Detection and Exploitation of Information Flow Leaks

Erkennung und Ausnutzung von Informationsflusslecks
Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte Dissertation
zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
vorgelegt von Quoc Huy Do Master of Information Technology geboren in Hanoi, Vietnam

Tag der Einreichung: 15.03.2017
Tag der Prüfung: 27.04.2017

1. Referent: Prof. Dr. Reiner Hähnle
2. Referent: Prof. Dr. David Sands

Erscheinungsort: Darmstadt
Erscheinungsjahr: 2017

Darmstädter Dissertation — D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Software Engineering

Detection and Exploitation of Information Flow Leaks
Erkennung und Ausnutzung von Informationsflusslecks

Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation von Quoc Huy Do Master of Information Technology aus Hanoi, Vietnam

1. Referent: Prof. Dr. Reiner Hähnle
2. Referent: Prof. Dr. David Sands

Tag der Einreichung: 15.03.2017
Tag der Prüfung: 27.04.2017

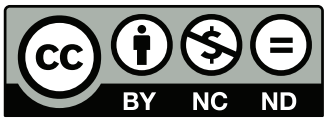
Erscheinungsort: Darmstadt
Erscheinungsjahr: 2017

Darmstädter Dissertation — D 17

Wissenschaftlicher Werdegang
Doktorand am Fachgebiet Software Engineering der Technischen Universität Darmstadt
von Juli 2013 bis April 2017

Studiengang Informatik: Master of Information Technology
University of Engineering and Technology, Vietnam National University, Hanoi
von Januar 2008 bis Juni 2010

Studiengang Informatik: Bachelor of Science (B.Sc.)
People's Security Academy, Hanoi
von September 1999 bis Juni 2004



Abstract

This thesis contributes to the field of language-based information flow analysis with a focus on detection and exploitation of information flow leaks in programs. To achieve this goal, this thesis presents a number of precise semi-automatic approaches that allow one to detect, exploit and judge the severity of information flow leaks in programs.

The first part of the thesis develops an approach to detect and demonstrate information flow leaks in a program. This approach analyses a given program statically using symbolic execution and self-composition with the aim to generate so-called *insecurity formulas* whose satisfying models (obtained by SMT solvers) give rise to pairs of initial states that demonstrate insecure information flows. Based on these models, small unit test cases, so-called *leak demonstrators*, are created that check for the detected information flow leaks and fail if these exist. The developed approach is able to deal with unbounded loops and recursive method invocation by using program specifications like loop invariants or method contracts. This allows the approach to be fully precise (if needed) but also to abstract and allow for false positives in exchange for a higher degree of automation and simpler specifications. The approach supports several information flow security policies, namely, noninterference, delimited information release, and information erasure.

The second part of the thesis builds upon the previous approach that allows the user to judge the severity of an information flow leak by exploiting the detected leaks in order to infer the secret information. This is achieved by utilizing a hybrid analysis which conducts an adaptive attack by performing a series of *experiments*. An experiment constitutes a concrete program run which serves to accumulate the *knowledge* about the secret. Each experiment is carried out with optimal low inputs deduced from the prior distribution and the knowledge of secret so that the potential leakage is maximized. We propose a novel approach to quantify information leakages as explicit functions of low inputs using symbolic execution and parametric model counting. Depending on the chosen security metric, general nonlinear optimization tools or Max-SMT solvers are used to find optimal low inputs, i.e., inputs that cause the program to leak a maximum of information.

For the purpose of evaluation, both approaches have been fully implemented in the tool KEG, which is based on the state-of-the-art program verification system KeY. KEG supports a rich subset of sequential Java programs and generates executable JUnit tests as leak demonstrators. For the secret inference, KEG produces executable Java programs and runs them to perform the adaptive attack.

The thesis discusses the planning, execution, and results of the evaluation. The evaluation has been performed on a collection of micro-benchmarks as well as two case studies, which are taken from the literature.

The evaluation using the micro-benchmarks shows that KEG detects successfully all information flow leaks and is able to generate correct demonstrators in case the supplied specifications are correct and strong enough. With respect to secret inference, it shows that the approach presented in this thesis (which computes optimal low inputs) helps an attacker to learn the secret much more efficiently compared to approaches using arbitrary low inputs.

KEG has also been evaluated in two case studies. The first case study is performed on an e-voting software which has been extracted in a simplified form from a real-world e-voting system. This case study focuses on the leak detection and demonstrator generation approach. The e-voting case study shows that KEG is able to deal with relatively complicated programs that include unbounded loops, objects, and arrays. Moreover, the case study demonstrates that KEG can be integrated with a specification generation tool to obtain both precision and full automation. The second case study is conducted on a PIN integrity checking program, adapted from a real-world ATM PIN verifying system. This case study mainly demonstrates the secret inference feature of KEG. It shows that KEG can help an attacker to learn the secret more efficiently given a good enough assumption about the prior distribution of secret.

Zusammenfassung

Diese Dissertation befasst sich mit dem Bereich der sprachbasierten Informationsflussanalyse und erweitert den aktuellen Kenntnisstand insbesondere in den Gebieten der Erkennung und Ausnutzung unsicheren Informationsflusses in Programmen.

Zum Erreichen dieses Ziels werden im Rahmen dieser Arbeit Verfahren entwickelt, die es ermöglichen, unsicheren Informationsfluss in Programmen zu erkennen und dessen Folgen fundiert einzuschätzen. Die entwickelten Verfahren zeichnen sich durch einen hohen Automatisierungsgrad sowie eine hohe Präzision aus.

Im ersten Teil der Dissertation wird ein Verfahren entwickelt, das es erlaubt unsicheren Informationsfluss zu entdecken und aufzuzeigen. Das entwickelte Verfahren kombiniert symbolische Ausführung mit Selbstkomposition (self-composition) von Programmen mit dem Ziel logische Formeln, sog. *Unsicherheitsformeln*, abzuleiten. Die Modelle dieser Formeln beschreiben initiale Zustandspaare, mit deren Hilfe unsicherer Informationsfluss demonstriert werden kann. Dazu werden Unit-Tests (*Leak Demonstrators*) erzeugt, die das Programm jeweils in einem der initialen Zustände ausführen und fehlschlagen, wenn ein unsicherer Informationsfluss entdeckt wird. Das entwickelte Verfahren unterstützt unbegrenzte Schleifenausführungen und rekursive Methodenaufrufe mit Hilfe von Programmspezifikationen wie Schleifeninvarianten und Methodenverträgen. Diese Vorgehensweise erlaubt auf der einen Seite eine hohe Präzision (d.h., keine False Positives) und auf der anderen Seite aber auch Abstraktion (und damit eine Zunahme von False Positives) bei höherem Automatisierungsgrad. Der Ansatz unterstützt die Informationsflusspolitiken *Noninterference*, *Delimited Information Release* und *Information Erasure*.

Der zweite Teil der Dissertation baut auf dem oben beschrieben Verfahren auf. Es erlaubt jedoch nicht nur die Demonstration des Vorhandenseins von unsicherem Informationsfluss, sondern ermöglicht es dessen Auswirkungen einzuschätzen. Dazu bettet es das bisherige Verfahren in eine hybride Analyse ein, die einen adaptiven Angriff auf das Programm unter Ausnutzung der entdeckten Unsicherheitsstellen realisiert. Das entwickelte Verfahren führt dazu eine Reihe von Experimenten durch und reichert systematisch das Wissen über das im Programm enthaltene Geheimnis an. Ein Experiment besteht aus einer konkreten Programmausführung mit optimal gewählten Eingabewerten basierend auf einer angenommenen A-priori-Verteilung der Werte des Geheimnisses und des durch die vorherigen Experiment angesammelten Wissens über das aktuell vorliegende Geheimnis. Optimal bedeutet, dass die Programmausführung einen maximalen Wissenszuwachs bedingt. Das entwickelte Verfahren grenzt sich von bisher existierenden Ansätzen dadurch ab, dass es den Informationsfluss als explizite Funktion in Abhängigkeit von öffentlichen Eingabewerten (low inputs), basierend auf Ergebnissen aus der symbolischen Programmausführung und parametrischer Modellzählung präsentiert. In Abhängigkeit von der gewählten Sicherheitsmetrik werden allgemeine nicht-lineare Optimierer und Max-SMT-Problemlöser verwendet, um optimale Eingabewerte zur Geheimnisextraktion zu bestimmen.

Die beiden Ansätze wurden implementiert und sind als Programmanalysewerkzeug KEG, basierend auf dem deduktiven Verifikationssystem KeY, verfügbar. KEG ermöglicht (zu einem hohen Grad) die Analyse sequentieller Java-Programme und erzeugt JUnit Testfälle, um un-

sicheren Informationsfluss zu demonstrieren. Für die Geheimnisextraktion erzeugt KEG zum Durchspielen eines Angriffs kleine Java Programme und führt diese aus.

Im Rahmen dieser Arbeit wurden die Verfahren mit Hilfe von Microbenchmarks und zweier Fallstudien evaluiert. Die Arbeit beschreibt die Planung sowie Durchführung der Evaluationen und diskutiert deren Ergebnisse.

Die Durchführung der Microbenchmarks zeigte, dass KEG alle unsicheren Informationsflüsse erkennen konnte und entsprechende Demonstratoren erzeugte, sofern die notwendigen Spezifikation korrekt und ausreichend vollständig waren. Es konnte auch nachgewiesen werden, dass KEG bei der Geheimnisextraktion wesentlich effizienter ist als ein Angreifer, der mit zufällig gewählten Experimenten arbeitet.

KEG bewies in den beiden Fallstudien, dass die Verfahren auch zur Analyse realistischerer Programme eingesetzt werden können. Die erste Fallstudie betrachtet eine vereinfachte Version einer realen e-Voting Software mit unbegrenzten Schleifenausführungensowie, Objekt- und Arraydatentypen. Neben der allgemeinen Anwendbarkeit von KEG konnte insbesondere demonstriert werden, dass sich KEG gut mit einem Spezifikationserzeugungswerkzeug kombinieren lässt, um eine vollautomatische Analyse des Programms zu erreichen.

Die zweite Fallstudie betrachtet einem der Literatur entnommenen Algorithmus zum Überprüfen der Integrität von PINs. KEG bewies hier, dass es unter Annahme einer ausreichend guten A-priori-Verteilung der Geheimniswerte einem Angreifer erlaubt, die geheimen PINs effizienter zu extrahieren.

Acknowledgments

First and foremost, I would like to take this opportunity to express my deepest gratitude to my thesis advisor, Prof. Dr. Reiner Hähnle, for everything he has done for me and my family. Reiner is a wonderful supervisor who always believes in me, encourages and gives me a lot of freedom to come up with new ideas, and provides me wise research guidance to keep me on track. Moreover, he is an extremely thoughtful, reliable friend whom I can always count on. His support was crucial for my family to settle down in Germany. I have learned a lot from him, not only in scientific research but also in many other aspects of life.

I would like to give my special thanks to Dr. Richard Bubel for all of his unconditional helps during my time in Darmstadt. Richard has always been the first one I look for whenever I have, for example, a raw research idea in mind to discuss, an implementation problem to solve, a piece of writing to revise, or even a mess of paperwork to clarify. He has never said no to any request from me. There is no doubt that his comments helped improve this thesis substantially.

I am grateful to Prof. Dr. David Sands for being the second reviewer of my thesis with many insightful feedbacks. I would like to thank all other members of the committee of my defense for their time and effort in reading and grading this thesis.

My sincere thanks go to my (former) colleagues who are also my good friends: Dr. Martin Hentschel for being my mentor in SED; Dr. Nathan Wasser for helping me in proof-reading and for showing me how interesting a boardgame could be; Antonio Flores Montoya, Dominic Steinhöfel, and Eduard Kamburjan for converting my office into a joyful place. Thank you all for the happiness and unforgettable memories we have together. I also would like to thank our secretary Gudrun Harris for all of her helps and also for being so nice to me in spite of my poor progress in German.

I was very lucky to be a member of two great communities: the KeY project and the RS^3 program. Both gave me the opportunity to meet a lot of excellent researchers. I would like to acknowledge all fruitful discussions and collaborations I had with them. Those helped expand my knowledge and also inspire many of my research ideas, some of which were realized in this thesis.

I thank program 165 for the scholarship that partly supports for the living of my family in Germany. My first contact with Reiner and Richard was carried out with the help of World University Service (WUS). I would like to thank people of WUS, especially Mr. Christoph Jöcker, for their support at the beginning of my PhD.

My family has adapted well in Germany thanks to the support of my big family in Vietnam as well as my friends in Germany. I am grateful to my parents and my parents in law for their unconditional support and love. I would like to thank all of my friends in Germany for all of their helps and more importantly, for the friendship that makes our life in Germany much happier.

Last but not least, I would like to express my deep appreciation to my wonderful wife Ha Thuong for her tremendous sacrifice and love. She postponed her career and gave up many opportunities in Vietnam to reunite with me in Germany. She gives me a true home for coming back after work and brings me two lovely angels, Hanh Chi and Mai Khue. Ha Thuong, I dedicate this thesis to you and our daughters. You are simply all of my life.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

We are living in a sharing, connecting world that heavily relies on a variety of IT systems to create, store and exchange information. With an enormous and dramatically increasing amount of data, protecting confidential data from improper observers/accessors is vital for such IT systems and their users. For example, online banking or e-commerce systems must keep the credit card number of a customer from the others. Mobile phone apps should not access or modify personal data such as photos, messages or locations without permission. Server-side programs must guarantee that user names and passwords of their clients cannot be read and changed by a third-party. We are talking about *confidentiality* and *integrity*, two of three main attributes of the well-known CIA triad in information security (the remaining is *availability*) [6]. This thesis focuses on *confidentiality*.

According to the standard for information security management systems (ISO27000), confidentiality "is the property, that information is not made available or disclosed to unauthorized individuals, entities, or processes". To keep confidential data from disclosure, several techniques and mechanisms have been devised and applied e.g. *access control* and *cryptography*. Access control restricts the access of authorized users to confidential resources by using, for example, an *access control list*. Cryptography provides confidentiality by encrypting private messages so that third-parties (i.e. adversaries), even if they are able to access the encrypted message (*cipher-text*), cannot infer the original information (*plain-text*).

The mentioned methods are definitely useful in the sense that they can limit the information that is released by a system. However, they are insufficient to guarantee about its *propagation* [104]. For example, access control is not able to ensure that confidential data, after being accessed by an authorized user, will not be sent to an unauthorized agent. Similarly, once ciphertext is decrypted, cryptography can do nothing to ensure that the secret information is not read by the adversary afterwards. General speaking, if unclassified data can somehow interfere confidential data (secret information) throughout program execution, one might gain partial or even complete knowledge about a secret by observing unclassified data at the time a program terminates. In this case confidentiality is likely to be broken.

To ensure end-to-end confidentiality of data for an IT system, *secure information flow*, which tracks and regulates information flows during program executions, is crucial. This thesis contributes to language-based information flow analysis with a focus on detection and exploitation of information flow leaks in programs. State-of-the-art approaches securing information flow in programs are discussed in Section 1.1. Section 1.2 sketches the approaches proposed in this thesis and highlights its main contributions. Important publications constituting the thesis are listed in Section 1.3. Section 1.4 outlines the structure and Section 1.5 fixes some notational conventions that are used throughout this thesis.

