) = In(P_1) = \{S\}$, $Out(P_1) = In(P_2) = \{T\}$, $Out(P) = Out(P_2) = \{O\}$, and let $\mathcal{S}, \mathcal{T}, \mathcal{O}$ be the corresponding sets of values, respectively. For a given $o \in \mathcal{O}$, $\mathrm{pre}_P(o)$ and $p(o)$, which are needed to compute $\mathrm{QIF}_1^P(o)$ and $\mathrm{QIF}_2^P(o)$ (see (3.7) and (3.9)), can be represented in terms of those of P_1 and P_2 as follows.

$$\mathrm{pre}_P(o) = \bigcup_{t \in (\mathrm{pre}_{P_2}(o) \cap \mathrm{post}_{P_1}(\mathcal{S}))} \mathrm{pre}_{P_1}(t), \tag{3.14}$$

$$p(o) = \sum_{s \in \mathcal{S}, t \in \mathcal{T}} p(s) p_1(t|s) p_2(o|t). \tag{3.15}$$

If $p(s)$ is given, we can compute (3.14) by enumerating $\mathrm{pre}_{P_1}(t)$ for $t \in (\mathrm{pre}_{P_2}(o) \cap \mathrm{post}_{P_1}(\mathcal{S}))$ and also for (3.15). In practice, $\mathrm{pre}_P(o)$ is computed by augmenting the information about an observed output value to the CNF that represents P, as illustrated in the flow of *CiA* and *CoD* in Figure 4.2. Then, the preimage can be either enumerated or counted, up to what is needed for the calculation. This approach can easily be generalized to the sequential composition of more than two programs, in which the enumeration is proceeded in a Breadth-First-Search fashion. However, in this approach, search space will often explode rapidly and lose the advantage of composition. Therefore, we come up with an approximation, which is explained in the next subsection, as an alternative.

B. Approximation

Let us assume that $P(S, O)$ is deterministic and S is uniformly distributed. In this subsection, we will derive both upper-bound and lower-bound of $|\mathrm{pre}_P(o)|$, which provide lower-bound and upper-bound of $\mathrm{QIF}_1^P(o) = \mathrm{QIF}_2^P(o)$ respectively. In general, our method can be applied to the sequential composition of more than two sub-programs.

Lower bound

To infer a lower bound of $|\mathrm{pre}_P(o)|$, we leverage Depth-First-Search (DFS) with a predefined timeout such that the algorithm will stop when the execution time exceeds the timeout and output the current result as the lower bound. The method is illustrated in Algorithm 3. For a program $P = P_1; P_2; \cdots; P_n$, an

observable output o of the last sub-program P_n and a predetermined *timeout*, the Algorithm 3 derives a lower bound of $|\mathrm{pre}_P(o)|$ by those n sub-programs. In

Algorithm 3 LowerBound($P_1, \cdots, P_n, o$, timeout)

```
 1: Pre[2..n] ← empty
 2: Stack ← empty
 3: level ← n
 4: acc_count ← 0
 5: Push(Stack, o)
 6: Pre[n] ← EnumeratePre(P_n, o)
 7: while not Stack.empty and execution_time < timeout do
 8:     if level = 1 then
 9:         acc_count ← acc_count+ CntPre(P_1, Stack.top)
10:         level ← level + 1
11:         Pop(Stack)
12:     else
13:         v ← PickNotSelected(Pre[level])
14:         if v = AllSelected then
15:             level ← level + 1
16:             Pop(Stack)
17:         else
18:             Push(Stack, v)
19:             level ← level − 1
20:             if level > 1 then
21:                 Pre[level] ← EnumeratePre(P_level, v)
22: return acc_count
```

Algorithm 3, *CntPre(Q, o)* counts $|\mathrm{pre}_Q(o)|$, *PickNotSelected(Pre[i])* selects an element of $Pre[i]$ that has not been traversed yet or returns *AllSelected* if there is no such element, and *EnumeratePre(P_i, v)* lists all elements in $\mathrm{pre}_{P_i}(v)$. $Pre[i]$ stores $\mathrm{pre}_{P_i}(o_i)$ for some o_i. For P_1, it is not necessary to store its preimage because we need only the size of the preimage. Lines 1 to 5 are for initialization. Line 6 enumerates $\mathrm{pre}_{P_n}(o)$. Lines 7 to 21 constitute the main loop of the algorithm, which is stopped either when the counting is done or when time is up. When

$level = 1$, lines 8 to 11 are executed and $CntPre$ will return $\text{pre}_{P_1}(Stack.top)$ in which $Stack.top$ is the input of P_2 that leads to output o of P_n, then back-propagate; lines 13 to 16 check if all elements in the preimage set of the current level is already considered and if so, back-propagate, otherwise push the next element onto the top of $Stack$ and go to the next level.

Theorem 3.5.1. In Algorithm 3, if $P_1, \cdots, P_n$ are deterministic, acc_count, which is returned at line 22, is a lower bound of the preimage size of o by $P_1, \cdots, P_n$. □

Upper bound

For an upper bound of $|\text{pre}_P(o)|$ we use Max#SAT problem [44], which is defined as follows.

Definition 3.5.1. Given a propositional formula $\varphi(X, Y, Z)$ over sets of variables X, Y and Z, the Max#SAT problem is to determine $max_X \#Y.\exists Z.\varphi(X, Y, Z)$.

If we consider a program Q, $In(Q)$, $Out(Q)$ and $Local(Q)$ as φ, Y, X and Z respectively, then, the solution X to the Max#SAT problem can be interpreted as the output value, which has the biggest size of its preimage set. In other words, $max_X \#Y.\exists Z.\varphi(X, Y, Z)$ is an upper bound of the size of pre_Q over all feasible outputs. Therefore, the product of those upper bounds of $|\text{pre}_{P_i}|$ over all i $(1 \leq i \leq n)$ is obviously an upper bound of $|\text{pre}_P|$. The Algorithm 4 computes this upper bound where $CntPre(P_n, o)$ returns the size of the preimage of o by P_n. Notice that, to avoid enumerating the preimages, which costs much computation time, we count only $|\text{pre}_{P_n}(o)|$. For $i = 1, ...n-1$, we compute $MaxCount(P_i)$ as an upper bound for pre_{P_i}, regardless of the corresponding output value. For prototyping, we used the tool developed by the authors of [44], which produces estimated bounds of Max#SAT with tunable confidence and precision. As explained in [44], the tool samples output values of k-fold self-composition of the original program. The greater k is, the more precise the estimation is, but also the more complicated the calculation of each sampling is. Note that $MaxCount(P_i)$ can be computed in advance only once. Though the precision is not always good in general, this approach provides a rather simple

Algorithm 4 UpperBound($P_1, \cdots, P_n, o$)

1: $Result \leftarrow CntPre(P_n, o)$
2: **for** $i \leftarrow 1$ to n **do**
3: $\quad Result \leftarrow Result * MaxCount(P_i)$
4: **return** $Result$

computation method. Also, note that the leakage is the logarithm of the model count.

Theorem 3.5.2. In Algorithm 4, if $P_1, \cdots, P_n$ are deterministic, then $Result$, which is returned at line 4, is an upper bound of the preimage size of o by $P_1, \cdots, P_n$. □

C. Extensibility to Probablistic Programs

When a program is probabilistic, a single input value may produce more than one output values. Therefore, when we count the preimage set of a specific output value, an input value may be counted multiple times, which results in an upper approximation. Though this does not invalidate our proposed algorithm, which computes an upper bound using Max#SAT, it could degrade the precision.

For the algorithms of exact counting and lower bounding, the following modifications will retain both their validity and precision as presented above when analyzing probabilistic programs. Notice that the Algorithm 3 gives exact count when *timeout* is sufficiently long.

- Maintain a list of distinct input values that produce the observed output value o'. In line 9 of Algorithm 3, instead of only counting models, enumerate all feasible input values, then update that list by adding the new input values. As the result, the final set is exactly the preimage set of o'.

- Provided the input distribution, i.e., the prior probability $p(s)$ of each input value s, QIF_1 can be calculated by taking the logarithm of the sum of $p(s)$ for s belonging to the preimage set, which is already computed by the above modification.

- Provided the channel matrix representing the conditional probability $p(o|s)$ of each output value o given an input value s, QIF_2 can be calculated by computing the probability $p(o')$ that the observed output value o' is produced.

3.5.2 Value domain decomposition

Another effective method for computing the dynamic leakage in a compositional way is to decompose the sets of input values and output values into several subsets, compute the leakage for the subprograms restricted to those subsets, and compose the results to obtain the leakage of the whole program. The difference between the parallel composition in [50] and the proposed method is that in the former case, a program under analysis itself is divided into two subprograms that run in parallel, and in the latter case, the computation of dynamic leakage is conducted in parallel by decomposing the sets of input and output values.

Let $P(S, O)$ be a program. Assume that the sets of input values and output values, $\mathcal{S}$ and $\mathcal{O}$, are decomposed into mutually disjoint subsets as

$$\begin{aligned} \mathcal{S} &= \mathcal{S}_1 \uplus \cdots \uplus \mathcal{S}_k, \\ \mathcal{O} &= \mathcal{O}_1 \uplus \cdots \uplus \mathcal{O}_l. \end{aligned}$$

For $1 \leq i \leq k$ and $1 \leq j \leq l$, let P_{ij} be the program obtained from P by restricting the set of input values to $\mathcal{S}_i$ and the set of output values to $\mathcal{O}_j$ where if the output value o of P for an input value $s \in \mathcal{S}_i$ does not belong to $\mathcal{O}_j$, the output value of P_{ij} for input s is undefined. In practice, this disjoint decomposition can be done simply by augmenting the program under analysis with appropriate constraints on input and output.

By definition, for a given $o \in \mathcal{O}_j$,

$$\mathrm{pre}_P(o) = \bigcup_{1 \leq i \leq k} \mathrm{pre}_{P_{i,j}}(o). \tag{3.16}$$

By (3.7) and (3.9), we can compute QIF_1 and QIF_2 in a compositional way. By Theorem 3.2.1, if P is deterministic and the prior probability of S is uniformly distributed, what we have to compute is $|\mathrm{pre}_P(o)|$, which can be obtained by

summing up each $|\text{pre}_{P_{i,j}}(o)|$ by (3.16):

$$|\text{pre}_P(o)| \quad = \quad \sum_{1 \leq i \leq k} |\text{pre}_{P_{i,j}}(o)|.$$

Otherwise, probabilistic programs with arbitrary input distribution can be handled in a manner similar to the one described in the last paragraph of the previous section.

3.5.3 Experiments

This section will investigate answers for the following questions: (1) How well does parallel computing based on the value domain decomposition improve the performance? (2) How well does sequential composition help improve the performance? (3) How well does approximation in the sequential composition work in terms of precision and speed? and (4) Is *CiA* always better than *CoD* or vice versa? We will examine those questions through a few examples: *Grade protocol* is for question (1), *Bit shuffle* and *Population count* are for (2) and (3), while (4) is considered based on both the former and the latter. The benchmarks and prototype are public. [6]

A. Setting up

The experiments were conducted on Intel(R) Xeon(R) CPU ES-1620 v3 @ 3.5GHz x 8 (4 cores x 2 threads), 32GB RAM, CentOS Linux 7. For parallel computation, we use OpenMP [38] library. At the very first phase, to transform C programs into CNFs, we leveraged the well-known CBMC. For the construction of a BDD from a CNF and the model counting and enumeration of the constructed BDD, we use an off-the-shell tool PC2BDD [7]. We use PC2DDNNF [8] for the d-DNNF counterpart. Both of the tools are developed by one of the authors in another project. Besides, as the ordering of Boolean variables of a CNF greatly affects the BDD generation performance, we utilize FORCE [5] to optimize the ordering

[6] `https://bitbucket.org/trungchubao-nu/dla-composition/src/master/dla-composition/`

[7] `https://git.trs.css.i.nagoya-u.ac.jp/t_isogai/cnf2bdd`

[8] `https://git.trs.css.i.nagoya-u.ac.jp/k-hasimt/gpmc-dnnf`

before transforming a CNF into a BDD. We use MaxCount [9] for estimating the answer of Max#SAT problem. We implemented a tool for Algorithm 3 and Algorithm 4, as well as the exact count in sequential compositions in Java.