1.1 Information Flow Analysis

Secure information flow has a long history since the 1970s. Denning [43] pioneered the use of a *lattice model* of security levels to specify the *information flow policies* to which the IT system

needs to adhere. In language-based secure information flow [104], each program location, i.e. field or variable, is labeled by a security level. Those security levels form a lattice that specifies a *noninterference* policy: if the security level of a program variable x is *higher* than the security level of a program variable y, then information flow from x to y is prohibited. The opposite direction is allowed. Conventionally, x is called *high variable* and y *low variable*. Noninterference is usually too strict for practical applications, therefore it is often downgraded by *declassification policies* [105] that specify either what, who, where or when an information leak can be accepted.

In the past decades, much theoretical and practical work on information flow analysis of programs was developed, for example, [7, 12, 16, 17, 37, 39, 45, 54, 63, 69, 70, 72, 91, 94, 103, 107, 110, 111, 113, 117]. Its focus is to ensure that an outside agent with well-defined properties cannot infer secret inputs by observing (and initiating) several runs of a program. The methods developed in the area of information flow analysis can be classified into either *qualitative* approaches that check *whether* a program leaks confidential information, or *quantitative* approaches that measure *how much* information might be leaked.

1.1.1 Qualitative Approaches

Qualitative information flow analysis is concerned with the development of methods ensuring that programs do not leak secret information, i.e., that it is not possible to learn secret information by looking at publicly accessible output. We discuss first *static approaches*.

Type-based approaches [12, 63, 91, 103, 117]) are light-weight approaches utilizing *type systems* to perform taint analysis on programs w.r.t. a lattice of security levels. These approaches are usually automatic and very efficient (they can be polynomial in the program's size [112]) for checking large programs. However, type-based approaches are value-insensitive, i.e. the actual values of program variables are not taken into account. Because of that, type-based approaches lack precision and often report many false positives.

Program dependence graph-based approaches [55, 58, 62] are another light-weight static information flow analysis. A program dependency graph (PDG) captures data-dependence and control-dependence information of its instructions that can be used to check information flow of the program. PDG-based approaches are relatively efficient that can scale to 100 kLOC [55]. However, they are also value-insensitive and, like type-based approaches, tend to raise false alarms. A formal comparison of the precision of a type-based and a PDG-based information flow analysis was conducted in [83] showing that they have the same precision.

In contrast to light-weight approaches that tend to be automatic and efficient but imprecise, heavy-weight *logic-based* approaches [17, 39, 107] tend to be precise, but require sometimes nontrivial user interactions and are not fully automatic. Roughly speaking, such approaches capture information flow properties of programs by means of a program logic and use verification tools i.e. theorem provers to prove those properties. Logic-based approaches are usually precise in the sense that they can formally prove whether a program is secure. However, they suffer from an inevitable drawback: constructing the proofs is often expensive and requires expert interactions. Hence, it is difficult to apply logic-based approaches for real-world programs that are usually large and complex.

With static qualitative information flow analysis approach, it is difficult to obtain precision and automation at the same time. However, this can be achieved *dynamically* for a concrete program run. Security monitors (e.g., [7, 37]) raise a warning as soon as a program violates a given

security policy and try to contain the leak. Secure multi-execution ([45] and several follow-up papers [15, 67, 100]) determines whether a given concrete run might violate a policy. However, such *dynamic approaches* have their own problems. Because they only consider the single current execution, it is difficult for them to establish confidentiality on *all* possible executions like static approaches [102, 104]. In addition, dynamic approaches usually require their own runtime infrastructures that might be expensive to construct.

1.1.2 Quantitative Approaches

Qualitative information flow analysis tries to establish that a program is secure and reject programs as insecure otherwise. However, in case of a leak (even if allowed by a given declassification policy) it does not provide details about how much information is leaked. *Quantitative* information flow analysis [3, 4, 9, 69, 72, 98, 110, 111] complements qualitative approaches by measuring the amount of leaked information. Developers/auditors can use this value to decide whether the leakage is acceptable. In addition, quantitative information flow analysis can provide a means to compare two programs in terms of insecurity, i.e. program A is considered as *more secure* than program B if A leaks *less* information than B.

All quantitative information flow analysis approaches are based on a common scenario: the attacker observes the (observable) outcome of the program to learn something about the initial value of high input (the secret). Generally, to compute how much information can be leaked by a program, one has to quantify the amount of unknown information in the secret (w.r.t. the attacker) before running the program (the attacker's initial uncertainty about the secret) and after observing the output (the attacker's remaining uncertainty about the secret). Then, the leakage is measured as the difference between the initial uncertainty and the remaining uncertainty. This can be expressed with the following equation:

$$\text{information leaked} = \text{initial uncertainty} - \text{remaining uncertainty}$$

Many approaches borrow the notion of *entropy* in information theory to interpret the attacker's uncertainty about the secret, that is treated as a *random variable*. Most common metrics are Shannon entropy [31, 44], min entropy [110] and guessing entropy [9, 70]. Such entropies are computed via the *prior probability distribution* (aka *a prior*) of secret. Thus, the leakage, computed using above entropies, also depends on that distribution.

Channel-capacity, which is the maximum leakage over all possible prior distributions of high inputs, is used to address *worst-case* analysis. It is first defined in the work of Denning [44] as the maximum Shannon entropy-based leakage over all possible probability distribution of high inputs (Shannon-capacity). In [110, 111], Smith extends the definition of channel capacity for the case of min entropy-based leakage (min-capacity) and proves that if the program is deterministic, Shannon-capacity and min-capacity coincide and both are the logarithm of the number of possible observable outputs.

Due to the hardness of quantitative information flow problems [30, 119], instead of computing exactly the leakage, which can be very expensive, it might be easier to estimate the *upper bound* of leakage. Channel-capacity, by definition, provides a natural, precise upper bound. Computing the channel-capacity of a deterministic program requires one to estimate the number of possible observable output values. This is addressed in many works in the field of quantitative information flow analysis [81, 86, 93, 98].

All secret data seems to be valued equally in above security metrics. To address the scenario that some secret data might be worth more than the others, e.g. leaking the credit card number of a user is more harmful than leaking the address, some approaches have been proposed. The approaches proposed in [3, 25] use a *gain function* to characterize the benefit that an attacker can derive from a certain guess. Alvim et al. [4] uses *worth assignment* to map each structure (a part of the whole secret) to a *worth* value.

Clarkson et al. [33] claim that uncertainty is not an adequate metric for the case when the attacker models his knowledge about the secret in terms of a probability distribution of secret. In this case, after observing the output of a program's run, the uncertainty of the attacker about the secret is reduced whereas the attacker might become more wrong about the secret. Clarkson et al. point out that the attacker's distribution is subjective, therefore it should be treated as a *belief*. To address this scenario, they introduce an alternative metric for leakage, namely *accuracy*, that is the distance from the attacker's distribution (her *belief*) to the actual distribution of the secret.

In practice, the observable outputs usually depend not only on high inputs but also low inputs. For example, a typical password checker outputs either `succeed` or `fail` depending on whether the value chosen by the user (low input) is equal to the actual password (high input). Although information leakage depends on low input, there are only a few approaches taking low inputs into account. Some papers [69, 94] parameterize information leakage on low inputs. They assume that the attacker will choose low inputs that *maximize* the leakage but do not point out how to find those optimal values. This problem is addressed in [70, 96] that target only side-channel attacks. Nevertheless, those approaches only either use channel-capacity as a security metric or consider the case that the prior distribution of high inputs is uniform.

Although channel-capacity is useful in the sense that it gives the precise upper bound of information that might be leaked, it is often too pessimistic. We illustrate this by the example taken from [8]. A typical password checker returns two possible observable outcomes `succeed` and `fail`, thus the capacity of the channel from secret passwords to observable outputs is 1 bits, corresponding to the distribution that assigns probability 0.5 for both outcomes. Consequently, a naive analysis might infer that an n-bit password can be discovered completely after at most n login attempts hence conclude that the system is insecure. However, that distribution is not likely to happen in practice. If the password is well-chosen from a large set of values so that its distribution is almost uniform, a login attempt leaks much less than one bit and the password checker is in fact secure.

On the other hand, assuming that the high inputs have a uniform distribution might severely restrict the applicability of a quantitative approach. For example, the password or PIN chosen by the users are non-uniform: typically some values can have higher probability, i.e. "`123456`" or "`password`" [24]. The adversary can obviously leverage the information of a secret's distribution to attack the system more effectively. Approaches that only work on the assumption that the secret has uniform distribution might not be adequate for such attack scenario, thus might fail to judge precisely about the vulnerability of the application.

Most quantitative information flow analysis approaches are static: program source code is analyzed to induce a numeric value indicating the severity of an information flow leak. Although static quantitative approaches might be able to estimate a program's vulnerability, or even to compare the vulnerabilities of different programs, they can be wrong *in practice* because of their approximative nature (the over-pessimistic judgment based on channel capacity metric for password checker is an example). Moreover, such existing approaches do not point out how an attacker can *exploit* the leak to infer the secret in practice. This suggests a complementarity

from dynamic analysis that actually runs a program. Although some hybrid/dynamic approaches exist [72, 85], they only concentrate on estimating the leakage instead of exploiting discovered leaks.

1.2 Approaches and Contributions of The Thesis

The work presented in this thesis is concerned with both qualitative and quantitative information flow analyses, and combines static and dynamic techniques. We highlight the approaches devised in this thesis in Section 1.2.1 and summarize its main contributions in Section 1.2.2.

1.2.1 Information Flow Analysis of The Thesis

Detection and Demonstration of Information Flow Leaks

This thesis contributes to qualitative information flow analysis by a novel approach detecting and demonstrating information flow leaks in programs. This approach connects the static and dynamic view: by static analysis we discover as many as possible information flow leaks and then produce test cases for such detected leaks, so called *leak demonstrators*, that are guaranteed to violate a given security policy. Such leak demonstrators can then be used independently for system testing, regression testing, documentation, etc. We take a similar view here as in functional verification: an information flow policy can be seen as a requirements specification. Its violation is a software fault witnessed by a test case, i.e. leak demonstrator.

In functional verification it is well-known that static verification is not a replacement for testing, but both techniques complement each other [19]: static verification gives strong guarantees for the modeled part of a system, while testing is incomplete, but validates actual executables and can also find problems in the underlying runtime environment or hardware platform. In our work we intend to achieve similar goals as in white-box software test generation frameworks [2, 51, 40]:

- Under the assumption that a sound and complete specification is provided, no further user interaction is required and the approach is automatic.
- Completenness is achieved only in specific cases (e.g., no loops present or else strong loop invariants are supplied), however, strong and precise coverage guarantees can be given.
- Like test cases, leak demonstrators can be used to validate a program in its actual runtime environment.
- Like test cases, leak demonstrators become part of a library that is regularly executed to protect against regression. They are useful even after changes were made.
- Like test cases, leak demonstrators can serve as a documentation and illustration of intended system behavior.

Like other white-box software test generation frameworks [2, 51, 40, 68], the leak detection and demonstrator generation approach proposed in this thesis is based on symbolic execution of the target program. The relational nature of information flow (two runs of a program must be

compared) is captured by the technique of self-composition (first introduced in [38]; the name self-composition was coined in [17]). The result is an *insecurity formula* for a given information flow policy that is satisfiable if and only if the policy is violated. Model generation with the help of SMT solvers [41] yields the input data for the leak demonstrator. In addition to standard information flow policies like noninterference, we also support relativized properties that tend to be used in practice, including delimited information release [105] and information erasure.

Unlike monitoring and multi-execution, no special runtime infrastructure is required, because we generate leak demonstrators in the form of self-contained JUnit tests. These run the program under test multiple times with the generated input in order to produce a security violation. In addition, the generated leak demonstrators come with statically determined coverage guarantees.

A huge problem in software testing is the creation of oracles [13] that tell whether a test succeeded or not. Among the problems are i) missing or insufficient specifications, ii) complexity of general functional specifications which might contain quantifications over all objects and similar, and iii) possible unintended side-effects of specification code (runtime exceptions and similar). Even in automatic test generation, these must often be supplied manually. But information flow policies (see [105] for an overview) can usually be expressed in a uniform manner for any given program. Hence, it is possible to generate test oracles automatically from them. In fact, our approach goes one step further in being *oracle-sensitive*: only such leak demonstrators that violate a given policy are generated at all.

In this thesis, the leak detection/demonstrator generation approach is explained in detail in Chapter 3. Chapter 6 presents the implementation and evaluation for this approach. An e-voting case study showcasing the approach is given in Chapter 7.

Quantification and Exploitation of Information Flow Leaks

As discussed in Section 1.1.2, existing approaches in quantitative information flow analysis do not provide a proper answer for the question “how easy an attacker can learn the secret input by observing the public output of a program in practice?”. This thesis proposes a novel approach tackling this problem for deterministic programs. We *exploit* detected leaks to construct a multiple-run adaptive attack that allows the attacker to learn the values of high inputs by performing a series of *experiments*. An experiment constitutes a concrete program run which serves to accumulate the *knowledge* about the secret (high inputs). Such knowledge is a set of logic constraints for the actual input values of high variables. It helps to infer the whole secret or at least to narrow down the space of possible values of the secret. The attack is adaptive because the attacker can use her latest knowledge of the secret to deduce the low inputs for the next experiment. During the attack, the secret is assumed to remain unchanged.

To judge the severity of information leaks, the work presented in this thesis applies techniques developed for quantified information flow analysis to guide the systematic creation of an (as small as possible) set of experiments to be conducted to gain *maximal* knowledge about a secret. The set of experiments is built incrementally. New experiments are added only if they are non-redundant and lead to a “maximal” knowledge gain. This sets our approach apart from previous approaches [9, 69, 98] that use a random set of experiments (or simply state the existence of such a set) and enables us to obtain a tighter characterization of secret.

The low inputs to be used in experiments are chosen so that at least they bring non-redundant experiments with respect to current knowledge (that the attacker can learn something new

about the secret), or in the best case they are *optimal*: they *maximize* the information leakage with respect to a specific security metric. To find optimal low inputs, we quantify information leakages as *explicit* functions of low inputs. Those functions are constructed using path conditions and symbolic output values of observable variables. Optimal low inputs are generated by solving an optimization problem, employing either (non-linear) optimization tools [21, 53] or Max-SMT solvers [14, 41]. The secret inference approach presented in this thesis differs from (and advances) [70, 96] in the sense that an arbitrary prior distribution of secret is taken into account, and information theoretic entropy-based leakage metrics, i.e. Shannon, min and guessing entropy are used, while [70, 96] either ignore that distribution and use channel-capacity as leakage metric, or deal only with the assumption that the possible secret values have a uniform distribution.

By choosing low inputs that potentially maximize gained knowledge, this approach also optimizes the attack strategy of the attacker and hence, can provide an upper bound for the severity of information leaks. Of course, finding optimal low inputs is more expensive than just simply choosing a random one. However, it might be crucial in many attack scenarios, especially with the case that the maximum number of experiments that the attacker can perform is limited, e.g. a password checker might accept at most three consecutive failed login attempts. Moreover, it addresses the worst-case scenario in which the computation power of the attacker is assumed to be unlimited.

The secret inference approach proposed in this thesis can be seen as an *exploit*-oriented approach in nature: it creates a small program to run the target system multiple times to learn the secret. Actually, information leaks, detected by the leak detection approach proposed in this thesis, can be exploited further: only risky paths (symbolic paths that might cause information leaks) and reachable paths are taken into account while computing the leakage and constructing the knowledge of secret.

In this thesis, the fundamental concepts of the secret inference approach is introduced in Chapter 4. Chapter 5 explains in detail how to quantify the information leakage by parametric counting or Max-SMT problem and how to find optimal low input maximizing such leakage. The implementation and evaluation of this approach is given in Chapter 6. Chapter 8 presents a PIN integrity checking case study for this approach.

1.2.2 Contributions

In summary, the main contributions of this thesis are:

- A novel approach for detecting all possible information flow leaks based on self-composition and symbolic execution (Sections 3.1). In particular, this approach utilizes program specifications i.e. loop invariants and method contracts to deal with unbounded loops and recursion (Section 3.4).
- A generalization for noninterference that supports more information flow polices, i.e. *information erasure* (Sections 3.2), and an extension of delimited information release policy addressing acceptable targets as well as conditions for declassification (Section 3.3).
- A technique to generate automatically *leak demonstrators* that can be used as fail test cases for leaks. Leak demonstrators can also be used for regression test and do not require any special runtime infrastructure (Section 3.5).

- A novel approach judging the severity of information leaks by simulating an adaptive attack to infer the secret (high inputs). This approach combines static and dynamic analysis to maximize the gained knowledge about the secret while minimizing the number of experiments/program runs (Sections 4.1, 4.3, and 4.4).

- A logic characterization of the secret synthesized from multiple concrete program runs and results obtained from symbolic execution (Section 4.2).

- A novel approach quantifying leakages as *explicit* functions of low inputs using symbolic execution and parametric counting. This approach uses Shannon, guessing or min entropy as security metric. In particular, this approach can take into account non-uniform distributions of the secret if the chosen security metric is Shannon or guessing entropy (Section 5.1).

- An algorithm finding optimal low inputs using Max-SMT solvers that min entropy-based leakage is quantified by means of a Max-SMT problem of low inputs (Section 5.2.2).

- An algorithm finding optimal low inputs using nonlinear optimization tools, applied for the cases i) the leakage metric is based on either Shannon or guessing entropy; and ii) the leakage metric is based on min entropy, and the prior distribution of the secret is uniform (Section 5.2.1).

- A tool, namely KEG, that implements the proposed approaches on top of the state-of-the-art deductive verification framework KeY and supports a rich subset of sequential Java (Sections 6.1 and 6.2).

- An evaluation for KEG using micro-benchmarks (Section 6.3) and two case studies:
  - An e-voting case study (Chapter 7): KEG is used to detect and demonstrate all possible information flow leaks in a simplified e-voting program adapted from a real-world e-voting system (Section 7.2). A fully automatic logic-based approach combining KEG with another specification generation tool is also proposed (Section 7.3).
  - A PIN integrity checking case study: KEG is used to infer the PIN value of a PIN integrity checking program adapted from a real-world ATM PIN verifying system (Chapter 8).

1.3 Publications

This thesis includes some previous publications on well established journal and conferences. The author of this thesis is main author of all publications being used in the thesis. To be more specific, the author of this thesis designed main theories, implemented proposed approaches, carried out all experiments/evaluations, and was responsible for the majority of writing.

Parts of Chapter 1, Chapter 2, Chapter 6 and Chapter 9 are based on [47] and [48]. The content of Chapter 3 heavily relies on [47] and [48]. Chapter 7 is a synthesis of [48] and [49]. Parts of Chapter 4, Chapter 5 and Chapter 8 appear in the technical report [46].

Further journal/conference publications of the author that are not directly relevant to this thesis are [28, 66, 65].

1.4 Structure of The Thesis

This thesis consists of ten chapters, beginning with the introduction given in this chapter. Chapter 2 introduces the necessary background for this thesis. The approach detecting and demonstrating information flow leaks is presented in Chapter 3. The secret inference approach is introduced in Chapter 4. Chapter 5 discusses in detail how to generate optimal low inputs maximizing information leakage. Chapter 6 describes the KEG tool, performs an evaluation using a collection of micro benchmarks, and gives some insightful discussions. Bigger case studies showcasing KEG's features are given in the next two chapters. Chapter 7 presents an e-voting case study that utilizes the leak detection and demonstrator generation features. Chapter 8 introduces a PIN integrity checking case study that mainly focuses on secret inference and leakage quantification functions of KEG. Related works are discussed in Chapter 9. Finally, Chapter 10 concludes this thesis and points out some future works.

1.5 Notational Conventions

Throughout this thesis, the following notational conventions are used:

- Similar to [56], we always use the standard equality symbol $=$ to indicate equality on the *meta-level*. To represent the equality predicate in logic, we use its counterpart $\doteq$. The same for the inequality sign $\neq$ and its logical counter parts $\not\doteq$.
- For three sets X, Y, Z, we write $X \dot\cup Y = Z$ to represent that Z is partitioned into X and Y, i.e. $X \cup Y = Z$ and $X \cap Y = \emptyset$.
- The power set of a set S is denoted by 2^S, $|S|$ is used to refer the cardinality of S.
- Given an ordered set of variables $V = \{v_1, \ldots, v_n\}$ and an ordered set $V' = \{v'_1, \ldots, v'_n\}$ having the same cardinality with V in which each v'_i is either a concrete value in the domain of v_i or a variable having the same domain as v_i, for an expression e we use $e[V'/V]$ to represent the expression obtained by replacing each v_i occurring in e by v'_i. In case of two disjoint variables sets V_1, V_2 we write $e[V'_1, V'_2 \,/\, V_1, V_2]$ instead of $e[V'_1/V_1][V'_2/V_2]$.
- For an ordered set of expressions $S_e = \{e_1, \ldots, e_n\}$, we write $S_e[V'/V]$ to denote the ordered set $\{e_1[V'/V], \ldots, e_n[V'/V]\}$. Similarly to the set $S_e[V'_1, V'_2 \,/\, V_1, V_2]$.
- If an expression is explicitly represented by an ordered set of all free variables occurring in it, i.e. $e(V)(V = \{v_1, \cdots, v_n\})$, for the sake of readability we simply use $e(V')$ $(V' = \{v'_1, \cdots, v'_n\})$ instead of $e(V)[V'/V]$. Likewise, we write $e(V'_1, \ldots, V'_m)$ to represent the expression obtained from $e(V_1, \ldots, V_m)$ by replacing all variables in V_i by their counterparts in V'_i.
- For two ordered, disjoint sets $S = \{x_1, \ldots, x_n\}$ and $S' = \{x'_1, \ldots, x'_n\}$ having the same cardinality, to represent the formula $\bigwedge_{i=1}^{n} x_i \doteq x'_i$ we simply write $S \doteq S'$. We also apply this for the negation form that $S \not\doteq S'$ means $\bigvee_{i=1}^{n} x_i \not\doteq x'_i$.
- For a logic formula f and V the set of all free variables occurring in f, we denote by $Sat(f)$ the set of all different concrete models of V that satisfy f.

- For a singleton set $\{x\}$, we usually use x instead of $\{x\}$ when the set is a parameter of a function while the context is unambiguous, or an operand in a set operation.
- All logarithms given without explicit base value are binary logarithm (the base is 2).

2 Preliminaries

This chapter provides the background of this thesis. The basic notions of qualitative and quantitative information flow security are introduced in the two first sections, including *insecurity policies* (Section 2.1) and *leakage metrics* (Section 2.2). The information flow analysis approach proposed in this thesis relies on *self-composition* that is presented in Section 2.3 and *symbolic execution* (Section 2.4). Section 2.5 mentions program specifications i.e. method contracts and loop invariants, as well as their representations in JML. Finally, Section 2.6 introduces the KeY system and its symbolic execution engine, that is the back-end for the implementation. Parts of this chapter are based on previous works of the author of this thesis, including two formal publications [47, 48] and a technical report [46].

2.1 Information Flow Policies

Before we can analyze that a program does not leak confidential information, we need to define the *security requirements*. This has two aspects: the *security level* of each program location (i.e. program variables and fields) as well as an *information flow policy* defining whether and what kind of information may flow between program locations with a different security level.

We recall the definitions of two well-known information flow policies supported by our approach.

2.1.1 Noninteference

Noninterference [35, 117] is the strongest possible information flow policy. It typically involves two security levels (high/confidential vs. low/public) and completely prohibits any information flow from program locations containing confidential information to publicly observable program locations. The opposite direction is allowed. This thesis considers only deterministic programs. In this case, noninterference can be formalized by comparing two program runs:

Definition 2.1 (Noninterference—Informal)**.** *A program has secure information flow with respect to noninterference, if any two executions of the program starting in initial states with identical values of the low variables, also end in final states which coincide on the values of the low variables.*

In other words the final value of low variables is solely determined by the initial value of low variables and does not depend on the initial values of high variables.

We define some basic notions required to formalise information flow policies. In the remaining thesis we use p to denote a program and $Var = \{x_1, \ldots, x_n\}$ to denote an ordered set of all program variables occurring in p.[1]

[1] To keep the presentation manageable, in the formal definitions we mention only variables, however, our implementation works also for fields

Definition 2.2 (Program State)**.** *A program state σ maps each program variable $v \in Var$ of type T (write $v : T$) to a value of its concrete domain D^T, i.e.:*

$$\sigma : Var \to D$$

with $\sigma(v : T) \in D^T$ and D being the union of all concrete domains. The set of all states for a given program p is denoted as $States_p$.

We define coincidence of program states relative to a set of program variables:

Definition 2.3 (State Coincidence)**.** *Given a set of program variables V and two states $\sigma^1, \sigma^2 \in States_p$. We write $\sigma^1 \simeq_V \sigma^2$ if and only if σ^1 and σ^2 coincide on V, i.e., $\sigma^1(v) = \sigma^2(v)$ for all $v \in V$.*

A *concrete execution trace* τ of a program p is a possibly infinite sequence of program states $\tau = \sigma_0\sigma_1\sigma_2\cdots$ produced by starting p in state σ_0. In this thesis, we concern ourselves only with terminating programs, consequently, all of our execution traces are finite. Then the big-step semantics is defined as follows: let X be a concrete execution of a program p defined by a trace τ^X. We represent X by a pair $\langle \sigma^X, \sigma^X_{out} \rangle$, where $\sigma^X \in States_p$ is the first state of τ^X and $\sigma^X_{out} \in States_p$ is the last. The set of all possible concrete executions of p is denoted as Exc_p. We can now formally define noninterference for two security levels low and high:

Definition 2.4 (Noninterference)**.** *Given a program p over variables Var and a* noninterference policy *$NI = H \not\rightsquigarrow L$ where $L \dot\cup H = Var$ such that L contains the low variables and H the high variables. Program p has secure information flow with respect to NI if and only if for all concrete executions $X, Y \in Exc_p$ it holds that if $\sigma^X \simeq_L \sigma^Y$ then $\sigma^X_{out} \simeq_L \sigma^Y_{out}$.*

Example 2.1. *The program*

```
if ( h > 0 ) { l = 2; }
```

with high variable `h` *and low variable* `l` *is insecure as it does not satisfy the noninterference property for the policy $NI = \{h\} \not\rightsquigarrow \{l\}$. Given two initial states σ^1 with $\sigma^1(h) = 5$, $\sigma^1(l) = 0$ and σ^2 with $\sigma^2(h) = -5$, $\sigma^2(l) = 0$, respectively. They satisfy $\sigma^1 \simeq_{\{l\}} \sigma^2$, but in the final states we have $\sigma^1_{out}(l) = 2 \neq \sigma^2_{out}(l) = 0$.*

2.1.2 Declassification

In practice noninterference is too restrictive. For instance, a program that authenticates users with their login password leaks the information whether an entered password is correct. Or take a database that may be queried for aggregated values like the average salary, but not for the income of an individual person.

Declassification is a class of information flow policies that allows one to express that some precisely specified confidential information may be leaked. The paper [105] provides an extensive survey of declassification approaches. Here we consider *delimited information release* as introduced in [103]. Delimited information release is a declassification policy which allows one to specify what kind of information may be released. To this end, so called *escape hatch*

expressions are specified in addition to the security level of the program locations. For instance, the escape hatch

$$\frac{\sum_{e \in Person} salary(e)}{|Person|}$$

can be used to declassify the average of the income of all persons in a database. The formal definition of delimited information release extends Definition 2.4. Both definitions coincide for trivial escape hatches such as $e = true$.

Definition 2.5 (Delimited Information Release)**.** *Given a program p over variables Var and a* delimited information release policy *$Decl = (L, H, E)$ with L, H as before and E denoting a set of escape hatch expressions. Program p has secure information flow with respect to Decl if and only if for all concrete executions $X, Y \in Exc_p$ it holds that if $\sigma^X \simeq_L \sigma^Y$ and for all $e \in E : [\![e]\!]_{\sigma^X} = [\![e]\!]_{\sigma^Y}$, then $\sigma^X_{out} \simeq_L \sigma^Y_{out}$. The expression $[\![e]\!]_\sigma$ denotes the semantic evaluation of e in state σ.*

Example 2.2. *Consider again the program from Example 2.1:*

```
if ( h > 0 ) { l = 2; }
```

Given the delimited information release policy $Decl = (\{l\}, \{h\}, \{h > 0\})$ where the escape hatch allows the sign of h to be leaked. The counter example for noninterference from Example 2.1 is no longer a counter example as

$$[\![h > 0]\!]_{\sigma_1} = true \neq false = [\![h > 0]\!]_{\sigma_2} \ .$$

In fact the program is secure for the given policy, as the decision whether l may be altered is only based on the guard. The same policy for the program

```
if ( h > 0 ) { l = h; }
```

is insecure as can be demonstrated by the following counter example: Given initial states σ^1 with $\sigma^1(h) = 3$, $\sigma^1(l) = 0$ and σ^2 with $\sigma^2(h) = 5$, $\sigma^2(l) = 0$ we observe (i) both states coincide on the value of the low variable l; and (ii) they evaluate the escape hatch expression to the same value $[\![h > 0]\!]_{\sigma_1} = [\![h > 0]\!]_{\sigma_2} = true$, but their final states differ on the value of l: $\sigma^1_{out}(l) = 3 \neq 5 = \sigma^2_{out}(l)$.

2.2 Quantification of Information Leakage

In this section we introduce notions of quantitative information flow analysis and leakage metrics from the literature to the degree required for this thesis. To take low inputs into account, we adapt the approach proposed in [119].