[9] `https://github.com/dfremont/maxcount`

B. Grade protocol

Table 3.3: Time for constructing data structure and counting model

		$n = 32$	$n = 8$	$n = 4$	$n = 1$
BDD Construction	$t = 32$	**218.53s**	–	–	–
	$t = 16$	222.27s	–	–	–
	$t = 8$	237.54s	**137.74s**	–	–
	$t = 4$	254.88s	144.55s	**155.90s**	–
	$t = 2$	376.21s	233.34s	214.65s	–
	$t = 1$	736.74s	450.85s	391.99s	**243.85**s
d-DNNF Construction	$t = 32$	93.17s	–	–	–
	$t = 16$	**91.49s**	–	–	–
	$t = 8$	107.31s	**123.48s**	–	–
	$t = 4$	141.27s	147.79s	**175.34s**	–
	$t = 2$	215.92s	226.93s	247.45s	–
	$t = 1$	398.99s	391.67s	457.38s	**304.88s**
Model Counting (*CiA* - BDD based)	$t = 32$	**0.21s**	–	–	–
	$t = 16$	0.22s	–	–	–
	$t = 8$	0.25s	**0.13s**	–	–
	$t = 4$	0.30s	0.16s	**0.16s**	–
	$t = 2$	0.65s	0.31s	0.24s	–
	$t = 1$	0.86s	0.36s	0.31s	**0.30s**
Model Counting (*CiA* - d-DNNF based)	$t = 32$	**0.05s**	–	–	–
	$t = 16$	**0.05s**	–	–	–
	$t = 8$	**0.05s**	**0.01s**	–	–
	$t = 4$	0.07s	**0.01s**	**0.01s**	–
	$t = 2$	0.12s	0.02s	0.02s	–
	$t = 1$	0.18s	0.04s	0.03s	**0.25s**
Model Counting (*CoD* - using GPMC)	$t = 1$	–	–	–	**44.69s**

This benchmark is taken from [74]. By this experiment, we investigated how well parallel computation improves the performance of counting models, hence of

quantifying dynamic leakage, in value domain decomposition. This benchmark sums up (then takes the average of) the grades of a group of students without revealing the grade of each student. We used the benchmark with *4 students* and *5 grades*, and all variables are of 16 bits. For model counting, we suppose the observed output (the sum of students' grades) to be **1**, and hence the number of models is 4. GPMC [10], one of the fastest tools for quantifying dynamic leakage as shown in [34], was chosen as the representative tool for *CoD* approach. We manually decompose the original program into 4, 8 and 32 sub-programs by adding constraints on input and output of the program based on the value domain decomposition (the set of output values is divided into 2 and the set of input values is divided into 2, 4 or 16 disjoint subsets). Table 3.3 is divided into sub-divisions corresponding to specific tasks: BDD construction, d-DNNF construction and model counting based on different approaches. In each sub-division, the **bold number** represents the shortest execution time in each column (i.e., the same number of decomposed sub-programs, but different numbers of threads). '−' represents cases when the number of threads is greater than the number of sub-programs, which are obviously meaningless to do experiments.

In Table 3.3, n: number of sub-programs decomposed from the original program; t: number of threads specified by *num_thread* compiling directive of OpenMP. Note that $n = 1$ means non-decomposition, $t = 1$ means a sequential execution and the number of physical CPUs is 8. From Table 3.3, we can make the following inferences:

- As for the answer to question (1), parallel computing speeds up the calculation several times *BDD Construction*: **137.74s** vs. **243.85s**; *d-DNNF Construction*: **91.49s** vs. **304.88s**; *Model Counting (CiA - BDD based)*: **0.13s** vs. **0.30s**. to tens times *Model Counting (CiA - d-DNNF)*: **0.01s** vs. **0.25s**.

- In general, increasing the number of threads (up to the number of sub-programs) does improve the execution time in both the construction of BDD, d-DNNF and the model counting.

- When the number of sub-programs is close to the number of physical CPUs,

[10] `https://www.trs.css.i.nagoya-u.ac.jp/~k-hasimt/tools/gpmc.html`

which is eight, the execution time is among the best if not the best.

The performance with d-DNNF is better than that with BDD in this example, but this seems due to the implementation of the tools.

C. Bit shuffle and Population count

population_count is the 16-bit version of the benchmark of the same name given in [74]. In this experiment, the original program is decomposed into three sub-programs in such a way that each sub-program performs one bit operation of the original. Inspired by *population_count*, we created the benchmark *bit_shuffle*, which consists of two steps: firstly it counts the number of bit-ones in a given secret number (by *population_count*, actually we took the count modulo 6 to increase the preimage size by the first part), then it shuffles those bits to produce an output value. This original program is divided into two sub-programs corresponding to the above-mentioned two steps. Though *bit_shuffle* is probabilistic (i.e., the shuffling part), the algorithm in Algorithm 3 still works, because there is always only one possible input value (i.e., the number of bit-ones) for an output of the sub-program corresponding to the latter step.

Table 3.4 shows execution times of constructing BDD and d-DNNF for two sample programs. The last three columns: *non-decompose*, *decompose (serial)* and *decompose (parallel)* are execution time when computed for the original program, computed sequentially and parallelly for the decomposed sub-programs, respectively. **Bold numbers** are the best execution times in those three.

For model counting, we let an output value be 3 (the number of models is 13110) for *bit_shuffle* and 7 (the number of models is 11440) for *population_count*. Table 3.5 presents the execution times for model counting where the underlined numbers are the exact counts, the **bold execution times** are the best results among approaches for the exact count of each benchmark and the *italic data* are of approximated calculations. The execution times for the lower bounds are predetermined timeouts, which were designed to be 1/2, 1/5 and 1/10 of the time needed by the exact count, followed by the time by *CoD*. In *bit_shuffle* benchmark, lower bounds based on d-DNNF were not improved (all are zero) even when the timeout was increased. This happened because an intermediate result of counting for one d-DNNF is unknown until the counting completes while this benchmark contains

Table 3.4: BDD and d-DNNF construction time for different approaches

		non-decompose	decompose (serial)	decompose (parallel)
BDD Construction	bit_shuffle	>1 hour	33.90s	**33.46s**
	population_count	0.48s	0.66s	**0.40s**
d-DNNF Construction	bit_shuffle	424.64s	50.28s	**48.39s**
	population_count	1.19s	0.71s	**0.69s**

Table 3.5: Model counting: execution time and the changing of precision

			bit_shuffle		population_count	
CoD using GPMC (non-decompose)			0.49s	13110	**0.09s**	11440
CiA-BDD based (decompose)	Exact count		1.47s	13110	10.98s	11440
	Approximation	Lower bound	*0.75s*	*6243*	*5.5s*	*5776*
			0.30s	*1918*	*2.2s*	*888*
			0.15s	*574*	*1.1s*	*312*
			0.49s	*3713*	*0.09s*	*0*
		Upper bound	0.02s	14025	0.07s	5898240
CiA-d-DNNF based (decompose)	Exact count		**0.27s**	13110	3.50s	11440
	Approximation	Lower bound	*0.13s*	*0*	*1.75s*	*4712*
			0.05s	*0*	*0.70s*	*1314*
			0.03s	*0*	*0.35s*	*52*
			0.49s	*13110*	*0.09s*	*0*
		Upper bound	0.07s	14025	0.13s	5898240

only two sub-programs and the size of the preimage by the second sub-program is always one (i.e., the number of times to count d-DNNFs is only two, one for the first sub-program and one for the second one).
From the experimental results, we obtain the following observations.

- As for the answer of question (2), in case of *bit_shuffle*, sequential composition helps speed up the construction of BDD more than 100 times (**33.46s** vs. > 1 hour) and d-DNNF more than 8 times (**48.39s** vs. 424.64s). However, in case of *population_count*, the improvement is insignificant. For model counting, in both of the samples, sequential composition either helps just little or does not help.

- As for the answer of question (3), in case of upper bound for *bit_shuffle*, the

precision is quite good. However, it was extremely low for *population_count*. For both of the samples, the calculation was very fast. For the lower bound, basically the precision gets better with longer timeout. In addition, because leakage is logarithm of model count, its upper bound and lower bound are much tighter than those of model count.

Note that, when we compare *CiA* with *CoD*, it is reasonable to ignore time of construction in *CiA*, because it happens only at the first time, then the result can be reused. For the question (4), in case of *grade protocol*, *CiA* shows a huge improvement over *CoD*, which is more than 4000 times (**0.01s** vs. **44.69s**). But in case of *population_count*, *CoD* is superior to *CiA* (**0.09s** vs. 3.50s). So, the answer is *NO*, i.e., sometimes *CiA* is better and the other time *CoD* is.

4 QIF of string manipulating programs

Calculating QIF relies hugely on model counting which is costly especially for complex data structures like strings. We proposed counting problems and the corresponding algorithms for recognizable series, recognizable tree series and algebraic series which are extensions of finite automata (FA), tree automata and context-free grammars respectively.

4.1 String counting

Strings is one of the most popular data types that plays the main role in many critical vulnerabilities such as SQL injection, Cross-site scripting, because most of data directly manipulated by users is represented under strings especially in web applications. Besides, since strings can be used to represent natural languages, the varieties of how, when and where those strings are generated and used make it difficult to check all possibilities that could be vulnerable.

```
PCTEMP_LHS_1 := T1_1.T2_1;
T2_1 == "=Online";
T1_1 == var_0xINPUT_2;
T_2 := T1_4.T2_4;
T2_4 == "Now";
T1_4 == PCTEMP_LHS_1;
T3 := T_2!="Hello=Joe=OnlineNow";
T_4 := !T_3;
ASSERT(T_4);
```

Above is an example of string constraint extracted from the well-known Kaluza benchmark. In that representation of the constraint, a variable is a character string that may includes only alphabets, digits and underscores, and a constant string is a chain of characters that is wrapped by a pair of double quotes. The dot "." represents the string concatenation. The last line `ASSERT(T_4)` indicates a checking command to verify if there is any assignment to the string variables that makes `T_4` true. There is one solution for this example, namely `var_0xINPUT_2 = "Hello=Joe"`.

Similarly to problems SAT and SMT introduced in Chapter 3, we are sometimes interested in the number of solutions of a string constraint, for instance given requirements for choosing a password, the number of possible passwords indicates the strength of the systems. However, when there is no limitation on the length of strings, the number may be infinite. Thus, a prerequisite is that the lengths of strings to count must be bounded to assure the answer is finite. String can serve as a background theory in an SMT problem. String counting, therefore, can be done as solving a #SMT problem. Therefore, if we develop an efficient algorithm of counting strings that satisfy a given constraint, we can use the algorithm as an engine for the background theory when we solve #SMT problems containing string constraints. Another approach is to use generating functions [61] to approximate the number of solutions by giving a lower and an upper bounds. Recently, an efficient calculating method based on automata was proposed [12]. This method can be regarded as a special case of the method proposed for counting recognizable series in the next section.

As presented in Chapter 2, Min entropy QIF can be calculated by counting the distinct output values of a program. Hence, combining string counter with a transformer, that can extract string constraints on output of the program under analysis, is another solution to calculate QIF. Java String Analyzer [1] and PHP String Analyzer [2] are two of such transformers. This chapter is dedicated to present our attempt to create a stronger, in terms of expressiveness, and faster string counter.

[1] `https://www.brics.dk/JSA/`

[2] `https://sv.c.titech.ac.jp/minamide/phpsa/`

4.2 The framework of formal series

This section is dedicated to present about recognizable series, recognizable tree series and the corresponding counting problems. Especially, the former that is extended from finite automata can be reduced to the state-of-the-art automata-based string counting algorithm [12].

4.2.1 Introduction

Formal power series (or formal series for short) are a natural extension of formal languages by assigning to each word (or string) in a language a value called the *coefficient* (or the *weight*) of the word taken from a semiring. A simple and important subclass of formal series is recognizable series, which can be regarded as an extension of regular languages. A recognizable series S is represented by a triple (λ, μ, γ) where λ is a row vector specifying the initial states, μ is a morphism from input words to state-transition matrices, and γ is a column vector specifying the final states. The coefficient of a word w is defined as $(S, w) = \lambda(\mu w)\gamma$. The support of a formal series is the language consisting of words w such that $(S, w) \neq 0$. For example, let S_a be the formal series such that $(S_a, w) = |w|_a$ for each word w where $|w|_a$ denotes the number of occurrences of a in w. For example, $(S_a, aba) = 2$ and $(S_a, bb) = 0$. Hence, the support of S_a is the language consisting of all words that contains at least one a. As shown in Example 4.2.1, S_a is a recognizable series. It is well-known that a language is regular if and only if the language is the support of a recognizable series with coefficients taken from natural numbers.

The coefficient of a word w can represent various quantities such as the cost needed for an operation on w, the probability that w is emitted from an information source, etc. One of the possible applications of formal series is the string counting in vulnerability analysis of software (also see related work below). Typical vulnerability analysis of a client-side code in web application reduces to counting the strings that are generated by the code and have a pattern abused by an attacker. Finite automata (FA) are a useful tool for the string counting because many operations on strings used in a code can be represented by FA and the class of regular languages has the decidability of basic problems and the closure

property on language operations. When we want to conduct a more elaborated analysis such as quantitative information flow (QIF) analysis based on Shannon entropy, the string counting does not suffice but we need to compute the probability that a string is generated by a code. For a program P with a secret input X and an observable output Y, QIF of P is defined as $H(X) - H(X|Y)$ where $H(X)$ is the entropy of X (the initial uncertainty of X) and $H(X|Y)$ is the conditional entropy of X given Y (the remaining entropy of X after observing Y), which is defined based on the probabilities for X and Y [67, 80]. Let us take an example. Assume that an attacker wants to guess the password X of a victim and he knows from Y that a string consisting of only a single character such as 'aaa' is not allowed to be a password by the security policy. Counting the possible candidate passwords would suffice for a simple vulnerability analysis. Let $\Sigma = \{a, b\}$ be an alphabet. The constraint for a string w to be a password is represented by the regular expression $r = a^+b\Sigma^* \cup b^+a\Sigma^*$. For example, the number of the possible passwords of length three is six, which can be computed by constructing an FA accepting $L(r)_d = \{\ w \mid w \text{ matches } r \text{ and } |w| = d\ \}$. (Note that $L(r)_d$ is a finite set.) If we want to conduct QIF analysis, we further need to compute the probability of the possible passwords of length d. If $p(a) = 2/3$ and $p(b) = 1/3$, the probability for $d = 3$ is $\sum_{|w|=3, w\neq aaa, bbb} p(w) = 1 - 8/27 - 1/27 = 18/27$. This computation can be done by using a formal series as follows: First, we construct a recognizable series S_e such that (S_e, w) is the probability for w, e.g., $(S_e, aba) = 4/27$ and $(S_e, aaa) = 0$, by augmenting an FA accepting $a^+b\Sigma^* \cup b^+a\Sigma^*$ with $p(a) = 2/3$ and $p(b) = 1/3$. Secondly, we compute the sum of the coefficients of each word w of length d in S_e, which will be denoted as $CC(S_e, d)$ in the sequel.