2.2.1 Measuring Leakage by Uncertainty

Given a program p and a noninterference policy $H \not\rightsquigarrow L$. Let $O \subseteq L$ (usually: $O = L$) be a subset of low variables whose value can be observed by an attacker after termination of p. We assume that before running p, the attacker knows about the values of low variables (or can even manipulate

them); and that the initial values of variables in H and L are independent (i.e. from an attacker's perspective knowledge about L does not entail any knowledge about H).

Conventionally, the amount of information that is leaked from H to O can be measured by quantifying the amount of unknown information about H's value (the secret) w.r.t. the attacker before running the program (the attacker's initial uncertainty about the secret) and after observing the output value of O (the attacker's remaining uncertainty about the secret). Then information leakage from H to O can be seen as the reduction of uncertainty of the attacker about the values of H that can be achieved by observing the final values of the variables in O after a run of program p:

$$\text{information leaked} = \text{initial uncertainty} - \text{remaining uncertainty}$$

Let $\mathbb{L}, \mathbb{H}$ denote the finite sets of all possible values of L and H, e.g., for two 32-bit integer program variables $H = \{h_1, h_2\}$, $\mathbb{H}$ is the Cartesian product $\mathbb{Z}_{32} \times \mathbb{Z}_{32}$ of their domain. Similarly, let $\mathbb{O}$ be the set of all possible output values of O. Let the function $\mathbb{O}_D : \mathbb{L} \to 2^{\mathbb{O}}$ that computes the set of all possible output values of O for a given low input be defined as follows:

$$\mathbb{O}_D : \bar{l} \mapsto \{\bar{o} \mid \bar{o} \text{ final values of } O \text{ after executing } p(\bar{l}, \bar{h}), \text{for each } \bar{h} \in \mathbb{H}\}$$

Each low input value $\bar{l}$ defines a random variable $O_{out}(\bar{l})$ corresponding to the observed output values in the set $\mathbb{O}_D(\bar{l})$ after running program p with fixed low level input $\bar{l}$. We denote with $O_{out}(L)$ the function from $\mathbb{L}$ to the space of random variables as defined above. The random variables corresponding to the initial values of H are denoted with H_{in}.

The following subsections describe different measures to compute information leakage of a program p with parameter L. We discuss three possible definitions of leakage: $\texttt{ShEL}_\texttt{p}(L)$, $\texttt{MEL}_\texttt{p}(L)$, and $\texttt{GEL}_\texttt{p}(L)$ based on Shannon entropy, min entropy, and guessing entropy, respectively. The last subsection introduces channel-capacity that is the upper bound of the leakage over all prior distribution of secret (the initial value of high variables).

2.2.2 Shannon Entropy

Shannon entropy, as introduced by C. E. Shannon [109], is widely used in quantifying information flow leaks [31, 44]. The formal definition of Shannon entropy and its conditional variant are given in Definition 2.6. Recall that all logarithms given without an explicit base are the binary logarithm (base is 2).

Definition 2.6 (Shannon and conditional Shannon entropy)**.** *Given random variables X, Y with sample space $\mathbb{X}$ and $\mathbb{Y}$, respectively. The* Shannon entropy *of X is defined as*

$$\mathscr{H}(X) = -\sum_{x \in \mathbb{X}} P(X = x)\texttt{log}(P(X = x))$$

and the conditional Shannon entropy *of X given Y as*

$$\mathscr{H}(X|Y) = \sum_{y \in \mathbb{Y}} P(Y = y)\mathscr{H}(X|Y = y)$$

where $\mathscr{H}(X|Y = y) = -\sum_{x \in \mathbb{X}} P(X = x|Y = y)\texttt{log}(P(X = x|Y = y))$.

Intuitively, $\mathcal{H}(X)$ is the average number of bits required to encode the values of X and $\mathcal{H}(X|Y=y)$ quantifies the average number of bits needed to describe the outcome of X under the condition that the value of Y is known.

Shannon entropy and its conditional variant are used to quantify information leakage as follows: the initial uncertainty of the attacker about the initial values of the high variables is interpreted as Shannon entropy of H_{in}, while the remaining uncertainty of the attacker about H_{in} when $O_{out}(L)$ is known is interpreted as conditional entropy. Then the information leakage is the *mutual information* of H_{in} and $O_{out}(L)$:

$$\mathtt{ShEL}_{\mathtt{p}}(L) = I(H_{in}; O_{out}(L)) = \mathcal{H}(H_{in}) - \mathcal{H}(H_{in}|O_{out}(L))$$

Theorem 2.1. *If program p is deterministic then the Shannon entropy-based leakage can be computed as*

$$\mathtt{ShEL}_{\mathtt{p}}(L) = \mathcal{H}(O_{out}(L)) \tag{2.1}$$

Proof. Because mutual information is symmetric, we get

$$\mathtt{ShEL}_{\mathtt{p}}(L) = I(H_{in}; O_{out}(L)) = \mathcal{H}(O_{out}(L)) - \mathcal{H}(O_{out}(L)|H_{in})$$

To prove Theorem 2.1, we will prove that $\mathcal{H}(O_{out}(L)|H_{in}) = 0$ in case program p is deterministic. By Definition 2.6 we have

$$\begin{aligned}\mathcal{H}(O_{out}(L)|H_{in}) &= \sum_{\overline{h}\in\mathbb{H}} P(H_{in}=\overline{h})\mathcal{H}(O_{out}(L)|H_{in}=\overline{h}) \\ &= -\sum_{\overline{h}\in\mathbb{H}} P(H_{in}=\overline{h}) \sum_{\overline{o}\in\mathbb{O}_D(L)} P(O_{out}(L)=\overline{o}|H_{in}=\overline{h})\mathtt{log}(P(O_{out}(L)=\overline{o}|H_{in}=\overline{h}))\end{aligned}$$

Let $\overline{l}_0$ be an arbitrary value of L and $\overline{o}_0$ an arbitrary value $\in \mathbb{O}_D(\overline{l}_0)$. Let $\mathbb{H}[\overline{l}_0, \overline{o}_0] \subseteq \mathbb{H}$ be the set of all values $\overline{h}_0 \in \mathbb{H}$ satisfying that at final state obtained after executing program p with initial state $L=\overline{l}_0, H=\overline{h}_0$ the output value of O is $\overline{o}_0$. Because $\overline{o}_0 \in \mathbb{O}_D(\overline{l}_0)$, by definition of $\mathbb{O}_D$ we have $\mathbb{H}[\overline{l}_0, \overline{o}_0] \neq \emptyset$. Because program p is deterministic, we can see that

$$P(O_{out}(\overline{l}_0)=\overline{o}_0|H_{in}=\overline{h}) = \begin{cases}1, & \text{if } \overline{h}\in\mathbb{H}[\overline{l}_0,\overline{o}_0] \\ 0, & \text{otherwise}\end{cases} \tag{2.2}$$

From (2.2), it is easy to see that

$$\sum_{\overline{o}\in\mathbb{O}_D(\overline{l}_0)} P(O_{out}(\overline{l}_0)=\overline{o}|H_{in}=\overline{h})\mathtt{log}(P(O_{out}(\overline{l}_0)=\overline{o}|H_{in}=\overline{h})) = 0$$

with $\overline{l}_0 \in \mathbb{L}$ is an arbitrary value of L. Hence $\mathcal{H}(O_{out}(L)|H_{in}) = 0$ and Theorem 2.1 is proven. □

Example 2.3. *Consider a simple password checker pwc as follows:*

$$\mathtt{pwc} \equiv \texttt{if (h==l) l=1; else l=0;}$$

Program `pwc` *models a simple password checker that determines whether a value given by the user is a correct password by comparing it with the actual secret password. Here* `h` *is the password (high value) and* `l` *is the guess chosen by the user (low value). For the sake of simplicity, we assume that the attacker can observe the output value of* `l` *to decide whether the input value of* `l` *is the password or not (in practice, a real-world password checker usually returns a notification for a wrong guess and accepts the correct one silently). It is easy to see that* `pwc` *is insecure w.r.t. the noninterference policy* $h \not\rightarrow l$. *We concentrate on measuring how much information of* `h` *could be leaked to the attacker after running* `pwc`.

Assume that `h` *is a* 32 *bits integer number, e.g.* $-2^{31} \leq h < 2^{31}$. *It is obvious that program* `pwc` *is deterministic. We make an assumption that the distribution of the input value of* `h` *is uniform. The initial uncertainty (measured by Shannon entropy) of the attacker about the input value of* `h` *is:*

$$\mathscr{H}(h_{in}) = -\sum_{i=-2^{31}}^{2^{31}-1} P(h_{in} = i)\mathtt{log}(P(h_{in} = i)) = -\sum_{i=-2^{31}}^{2^{31}-1} \frac{1}{2^{32}}\mathtt{log}(\frac{1}{2^{32}}) = 32$$

Denote the initial value of `l` *by* l_0 (l_0 *is chosen by the attacker), the remaining uncertainty of the attacker after observing the output value of* `l` *is:*

$$\begin{aligned}\mathscr{H}(h_{in}|l_{out}(l_0)) = &-P(l_{out}(l_0) = 1)\sum_{i=-2^{31}}^{2^{31}-1} P(h_{in} = i|l_{out}(l_0) = 1)\mathtt{log}(P(h_{in} = i|l_{out}(l_0) = 1))\\ &-P(l_{out}(l_0) = 0)\sum_{i=-2^{31}}^{2^{31}-1} P(h_{in} = i|l_{out}(l_0) = 0)\mathtt{log}(P(h_{in} = i|l_{out}(l_0) = 0))\end{aligned}$$

If $l_0 \notin [-2^{31}, 2^{31}-1]$, *we have*

$$P(l_{out}(l_0) = 1) = 0,\ P(l_{out}(l_0) = 0) = 1$$
$$\forall i \in [-2^{31}, 2^{31}-1].\ P(h_{in} = i|l_{out}(l_0) = 0) = \frac{1}{2^{32}}$$

In this case $\mathscr{H}(h_{in}|l_{out}(l_0)) = 32 = \mathscr{H}(h_{in})$, *hence* $\mathtt{ShEL}_{\mathtt{pwc}}(\mathtt{l}_0) = 0$ *which means that there is no information of* `h` *being leaked.*

If $l_0 \in [-2^{31}, 2^{31}-1]$, *we have:*

$$P(l_{out}(l_0) = 1) = \frac{1}{2^{32}},\ P(l_{out}(l_0) = 0) = \frac{2^{32}-1}{2^{32}}$$
$$\forall i \in [-2^{31}, 2^{31}-1].\ P(h_{in} = i|l_{out}(l_0) = 1) = \begin{cases}1, & \text{if } i = l_0\\ 0, & \text{otherwise}\end{cases}$$
$$\forall i \in [-2^{31}, 2^{31}-1].\ P(h_{in} = i|l_{out}(l_0) = 0) = \begin{cases}0, & \text{if } i = l_0\\ \frac{1}{2^{32}-1}, & \text{otherwise}\end{cases}$$

In this case we can compute the remaining uncertainty $\mathscr{H}(h_{in}|l_{out}(l_0)) \approx 31.9999999922$ *which means that the leakage is* $\mathtt{ShEL}_{\mathtt{pwc}}(\mathtt{l}_0) \approx 0.0000000078$.

Now we consider the case that the space of password is narrowed down, i.e. `h` *is an 8 bits integer number (*$-128 \leq h \leq 127$*). Similarly to above, we can compute the Shannon entropy-based leakage for the case* $l \in [-128, 127]$ *is approx.* 0.037, *which is much greater than the case of 32 bits password. The quantitative information flow analysis confirms the intuition that password checking program is relatively safe w.r.t. brute force attack if the space of the password is large enough and the distribution of the password is almost uniform.*

2.2.3 Min Entropy

While Shannon entropy is a natural way to quantify leakage, it fails to reflect the vulnerability that high values might be guessed correctly in a single try. Consider the two programs

$$p_1 \equiv \texttt{if (h\%8==0) l=h else l=1}, \qquad p_2 \equiv \texttt{l=h\&0777}$$

taken from [110]. Using Shannon entropy, the mutual information leakage of program p_1 is smaller than that of p_2, i.e., p_1 is considered to be more secure than p_2. However, the risk of leaking the complete value of h in a single run is significantly higher for p_1 than for p_2. Smith [111] proposed *min entropy* as an alternative metric to address this problem. As Smith focuses on programs without low input, we use the extension given in [119]:

Definition 2.7 (Min and conditional min entropy)**.** *Given random variables X, Y with sample space $\mathbb{X}$ and $\mathbb{Y}$, respectively. The* min entropy *of X is defined as*

$$\mathscr{H}_\infty(X) = -\log \mathscr{V}(X)$$

and the conditional min entropy *of X given Y as*

$$\mathscr{H}_\infty(X|Y) = -\log \mathscr{V}(X|Y)$$

where $\mathscr{V}(X) = max_{x\in\mathbb{X}} P(X=x)$ and $\mathscr{V}(X|Y) = \sum_{y\in\mathbb{Y}} P(Y=y) max_{x\in\mathbb{X}} P(X=x|Y=y)$.

Intuitively, the min entropy of X represents the highest probability that X can be guessed in a single try. Using min entropy allows to measure information leakage as follows: the initial uncertainty is interpreted as min entropy of H_{in} and the remaining uncertainty is the conditional min entropy of H_{in} given O_{out}. The min entropy-based leakage becomes then

$$\mathtt{MEL}_\mathtt{p}(L) = \mathscr{H}_\infty(H_{in}) - \mathscr{H}_\infty(H_{in}|O_{out}(L)) = \log \frac{\mathscr{V}(H_{in}|O_{out}(L))}{\mathscr{V}(H_{in})} \tag{2.3}$$

Theorem 2.2. *If program p is deterministic and H_{in} is uniformly distributed then the min entropy-based leakage can be computed as*

$$\mathtt{MEL}_\mathtt{p}(L) = \log|\mathbb{O}_D(L)| \tag{2.4}$$

Proof. Because the probability distribution of H_{in} is uniform, by definition 2.7 we have

$$\mathscr{V}(H_{in}) = \frac{1}{|\mathbb{H}|} \tag{2.5}$$

By Definition 2.7, we have

$$\begin{aligned}
\mathscr{V}(H_{in}|O_{out}(L)) &= \sum_{\overline{o}\in\mathbb{O}_D(L)} P(O_{out}(L)=\overline{o}) \max_{\overline{h}\in\mathbb{H}} P(H_{in}=\overline{h}|O_{out}(L)=\overline{o}) \\
&\underset{\text{Bayes}}{=} \sum_{\overline{o}\in\mathbb{O}_D(L)} P(O_{out}(L)=\overline{o}) \max_{\overline{h}\in\mathbb{H}} \frac{P(O_{out}(L)=\overline{o}|H_{in}=\overline{h})P(H_{in}=\overline{h})}{P(O_{out}(L)=\overline{o})} \\
&= \sum_{\overline{o}\in\mathbb{O}_D(L)} \max_{\overline{h}\in\mathbb{H}} P(O_{out}(L)=\overline{o}|H_{in}=\overline{h})P(H_{in}=\overline{h})
\end{aligned}$$

Because H_{in} has uniform distribution, we have $\forall \overline{h} \in \mathbb{H}.\ P(H_{in} = \overline{h}) = \frac{1}{|\mathbb{H}|}$. Hence

$$\mathscr{V}(H_{in}|O_{out}(L)) = \frac{1}{|\mathbb{H}|} \sum_{\overline{o} \in \mathbb{O}_D(L)} \max_{\overline{h} \in \mathbb{H}} P(O_{out}(L) = \overline{o} | H_{in} = \overline{h}) \tag{2.6}$$

Because program p is deterministic, we can derive from (2.2) that

$$\max_{\overline{h} \in \mathbb{H}} P(O_{out}(L) = \overline{o} | H_{in} = \overline{h}) = 1$$

Therefore, from (2.6) we have

$$\mathscr{V}(H_{in}|O_{out}(L)) = \frac{1}{|\mathbb{H}|} \sum_{\overline{o} \in \mathbb{O}_D(L)} 1 = \frac{|\mathbb{O}_D(L)|}{|\mathbb{H}|} \tag{2.7}$$

Theorem 2.2 is proven by combining (2.3), (2.5) and (2.7). □

Example 2.4. *We continue with password checking program* `pwc` *presented in Example 2.3. Again we denote the initial value of* `l` *with* l_0*. Assume that the value of* `h` *ranges between lower bound lb and upper bound ub (*$lb \leq h \leq ub$*). If* $l_0 \notin [lb, ub]$ *then we have the set of output values of* `l` *is* $\{0\}$*. By Theorem 2.2 we have the min entropy-based leakage of program* `pwc` *is* $\mathtt{MEL}_{\mathtt{pwc}}(\mathtt{l}_0) = \mathtt{log}(1) = 0$*. If* $l_0 \in [lb, ub]$*, we have the set of output value of* `l` *is* $\{1, 0\}$ *and the min entropy-based leakage is* $\mathtt{MEL}_{\mathtt{pwc}}(\mathtt{l}_0) = \mathtt{log}(2) = 1$*. We can see that with password checking program, the min entropy-based leakage does not depend on the size of password space and is always* 1 *if the chosen guessing value is in the range of the password.*

2.2.4 Guessing Entropy

The formal definitions of guessing entropy and its conditional variant are given below:

Definition 2.8 (Guessing entropy and conditional variant)**.** *Given random variables* X, Y *with sample space* $\mathbb{X}$ *and* $\mathbb{Y}$*, respectively. The* guessing entropy *of* X *is defined as*

$$\mathscr{G}(X) = \sum_{1 \leq i \leq m} i \cdot P(X = x_i) \qquad (m = |\mathbb{X}|)$$

where $x_1, \ldots, x_m$ *satisfy* $\forall i, j.(i \leq j \rightarrow P(X = x_i) \geq P(X = x_j))$.

The conditional guessing entropy *of* X *given* Y *is defined as*

$$\mathscr{G}(X|Y) = \sum_{y \in \mathbb{Y}} P(Y = y) \mathscr{G}(X|Y = y)$$

where

$$\mathscr{G}(X|Y = y) = \sum_{1 \leq i \leq m} i \cdot P(X = x|Y = y)$$

and $x_1, \ldots, x_m$ *satisfy* $\forall i, j.(i \leq j \rightarrow P(X = x_i|Y = y) \geq P(X = x_j|Y = y))$.

Intuitively, the *guessing entropy* of a random variable X is the average number of questions of the kind: "Is the value of X equal to x?" that are needed to infer the value of X correctly [84].

The derivation of the computation of the guessing entropy-based leakage is similar to the previous ones and yields:

$$\mathtt{GEL}_{\mathtt{p}}(L) = \mathscr{G}(H_{in}) - \mathscr{G}(H_{in}|O_{out}(L)) \tag{2.8}$$

Example 2.5. *Consider the password checking program* `pwc` *from Example 2.3. We compute the guessing entropy-based leakage of* `pwc` *for the case that* `h` *has integer type whose value is in the range* $[lb, ub]$ *and has uniform distribution. By Definition 2.8, the initial guessing-uncertainty of* `h` *is:*

$$\mathscr{G}(h_{in}) = \frac{1}{ub-lb+1}\sum_{i=1}^{ub-lb+1} i = \frac{ub-lb+2}{2}$$

If the input value l_0 *of* `l` *is not in the range* $[lb, ub]$*, we can see easily that* $P(\mathtt{l}_{out}(l_0) = 1) = 0, P(\mathtt{l}_{out}(l_0) = 0) = 1$*. Similar to Example 2.3, the remaining uncertainty is:*

$$\mathscr{G}(h_{in}|\mathtt{l}_{out}(l_0)) = P(\mathtt{l}_{out}(l_0) = 0)\mathscr{G}(h_{in}|\mathtt{l}_{out}(l_0) = 0) = \frac{1}{ub-lb+1}\sum_{i=1}^{ub-lb+1} i = \frac{ub-lb+2}{2} = \mathscr{G}(h_{in})$$

The guessing entropy-based leakage is 0 *in this case. If* $l_0 \in [lb, ub]$*, we have:*

$$\mathscr{G}(h_{in}|\mathtt{l}_{out}(l_0)) = P(\mathtt{l}_{out}(l_0) = 0)\mathscr{G}(h_{in}|\mathtt{l}_{out}(l_0) = 0) + P(\mathtt{l}_{out}(l_0) = 1)\mathscr{G}(h_{in}|\mathtt{l}_{out}(l_0) = 1) = \frac{ub-lb}{ub-lb+1}\frac{1}{ub-lb}\sum_{i=1}^{ub-lb} i + \frac{1}{ub-lb+1}1 = \frac{(ub-lb+1)(ub-lb)+2}{2(ub-lb+1)}$$

The guessing entropy-based leakage in case low the input value is in the range $[lb, ub]$ *is* $\frac{ub-lb+2}{2} - \frac{(ub-lb+1)(ub-lb)+2}{2(ub-lb+1)} = 1 - \frac{1}{ub-lb+1}$*. If the domain of password is large then* $\mathtt{GEL}_{\mathtt{pwc}}(\mathtt{l}_0) = 1 - \frac{1}{ub-lb+1} \approx 1$*. This leakage value can be intuitively explained as follows: after observing the outcome of the password checking, one wrong value is removed from the space of possible passwords, hence the average number of guesses needed to discover the actual password is also reduced by one.*

2.2.5 Channel Capacity

The information leakage computed using Shannon, min, or guessing entropy is parameterized on the probability distribution of H_{in}. Hence, computing the leakage using those metrics requires that the probability distribution of high input is given explicitly. In case that the distribution of high input is unknown, *channel capacity* [44] can be used to measure the upper bound of the leakage.

Definition 2.9 (Channel capacity - Informal)**.** *Channel capacity is the maximum amount of information leakage (measured by using Shannon entropy or min entropy) over all possible probability distributions of high input. The channel capacities calculated using Shannon entropy and min entropy are named* Shannon-capacity *and* min-capacity *respectively.*

Because Shannon entropy-based leakage and min entropy-based leakage depend on low input, Shannon-capacity and min-capacity also depend on low input. Denote the sets of all Shannon-capacities and min-capacities of program p by $\mathbb{SCS}_p, \mathbb{MCS}_p$ respectively. Let $\mathtt{SC}_p : \mathbb{L} \mapsto \mathbb{SCS}_p$ be a function computing the Shannon-capacity of program p with a low input, i.e. $\mathtt{SC}_p(\bar{l})$ is the Shannon-capacity of program p given that $\bar{l}$ is the input value of L. Similarly, we define function $\mathtt{MC}_p : \mathbb{L} \mapsto \mathbb{MCS}_p$ for min-capacity.

Theorem 2.3. *If program p is deterministic then $\mathtt{SC}_p(L) = \mathtt{MC}_p(L) = \mathtt{log}|\mathbb{O}_D(L)|$.*

Proof. Let $\bar{l}_0 \in \mathbb{L}$ be an arbitrary concrete value of L. From Theorem 3.3 in [111] we derive $\mathtt{SC}_p(\bar{l}_0) = \mathtt{MC}_p(\bar{l}_0) = \mathtt{log}|\mathbb{O}_D(\bar{l}_0)|$. Thus we have $\forall \bar{l} \in \mathbb{L}.\ \mathtt{SC}_p(\bar{l}) = \mathtt{MC}_p(\bar{l}) = \mathtt{log}|\mathbb{O}_D(\bar{l})|$ and Theorem 2.3 is proven. □

Channel capacity is useful if we need a *worst case analysis* of quantitative information flow. For example, the attacker can exploit her knowledge of the distribution of H_{in} to derive an attack strategy that maximizes the amount of leakage, in this case channel capacity can give a precise upper bound of the amount of secret information that the attacker can learn, no matter what attack strategy is used. However, because of taking only the worst case into account, in some cases channel-capacity becomes over-pessimistic. We recall here the example adapted from [8] that has been mentioned already in Section 1.1.2 of this thesis.

Example 2.6. *Consider the password checking program* `pwc` *in Example 2.3 where the password* h *is a 32-bit integer number. Because the set of observable output values is* $\{0, 1\}$*, the Shannon-capacity of program* `pwc` *is* $\mathtt{log}(2) = 1$*, which means that after one guess, the attacker can reveal one bit from the password. This amount of leakage can be obtained with a distribution of input value of* h *that assigns probability 0.5 to both output value* 0 *and* 1*. Because of this, a naive analysis can conclude that* `pwc` *is insecure because the value of* h *can be found within 32 guesses. However, if* h *is uniform, program* `pwc` *leaks a very small amount of password's information (see Example 2.3). Hence, if the value of* h *is well-chosen (the distribution is almost uniform), program* `pwc` *can be considered as secure.*

2.3 Self-composition

Self-composition [17, 38, 39] is a technique to formalize information policies with respect to a program as the functional/behavioral properties of its self-composed version. The self-composed program is specified in the following definition.

Definition 2.10 (Self-composed program)**.** *Given a program p with the set of program variables Var, the self-composed program of p is the program $p; p(Var')$ where*

- *$p(Var')$ is constructed from p by renaming all program variables of p, i.e. from v to v', Var' is the set of all program variables of $p(Var')$*
- *$Var \cap Var' = \emptyset$*
- *p and $p(Var')$ do not share any memory*

Self-composition can be used to formalize information flow policies like noninterference and declassification by means of a classic program logic like Hoare logic, temporal logic or dynamic logic. We demonstrate this formalization explicitly using Hoare logic [61].

The Hoare triple $\{Pre\}\ p\ \{Post\}$ characterizes that whenever the program p, started in an initial state satisfying *Pre* (precondition) and terminates, *Post* (postcondition) must hold in the final state. Noninterference, as given in Definition 2.4 requires the comparison of two program runs. Let without loss of generality $\mathtt{l} \in L$, $\mathtt{h} \in H$ be the only variables of $p = p(\mathtt{l},\mathtt{h})$. Further, let $p(\mathtt{l'},\mathtt{h'})$ be the copied program constructed from p by renaming variable $\mathtt{l}$ to $\mathtt{l'}$and $\mathtt{h}$ to $\mathtt{h'}$ as in Definition 2.10. Then

$$\{\mathtt{l} \doteq \mathtt{l'}\}\mathtt{p(l,h);\ p(l',h')}\{\mathtt{l} \doteq \mathtt{l'}\}$$

is a direct formalization of noninterference.

The delimited information release policy $Decl = \{\{\mathtt{l}\},\{\mathtt{h}\},\{e\}\}$ (see Definition 2.5) can be formalized similarly as following:

$$\{\mathtt{l} \doteq \mathtt{l'} \wedge e \doteq e[\mathtt{l'},\mathtt{h'}/\mathtt{l},\mathtt{h}]\}\mathtt{p(l,h);\ p(l',h')}\{\mathtt{l} \doteq \mathtt{l'}\}$$

Example 2.7. *Consider program p extracting the last bit of high variable h to the low variable l:*

```
l = h % 2;
```

The formalization of noninterference policy $h \not\rightarrow l$ in Hoare logic is as follows:

```
{ l=l'} l = h % 2; l' = h' % 2 { l=l'}
```

The program is insecure which can be confirmed by a counter example for the above Hoare triple: $l = l' = 0; h = 0; h' = 1$.

Self-compostion enables existing general-purpose verification tools e.g. KeY [1] to be used to verify the information flow security of programs. A major drawback of this formalization is that it requires program p to be analyzed twice. Several refinements have been presented to avoid the repeated execution [16, 113]. The approach proposed in this thesis is based on symbolic execution. The fundamental idea is to execute the program symbolically only once and then to use the path conditions and symbolic states to carry out self-composition. Instead of copying program and renaming all the variables, it is sufficient to copy path conditions and symbolic values, replacing the symbolic input values with fresh copies. The technical details are given in Chapter 3.

2.4 Symbolic Execution

Symbolic execution (SE) [68] is a versatile technique used for various static program analyses. Symbolic execution of a program means to run it with symbolic input values instead of concrete ones. Such a run results in a tree of symbolic execution traces, which cover all possible concrete executions.

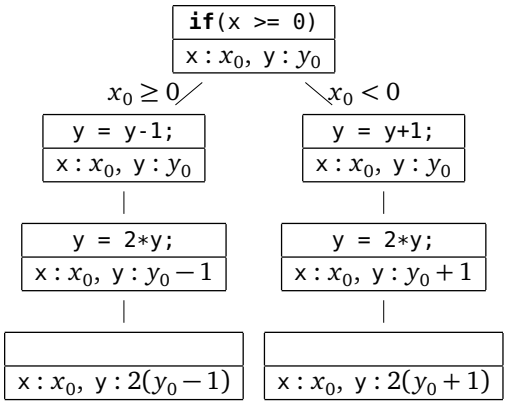
A symbolic execution tree represents all possible concrete execution paths; moreover, a single symbolic execution path may represent infinitely many concrete execution paths. Each node

of a symbolic execution path corresponds to a code location and contains the symbolic state at that point: a mapping from program variables to their symbolic value and a path condition. The *path condition* is obtained as the conjunction of all *branch conditions* up to the current point of execution and unambiguously defines the execution path to be taken. The initial state of any execution path through a node with path condition *pc* must necessarily satisfy *pc*. As long as the program does not contain loops or method invocations, a path condition is a quantifier-free formula in first-order logic.

Example 2.8. *Consider the simple program depicted in Figure 2.1a.*

```
1  if (x >= 0){
2    y=y-1;
3  }
4  else {
5    y=y+1;
6  }
7  y=2*y;
```

(a) Program source code



(b) Symbolic execution tree for symbolic input values $(x = x_0, y = y_0)$

Figure 2.1: A program and its symbolic execution tree

We start symbolic execution at the first statement in line 1 of the listing in Figure 2.1a. We begin with an initial state where x and y have symbol input values x_0 and y_0, respectively. The root node is a branching node whose outgoing edges are annotated by their branch conditions*. Here the symbolic execution splits into two branches: the left one for the case where the symbolic value x_0 is non-negative and the right one for a negative x_0. We continue symbolic execution on the first branch, the next statement (line 2) is the assignment of value $y_0 - 1$ to y. The final statement on the first branch is in line 7 that doubles the value of y. Thus, in case of a non-negative input value for x, the program terminates in a final state in which the final value of y is $2(y_0 - 1)$. Similarly, symbolically executing next statements on second branch whose branch condition is $x_0 < 0$ (lines 5 and 7) gives us a symbolic final state in which the final value of y is $2(y_0 + 1)$. In contrast to y, there is no statements assigning value to x, hence in both branches, the final value of x remains unchanged (i.e., x_0) at the final state. The obtained symbolic execution tree having two symbolic execution paths with path conditions and symbolic states of each node is shown in Figure 2.1b.*

Path conditions and symbolic values are always expressed relative to the initial symbolic values present in the initial symbolic state as illustrated in Example 2.8. In the following, instead of introducing a new constant symbol v_0 to refer to the initial value of a program variable v, we simply use the program variable v itself. This means program variables occurring in path conditions and symbolic values refer always to their initial value.