In this section, we define counting problems for formal series and propose algorithms for the problems for recognizable and algebraic series. We first define two notions on counting for formal series, the coefficient count of S with length d (denoted $CC(S, d)$) and the support count of S with length d (denoted $SC(S, d)$), which are natural but different extensions of the number of words of length d belonging to a given language. We then show that for a given recognizable series S and a natural number d, $CC(S, d)$ can be computed in $O(\eta \log d)$ time where η is an upper-bound of time needed for a single matrix operation, and if the state-transition matrices are commutative, $SC(S, d)$ can be computed in a polynomial

order of d. The set of strings generated by a recursive program is expressed by a context-free grammar (CFG) more precisely than FA. However, the previous study translates the obtained CFG into an FA [32]. Algebraic series is an extension of CFGs in the same sense as recognizable series is an extension of FAs. In this section, we propose an algorithm that computes $CC(S, d)$ in square time of d for an algebraic series S.

There are various data structures other than strings generated in a program, such as trees, linked lists and graphs. For QIF analysis or probabilistic testing of programs that dynamically generate such data structures, counting methods for those data structures have also been studied [43]. Tree series is an extension of formal series from strings to trees, which assigns a value of a semiring to each tree [41]. This research attempts to extend the proposed notion of $CC(S, d)$ and $SC(S, d)$ to trees. Unlike strings, even if the size is determined, trees can take different shapes. We extend the coefficient count problem to tree series, denoted as $CC(S, \tau)$, as follows: for a given tree series S and a tree τ, compute the sum of the coefficients of all the trees in S that have the same shape as τ. We extend the support count problem to tree series similarly and propose algorithms for computing $CC(S, \tau)$ and $SC(S, \tau)$. For an application to QIF analysis of a program that generates trees, we can combine an algorithm A for $CC(S, \tau)$ and a tree enumeration algorithm B that enumerates all the unlabeled trees of a given size without repetition (e.g., [69]), in such a way that for a given size d, we run B and for each output from B, we run A and sum up the counting results.

An overview of a vulnerability analysis based on counting can be illustrated in Figure 4.1 where formal model corresponds to formal series in this thesis while it corresponds to FA or logical formula in the previous work. This research focuses on counting (within the dashed line in the figure) and the other steps is out of the scope. To empirically evaluate the efficiency of the proposed algorithms, we show the CPU time to compute $CC(S, d)$ for some CFGs as S, one of which represents the syntax of C language. Finally, to examine an applicability of the proposed method to vulnerability analysis, we have implemented a prototype counting tool that takes a constraint with a restricted syntax, translates it into a CFG and applies our proposed counting algorithm to the translated CFG. We present the results conducted by the tool for more than 17,000 instances taken from Kaluza

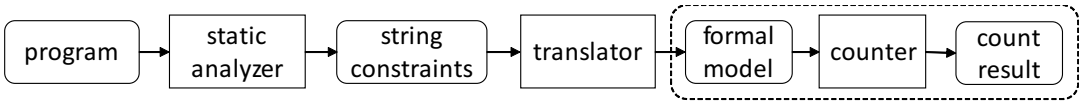

Figure 4.1: A vulnerability analysis

Benchmark.

4.2.2 Counting for recognizable series

A. Formal series and recognizable series

Let A be an alphabet and A^* be the set of all strings over A. An element of A is called a *letter* and an element of A^* is called a *word* over A. Let ε denote the empty word. Let $\mathbb{R}$ be a semiring. A *formal series* S (over A with coefficients in $\mathbb{R}$) is a function $S : A^* \to \mathbb{R}$. The image by S of a word w is denoted by (S, w) and it is called the coefficient of w in S. The class of formal series over A with coefficients in $\mathbb{R}$ is denoted by $\mathbb{R}\langle\langle A\rangle\rangle$. A formal series S is also written in the summation notation as follows:

$$S = \sum_{w \in A^*} (S, w)w.$$

Definition 4.2.1 (support)**.** For a formal series $S \in \mathbb{R}\langle\langle A\rangle\rangle$, the *support* of S, denoted by $\mathtt{supp}(S)$ is defined by

$$\mathtt{supp}(S) = \{w \in A^* \mid (S, w) \neq 0.\}$$

□

Let I and J be index sets. $\mathbb{R}^{I \times J}$ denotes the set of matrices over $\mathbb{R}$ indexed by $I \times J$. If $I = \{1, 2, \ldots, m\}$ and $J = \{1, 2, \ldots, n\}$, $\mathbb{R}^{I \times J}$ is also written as $\mathbb{R}^{m \times n}$.

Definition 4.2.2 (recognizable series)**.** A formal series $S \in \mathbb{R}\langle\langle A\rangle\rangle$ is *recognizable* if there is an integer $n \geq 1$ and a morphism of monoids $\mu : A^* \to \mathbb{R}^{n \times n}$ (i.e., $\mu\varepsilon$ is the identity matrix E and $\mu(w_1 w_2) = (\mu w_1)(\mu w_2)$ for every $w_1, w_2 \in A^*$) and two matrices $\lambda \in \mathbb{R}^{1 \times n}$ and $\gamma \in \mathbb{R}^{n \times 1}$ such that for all words w,

$$(S, w) = \lambda(\mu w)\gamma.$$

The triple (λ, μ, γ) is called a linear representation of S (or S is represented by (λ, μ, γ)), and n is called the dimension of (λ, μ, γ). □

Example 4.2.1 ([21])**.** Let $A = \{a, b\}$ and define $|w|_a$ as the number of occurrences of the letter a in w. Let $\mathbb{N}$ be the semiring of natural numbers. The formal series $\mathbb{N}\langle\langle A\rangle\rangle$ defined by

$$S_a = \sum_{w \in A^*} |w|_a w$$

is recognizable. In fact, S_a is represented by a linear representation (λ, μ, γ) where

$$\lambda = (1, 0), \gamma = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \mu a = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \mu b = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

□

Proposition 4.2.1 ([21, 41])**.** Let $\mathbb{R}$ be $\mathbb{N}$ or the Boolean semiring $\mathbb{B}$. For any recognizable series $\mathbb{R}\langle\langle A\rangle\rangle$, $\mathtt{supp}(S)$ is a regular language. Conversely, if $L \subseteq A^*$ is a regular language, there is a recognizable series $S \in \mathbb{R}\langle\langle A\rangle\rangle$ such that $L = \mathtt{supp}(S)$. □

Example 4.2.2. Let $A = \{a, b\}$, $\mathbb{Z}$ be the semiring of integers, and let $\mathbb{Z}\langle\langle A\rangle\rangle$ be the series

$$S_{\mathtt{diff}} = \sum_{w \in A^*} (|w|_a - |w|_b) w.$$

A linear representation (λ, μ, γ) of $S_{\mathtt{diff}}$ is defined by

$$\lambda = (1, 0), \gamma = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \mu a = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \mu b = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}.$$

The support of $S_{\mathtt{diff}}$ is:

$$\mathtt{supp}(S_{\mathtt{diff}}) = \{w \in A^* \mid |w|_a \neq |w|_b\},$$

which is not regular. □

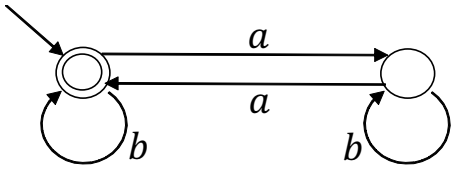

Figure 4.2: Dfa $S_{\mathtt{ae}}$

Note that a deterministic finite automaton (dfa) $M = (Q, A, \delta, q_1, F)$ where $Q = \{q_1, \ldots, q_n\}$ is a finite set of states, A is an input alphabet, $\delta : Q \times A \to Q$ is a transition function, $q_1 \in Q$ is the initial state and $F \subseteq Q$ is the set of final states, can be regarded as a recognizable series $S_M \in \mathbb{R}\langle\langle A \rangle\rangle$ where $\mathbb{R}$ is either $\mathbb{N}$ or $\mathbb{B}$, represented by a linear representation (λ, μ, γ) such that

- $\lambda = (1, 0, \ldots, 0)$,
- $\gamma = \begin{pmatrix} f_1 \\ : \\ f_n \end{pmatrix}$ where $f_i = 1$ if $q_i \in F$ and $f_i = 0$ otherwise.
- for every $a \in A$, $\mu a[i,j] = \begin{cases} 1 & if\ \delta(q_i, a) = q_j, \\ 0 & otherwise. \end{cases}$

In what follows, we identify S_M with M and call S_M a dfa.

Example 4.2.3. Let $S_{\mathtt{ae}}$ be the recognizable series represented by (λ, μ, γ) where

$$\lambda = (1, 0), \gamma = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \mu a = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \mu b = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

which is the dfa shown in Figure 4.2. □

B. Counting problems and algorithms

For a language $L \subseteq A^*$, let $L_d = \{w \in L \mid |w| = d\}$ where $|w|$ denotes the length of w.

Definition 4.2.3 (coefficient count and support count)**.** Let S be a formal series over A and d be a natural number. Define

$$CC(S,d) = \sum_{|w|=d} (S,w),$$

and

$$\begin{aligned} SC(S,d) &= |\mathtt{supp}(S)_d| = |\{w \in A^* \mid (S,w) \neq 0, |w| = d\}| \\ &= \sum_{|w|=d} \Delta[(S,w)] \end{aligned}$$

$$\Delta[C] = \begin{cases} 1 & \text{if } C \neq 0 \\ 0 & \text{if } C = 0 \end{cases},$$

which are called the coefficient count of S with length d and the support count of S with length d, respectively. □

$CC(S,d)$ is the summation of the coefficients of words of length d in S whereas $SC(S,d)$ is the number of words of length d that belong to $\mathtt{supp}(S)$.

Theorem 4.2.1. For a recognizable series S over A represented by (λ, μ, γ) and a natural number $d \in \mathbb{N}$,

$$CC(S,d) = \lambda \overline{\mu}^d \gamma$$

where $\overline{\mu} = \sum_{a \in A} \mu a$. $CC(S,d)$ can be computed in $O(\eta \log d)$ time where η is an upper-bound of the time needed for a single matrix operation (addition and multiplication) [3].

(Proof)

$$\begin{aligned} CC(S,d) &= \sum_{|w|=d} (S,w) = \sum_{c_i \in A, 1 \leq i \leq d} \lambda(\mu c_1 \cdots \mu c_d)\gamma \\ &= \lambda(\sum_{a \in A} \mu a)^d \gamma = \lambda \overline{\mu}^d \gamma. \end{aligned}$$

Let $d = \sum_{k=0}^{L} b_k 2^k$ be the binary representation of d where $L = \lfloor \log d \rfloor$ and $b_k = 0$ or 1 for $0 \leq k \leq L$. Then, we have $\overline{\mu}^d = \Pi_{b_k=1, 0 \leq k \leq L} \overline{\mu}^{2^k}$, which can be computed within the required time by matrix exponentiation operations. □

[3] Note that $\eta \in O(n^3)$ where n is the dimension of (λ, μ, γ).

A linear representation (λ, μ, γ) is *commutative* if $\mu a \mu b = \mu b \mu a$ holds for every $a, b \in A$.

Theorem 4.2.2. Let S be a recognizable series over $A = \{a_1, \ldots, a_m\}$ represented by a commutative linear representation (λ, μ, γ) and $d \in \mathbb{N}$.

$$SC(S,d) = \sum_{d_1+\cdots+d_m=d} \binom{d}{d_1,\ldots,d_m} \Delta[\lambda(\mu a_1)^{d_1} \cdots (\mu a_m)^{d_m}\gamma]$$

where $\binom{d}{d_1,\ldots,d_m}$ is the multinomial coefficient of $d = d_1 + \cdots + d_m$ defined as $\frac{d!}{d_1! \cdots d_m!}$. $SC(S,d)$ can be computed in $O(d^m(\eta m + d^2 \log d) \log d)$ time where η is an upper-bound mentioned in the previous theorem.