We make some notational conventions: We use SET_p to refer to the symbolic execution tree of program p and N_p to refer to the number of symbolic execution paths of SET_p. For each leaf node of a symbolic execution path i $(1 \leq i \leq N_p)$ the corresponding path condition is denoted

with pc_i and the symbolic value of variable $v \in Var$ in the final state of path i is denoted with the expression f_i^v. With an ordered set of program variables $V = \{v_1, \ldots, v_m\}$, we use $\overline{f_i^V}$ to refer to the ordered set of symbolic values of all variables of V in the final state of path i, i.e. $\overline{f_i^V} = \{f_i^{v_1}, \ldots, f_i^{v_m}\}$.

Symbolic execution suffers from the severe problem of path explosion. The number of symbolic execution paths grows exponentially in the number of program branches. If the program contains unbounded loops or recursive method calls, the symbolic execution tree even becomes infinite. We illustrate the case of unbounded loops by an example. Consider a Java class `IntArray` wrapping an integer array and its method `max` returning the maximum value of the array as given in Listing 2.1. Method `max` returns the maximum value of array `arr` in case `arr` is not **`null`** and has at least one element, otherwise it returns an exception. In case `arr` is not empty, the maximum value is assigned by the first element (line 10), then it is compared to all other elements of the array. If the current maximum value is smaller than an array element, this element's value becomes new maximum value. This is implemented by a for-loop construct from lines 11 - 13.

Listing 2.1: Method `max` containing unbounded loop

```
1  public class IntArray{
2    private int[] arr;
3    ...
4    public int max(){
5      if(arr==null)
6        throws new Exception("array is null");
7      else if(arr.length==0)
8        throws new Exception("array contains no elements");
9      else{
10       int max = arr[0];
11       for(int i=1; i<arr.length; i++)
12         if(max< arr[i])
13           max = arr[i];
14       return max;
15     }
16   }
17   ...
18 }
```

We perform symbolic execution on method `max` taking `a0` as symbolic value of array `arr` in the initial state. The if-construct checks first whether `a0` is null or an array of length 0. If that is not the case, the maximum is computed in lines 10 - 14. Before entering the loop, the value of `max` and `i` is assigned `a0[0]` and `1`, respectively. The loop body is executed iff the *loop guard* (`i < arr.length`) holds. Checking the loop guard causes two branches: if its value is false (`a0.length = 1`), then the **`return`** statement (line 14) is executed yielding the maximum value of the array. Otherwise (`a0.length > 1`), the current maximum value stored in variable `max` (`a0[0]`) is compared to the element of array `a0` indexed by `i`. If the current maximum value is smaller than considering array's element (`a0[0] < a0[1]`), then `max` is assigned a new maximum value (`a0[1]`), otherwise there is no need to change the value of `max`. Afterwards, the index `i` is increased by `1` and the loop guard is checked again. The symbolic execution only

terminates if the loop guard cannot be **true** anymore. Under the assumption that the length of an integer array is unbounded (actually in Java the maximum length of an array is bounded by `Integer.MAX_VALUE`), the loop guard is always possible to hold, no matter what the value of indexing number is. Thus symbolically executing method `max` produces an infinite symbolic execution tree that is sketched in Figure 2.2.

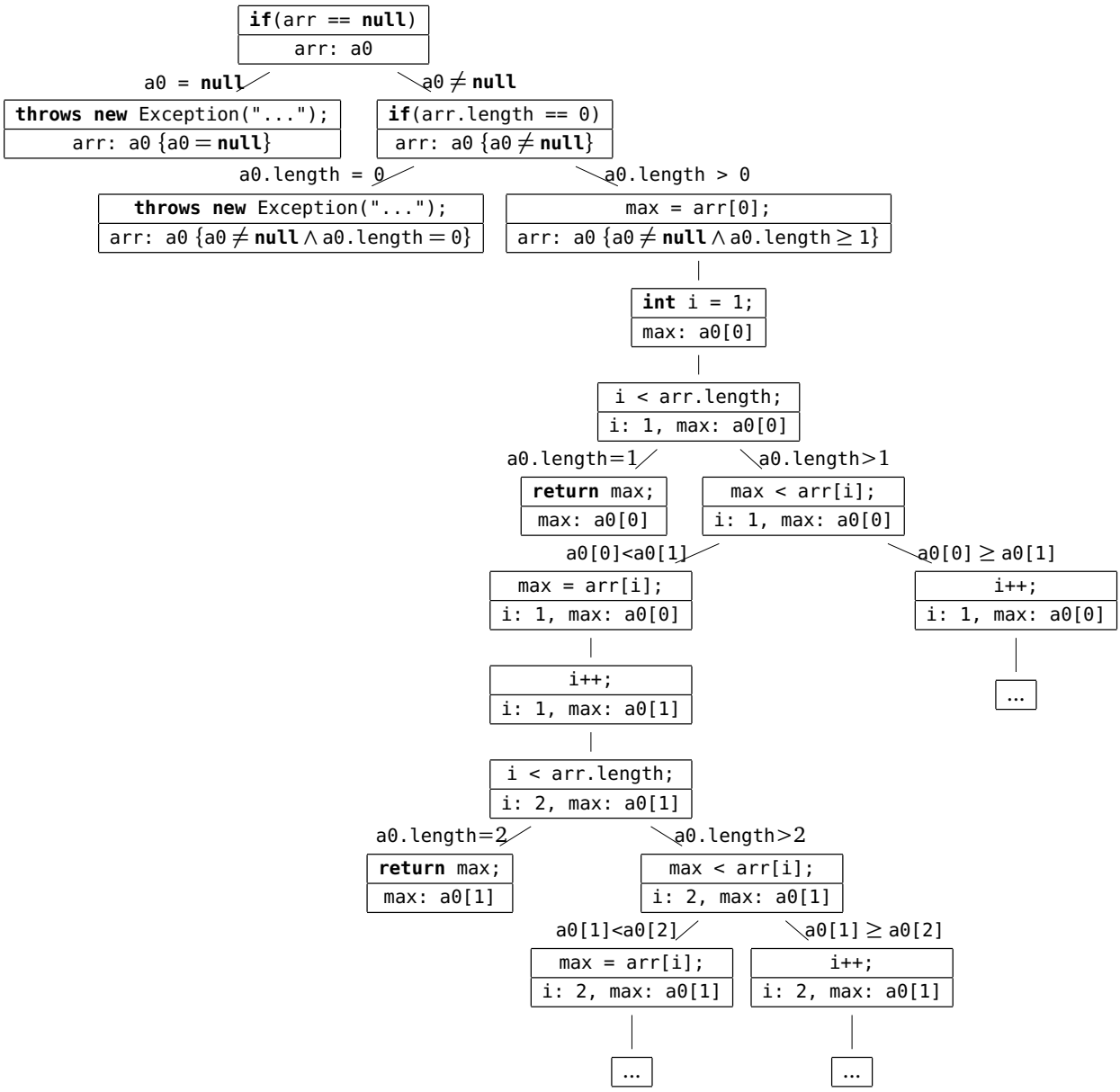


Figure 2.2: Infinite symbolic execution tree of method `max` in Listing 2.1

To deal with loops and recursions, a number of solutions have been introduced. One natural approach implemented within popular symbolic execution tools such as KLEE [29], Java Pathfinder [5] or Pex [114] is to unfold loops or inlining method calls up to a fixed depth. This solution brings an under-approximation of the infinite symbolic execution tree, hence some critical behaviors might not be taken into account. The paper [59] proposes an alternative approach achieving a finite representation of symbolic execution tree by making use of specifications. This is realized in the symbolic execution engine of the verification system KeY [1], that is used as implementation basis of the information flow analysis approach proposed in this thesis.

2.5 Program Specification with JML

Formal program specifications describe the intended program behavior. We introduce here briefly the program paradigm and specification language used throughout the thesis.

2.5.1 JML

The Java Modeling Language (JML) [78, 79] is a specification language for Java programs. It specifies Java modules following the *design-by-contract* paradigm [87]. JML specifications are added into Java source code as special comments beginning with an `@` symbol. A JML comment either has the form

```
//@ ...
```

or

```
/*@ ... @*/
```

in which the latter can be used for multi-line specifications.

The following sections show how to specify some method contracts and loop invariants in JML.

2.5.2 Method Contract

Contracts are central to the *design-by-contract* paradigm [87] and used to specify the input/output behavior of methods. The behavior of a program method is specified by means of preconditions and postconditions that constitute a method contract. We give here a formal definition for method contracts adapted from [59].

Definition 2.11 (Method contract)**.** *Given a method* `m`*, a contract* $C_{\texttt{m}}$ *for* `m` *is a triple* $(Pre_{\texttt{m}}, Post_{\texttt{m}}, Mod_{\texttt{m}})$ *with precondition* $Pre_{\texttt{m}}$*, postcondition* $Post_{\texttt{m}}$ *and modifies (or assignable) clause* $Mod_{\texttt{m}}$*.* $Pre_{\texttt{m}}$ *and* $Post_{\texttt{m}}$ *are formulas.* $Mod_{\texttt{m}}$ *is the set of all locations whose value* `m` *can possibly change.*

Method contract $C_{\texttt{m}}$ defines an agreement between method `m` and its callers: given that the precondition $Pre_{\texttt{m}}$ holds when calling `m`, `m` guarantees that the postcondition $Post_{\texttt{m}}$ is established when it returns. In JML, $Pre_{\texttt{m}}$ and $Post_{\texttt{m}}$ are boolean JML expressions (super set of Java boolean expressions). Preconditions are specified by expressions placed after the keyword **`requires`**, while the keyword **`ensures`** is used for postconditions. Keyword **`assignable`** is used to specify the set of all locations $Mod_{\texttt{m}}$ whose values can be changed by executing `m`. Some special keywords can be used to express the set $Mod_{\texttt{m}}$: **`\everything`** means that all locations can be changed by `m` and **`\nothing`** states that no location is changed.

Example 2.9. *Consider class* `IntArray` *and its method* `max` *taken from Listing 2.1. We specify a contract for* `max` *claiming that the maximum value of array* `arr` *is returned if* `arr` *is neither* **`null`** *nor empty. Listing 2.2 shows this contract in the form of a JML specification, from line 4 to line 8. The precondition at line 5 claims that method* `max` *should only be called if the array has been instantiated with at least one element (*`arr.length > 0`*). If this precondition holds, then* `max`

Listing 2.2: Contract of method `max` in Listing 2.1

```
1 public class IntArray{
2   private int[] arr;
3   ...
4   /*@ normal_behavior
5   @ requires this.arr!=null && this.arr.length>0
6   @ ensures \result ==(\max int i; 0<=i && i < this.arr.length; this.arr[i]);
7   @ assignable \nothing;
8   @*/
9   public int max(){
10    ...
11  }
12  ...
13 }
```

*guarantees that it returns the maximum value of the array (**`ensures`** clause at line 6). The JML expression **`\result`** is the identifier for the return value of the considered method. The maximum value of array `arr` is expressed by JML generalized quantifier **`\max`**. The **`assignable`** clause at line 7 claims that `max` does not change the heap (but may create new objects). Method `max` also guarantees that no exception is thrown (**`normal_behavior`**) if the array is non empty.*

2.5.3 Loop Specification

A *Loop invariant* [52, 61] is a logical assertion that has to be true before (and after) each iteration of a loop. Basically, it over-approximates the properties of all program states reached after each loop iteration. Loop invariants are a popular approach to deal with loops in program verification.

To help a program verification tool to deal with loops, not only the loop invariant is needed but also additional information about the effect of the loop on program locations is required. They are encompassed within a *loop specification*.

Definition 2.12 (Loop specification)**.** *A loop specification is a tuple* (I, mod) *in which* I *is the* loop invariant *formula and mod is a set of program locations that the loop can possibly modify.*

Definition 2.12 is a simplified version of the one given in [1] (Definition 3.23). It drops the witness of loop termination that does not contribute for constructing the symbolic execution tree (introduced in Section 2.6). The JML loop specification is added right before the corresponding loop. The loop invariant is specified after the keyword **`loop_invariant`**. Similar to method contracts, the set of modifiable locations is declared by the keyword **`assignable`**. The termination witness is placed after the keyword **`decreases`**.

Example 2.10. *Consider the loop structure finding the maximum value of a non-empty integer array in Listing 2.1, its specification is given in Listing 2.3 from line 1 to line 5. The loop invariant, specified at lines 1 and 2, preserves that the current value of `max` is the maximum value over the fragment of array `arr`, from the beginning to position `i-1`, and that the value of index variable `i` cannot exceed the array's length. The **`assignable`** clause at line 3 claims that only `max` and `i` can*

Listing 2.3: Loop specification for `for`-loop of method `max` in Listing 2.1

```
1 /*@ loop_invariant i>=1 && i <= arr.length &&
2           max == (\max int j; 0 <= j && j < i; arr[j]);
3 @ assignable max, i;
4 @ decreases arr.length-i;
5 @*/
6 for(int i=1; i<arr.length; i++)
7   ...
```

be changed by loop execution. Finally, the difference between the array's length and index `i` is used as the termination witness (line 4). Because the value of `i` is increased after each iteration, this is strictly decreased by each iteration and always ≥ 0, that is a guarantee of loop termination.

2.6 The KeY System

This section introduces the deductive verification system KeY, that is used as the back-end symbolic execution engine for the implementation of the approaches proposed in this thesis. We briefly present the main architecture of KeY, then we explain how KeY generates the symbolic execution tree for a Java program.

2.6.1 Architecture

The KeY tool is a deductive verification system targeting Java programs at source code level. In recent years, KeY has evolved from a formal verification tool [20] into a framework that benefits the developers during software development process in many aspects [1]. The main architecture of the KeY framework is depicted in Figure 2.3.