(Proof)

$$\begin{aligned} SC(S,d) &= \sum_{|w|=d} \Delta[(S,w)] = \sum_{|w|=d} \Delta[\lambda(\mu w)\gamma] \\ &= \sum_{c_i \in A, 1 \le i \le d} \Delta[\lambda(\mu c_1 \cdots \mu c_d)\gamma] \\ &= \sum_{d_1+\cdots d_m=d} \binom{d}{d_1,\ldots,d_m} \Delta[\lambda(\mu a_1)^{d_1} \cdots (\mu a_m)^{d_m}\gamma]. \end{aligned}$$

The number of combinations $d_1, \ldots, d_m$ is $O(d^m)$. In a similar way to the previous theorem, we can show for given $d_1, \ldots, d_m$, it takes $O(\eta m \log d)$ time to compute $\lambda(\mu a_1)^{d_1} \cdots (\mu a_m)^{d_m}\gamma$, and $O(d^2 \log^2 d)$ time to calculate $\binom{d}{d_1,\ldots,d_m}$. □

Example 4.2.4. For S_a in Example 4.2.1, $\overline{\mu} = \begin{pmatrix} 2 & 1 \\ 0 & 2 \end{pmatrix}$ and so $\overline{\mu}^3 = \overline{\mu}^2\overline{\mu} = \begin{pmatrix} 4 & 4 \\ 0 & 4 \end{pmatrix}\begin{pmatrix} 2 & 1 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 8 & 12 \\ 0 & 8 \end{pmatrix}$. Therefore, $CC(S_a, 3) = (1,0)\begin{pmatrix} 8 & 12 \\ 0 & 8 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = 12$.
Since $\mu a \mu b = \mu b \mu a$, we can apply Theorem 4.2.2. We have

$$\lambda(\mu a)^i(\mu b)^j\gamma \begin{cases} = 0 & if\ i = 0 \\ \ge 1 & if\ i \ge 1 \end{cases}$$

and so $SC(S_a, 3) = 0 + 3 + 3 + 1 = 7$. □

Example 4.2.5. Let us consider $S_{\texttt{diff}}$ in Example 4.2.2. Since $\overline{\mu} = 2E$, $\lambda\overline{\mu}^d\gamma = 2^d\lambda\gamma = 0$ for every $d \geq 1$. Therefore, $CC(S_{\texttt{diff}}, d) = 0$ for every $d \geq 1$. Since $\mu a\mu b = \mu b\mu a$, we can apply Theorem 4.2.2. Note that $\lambda(\mu a)^i(\mu b)^j\gamma \geq 1$ if $i \neq j$. Therefore, $SC(S_{\texttt{diff}}, 3) = \sum_{k=0}^{3}\binom{3}{k} = (1+1)^3 = 8$. □

Corollary 4.2.1. For a dfa S, $SC(S, d) = CC(S, d)$ for every $d \in \mathbb{N}$. $SC(S, d)$ can be computed in $O(\eta \log d)$ time where η is an upper-bound mentioned in Theorem 4.2.1.

(Proof) Let S be the dfa represented by (λ, μ, γ). By definition, for an arbitrary $w \in A^*$, exactly one component of $\lambda(\mu w)$ is 1 and the other components are all zero. Hence, $(S, w) = 1$ when $(S, w) \neq 0$ and the claim holds. □

As shown above, we obtain the main theorem of [12] as a corollary of Theorem 4.2.1 although [12] did not explicitly mention an upper-bound of the computational complexity. In fact, [12] uses generating functions for deriving an algorithm that counts the words of a given length in a regular language by multiplication of the "transfer matrix" $\overline{\mu}$. A generating function is a power series over one variable (or a singleton alphabet, say $A = \{z\}$) and can be regarded as a special case of a formal series. A technique common to [12] and this research is to transform every letter $a \in A$ into an identical letter, say z so that we can merge all the different runs of a dfa (or a recognizable series) for different inputs of the same length into a single run having a weight (a value of the semiring) greater or equal to one, which represents the multiplicity of the runs. This can be done in a simple way by using $\overline{\mu} = \sum_{a\in A}\mu a$ in this research.

Example 4.2.6. The third example of counting is for $S_{\texttt{ae}}$ in Example 4.2.3. We have $\overline{\mu} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$. By Theorem 4.2.1 and Corollary 4.2.1,

$$\begin{aligned} SC(S_{\texttt{ae}}, d) &= CC(S_{\texttt{ae}}, d) = (1,0)\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}^d\begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= (1,0)\begin{pmatrix} 2^{d-1} & 2^{d-1} \\ 2^{d-1} & 2^{d-1} \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = 2^{d-1} \end{aligned}$$

for $d \geq 1$ and

$$SC(S_{\mathtt{ae}}, 0) = CC(S_{\mathtt{ae}}, 0) = (1, 0) \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1.$$

By the way,

$$S_{\mathtt{ae}} = \varepsilon + b + aa + bb + aab + aba + baa + bbb + \cdots .$$

By replacing every a and b with z in $S_{\mathtt{ae}}$, we obtain the generating function $GF(z)$ of $SC(S_{\mathtt{ae}}, d)$:

$$GF(z) = 1 + z + 2zz + 4zzz + \cdots = 1 + \sum_{k=1}^{\infty} 2^{k-1} z^k$$

where we write 1 instead of ε according to the usual notation of the unit element in algebra. We have $(GF(z), z^d) = 2^{d-1}$ for $d \geq 1$ and $(GF(z), 1) = 1$, which coincide with the above results. □

4.2.3 Extension to recognizable tree series

Let F be a ranked alphabet where the rank of $f \in F$ is denoted as $r(f)$. Let $F_r = \{f \mid f \in F, r(f) = r\}$. Define the set of trees over F as the smallest set satisfying: $f(t_1, \ldots, t_{r(f)}) \in T(F)$ whenever $t_i \in T(F)$ for $1 \leq i \leq r(f)$.

For a semiring $\mathbb{R}$, a tree series S over F with coefficients in $\mathbb{R}$ is a function $S : T(F) \to \mathbb{R}$. Similarly to a formal series, the image of a tree $t \in T(F)$ by S is denoted as (S, t) and we write

$$S = \sum_{t \in T(F)} (S, t)t.$$

Let $\mathbb{R}\langle\langle F \rangle\rangle$ denote the class of tree series over F with coefficients in $\mathbb{R}$. For a tree series $S \in \mathbb{R}\langle\langle F \rangle\rangle$, the *support* of S is defined as $\mathtt{supp}(S) = \{t \in T(F) \mid (S, t) \neq 0\}$.

We extend recognizability to tree series. A pair (μ, γ) is a *representation* for F in $\mathbb{R}$ where $Q = \{1, \ldots, n\}$ is a finite set of indices (or states), $\mu = \bigcup_{r \geq 0} \mu_r$ with $\mu_r : (\mathbb{R}^{Q^r \times Q})^{F_r}$ and $\gamma : \mathbb{R}^Q$. (We will interchangeably write $\mu_r : F_r \to \mathbb{R}^{Q^r \times Q}$, $\mu_r : (\mathbb{R}^{n^r \times n})^{F_r}$, $\gamma : \mathbb{R}^{Q \times 1}$, and so on.)

Let $t = f(t_1, \ldots, t_r)$ where $r = r(f)$. We extend μ to a tree homomorphism in $T(F)$ by:

$$\mu t[q] = \sum_{q_1,\ldots,q_r \in Q} \mu t_1[q_1] \cdots \mu t_r[q_r] \mu_r(f)[(q_1, \ldots, q_r), q].$$

Also, we define $(\mu t)\gamma = \sum_{q \in Q} \mu t[q]\gamma[q]$.

Definition 4.2.4 (recognizable tree series [41])**.** A tree series $S \in \mathbb{R}\langle\langle F \rangle\rangle$ is *recognizable* if there is a representation (μ, γ) such that for all trees t,

$$(S, t) = (\mu t)\gamma.$$

We call (μ, γ) a representation of S (or S is represented by (μ, γ)), and $|Q|$ is called the dimension of (μ, γ). □

Example 4.2.7. Let $F = \{f, g, a, b\}$ with $r(f) = r(g) = 2$, $r(a) = r(b) = 0$ and let $S_{\texttt{tdiff}} \in \mathbb{Z}\langle\langle F \rangle\rangle$ be a recognizable tree series represented by (μ, γ) where

- the index set is $Q = \{\mathbf{1}, \mathbf{2}\}$,
- $\mu_0 a = (1, 1)$, $\mu_0 b = (1, -1)$,
- $\mu_2 f = \begin{matrix}(\mathbf{1},\mathbf{1})\\(\mathbf{1},\mathbf{2})\\(\mathbf{2},\mathbf{1})\\(\mathbf{2},\mathbf{2})\end{matrix}\begin{pmatrix}1 & 1\\ 0 & 1\\ 0 & 1\\ 0 & 0\end{pmatrix}$, and $\mu_2 g = \begin{matrix}(\mathbf{1},\mathbf{1})\\(\mathbf{1},\mathbf{2})\\(\mathbf{2},\mathbf{1})\\(\mathbf{2},\mathbf{2})\end{matrix}\begin{pmatrix}1 & -1\\ 0 & 1\\ 0 & 1\\ 0 & 0\end{pmatrix}$
  where the cell at the $(\mathbf{i}, \mathbf{j})$-th row and the $\mathbf{k}$-th column of $\mu_2 f$ denotes $\mu_2 f[(\mathbf{i}, \mathbf{j}), \mathbf{k}]$ and similarly for $\mu_2 g$,
- $\gamma = \begin{pmatrix}0\\1\end{pmatrix}$.

For example,

- $\mu g(a, b)[\mathbf{1}] = \sum_{1 \le \mathbf{i},\mathbf{j} \le 2} \mu a[\mathbf{i}] \mu b[\mathbf{j}] \mu_2 g[(\mathbf{i}, \mathbf{j}), \mathbf{1}] = 1 \cdot 1 \cdot 1 + 1 \cdot (-1) \cdot 0 + 1 \cdot 1 \cdot 0 + 1 \cdot (-1) \cdot 0 = 1$, and similarly,
- $\mu g(a, b)[\mathbf{2}] = 1 \cdot 1 \cdot (-1) + 1 \cdot (-1) \cdot 1 + 1 \cdot 1 \cdot 1 + 1 \cdot (-1) \cdot 0 = -1$, i.e., $\mu g(a, b) = (1, -1)$.

- $\mu f(g(a,b),a) = (1,1)$.

In general, $(S_{\texttt{tdiff}}, t) = (|t|_f + |t|_a) - (|t|_g + |t|_b)$. □

As recognizable series can be seen as an extension of dfa, deterministic bottom-up tree automata can be regarded as a special case of recognizable tree series. A deterministic bottom-up tree automaton (dbta) is a tuple $M = (Q, F, R, Q_F)$ where Q is a finite set of states, F is a ranked alphabet, R is a finite set of transition rules satisfying the conditions described below, $Q_F \subseteq Q$ is the set of final states. A transition rule in R has the shape $f(q_1, \ldots, q_r) \to q$ where $q_1, \ldots, q_r, q \in Q$. For every $f \in F_r, q_1, \ldots, q_r, q \in Q$, there is exactly one rule in R whose left-hand side is $f(q_1, \ldots, q_r)$ (i.e., we assume M is complete without loss of generality). M can be regarded as a recognizable tree series represented by (μ, γ) where

- $\mu_r(f)[(q_1, \ldots, q_r), q] = \begin{cases} 1 \text{ if } f(q_1, \ldots, q_r) \to q \in R, \\ 0 \text{ otherwise,} \end{cases}$
- $\gamma[q] = 1$ iff $q \in Q_F$.

We define the equivalence relation $\simeq$ over $T(F)$ as the smallest relation satisfying: for every $f, g \in F$ with $r(f) = r(g) = r$, $f(s_1, \ldots, s_r) \simeq g(t_1, \ldots, t_r)$ whenever $s_i \simeq t_i$ $(1 \leq i \leq r)$. If $s \simeq t$, we say s is *shape equivalent* with t.

Similarly to the coefficient and support counts of formal series, we will define two countings of a given tree series with respect to the shape equivalence with a given tree, instead of the length of words.

Definition 4.2.5 (coefficient count and support count)**.** Let $L \subseteq T(F)$ be a tree language over F and τ be a tree in $T(F)$. Define

$$L_{\simeq\tau} = \{t \in L \mid t \simeq \tau\}.$$

Let S be a tree series over F and τ be a tree in $T(F)$. Define

$$CC(S, \tau) = \sum_{t \simeq \tau} (S, t),$$

and

$$\begin{aligned} SC(S,\tau) &= |\mathtt{supp}(S)_{\simeq\tau}| = |\{t \in T(F) \mid (S,t) \neq 0, t \simeq \tau\}| \\ &= \sum_{t\simeq\tau} \Delta[(S,t)], \end{aligned}$$

which are called the coefficient count of S with shape τ and the support count of S with shape τ, respectively. □

We do not yet know an efficient computing method similar to matrix exponentiation in recognizable series. The following theorem provides an algorithm that utilizes the summation $\sum_{f\in F_r} \mu_r(f)$ as the state transition (instead of tracing an independent transition $\mu_r(f)$ and summing up), and computes $CC(S,\tau)$ recursively on the structure of τ.

Theorem 4.2.3. For a recognizable tree series S over F represented by (μ,γ) and a tree $\tau \in T(F)$,

$$CC(S,\tau) = (\overline{\mu}\tau)\gamma$$

where $\overline{\mu} = \bigcup_{r\geq 0} \overline{\mu_r}$, $\overline{\mu_r}(\varphi) = \sum_{f\in F_r} \mu_r(f)$ for any $\varphi \in F_r$.

$CC(S,\tau)$ can be computed in $O(\xi n^{\rho+1}|\tau|)$ time where

- ξ is an upper-bound of the time needed for computing $v_1 \cdots v_r \mu_r(f)[(q_1,\ldots,q_r),q]$ for given $v_i \in \mathbb{R}$, $q_i \in Q$ $(1 \leq i \leq r)$ and $q \in Q$,
- n is the dimension of (μ,γ),
- ρ is the maximum rank of $f \in F$, and
- $|\tau|$ is the size of τ defined by $|\varphi(\tau_1,\ldots,\tau_r)| = 1 + \sum_{i=1}^{r} |\tau_i|$.

(Proof) Let $\tau = \varphi(\tau_1,\ldots,\tau_r) \simeq f(t_1,\ldots,t_r) = t$. By definition,

$$\overline{\mu}\tau[q] = \sum_{q_1,\ldots,q_r\in Q} \overline{\mu}\tau_1[q_1] \cdots \overline{\mu}\tau_r[q_r]\overline{\mu_r}(\varphi)[(q_1,\ldots,q_r),q].$$

$$
\begin{aligned}
\sum_{t\simeq\tau}(S,t) &= \sum_{t\simeq\tau}(\mu\tau)\gamma \\
&= \sum_{t\simeq\tau}\sum_{q\in Q}\mu t[q]\gamma[q] \\
&= \sum_{t\simeq\tau}\sum_{q\in Q}\sum_{q_1,\ldots,q_r\in Q}(\mu t_1[q_1]\cdots\mu t_r[q_r]\mu f[(q_1,\ldots,q_r),q]\gamma[q]) \\
&= \sum_{q\in Q}(\sum_{q_1,\ldots,q_r\in Q}(\sum_{t\simeq\tau}\mu t_1[q_1]\cdots\mu t_r[q_r]\mu f[(q_1,\ldots,q_r),q]))\gamma[q] \\
&= \sum_{q\in Q}(\sum_{q_1,\ldots,q_r\in Q}(\sum_{t_1\simeq\tau_1}\mu t_1[q_1]\cdots\sum_{t_r\simeq\tau_r}\mu t_r[q_r]\overline{\mu}(\varphi)[(q_1,\ldots,q_r),q]))\gamma[q] \\
&= \sum_{q\in Q}(\sum_{q_1,\ldots,q_r\in Q}\overline{\mu}\tau_1[q_1]\cdots\overline{\mu}\tau_r[q_r]\overline{\mu}(\varphi)[(q_1,\ldots,q_r),q])\gamma[q] \\
&= \sum_{q\in Q}\overline{\mu}\tau[q]\gamma[q] \\
&= (\overline{\mu}\tau)\gamma
\end{aligned}
$$

The time needed for computing $(\overline{\mu}\tau)\gamma$ is the summation of the time needed for $\overline{\mu}\tau[q]\gamma[q]$ for all $q \in Q$. We can compute $\overline{\mu}\tau[q]$ recursively on the structure of the tree τ, taking the sum for arbitrary combinations of $q_1,\ldots,q_r$ of states at the argument positions. Hence, the total computation time is $O(\xi n^{\rho+1}|\tau|)$. □

Example 4.2.8. Consider $S_{\texttt{tdiff}}$ in Example 4.2.7. By definition,

- $\overline{\mu}_0 a = \overline{\mu}_0 b = (1,1) + (1,-1) = (2,0)$,
- $\overline{\mu}_2 f = \overline{\mu}_2 g = \begin{matrix}(\mathbf{1},\mathbf{1})\\(\mathbf{1},\mathbf{2})\\(\mathbf{2},\mathbf{1})\\(\mathbf{2},\mathbf{2})\end{matrix}\begin{pmatrix}2&0\\0&2\\0&2\\0&0\end{pmatrix}$.

We will show that for any $\tau \in T(F)$, $\overline{\mu}\tau = (2^{|\tau|},0)$ by the structural induction on τ. The claim holds in the basis case because $\overline{\mu}_0 a = \overline{\mu}_0 b = (2,0)$. Let $\tau = \varphi(\tau_1,\tau_2) \in T(F)$, and assume that the claim holds for τ_1, and τ_2. Then, $\overline{\mu}\varphi(\tau_1,\tau_2)[\mathbf{1}] = \overline{\mu}\tau_1[\mathbf{1}] \times \overline{\mu}\tau_2[\mathbf{1}] \times 2 = 2^{|\tau_1|}\cdot 2^{|\tau_2|}\cdot 2 = 2^{|\tau|}$, and $\overline{\mu}\varphi(\tau_1,\tau_2)[\mathbf{2}] = \overline{\mu}\tau_1[\mathbf{1}] \times \overline{\mu}\tau_2[\mathbf{2}] \times 2 + \overline{\mu}\tau_1[\mathbf{2}] \times \overline{\mu}\tau_2[\mathbf{1}] \times 2 = 0$. Namely, $\overline{\mu}\varphi(\tau_1,\tau_2) = (2^{|\tau|},0)$ and the claim holds in the inductive case. Thus, the claim holds by the induction. By Theorem 4.2.3, $CC(S_{\texttt{tdiff}},\tau) = (2^{|\tau|},0)\begin{pmatrix}0\\1\end{pmatrix} = 0$. □

For $SC(S, \tau)$, we even do not know a way of merging runs of S on different trees that have the same shape τ. An upper-bound of the time to compute $SC(S, \tau)$ is given in the next theorem.

Theorem 4.2.4. Let S be a recognizable tree series over F represented by (μ, γ) and let $\tau \in T(F)$. $SC(S, \tau)$ can be computed in $O(\xi n^{\rho+1}|\tau||F|^{|\tau|})$ time where ρ is the maximum rank of f in F. □

If S is a dbta, $SC(S, \tau)$ coincides with $CC(S, \tau)$, which is similar to the fact that $SC(S, d) = CC(S, d)$ for a dfa S, and we can use the algorithm in Theorem 4.2.3 to compute $SC(S, \tau)$.

Corollary 4.2.2. For a dbta S over F, $SC(S, \tau) = CC(S, \tau) = (\overline{\mu}\tau)\gamma$ for every $\tau \in T(F)$.

(Proof) We can show that for every $t \in T(F)$, there is exactly one $q \in Q$ such that $\mu t[q] = 1$ and $\mu t[q'] = 0$ for every $q' \neq q$. Hence, $(S, t) = \Delta[(S, t)]$ holds and the claim follows. □

4.3 Context free grammar-based string counting

While automata-based counting is fast, the weak expressiveness of finite automata limits the language of string constraints can be handled by this method. This section show an attempt to use context-free grammars, which has a stronger expressiveness than FA, to represent string constraints and a method to count strings that satisfy those constraints.

4.3.1 Counting for algebraic series

In this section, we consider counting problems for algebraic series. While counterparts of recognizable series in formal language theory are regular languages, counterparts of algebraic series are context-free languages. We first introduce definitions and well-known properties of algebraic series based on [41, 54] and then present an algorithm that computes $CC(S, d)$, the coefficient count of a given algebraic series S by dynamic programming. Computing coefficients of an algebraic series has applications to static analysis of programs because the behavior of a

recursive program can often be modeled by a context-free language and the result of static analysis such as dataflow analysis can be represented as a coefficient of the corresponding algebraic series (see [77] for example).

A *polynomial* $S \in \mathbb{R}\langle\langle A \rangle\rangle$ is a formal series such that $\mathtt{supp}(S)$ is finite. The class of all polynomials over A with coefficients in $\mathbb{R}$ is denoted as $\mathbb{R}\langle A \rangle$.

We fix an alphabet A and a semiring $\mathbb{R}$ and let Y be a countable set of variables disjoint with A.

Definition 4.3.1 (algebraic system)**.** An ($\mathbb{R}$-)*algebraic system* is a finite set of equations of the form

$$y_i = p_i \quad (1 \leq i \leq n)$$

where $y_i \in Y$ and $p_i \in \mathbb{R}\langle A \cup Y \rangle$. The system is called *proper* if $(p_i, \varepsilon) = (p_i, y_j) = 0$ for all i, j $(1 \leq i, j \leq n)$. □

A solution of a given algebraic system consists of n formal series $R_1, \ldots, R_n \in \mathbb{R}\langle\langle A \rangle\rangle$ satisfying the system in the sense that when each y_i $(1 \leq i \leq n)$ is replaced with R_i, y_i becomes equal to p_i for every i $(1 \leq i \leq n)$. For a vector $\mathbf{v}$, let $\mathbf{v}^T$ denote the transpose of $\mathbf{v}$. Formally, let $R = (R_1, \ldots, R_n)^T \in \mathbb{R}\langle\langle A \rangle\rangle^{n \times 1}$ be an n-column vector of formal series and define the morphism $h_R : (A \cup Y)^* \to \mathbb{R}\langle\langle A \rangle\rangle$ by $h_R(y_i) = R_i$ $(1 \leq i \leq n)$ and $h_R(a) = a$ for $a \in A$. We furthermore extend h_R to a polynomial $p \in \mathbb{R}\langle A \cup Y \rangle$ by

$$h_R(p) = \sum_{w \in (A \cup Y)^*} (p, w) h_R(w).$$

R is a *solution* of the algebraic system $y_i = p_i$ $(1 \leq i \leq n)$ if $R_i = h_R(p_i)$ for all i $(1 \leq i \leq n)$.

We say $S \in \mathbb{R}\langle\langle A \rangle\rangle$ is *quasi-regular* if $(S, \varepsilon) = 0$. It is well-known that every proper $\mathbb{R}$-algebraic system $y_i = p_i$ $(1 \leq i \leq n)$ has a unique solution $R = (R_1, \ldots, R_n)^T$ such that every component R_i $(1 \leq i \leq n)$ is quasi-regular. R is called the *strong* solution of the system. The strong solution of a proper algebraic system $y_i = p_i$ $(1 \leq i \leq n)$ can be given by the limit $R = \lim_{j \to \infty} R^j$ of the sequence R^j defined by $R^0 = (0, \ldots, 0)^T$ and $R^{j+1} = (h_{R^j}(p_1), \ldots, h_{R^j}(p_n))^T$ for $j \geq 0$.

Example 4.3.1. (1) For an $\mathbb{N}$-algebraic system $y = ayb + ab$, $R^0 = 0$, $R^1 = ab$, $R^2 = ab + a^2b^2$, …, and $R = \sum_{n \geq 1} a^n b^n$.

(2) For an $\mathbb{N}$-algebraic system $y = yy + a$, $R^0 = 0$, $R^1 = a$, $R^2 = a + a^2$, $R^3 = a + (a + a^2)(a + a^2) = a + a^2 + 2a^3 + a^4$, ... $R = \sum_{n \geq 1} C_{n-1} a^n$ where C_n is the n-th Catalan number: $C_n = \frac{1}{n+1}\binom{2n}{n}$.

□

Definition 4.3.2 (algebraic series)**.** A formal series $S \in \mathbb{R}\langle\langle A \rangle\rangle$ is $\mathbb{R}$-*algebraic* if $S = (S, \varepsilon)\varepsilon + S'$ where S' is some component of the strong solution of a proper $\mathbb{R}$-algebraic system. The class of $\mathbb{R}$-algebraic series over A is denoted by $\mathbb{R}^{\mathtt{alg}}\langle\langle A \rangle\rangle$.

A language $L \subseteq A^*$ is $\mathbb{R}$-*algebraic* if $L = \mathtt{supp}(S)$ for some $S \in \mathbb{R}^{\mathtt{alg}}\langle\langle A \rangle\rangle$. □

A context-free grammar (CFG) is a tuple (N, T, P, S) where N and T are finite sets of nonterminal and terminal symbols, respectively, P is a finite set of (production) rules in the form $A \to \gamma$ where $A \in N$ and $\gamma \in (N \cup T)^*$ and $S \in N$ is the start symbol. The derivation relation $\overset{*}{\Rightarrow}_G$ is defined in the usual way. For $A \in N$, let $L_G(A) = \{w \in T^* \mid A \overset{*}{\Rightarrow}_G w\}$. The context-free language generated by G is $L(G) = L_G(S)$.

Proposition 4.3.1. [41, 54] A language L is context-free if and only if L is $\mathbb{N}$-algebraic if and only if L is $\mathbb{B}$-algebraic.

Proof sketch. For a given algebraic system $y_i = p_i$ $(1 \leq i \leq n)$, construct CFG $G = (\{y_1, \ldots, y_n\}, A, P, y_1)$ where $y_i \to w \in P$ iff $(p_i, w) \neq 0$. Conversely, for a given CFG $G = (\{y_1, \ldots, y_n\}, A, P, y_1)$, construct the algebraic system $y_i = p_i$ $(1 \leq i \leq n)$ where $(p_i, w) = 1$ if $y_i \to w \in P$ and $(p_i, w) \to 0$ otherwise. □

Definition 4.3.3 (Chomsky normal form)**.** An $\mathbb{R}$-algebraic system $y_i = p_i$ $(1 \leq i \leq n)$ is in *Chomsky normal form* if $\mathtt{supp}(p_i) \subseteq A \cup Y^2$ for every i $(1 \leq i \leq n)$. □

As is the case of context-free languages, for a proper $\mathbb{R}$-algebraic system $\alpha : y_i = p_i$ $(1 \leq i \leq n)$, we can construct an $\mathbb{R}$-algebraic system β in Chomsky normal form such that every component of the strong solution of α is a component of the strong solution of β. Hence, we assume without loss of generality that a given proper $\mathbb{R}$-algebraic system is in Chomsky normal form.