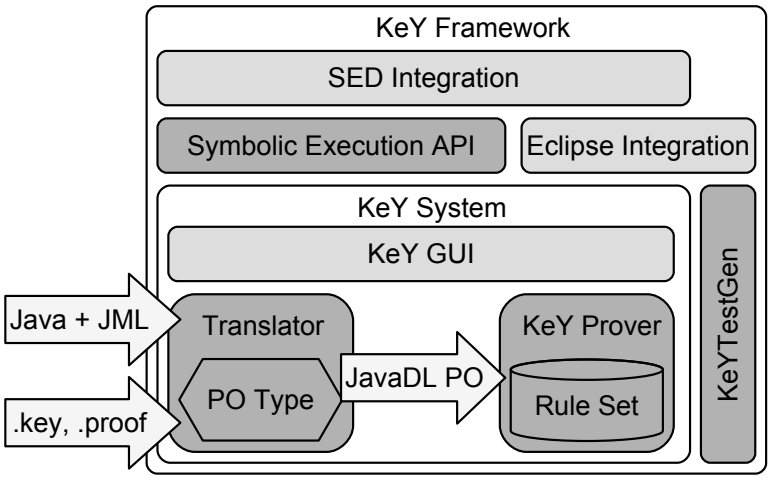


Figure 2.3: Architecture of the KeY tool set (simplified version of [1, chapter 1])

The central task of KeY is to verify the functional correctness of Java programs. There are two ways to provide a verification problem for KeY: defining the problem (along with the Java source code) in a `.key` file or specifying the problem by JML specifications (usually in the form

of a method contract) annotated in a `.java` file. The latter is more natural and allows users to define multiple verification problems in one `.java` file. The annotated Java source code is then translated into corresponding *proof obligations* expressed in Java Dynamic Logic (Java DL) [1, 99] formulas.

To prove the validity of a Java DL formula, the *KeY Prover* - heart of the KeY framework - is used. The KeY prover uses *sequent calculus* to construct the proof: the formula to be proven is expressed as a *sequent* of the form $\phi_1, \ldots, \phi_n \Longrightarrow \psi_1, \ldots, \psi_m$ where $\phi_1, \ldots, \phi_n, \psi_1, \ldots, \psi_m$ are Java DL formulas. A sequent is valid if and only if the formula $\bigwedge_{i=1}^{n} \phi_i \rightarrow \bigvee_{j=1}^{m} \psi_j$ is valid. KeY proves sequents by constructing a *proof tree* where: the root is the original proof obligation; the child nodes are obtained from the parent node by applying rules on its sequent. A leaf of a proof tree is called "closed goal" if its sequent is evaluated to true by a *closing rule*, otherwise it is an "open goal". A formula is proven if all leaves of its proof tree are closed. Rules can be applied either automatically with help of a proof strategy, or manually by the user using the *KeY GUI*.

Java source code to be verified is embedded in Java DL formulas. While constructing the proof for a Java DL formula, its embedded source code can be symbolically executed by applying a number of symbolic execution rules. The symbolic execution tree extracted from the proof tree by *Symbolic Execution API* can be used for different purposes such as test case generation or program debugging. KeY as a symbolic execution engine plays an important role as the backend of information flow analysis approaches in this thesis. That engine will be explained in the next section.

Beside the theorem proving engine and the symbolic execution engine, KeY also offers test case generation with component *KeyTestGen*. The KeY framework also supplies the *Symbolic Execution Debugger (SED)* as an interactive symbolic execution and debugging tool in Eclipse. SED is made available to be used in Eclipse by *SED Integration*. The developer can also use KeY directly within Eclipse with the help of *Eclipse Integration*.

2.6.2 KeY as Symbolic Execution Engine

KeY can be used as a stand-alone symbolic execution engine for Java programs. It supplies a symbolic execution API for general usage. Figure 2.4 illustrates the process generating the symbolic execution tree of a Java method by KeY.

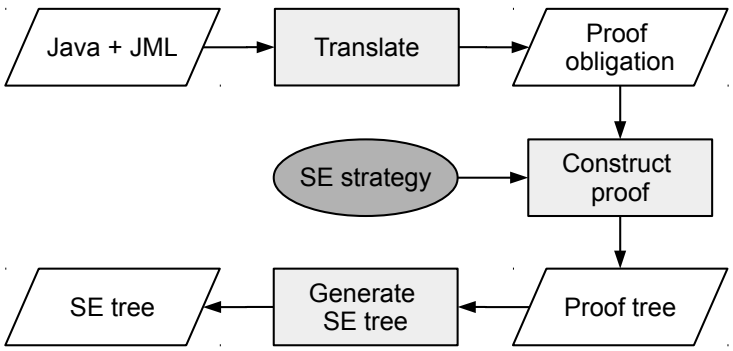


Figure 2.4: Generating symbolic execution tree by KeY

First of all, a Java method annotated by JML specification is translated into proof obligations. Then, the proof tree will be constructed. Instead of using the standard proof strategy, KeY offers a special proof strategy, namely *Symbolic Execution Strategy*, that helps to build a proof

optimized for generating the symbolic execution tree. During the proof construction, infeasible paths are also pruned. Whenever the proof tree has been built, the symbolic execution tree is generated. Path conditions and symbolic state are extracted from the logical content contained in the sequent of the proof nodes.

To deal with unbounded loops and recursive method calls, the symbolic execution engine of KeY uses *loop specifications* and *method contracts* to achieve finite representations of infinite symbolic execution trees. The basic idea is that all possible loop executions can be represented by loop specification and the guard of loop, while method contracts can be used to describe the effect of method call from the caller's side. When the symbolic execution reaches a loop or method call, instead of unfolding the loop or inlining the called method (that might result enormous, or even infinite symbolic execution branches), only two branches are generated, one of them allows symbolic execution to go further to the statement after the loop/method call. Branch conditions and the modification of symbolic states are contributed by loop specifications and method contracts. Details can be found in [59].

Example 2.11. *Consider method* `max()` *whose infinite symbolic execution tree is sketched in Figure 2.2, we show how a finite presentation of that tree is achieved by KeY using the loop specification supplied in Listing 2.3. Let* `a0` *be the symbolic value of array* `arr` *in the initial state. When reaching the loop (line 6, Listing 2.1), the loop specification, including loop invariant and set of modifiable locations is taken into account and is represented as a node of the symbolic execution tree. Then two execution branches are generated. Both branch conditions contain the loop invariant (*$i \geq 1 \wedge i \leq$ `arr.length` $\wedge$ `max` $= max_{j=0}^{i-1}($`arr[j]`$)$*) as a conjunct, and they vary on the value of the loop guard (*$i <$ `arr.length` *and* $i \geq$ `arr.length`*). The values of* `max` *and* `i` *are fresh symbolic values:* `max0, i0` *in the first branch and* `max1, i1` *in the second to eliminate the knowledge of their initial values as they might be changed by the previous loop iteration. The first branch leads to a subtree representing for all loop executions that are performed if the loop guard holds (*$i <$ `arr.length`*), in which all statements of loop structure are symbolically executed (using* `max0, i0` *as symbolic values of* `max, i` *respectively). The second branch leads to another subtree representing the execution of program after the loop is terminated. In this case the loop guard does not hold anymore (*$i \geq$ `arr.length`*) and the symbolic execution is continued with the first statement after the loop structure, here the* **`return`** *statement, using* `max1, i1` *as symbolic values of* `max` *and* `i`*. The constraints for fresh variables are defined in corresponding branch conditions. The finite symbolic execution for* `max()` *using loop specification is represented in Figure 2.5.*

The usage of specifications allows KeY to obtain a finite symbolic execution tree. The drawback is that correct and strong specifications are required to be provided by the users. Details of how to use program specification for leak detection are explained in Section 3.4.

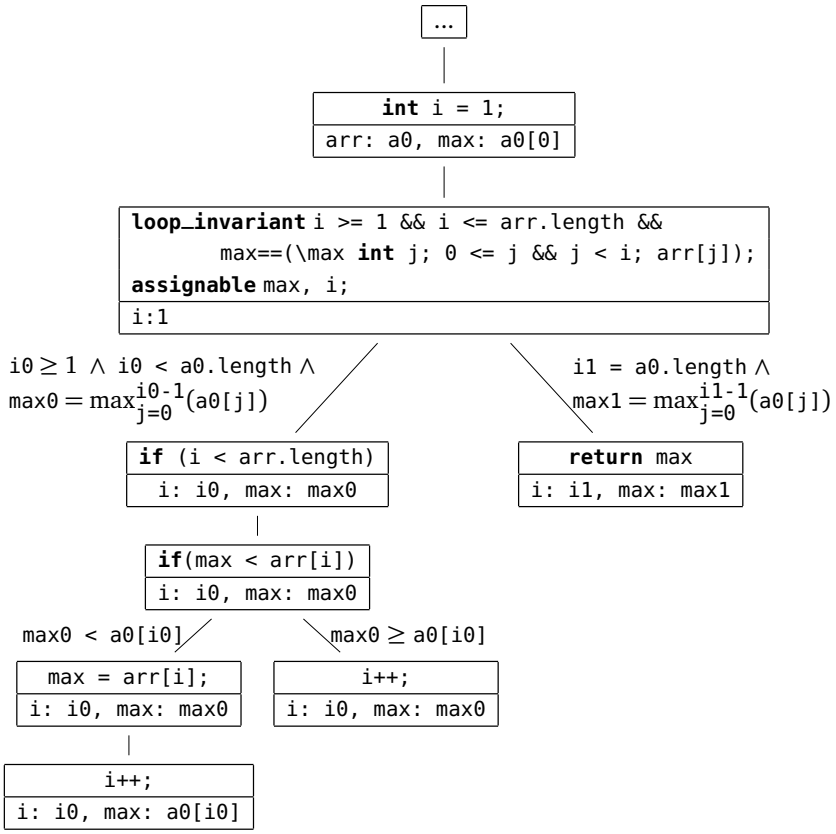


Figure 2.5: Finite representation of an infinite symbolic execution tree for method `max` using the loop specification in Listing 2.3

3 Detection and Demonstration of Information Flow Leaks

This chapter introduces a novel approach to detect and demonstrate information flow leaks with respect to a given information flow policy, i.e. noninterference and declassification. First, the formalization in logic for insecurity is given in Section 3.1. This formalization is extended to incorporate generalized noninterference policies that support *information erasure* (Section 3.2) and *targeted conditional delimited release* as declassification policy (Section 3.3). Section 3.4 demonstrates the usage of program specifications, i.e. method contract and loop specification in leak detection. Finally, Section 3.5 explains how the demonstrations of information leaks can be generated. Parts of this chapter are based on previous publications [47, 48] of the author of this thesis.

3.1 Logic Characterization of Insecurity

This section describes the formalization in logic for various information flow policies. Given a complete symbolic execution tree for a program and an information flow policy, we construct formulas that are unsatisfiable when the program is secure and satisfiable if the policy can be violated by some inputs. This approach permits to use an SMT solver or model finder to search for satisfying models from which one can then read off concrete input states for two program runs that demonstrate a violation of the given policy.

First, we show how to construct a formula that characterizes noninterference (Definition 2.4) from a complete symbolic execution tree for program p with paths $i \in \{1, \ldots, N_p\}$. We recall here some notational conventions defined in Chapter 2, Section 2.4: pc_i denotes the path condition that uniquely determines path i, f_i^v represents the symbolic value of variable v at the final state corresponding to symbolic path i, and $\overline{f_i^V}$ denotes $\{f_i^{v_1}, \ldots, f_i^{v_m}\}$ where $V = \{v_1, \ldots, v_m\}$. The noninterference policy involved is $NI = H \not\rightsquigarrow L$ with H, L are set of high and low variables, respectively and $L \dot\cup H = Var$ (Var is the set of all program variables of p).

To represent two independent program runs, we create a copy of all program variables $Var' = \{v' \mid v \in Var\}$ with the notational convention that v' always refers to the copy of v. Now we obtain the sets L' and H' as copies of L and H, i.e., $L' = \{l' \mid l \in L\}$ (analogously H'). To refer to the initial (symbolic) value of a program variable v, instead of introducing a new constant symbol v_0, we simply use the program variable v itself. Intuitively, the first run is performed using Var, while the second one uses the copy Var'. Both runs are independent as they do not share any common memory.

Then the *NI-insecurity formula*

$$\bigvee_{1 \le i \le j \le N_p} \left(L \doteq L' \wedge pc_i \wedge pc_j[Var'/Var] \wedge \overline{f_i^L} \neq \overline{f_j^L}[Var'/Var] \right) \tag{3.1}$$

is satisfied if and only if there is a model (i.e., concrete state) σ assigning values to the program variables Var, Var' such that

- the input values of the low variables L coincide with the values of their copies in L',
- there are two paths i, j ($i = j$ possible) with consistent path conditions (i.e., both paths can actually be taken), but
- for which the final value of at least one low variable differs.

In other words, the model σ assigns concrete values to Var and Var' such that p produces different low level output for two runs from initial states with identical low input.

Example 3.1. *The insecurity formula* (3.1) *for the example program from Figure 2.1 on page 22 and the NI policy* $\{x\} \not\rightarrow \{y\}$ *is:*

$$\begin{aligned} & y_0 \doteq y_0' \wedge x_0 \geq 0 \wedge x_0' \geq 0 \wedge 2(y_0 - 1) \neq 2(y_0' - 1) \\ \vee\ & y_0 \doteq y_0' \wedge x_0 \geq 0 \wedge x_0' < 0 \wedge 2(y_0 - 1) \neq 2(y_0' + 1) \\ \vee\ & y_0 \doteq y_0' \wedge x_0 < 0 \wedge x_0' < 0 \wedge 2(y_0 + 1) \neq 2(y_0' + 1) \end{aligned}$$

It is easy to see that the first and third disjunct are unsatisfiable, but the second disjunct is satisfiable, e.g., for the model $x_0 \mapsto 0$, $x_0' \mapsto -1$, $y_0 \mapsto 1$, $y_0' \mapsto 1$.

The NI-insecurity formula (3.1) can be rewritten into the equivalent formula:

$$\bigvee_{l \in L} \bigvee_{1 \leq i \leq j \leq N_p} \overbrace{\left(L \doteq L' \wedge pc_i \wedge pc_j[Var'/Var] \ \wedge f_i^l \neq f_j^l[Var'/Var]\right)}^{Leak^{NI}(H,L,l,i,j)} \tag{3.2}$$

This formulation will be easier to incorporate declassification into. The intuition behind the formula $Leak^{NI}(H, L, l, i, j)$ is that it allows us to ascribe leaks to a specific target, i.e., it is satisfiable, if some information is leaked from the program variables in H to variable l.

The copy of the low level variables is actually not needed (as we require their equality for all models), so formulas (3.1), (3.2) can be made more succinct by replacing L' with L and omitting the first conjunct, which states $L \doteq L'$. In the future we will tacitly perform this simplification. Then the first disjunct in Example 3.1 becomes $x_0 \geq 0 \wedge x_0' \geq 0 \wedge 2(y_0 - 1) \neq 2(y_0 - 1)$.

3.2 Generalized Noninterference Policy

Sometimes it is not sufficient to simply ensure that no information is leaked, but one wants also to guarantee that secret data is not kept longer than needed, because of legal reasons or to make data dumps (initiated by an attacker) less useful. *Information erasure* is for a desired property for cryptographic devices (secret keys must be erased after usage), online transactions (credit card information must be erased after the transaction is completed), e-voting (all data connecting voter and ballot must be erased after the result has been published), etc. Information erasure policies have been presented in [42, 64].