We provide an algorithm that computes $CC(S, d)$ for an algebraic series S and $d \in \mathbb{N}$.

Theorem 4.3.1. Let $\alpha : y_i = p_i$ $(1 \leq i \leq n)$ be an $\mathbb{R}$-algebraic system in Chomsky normal form and let $R = (R_1, \ldots, R_n)^T$ be the strong solution of α. Then, for each i $(1 \leq i \leq n)$ and $d \in \mathbb{N}$,

$$\begin{aligned} CC(R_i, d) &= \sum_{y_j y_m \in \mathtt{supp}(p_i) \cap Y^2} (p_i, y_j y_m) \sum_{k=1}^{d-1} CC(R_j, k) CC(R_m, d-k) \\ &\quad + \sum_{a \in \mathtt{supp}(p_i) \cap A} (p_i, a) \Delta[d = 1]. \end{aligned} \tag{4.1}$$

Moreover, $CC(R_i, d)$ can be computed in $O(\xi d^2)$ time where $\xi = n \max_{1 \leq i \leq n} |p_i|$.

(Proof) By definition of strong solution, for every i $(1 \leq i \leq n)$,

$$\begin{aligned} R_i &= h_R(p_i) = \sum_{w \in (A \cup Y)^*} (p_i, w) h_R(w) \\ &= \sum_{y_j y_m \in \mathtt{supp}(p_i) \cap Y^2} (p_i, y_j y_m) R_j R_m + \sum_{a \in \mathtt{supp}(p_i) \cap A} (p_i, a) a. \end{aligned}$$

Therefore, for every i $(1 \leq i \leq n)$,

$$\begin{aligned} (R_i, w) &= \sum_{y_j y_m \in \mathtt{supp}(p_i) \cap Y^2} (p_i, y_j y_m) \sum_{w = uv} (R_j, u)(R_m, v) \\ &\quad + \sum_{a \in \mathtt{supp}(p_i) \cap A} (p_i, a) \Delta[w = a], \text{ and} \\ CC(R_i, d) &= \sum_{|w| = d} (R_i, w) \\ &= \sum_{y_j y_m \in \mathtt{supp}(p_i) \cap Y^2} (p_i, y_j y_m) \sum_{|w| = d, w = uv} (R_j, u)(R_m, v) \\ &\quad + \sum_{a \in \mathtt{supp}(p_i) \cap A} (p_i, a) \sum_{|w| = d} \Delta[w = a] \\ &= (4.1). \end{aligned}$$

$CC(R_i, d)$ can be computed simultaneously for all i $(1 \leq i \leq n)$ in $O(\xi d^2)$ time by dynamic programming where $\xi = n \max_{1 \leq i \leq n} |p_i|$. □

We can extend the above algorithm to apply to an arbitrary algebraic series. [4]

[4] We can keep the time complexity $O(\xi d^2)$ through this extension by revising the algorithm in such a way that an equation of length three or more is virtually divided into equations of length two on the fly.

Example 4.3.2. Consider the following algebraic system:

$$y_1 = y_2y_3 + y_4y_5, \quad y_2 = ay_2 + a, \quad y_3 = by_3c + bc,$$
$$y_4 = ay_4b + ab, \quad y_5 = cy_5 + c.$$

Let $R = (R_1, \ldots, R_5)^T$ be the strong solution of the system. $CC(R_i, d)$ $(1 \leq i \leq 5,$ $1 \leq d \leq 5)$ can be computed by Theorem 4.3.1 as shown below.

	R_1	R_2	R_3	R_4	R_5
1	0	1	0	0	1
2	0	1	1	1	1
3	2	1	0	0	1
4	2	1	1	1	1
5	4	1	0	0	1

Also,

$$\begin{aligned} CC(R_1, 6) &= \sum_{k=1}^{5}(CC(R_2, k)CC(R_3, 6-k) + CC(R_4, k)CC(R_5, 6-k)) \\ &= 4. \end{aligned}$$

$R_1 = \sum_{i\neq j, i\geq 1, j\geq 1}(a^ib^jc^j + a^ib^ic^j) + \sum_{i\geq 1} 2a^ib^ic^i$ and $\mathtt{supp}(R_1) = \{a^ib^jc^j \mid i, j \geq 1\} \cup \{a^ib^ic^j \mid i, j \geq 1\}$ is an (inherently) ambiguous context-free language. □

For a given CFG G, let $\alpha_G : y_i = p_i$ $(1 \leq i \leq n)$ be the algebraic system constructed in the proof of Proposition 4.3.1. Let $R = (R_1, \ldots, R_n)^T$ be the strong solution of α_G. By the construction of α_G, (R_1, w) represents the number of different derivation trees of w in G. Hence, we have the following corollary similar to Corollary 4.2.1, which states that $SC(S, d) = CC(S, d)$ and thus $SC(S, d)$ can be computed in square time of d if an algebraic series S corresponds to an unambiguous CFG.

Corollary 4.3.1. Let G be a CFG and α_G be the algebraic system constructed in the proof of Proposition 4.3.1. Let $R = (R_1, \ldots, R_n)^T$ be the strong solution of α_G. If G is unambiguous, $|L(G)_d| = SC(R_1, d) = CC(R_1, d)$. □

Table 4.1: Size of G_1

No. of terminal symbols	151
No. of non-terminal symbols	143
No. of rules	431

4.3.2 Experimental evaluation

To evaluate the efficiency and applicability of the algorithm proposed in Section 4.3.1, we conducted the following experiments [5]. Based on Theorem 4.3.1, we implemented an algorithm for solving the following string counting problem for CFGs and conducted experiments for a few CFGs (Section 4.3.2).

Input: CFG $G = (N, T, P, S)$, $A \in N$ and $d \in \mathbb{N}$.

Output: $\Sigma_{d' \leq d} |L_A(G)_{d'}|$, the number of strings derivable from A in G and of length not greater than d.

Next, on top of the above implementation, we prototyped a counting tool for string constraints (Section 4.3.2). The tool was implemented in C++ and the experiments were conducted in the environment: core i7-6500U, CPU@2.5GHz x 4, 8GB RAM, Ubuntu 16.10 64bits. GMP library is utilized to deal with big numbers.

A. Basic performance

We first show the experimental results on two CFGs G_1 and G_2. Both of them are taken from [20], with minor modification on G_1 to make it fit to be input into our implementation. Then, we empirically estimate the time complexity of the algorithm and compare it with the result given in Theorem 4.3.1. Finally, we discuss the counting precision by using another simple CFG.

G_1 represents the syntax of C programming language. The size of G_1 is shown in Table 4.1. By Theorem 4.3.1, the time complexity of the algorithm is $O(\xi d^2) = O(n \max_{1 \leq i \leq n} |p_i| d^2)$ where d is the length bound. We estimate

[5] We also conducted experiments for the algorithms for recognizable series, which will be reported elsewhere.

Table 4.2: Experimental result for G_1

Bound	CPU Time (ms)	Count (approx.)
10	3344.53	2.6×10^{19}
15	3797.49	1.1×10^{31}
20	3476.46	6.3×10^{42}
30	3971.00	3.2×10^{66}
40	4838.96	2.0×10^{90}
60	11969.30	1.1×10^{138}
80	45097.40	7.6×10^{185}
120	421563.00	4.6×10^{281}
160	2320180.00	3.3×10^{377}

the time complexity of the algorithm by interpolation on the experimental result shown in Table 4.2. If we assume the running time is $t = \xi d^{\alpha}$, we obtain $\alpha = 2.229$ while the theoretical value is $\alpha = 2$. G_2 is an unambiguous CFG that specifies a simplified syntax of English language, consisting of 16 rules. The experimental results on G_2 is shown in Table 4.3. We fit the result to $t = \xi d^{\alpha}$ and we obtain $\alpha = 2.472$.

There are two questions about the estimated values of α for the two experiments. (1) Besides the length bound, on what factors does the running time of counting algorithm depend? (2) Why are the values of α estimated from the experimental results larger than the theoretical result $\alpha = 2$? For the first question (1), the empirical results show that beside length bound, the size of the input CFG also affects the running time. It took about 3 seconds to count for G_1 at the length bound of 20 but the same amount of time is sufficient to count at length bound of 400 for G_2 whose size is roughly 20 times smaller than G_1's. The answer for the second question (2) is that Theorem 4.3.1 assumes Chomsky Normal Form whilst G_1 and G_2 are not. Our implementation uses a naive extension of the algorithm in Theorem 4.3.1 so that the time complexity becomes (ηd^m) where m is the maximum length of the right-hand sides of the rules [6]. This made the

[6]It is possible to explicitly transform a given CFG to a Chomsky normal form or revise the algorithm so that the given CFG is processed as pointed out in the footnote after Theorem 4.3.1.

Table 4.3: Experimental result for G_2

Bound	CPU Time (ms)	Count (approx.)
400	3655.17	3.3×10^{14}
600	5107.08	8.3×10^{21}
800	9002.97	2.4×10^{29}
1200	21057.50	1.8×10^{44}
1600	47927.00	1.3×10^{59}
2400	161622.00	7.3×10^{88}
3200	393007.00	4.0×10^{118}
4800	1360400.00	1.2×10^{178}

value of α slightly larger than 2.

Lastly, we discuss the counting precision. Remember that $CC(S, d)$ is the summation of the coefficients of words of length d in S and under the interpretation of a CFG as an algebraic series shown in the proof of Proposition 4.3.1, the coefficients of a word w corresponds to the number of derivation trees of w. Consequently, while our implementation gives the exact answer to the counting problem for an unambiguous CFG, it only outputs an upper bound of the answer for an ambiguous CFG. Let's consider the following ambiguous CFG [20].

S –> A.B| a
A –> S.B| b
B –> B.A| a

Table 4.4 shows the exact count and the output of our implementation. As seen in the table, the precision goes down rapidly when the length bound increases.

B. Applicability to string counting

In this subsection, we give the experimental results conducted by our prototype counting tool and compare it to the state-of-the-art string counter ABC on Kaluza benchmark [61], [12].

Figure 4.3 shows the syntax of string constraints that our tool accepts as input. This tool cannot handle string operations `indexOf`, `substring`, `replace`, `regex` (regular expression). To process constraints accepted by this syntax we made

Table 4.4: Counting for an ambiguous CFG

Length	Exact Count	Prototype's Output	Ratio
1	1	1	1.00
2	1	1	1.00
3	2	2	1.00
4	4	5	0.80
5	8	14	0.57
6	16	42	0.38
7	32	132	0.24
8	64	429	0.15

some simple extension on the algorithm in Theorem 4.1 about constraints related to *const-string* and string length with *const-integer* (lines 2, 3 in the Figure 4.3). For instance, *len(A)* $\neq 3$ is interpreted to $CC(A, 3) = 0$ instead of calculating the right hand side of expression (1) in Theorem 4.1. The tool was implemented on top of the implementation in Section 4.3.2 and, as noted there, the tool is not guaranteed to precisely work for all the constraints accepted by the syntax.

Kaluza bechmark consists of two parts, *small* and *big*. The former includes 17,554 files and the latter does 1,342 files. Almost all of files in Kaluza small benchmark satisfy the syntax of our tool with simple equivalence transformation. In fact, there are some regular expression constraints of the form *x in /regex/* in the benchmark, but almost all of the *regex* represent only a constant string c and thus they can be converted into $x{=}c$. Our tool cannot handle precisely the conjunction of rules of CFGs (called CF constraints) because context-free languages (CFLs) are not closed by the intersection. 1,019 out of 1,342 files in Kaluza big include the conjunction of CF constrains rooted at the queried variable, while those in Kaluza small are only 163 out of 17,554 files. Taking this fact into consideration, we compared our tool with ABC on Kaluza small.

We cloned ABC source code and benchmark from: `https://github.com/vlab-cs-ucsb/ABC`[7]. We ran both ABC and our tool at length bounds 1, 2 and

[7]There were several branches without mentioning about a stable release in the repository. So, we cloned the master branch as of 2017/12/27.

constraint ::= *non-terminal* = rhs
| *non-terminal* op1 *const-string*
| len(*non-terminal*) op2 *const-integer*

rhs ::= *non-terminal*
| rhs ∘ *non-terminal*

op1 ::= = | ≠

op2 ::= < | = | > | ≠ | ≤ | ≥

Figure 4.3: Grammar for input of the prototype. ∘ is the concatenation operator.

Table 4.5: Our prototype vs ABC on Kaluza *small* at length bounds 1, 2 and 3

Ours ≡ ABC	**Ours ≢ ABC**			
17,183 files	*Ours right*	*Ours better*	*ABC better*	*ABC right*
	227 files (18 groups)	143 files (2 groups)	0 files (0 groups)	1 file (1 group)
	371 files			
Total = 17,544 files				

3 so that we can manually confirm the counts. There was no big difference about running time between the tools noticed. The comparison results were consistent through different length bounds (1, 2 and 3) in which ABC and our tool give the same results on 17,183 files. All of those files did not include the conjunction of CF constraints. We manually investigated the other 371 files and found that they can be classified into 21 groups such that all constraints in a same group are very similar. Among those 21 groups, our tool gave correct answers for 18 groups, more precise than ABC for 2 groups, and less precise than ABC for 1 group. The last group included the conjunction of CF constraints which our tool could not accept. 163 files (distributed into 1 group in *Ours right*, 2 groups in *Ours better*) among those 371 files included the conjunction of CF constraints. The reason why our tool could precisely answer for 18 groups was that those conjuncts were

the repetition of same constraints. Overall, the experimental result shows that our counting algorithm is potentially applicable to a qualified benchmark and it is promising to extend the algorithm.

5 Data circulation on QIF analysis

As introduced in the last section of Chapter 1, data circulation on QIF analysis has two targets: information leakage threshold and confidential data. This chapter elaborates the discussion by describing the data circulation in details as well as showing the social value that can be yielded.

5.1 The data circulation

Figure 5.1 presents two cycles corresponding to data circulation on QIF analysis for information leakage threshold and confidential data. The first cycle with thicker arrows represents data circulation on QIF analysis to adapt the security policy, which decides whether software is secure or not, to the change of real-world. In the phase *Acquisition*, security incident reports, vulnerability reports and other data that relates to the security and usability of software, are collected and transferred to the next phase *Analysis*. In this phase, based on evaluating the collected data, the information leakage threshold is adjusted. The new value of the threshold is applied to the software development in the phase *Implementation*. The software development is a loop where QIF analysis is executed. The software after being developed will be checked if the QIF is smaller than the threshold. If not, the software is modified to decrease the QIF. This is repeated until the software has the QIF as required. After that, the software is released to be used in real-world.

The second cycle with thinner arrows represents data circulation on QIF analysis to keep confidential data safe under the change of real-world. There is confidential data, which is kept secret to protect other highly confidential data, such

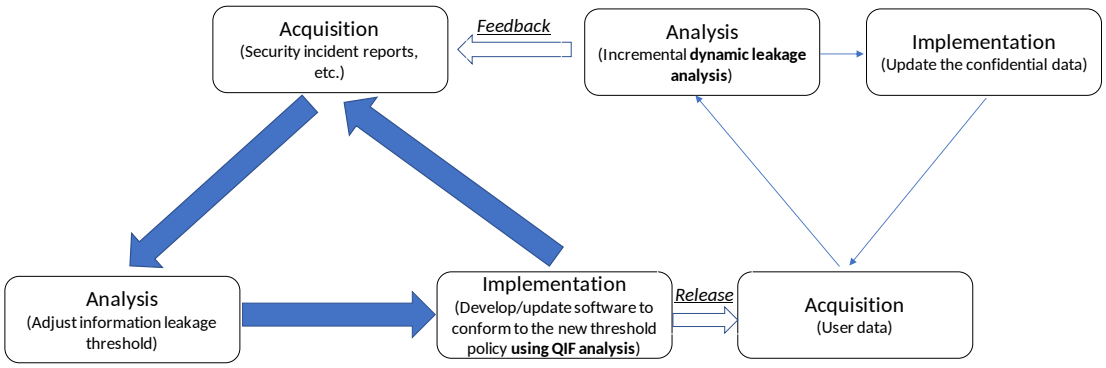

Figure 5.1: Data circulation on QIF analysis

as passwords or some system configurations that can be changed. By monitoring the accumulated information leakage, we can know when those passwords and configurations need to be updated. In *Acquisition*, user data such as IP addresses, user identity and so on are collected along with the data of using the software. The collected data is then transferred to *Analysis* where incremental dynamic leakage calculation is performed. The result is passed through *Implementation* where an update, if needed, on the confidential data is executed.

The two cycles are not independent but interact with each other. The first cycle releases new or updated software to the second one. On the other hand, the second cycle sends feedback on using the software to the first one, for instance, when the confidential data is required to update too frequently, the information leakage threshold should be smaller. In that way, data goes back and forth between the two cycles to constitute one big cycle.

5.2 The social value

We are witnessing the transformation from analog to digital, real-world to virtual world, in an extremely fast pace that might never happened before. We have used mobile phones, then smartphones and now Internet of Things. We watched black-white TVs, then color TVs, and now virtual reality is coming closer. Internet becomes essential to our daily life. Many of us live two lives, real one and virtual

one. Because of the trend, information security becomes much more relevant to us. On the other hand, real-world data, which includes all the data traveling on the internet, affects hugely the security. A fixed security policy will face difficulties in this rapidly changing world. The data circulation on QIF analysis presented in the previous section, however, will adapt well. Thus, **providing an adaptive information security solution** is the most notable social value of this data circulation. This is quite similar to the case of Learning Health System (LHS) [1] where real-world data, generated during routine clinical practice, is used to improve the clinical practice itself. This system makes the treatment adaptive to a targeted case of a specific patient.

[1] `https://en.wikipedia.org/wiki/Learning_health_systems`

6 Conclusion

6.1 Summary

Focusing on the needs for automated and quantitative analysis of software security, this dissertation presented research attempts to answer the following questions related to QIF:

- (Q1) *Is there any method of counting models that directly analyzes variables with complex structures as what they are: strings, arrays, linked lists, trees and so on?*
- (Q2) *How can we address reasonably and naturally the leakage of secure information when executing programs through a specific output value?*
- (Q3) *How to calculate the dynamic leakage in executing a program P when the dynamic leakages of the sub-programs of P are known?*

For (Q1), we propose a string counting method based on CFGs (Context Free Grammars). Our prototype performs well against the state-of-the-art tool over a well-known Kaluza benchmark. Besides the CFG-based counting, we propose a framework of formal power series in which recognizable and algebraic series which are extensions of finite-automata and CFGs, respectively. Although recognizable tree series has been discussed without a concrete implementation, we expect that the tree series is promising to work for more complex data structures.

For (Q2), we introduce two new notions called QIF_1 and QIF_2 for dynamic leakage that is associated with a specific output value of a program. Similar to the traditional (static) QIF, the problem of quantifying QIF_1 and QIF_2 of a program is proved to be not easier than solving #SAT as the counting version of the well-known NP-complete problem is SAT. Though these problems are computationally

hard to solve in the worst-case, we propose a quantifying method of QIF_1 and QIF_2 based on model counting and implement a prototype to show the method is useful in practice. Our experimental results show the feasibility of the method but the method does not scale to programs with hundreds lines of code.

Naturally, we ask (Q3) as a solution for speeding up the calculation of dynamic leakage based on "divide-and-conquer" strategy. Sequential and value domain-based compositions, which provide the methods to calculate dynamic leakage of a program from the constitutional sub-programs, are introduced as the answer for (Q3). The former and the latter are promising to combine to utilize the strength of each other, which is modular computing for sequential composition and parallel computing for value domain-based composition.

6.2 Future work

A possible extension for the CFG-based counting method is to improve the precision of counting when CFGs are ambiguous. Also, another possible future work is to develop a QIF calculating system using this CFG-based counting. Other complex data structures like trees, linked lists are promising to be the next target of a counting method based on the proposed recognizable tree series.

As for dynamic leakage, experiments on bigger benchmarks should be conducted to justify the conclusion. Furthermore, in sequential and value domain-based compositions, currently there is no criteria to evaluate if a decomposition is good or bad in terms of optimizing calculation time. Thus, to create the criteria and a guided method to automatically decompose a certain program to optimize the calculation is needed to be studied.

References

[1] P. A. Abdulla et al.: *String constraints for verification*. In: Proceedings of the 26th International Conference on Computer Aided Verification, CAV 2014, pp. 150–166. LNCS, volume 8559.

[2] P. A. Abdulla et al.: *Norn: an SMT solver for string constraints*. In: Proceedings of the 27th International Conference on Computer Aided Verification, CAV 2015, pp. 462–469. LNCS, volume 9206.

[3] P. S. Aldous: *Noninterference in expressive low-level languages*. In: Doctoral dissertation, The University of Utah, 2017.

[4] M. Alkhalaf , T. Bultan and J. L. Gallegos: *Verifying client-side input validation functions using string analysis*. In: Proceedings of the 34th International Conference on Software Engineering, ICSE 2012, pp. 947–957.

[5] F. Aloul, I. Markov and K. Sakallah: *FORCE: a fast and easy-to-implement variable-ordering heuristic*. In: Proceedings of the 13th ACM Great Lakes symposium on VLSI, GLSVLSI 2003, pp. 116–119.

[6] R. Alur, K. Etessami and M. Yannakakis: *Analysis of recursive state machines*. In: Proceedings of the 13th International Conference on Computer Aided Verification, CAV 2001, pp. 304–313. LNCS, volume 2102.

[7] M. Alvim, M. Andrés, K. Chatzikokolakis and C. Palamidessi: *Quantitative information flow and applications to differential privacy*. In Proceedings of the 11th International School on Foundations of Security Analysis and Design, FOSAD 2011, pp. 211–230. LNCS, volume 6858.

[8] M. Alvim, M. Andrés, K. Chatzikokolakis and C. Palamidessi: *On the relation between differential privacy and quantitative information flow*. In: Pro-

ceedings of the 38th International Colloquium on Automata, Languages and Programming, ICALP 2011, pp. 60–76. LNCS, volume 6756.

[9] M. S. Alvim, K. Chatzikokolakis, A. Mciver, C. Morgan, C. Palamidessi and G. Smith: *Axioms for Information Leakage*. In: Proceedings of the 2016 IEEE 29th Computer Security Foundations Symposium, CSF 2016, pp. 77–92.

[10] M. S. Alvim, K. Chatzikokolakis, C. Palamidessi and G. Smith: *Measuring information leakage using generalized gain functions*. In: Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium, CSF 2012, pp. 280–290.

[11] K. Anjaria and A. Mishra: *Theoretical framework of quantitative analysis based information leakage warning system*. In: Karbala International Journal of Modern Science, 4(1), 2018, pp. 151–163.

[12] A. Aydin, L. Bang and T. Bultan: *Automata-based model counting for string constraints*. In: Proceedings of the 27th International Conference on Computer Aided Verification, CAV 2015, pp. 255–272. LNCS, volume: 9206.

[13] A. Aydin et al.: *Parameterized model counting for string and numeric constraints*. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, pp. 400–410.

[14] R. A. Aziz, G. Chu, C. Muise and P. Stuckey: *#∃SAT: projection model counting*. In: Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing, SAT 2015, pp. 121–137. LNCS, volume 9340.

[15] M. Backes, M. Berg and B. Köpf: *Non-uniform distributions in quantitative information flow*. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, pp. 367–375.

[16] M. Backes, B. Köpf and A. Rybalchenko: *Automatic discovery and quantification of information leaks*. In: Proceedings of the 30th IEEE Symposium on Security and Privacy, S&P 2009, pp. 141–153.

[17] L. Bang, A. Aydin, Q. S. Phan, C. S. Păsăreanu and T. Bultan: *String analysis for side channels with segmented oracles*. In: Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE 2016, pp. 193–204.

[18] C. Barrett, R. Sebastiani, S. Seshia and C. Tinelli: *Chapter 12, Handbook of satisfiability* IOS Press, 2008.

[19] G. Barthe and B. Köpf: *Information-theoretic bounds for differentially private mechanisms*. In: Proceedings of the 24th Computer Security Foundations Symposium, CSF 2011, pp. 191–204.

[20] H. J. S. Basten: *Ambiguity detection methods for context-free grammars*. In: Master's thesis, University of Amsterdam, 2007.

[21] J. Berstel and C. Reutenauer: *Rational series and their languages*, pp. 1–49, Springer, 1988.

[22] F. Besson, N. Bielova and T. Jensen: *Hybrid monitoring of attacker knowledge*. In: Proceedings of the 29th Computer Security Foundations Symposium, CSF 2016, pp. 225–238.

[23] N. Bielova: *Short paper: Dynamic leakage: A need for a new quantitative information flow measure*. In: Proceedings of the 11th ACM Workshop on Programming Languages and Analysis for Security, PLAS 2016, pp. 83–88.

[24] N. Bielova and R. Rezk: *A taxonomy of information flow monitors*. In: Proceedings of the 5th International Conference on Principles of Security and Trust, POST 2016, pp. 46–67. LNCS, volume 9635.

[25] F. Biondi, M. A. Enescu, A. Heuser, A. Legay, K. S. Meel and J. Quilbeuf: *Scalable approximation of quantitative information flow in programs*. In: Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2018, pp. 71–93. LNCS, volume 10747.

[26] F. Biondi, Y. Kawamoto, A. Legay and L. M. Traonouez: *HyLeak: hybrid analysis tool for information leakage*. In: Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis, ATVA 2017, pp. 156–163. LNCS, volume 10482.

[27] F. Biondi, A. Legay, L. Traonouez and A. Wasowski: *QUAIL: a quantitative security analyzer for imperative code*. In: Proceedings of the 25th International Conference on Computer Aided Verification, CAV 2013, pp. 702–707. LNCS, volume 8044.

[28] A. Bouajjani, J. Esparza and O. Maler: *Reachability analysis of pushdown automata: application to model-checking*. In: Proceedings of the 8th International Conference on Concurrency Theory, CONCUR 1997, pp. 135–150. LNCS, volume 1243.

[29] R. Chadha and M. Ummels: *The complexity of quantitative information flow in recursive programs*. In: Research Report LSV-2012-15, Laboratoire Spécification & Vérification, École Normale Supérieure de Cachan, 2012.

[30] S. Chakraborty, K. S. Meel and M. Y. Vardi: *Algorithmic improvements in approximate counting for probabilistic inference: from linear to logarithmic SAT calls*. In: Proceedings of the 25th International Joint Conference on Artificial Intelligence, IJCAI 2016, pp. 3569–3576.

[31] T. Chothia, Y. Kawamoto and C. Novakovic: *LeakWatch: estimating information leakage from Java programs*. In: Proceedings of the 19th European Symposium on Research in Computer Security, ESORICS 2014, pp. 219–236. LNCS, volume 8713.

[32] A. S. Christensen, A. Møller and M. I. Schwartzbach: *Precise analysis of string expressions*. In: Proceedings of the 10th International Symposium on Static Analysis, SAS 2003, pp. 1–18. LNCS, volume 2694.

[33] B. T. Chu, K. Hashimoto, and H. Seki: *Counting algorithms for recognizable and algebraic series*. In: IEICE Transaction on Information and Systems, E101-D(6), June 2018, pp. 1479–1490.

[34] B. T. Chu, K. Hashimoto, and H. Seki: *Quantifying dynamic leakage: complexity analysis and model counting-based calculation*. In: IEICE Transactions on Information and Systems, E102-D(10), October 2019, pp. 1952–1965.

[35] B. T. Chu, K. Hashimoto, and H. Seki: *On the compositionality of dynamic leakage and its application to the quantification problem*. In: Proceedings of the 13th International Conference on Emerging Security Information, Systems and Technologies, SECURWARE 2019, pp. 1–8.

[36] M. R. Clarkson, A. C. Myers and F. B. Schneider: *Quantifying information flow with beliefs*. In: Proceedings of the 18th Computer Security Foundations Symposium, CSF 2009, pp. 655–701.

[37] L. D'Antoni and M. Veanes: *Static analysis of string encoders and decoders*. In: Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2013, pp. 209–228. LNCS, volume 7737.

[38] Dagum L. and Menon R.: *OpenMP: an industry-standard API for shared-memory programming*. In: IEEE Computational Science & Engineering, 5(1), January 1998, pp. 46–55.

[39] A. Darvas, R. Hähnle and D. Sands: *A theorem proving approach to analysis of secure information flow*. In: Proceedings of the 2nd International Conference on Security in Pervasive Computing, SPC 2005, pp. 193–209. LNCS, volume 3450.

[40] A. Darwiche: *On the tractability of counting theory models and its application to belief revision and truth maintenance*. In: Journal of Applied Non-Classical Logics, 11(1-2), 2001, pp. 11–34.

[41] M. Droste, W. Kuich and H. Vogler: *Handbook of weighted automata*, Springer, 2009.

[42] K. Etessami and M. Yannakakis: *Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations*. In: Proceedings of

the 22nd Annual Symposium on Theoretical Aspects of Computer Science, STACS 2005, pp. 340–352. LNCS, volume 3404.

[43] A. Filieri, M. F. Frias, C. S. Păsăreanu and W. Visser: *Model counting for complex data structures*. In: Proceedings of the 22nd International SPIN Workshop on Model Checking of Software, SPIN 2015, pp. 222–241. LNCS, volume 9232.

[44] D. J. Fremont, M. N. Rabe and S. A. Seshia: *Maximum Model Counting*. In: Proceedings of the 31st AAAI Conference on Artificial Intelligence, 2017, pp. 3885–3892.

[45] J. A. Goguen and J. Meseguer: *Security policies and security models*. In: Proceedings of the 3rd IEEE Symposium on Security and Privacy, S&P 1982, pp. 11–20.

[46] P. Hooimeijer, B. Livshits and D. Molnar: *Fast and precise sanitizer analysis with BEK*. In: Proceedings of the 20th USENIX Conference on Security, SEC 2011, pp. 1.

[47] P. Hooimeijer and W. Wiemer: *A decision procedure for subset constraints over regular languages*. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 188–198.

[48] P. Hooimeijer and W. Wiemer: *Solving string constraints lazily*. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, ASE 2010, pp. 378–386.

[49] X. Huang and P. Malacaria: *SideAuto: quantitative information flow for side-channel leakage in web applications*. In: Proceedings of the 12th ACM Workshop on Privacy in The Electronic Society, WPES 2013, pp. 285–290.

[50] Y. Kawamoto, K. Chatzikokolakis and C. Palamidessi: *On the compositionality of quantitative information flow*. In: Logical Methods in Computer Science, 13(3:11), 2017, pp. 1–31.

[51] A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer and M. D. Ernst: *HAMPI: A solver for string constraints*. In: Proceedings of the 18th International Symposium on Software Testing and Analysis, ISSTA 2009, pp. 105–116.

[52] V. Klebanov: *Precise quantitative information flow analysis - a symbolic approach*. In: Theoretical Computer Science, 538, 2014, pp. 124–139.

[53] V. Klebanov, N. Manthey and C. Muise: *SAT-based analysis and quantification of information flow in programs*. In: Proceedings of the 10th International Conference on Quantitative Evaluation of Systems, QEST 2013, pp. 177-192.

[54] W. Kuich and A. Salomma: *Semirings, automata, languages*, Springer, 1986.

[55] B. Köpf and D. Basin: *Automatically deriving information-theoretic bounds for adaptive side-channel attacks*. In: Journal of Computer Security, 19(1), 2011, pp. 1–31.

[56] 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 2012, pp. 564–580. LNCS, volume 7358.

[57] B. Köpf and A. Rybalchenko: *Approximation and randomization for quantitative information flow analysis*. In: Proceedings of the 23rd Computer Security Foundations Symposium, CSF 2010, pp. 3–14.

[58] G. Li and I. Ghosh: *PASS: String solving with parameterized array and interval automaton*. In: Proceedings of the 9th International Haifa Verification Conference, HVC 2013, pp. 15–31. LNCS, volume 8244.

[59] T. Liang, A. Reynolds, C. Tinelli, C. Barrett and M. Deters: *A DPLL($\mathcal{T}$) theory solver for a theory of strings and regular expressions*. In: Proceedings of the 26th International Conference on Computer Aided Verification, CAV 2014, pp. 646–662. LNCS, volume 8559.

[60] Lin A. W. and Barceló P.: *String solving with word equations and transducers: Towards a logic for analysing mutation XSS*. In: Proceedings of the 43rd

Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 123–136.

[61] L. Luu, S. Shinde, P. Saxena and B. Demsky: *A model counter for constraints over unbounded strings*. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 565–576.

[62] P. Malacaria, M. Khouzani, C. S. Pasareanu, Q. S. Phan and K. Luckow: *Symbolic side-channel analysis for probabilistic programs*. In: Proceedings of the 31st Computer Security Foundations Symposium, CSF 2018, pp. 313–327.

[63] S. McCamant and M. D. Ernst: *Quantitative information flow as network flow capacity*. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, pp. 193–205.

[64] Z. Meng and G. Smith: *Calculating bounds on information leakage using two-bit patterns*. In: Proceedings of the 6th Workshop on Programming Languages and Analysis for Security, PLAS 2011, pp. 1–12.

[65] Y. Minamide: *Static approximation of dynamically generated web pages*. In: Proceedings of the 14th World Wide Web Conference, WWW 2005, pp. 432–441.

[66] M. Mitzenmacher and E. Upfal: *Probability and computing: randomized algorithms and probabilistic analysis*, Cambridge, 2005, pp. 167–173.

[67] C. Mu: *Quantitative information for security: a survey*. In: Technical Report TR-08-06, Department of Computer Science, King's College London, 2008. Updated 2010. `http://www.dcs.kcl.ac.uk/technical-reports/papers/TR-08-06.pdf`

[68] C. Muise, S. A. McIlraith, J. C. Beck and E. Hsu: *DSHARP: fast d-DNNF compilation with sharpSAT*. In: Proceedings of the 25th Canadian Conference on Artificial Intelligence, Canadian AI 2012, pp. 356–361. LNCS, volume 7310.

[69] S. Nakano and T. Uno: *Efficient generation of rooted trees*. In: NII Technical Report NII-2003-005E, National Institute of Informatics, 2003.

[70] S. Nakashima, B. T. Chu, K. Hashimoto, M. Sakai and H. Seki: *Efficiency improvement in #SMT-based quantitative information flow analysis*. In: IEICE Technical Report, SS2016-26, 116(277), 2016, pp. 49–54.

[71] C. S. Pasareanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz and N. Rungta: *Symbolic Path Finder: integrating symbolic execution with model checking for Java bytecode analysis*, Automated Software Engineering, 20(3), 2013, pp. 391–425.

[72] M. Pettai and P. Laud: *Combining differential privacy and mutual information for analyzing leakages in workflows*. In: Proceedings of the 6th International Conference on Principles of Security and Trust, POST 2017, pp. 298–319. LNCS, volume 10204.

[73] L. H. Pham, Q. L. Le, Q. S. Phan, J. Sun and S. Qin: *Poster: Testing heap-based programs with Java StarFinder*. In: Poster of the 40th IEEE/ACM International Conference on Software Engineering: Companion, ICSE-Companion 2018.

[74] Q. S. Phan: *Model counting modulo theories*. In: Doctoral dissertation, Queen Mary University of London, 2015.

[75] Q. S. Phan and P. Malacaria: *All-solution satisfiability modulo theories: applications, algorithms and benchmarks*. In: Proceedings of the 10th International Conference on Availability, Reliability and Security, ARES 2015, pp. 100–109.

[76] Q. S. Phan, P. Malacaria, O. Tkachuk and C. S. Pasareanu: *Symbolic quantitative information flow*. In: ACM SIGSOFT Software Engineering Notes, 37(6), November 2012, pp. 1–5.

[77] T. Reps, S. Schwoon, S. Jha and D. Melski: *Weighted pushdown systems and their application to interprocedural dataflow analysis*. In: Proceedings of

the 10th International Symposium on Static Analysis, SAS 2003, pp. 189–213. LNCS, volume 2694. Revised Version appears in: Science of Computer Programming, 58(1-2), October 2005, pp. 206–263.

[78] N. Rosner, J. Geldenhuys, N. M. Aguirre, W. Visser and M. F. Frias: *BLISS: improved symbolic execution by bounded lazy initialization with SAT support*. In: IEEE Transaction on Software Engineering, 41(7), 2015, pp. 639–660.

[79] P. Saxena, P. Akhawe, D. Hanna, S. Mao, F. McCamant and S. Song: *A symbolic execution framework for JavaScript*. In: Proceedings of the 30th IEEE Symposium on Security and Privacy, S & P 2010, pp. 513–528.

[80] 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 2009, pp. 288–302. LNCS, volume 5504.

[81] F. Somenzi: *Binary decision diagrams*, 1999. `http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/reading/somenzi99bdd.pdf`

[82] R. Suzuki, K. Hashimoto, M. Sakai: *Improvement of projected model-counting solver with component decomposition using SAT solving in components*. In: JSAI Technical Report, SIG-FPAI-506-07, 2017, pp. 31–36 (in Japanese).

[83] T. Tateishi, M. Pistoia and O. Tripp: *Path- and index-sensitive string analysis based on monadic second-order logic*. In: Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, pp. 166–176.

[84] M. T. Trinh, D. H. Chu and J. Jaffar: *S3: A symbolic string solver for vulnerability detection in web applications*. In: Proceedings of the 21st ACM Conference on Computer and Communications Security, CCS 2014, pp. 1232–1243.

[85] M. T. Trinh, D. H. Chu and J. Jaffar: *Model counting for recursively-defined strings*. In: Proceedings of the 29th International Conference on Computer Aided Verification, CAV 2017, pp. 399–418. LNCS, volume 10427.

[86] C. G. Val, M. A. Enescu, S. Bayless, W. Aiello and A. J. Hu: *Precisely measuring quantitative information flow: 10k lines of code and beyond*. In: Proceedings of the 1st IEEE European Symposium on Security and Privacy, EuroS&P 2016, pp. 31–46.

[87] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar and N. Bjorner: *Symbolic finite state transducers: Algorithms and applications*. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, pp. 137–150.

[88] W. Visser, K. Havelund, G. Brat, S. J. Park and F. Lerda: *Model checking programs*. In: Automated Software Engineering, 10(2), 2003, pp. 203–232.

[89] W. Wang, L. Ying and J. Zhang: *On the relation between identifiability, differential privacy, and mutual-information privacy*. In: IEEE Transaction on Information Theory, 62(9), 2016, pp. 5018–5029.

[90] H. Yasuoka and T. Terauchi: *Quantitative information flow - verification hardness and possibilities*. In: Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, pp. 15–27.

[91] H. Yasuoka and T. Terauchi: *On bounding problems of quantitative information flow*. In: Journal of Computer Security, 19(6), December 2011, 1029–1082.

[92] M. Ying, Y. Feng and N. Yu: *Quantum information-flow security: noninterference and access control*. In: Proceedings of the 26th IEEE Computer Security Foundations Symposium, CSF 2013, pp. 130–144.

[93] F. Yu, M. Alkhalaf and T. Bultan: *STRANGER: An automata-based string analysis tool for PHP*. In: Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2010, pp. 154–157. LNCS, volume 6015.

[94] F. Yu, M. Alkhakaf, T. Bultan and O. H. Ibarra: *Automata-based symbolic string analysis for vulnerability detection*. In: Formal Methods in System Design, 44(1), 2014, pp. 44–70.

[95] Y. Zheng, X. Zhang and V. Ganesh: *Z3-str: A Z3-based string solver for web application analysis*. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 114–124.