Listing 3.1: Ticket vending machine

```
1 void buy() {
2   charge(ticketCost, ccNumber);
3   log();
4   ccNumber = 0;
5 }
```

Example 3.2. *Consider a simple ticket vending machine model (adapted from [42]) as shown in Listing 3.1. Assume that before executing the program the buyer's credit card number was read from a terminal and stored in variable* `ccNumber`*. To complete the purchase, method* `buy` *is called and the card account debited. After logging the purchase, the credit card number is erased from memory by setting* `ccNumber` *to* `0`*. This ensures that even a powerful attacker who can dump the memory of the vending machine to read the location of variable* `ccNumber` *cannot learn anything about the credit card number of the buyer.*

In more technical terms, we want to ensure that after execution of `buy` *the value of* `ccNumber` *is permitted to flow to any low location. This policy is not expressible within the standard noninterference framework. A naive and ad hoc extension would be to classify* `ccNumber` *as high and to add a constraint* $f^{\texttt{ccNumber}}(L,H) \neq \texttt{0}$ *to the insecurity formula. But this does not work, for instance, if we want to erase the secret with a random number.*

To provide support for some of information erasure policies [42, 64] we generalize our notion of interference (cf. Definition 2.4)

Definition 3.1 (Generalized Noninterference)**.** *Given a program p over variables Var. A* generalized noninterference policy *(GNI) is an ordered pair, written as $H \not\rightsquigarrow_{GNI} L$, where $L, H \subseteq Var$ are sets of low and high variables. Program p has secure information flow with respect to GNI if and only if for all concrete executions $X, Y \in Exc_p$ it holds that if $\sigma^X \simeq_{Var \setminus H} \sigma^Y$ then $\sigma^X_{out} \simeq_L \sigma^Y_{out}$.*

The definition omits the requirement that L and H form a partitioning of Var, i.e. a variable v is allowed to be a member of both variable sets. In addition, it potentially strengthens the condition on the output values of the low variables in the final states. The output values of the low variables in the final states must now be identical for any two initial states that coincide on the variable set of $Var \setminus H$. The set $Var \setminus H$ might not contain all variables in L (e.g., variable v from before would not be in H) and hence allows for more pairs of initial states to be considered. Generalized noninterference reduces to standard noninterference when $L \dot\cup H = Var$.

Example 3.3 (Example 3.2 Continued)**.** *To ensure that the credit card number is erased we specify the GNI policy*

$$H = \{\texttt{ccNumber}\} \not\rightsquigarrow_{GNI} \{\texttt{ccNumber}, \texttt{ticketCost}\} = L \ .$$

To keep the analysis simple, assume that methods `charge(int, int)` *and* `log()` *do nothing. First we assume that line 4 that peforms the erasure was forgotten. The following test case demonstrates a violation of the policy: Given initial states σ_1, σ_2 with:*

	`ccNumber`	`ticketCost`
σ_1	*1234*	*50*
σ_2	*5678*	*50*

Both initial states coincide on the variable set $Var \setminus H = \{\texttt{ticketCost}\}$. *To adhere to the GNI policy, the final states* $\sigma^1_{out}, \sigma^2_{out}$ *must coincide on all low variables, i.e. on the variables* `ticketCost` and `ccNumber`. *But as none of the values is changed by the runs, they still differ in the value of* `ccNumber`, *hence, the GNI property is not valid. Now assume line 4 is present; then, in both executions the final value of* `ccNumber` *is* 0 *and the GNI property holds.*

The logic formalisation of the corresponding insecurity formula is almost identical to (3.2):

$$\bigvee_{l \in L} \bigvee_{1 \le i \le j \le N_p} \overbrace{\left(Var \setminus H \doteq Var' \setminus H' \ \wedge \ pc_i \ \wedge \ pc_j[Var'/Var] \ \wedge \ f_i^l \not\doteq f_j^l[Var'/Var]\right)}^{Leak^{GNI}(H,L,l,i,j)} \tag{3.3}$$

3.3 Targeted Conditional Delimited Release

We further extend the insecurity formula for generalized noninterference (3.3) to *delimited information release* (DIR) [103]. In contrast to the standard version of DIR, our policy describes not only *what* information can be released by escape hatches, but also allows to express under which condition and to *whom* (target) the information might be leaked.

Definition 3.2 (Targeted Conditional Delimited Release)**.** *Given a program* p *over variables* Var *and a* $GNI = H \not\rightsquigarrow_{GNI} L$. *A* Targeted Conditional Delimited Release *(TCD) policy* $(\mathscr{D}, GNI)$ *is a set of* specification triples *where each* $(e, C, T) \in \mathscr{D}$ *consists of*

- *an escape hatch expression (i.e. first-order term)* e *over* Var,
- *a declassification condition formula* C *over* Var, *and*
- $T \subseteq L$, *a set of program variables to which the specified escape hatch is allowed to be leaked.*

A program satisfies a given TCD policy $(\mathscr{D}, GNI)$ *if it satisfies the GNI policy, except for the cases covered by a triple* $(e, C, T) \in \mathscr{D}$. *Here, the program is free to release the information captured by the escape hatch expression* e *to a location in* T, *provided that condition* C *is satisfied in the initial state of the execution.*

Given a TCD policy $(\mathscr{D}, GNI)$ and a program p. We give the insecurity formula for the case that $\mathscr{D} = \{(e, C, T)\}$ consists of a single TCD specification triple:

$$\bigvee_{l \in L} \bigvee_{1 \le i \le j \le N_p} \overbrace{\left(Leak^{GNI}(H, L, l, i, j) \wedge \left(l \dot{\in} T \wedge C \wedge C[Var'/Var] \rightarrow e \doteq e[Var'/Var]\right)\right)}^{Leak^{(\mathscr{D},GNI)}(H,L,l,i,j)} \tag{3.4}$$

The formula coincides with the noninterference insecurity formula for locations $l \notin T$ that are not among the allowed release targets. Otherwise, the new second conjunct adds

$$C \wedge C[Var'/Var] \rightarrow e \doteq e[Var'/Var] \tag{3.5}$$

as an additional restriction to the initial states for both runs: if both initial states satisfy the declassification condition C then they must also coincide on the value of the escape hatch expression. The justification is that if there are two runs such that their initial states coincide on the low level input and on the escape hatches and if the final value for an allowed target differs, then more information than just the escape hatch must have been released.

Example 3.4 (Example 2.2 Continued)**.** *Consider the program from Example 2.2:*

```
if ( h > 0 ) { l = 2; }
```

Let $(\{\texttt{h>0}, true, \{\texttt{l}\}\}, \{h\} \not\rightsquigarrow_{GNI} \{\texttt{l}\})$ *be a declassification TCD policy. This is the insecurity formula resulting from symbolic execution starting in* (l_0, h_0)*:*

$$(l_0 \doteq l_0' \wedge h_0 > 0 \wedge h_0' > 0 \wedge 2 \not\doteq 2) \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h_0' > 0)) \vee$$
$$(l_0 \doteq l_0' \wedge h_0 \leq 0 \wedge h_0' > 0 \wedge l_0 \not\doteq 2) \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h_0' > 0)) \vee$$
$$(l_0 \doteq l_0' \wedge h_0 \leq 0 \wedge h_0' \leq 0 \wedge l_0 \not\doteq l_0') \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h_0' > 0))$$

The first and third disjunct are trivially invalid. The second disjunct is invalid, because the second conjunct implies $h_0 > 0$ *if and only if* $h_0' > 0$ *which contradicts the path condition in the first conjunct. Consequently, the insecurity formula is unsatisfiable, i.e. the program is secure for the specified policy. Consider a slightly altered program:*

```
if ( h >= 0 ) { l = 2; }
```

We analyze it with the same policy and initial state as above. Now in the resulting insecurity formula

$$(l_0 \doteq l_0' \wedge h_0 \geq 0 \wedge h_0' \geq 0 \wedge 2 \not\doteq 2) \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h_0' > 0)) \vee$$
$$(l_0 \doteq l_0' \wedge h_0 \leq 0 \wedge h_0' \geq 0 \wedge l_0 \not\doteq 2) \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h_0' > 0)) \vee$$
$$(l_0 \doteq l_0' \wedge h_0 \leq 0 \wedge h_0' \leq 0 \wedge l_0 \not\doteq l_0') \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h_0' > 0))$$

the second disjunct is *satisfiable, for instance, when the initial value of* `h` *is* -1 *and the initial value of* `h'` *is* 0*. Consequently, the program does not adhere to the specified policy.*

3.4 Leak Detection Using Program Specification

Code with unbounded loops or recursive method calls gives rise to infinite symbolic execution trees. Another difficulty is posed by calls to library methods for which no source code is available. And in general symbolic execution trees tend to become infeasibly large when the implementations of called methods are simply inlined. To overcome these problems we annotate programs with specifications in the form of loop invariants and method contracts. This allows to approximate a loop by an invariant and a method call by a contract. In the paper [59] it is shown how to use such specifications during symbolic execution and we can adapt that solution to our setting. To keep the presentation readable, in this section we focus on the standard noninterference analysis case. The extension to declassification and erasure is straightforward.

3.4.1 Loop Specification

To compute the path conditions and the final values of symbolic execution paths we need to be able to execute unbounded loops without unwinding them infinitely often. In program verification this is achieved by providing a *loop specification*. According to Definition 2.12, a loop specification $LS = (I, mod)$ consists of a *loop invariant* formula I and a set of program variables *mod* that contains at least those program variables the loop can possibly modify.

We need to integrate loop specifications into the NI-insecurity formula (3.2). Let `b` be the guard of a loop and $LS = (I, mod)$ its specification. The basic idea in [59] is that the loop specification describes the state after exiting the loop. This means, we can treat the loop as a black-box and continue execution after the loop in a state for which the variables *mod* that might have been modified by the loop are set to an unknown value. Unknown values are represented by fresh symbolic values V_{mod}. The only knowledge about these values is provided by the loop invariant and by the fact the loop guard `b` must be *false* after exiting the loop.

Our insecurity formulas express a constraint over the initial state. For instance, the final value f_i^l of variable l is given in terms of the initial symbolic values of the program variables. The same holds for the path conditions. We make this implicit weakest precondition computation here explicit for the loop guard and the invariant, i.e., I^{wp} is the weakest precondition of I computed in the state directly after the loop (similar for the loop guard).

For the sake of simplicity, we only show how to adapt $Leak^{NI}(H, L, l, i, j)$ for the case that both paths i, j contain the same loop:

$$Leak^{NI}(H,L,l,i,j) \;=\; L \doteq L' \wedge pc_i \wedge pc_j[V_S'/V_S] \wedge \\ I^{wp} \wedge \neg b^{wp} \wedge I^{wp}[V_S'/V_S] \wedge \neg b^{wp}[V_S'/V_S] \wedge f_i^l \not\doteq f_j^l[V_S'/V_S] \quad (3.6)$$

Here $V_S = Var \cup V_{mod}$ and b^{wp} is the symbolic value of the guard after the loop terminates expressed in terms of the initial values of V_S. If one or both paths i, j do not contain the loop or a different loop, then the conjuncts corresponding to the invariants and loop guards are omitted or added accordingly.

Example 3.5. *We illustrate formula* (3.6). *Consider the loop below with low variable* `l` *and high variable* `h`. *We want to establish whether this code is secure with respect to the policy* $\{h\} \not\rightarrow \{l\}$. *The loop specification is* $(l \geq 0, \{l\})$. *This loop invariant could easily have been inferred with automated methods.*

```
l = h * h;
while (l > 0) { l = l - 1; }
l = l + h;
```

Let l_{mod}, l'_{mod} *be the fresh values representing the value of* `l` *directly after the loop. Computing the weakest precondition of the invariant gives us* $l_{mod} \geq 0$ *and for the guard* $l_{mod} > 0$ *for the first run (analogous for the second run). The resulting formula is (please note that there is only one path and no path condition):*

$$l_0 \doteq l'_0 \wedge l_{mod} \geq 0 \wedge \neg(l_{mod} > 0) \wedge l'_{mod} \geq 0 \wedge \neg(l'_{mod} > 0) \wedge l_{mod} + h_0 \not\doteq l'_{mod} + h'_0$$

The formula is satisfiable, for example, with $l_0 = l'_0 = 10$, $l_{mod} = l'_{mod} = 0$, $h_0 = 1$ *and* $h'_0 = 2$. *Indeed, the program is insecure. Removing the final statement would make it secure. In this case the final conjunct in the insecurity formula would change to* $l_{mod} \not\doteq l'_{mod}$ *which renders it unsatisfiable.*

3.4.2 Method Contracts

We recall here the Definition 2.11 for method contract. Let `m` be a method name. A contract $C_{\mathtt{m}}$ for `m` is a triple $(Pre_{\mathtt{m}}, Post_{\mathtt{m}}, Mod_{\mathtt{m}})$ with precondition $Pre_{\mathtt{m}}$, postcondition $Post_{\mathtt{m}}$ and modifies

(or assignable) clause $Mod_{\mathtt{m}}$. The latter is the set of all program variables whose value m can possible change (similar as *mod* in loop specifications).

A method satisfies its contract, if it ensures that when invoked in a state for which the precondition is satisfied, then in the final state the postcondition holds and at most the value of program variables in the assignable clause has been modified.

Analysing Noninterference Relative to a Precondition.

Given a method m with contract $C_{\mathtt{m}}$. We want to analyze whether m respects a noninterference policy $H \not\rightsquigarrow L$ under the condition that m is only invoked in states satisfying its precondition $Pre_{\mathtt{m}}$. Adapting the noninterference formula (3.2) is straightforward and merely requires to add a restriction to the initial states that they must satisfy the method's precondition:

$$Leak^{NI}(H, L, l, i, j) \wedge Pre_{\mathtt{m}} \wedge Pre_{\mathtt{m}}[Var'/Var]$$

Example 3.6 (Example 3.1 Continued)**.** *The insecurity formula checking whether the program from Figure 2.1 on page 22 respects the NI policy $\{x\} \not\rightsquigarrow \{y\}$ under the condition that the initial state satisfies $x > 0$ is:*

$$\begin{aligned} & y_0 \doteq y_0' \wedge x_0 \geq 0 \wedge x_0' \geq 0 \wedge 2(y_0 - 1) \not\doteq 2(y_0' - 1) \wedge x_0 > 0 \wedge x_0' > 0 \\ \vee\ & y_0 \doteq y_0' \wedge x_0 \geq 0 \wedge x_0' < 0 \wedge 2(y_0 - 1) \not\doteq 2(y_0' + 1) \wedge x_0 > 0 \wedge x_0' > 0 \\ \vee\ & y_0 \doteq y_0' \wedge x_0 < 0 \wedge x_0' < 0 \wedge 2(y_0 + 1) \not\doteq 2(y_0' + 1) \wedge x_0 > 0 \wedge x_0' > 0 \end{aligned}$$

The precondition $x > 0$ and path condition $x < 0$ are exclusive, hence the second and third disjunct are unsatisfiable. The first disjunct that corresponds to path condition $x \geq 0$ is also unsatisfiable because it is obvious that $y_0 \doteq y_0' \wedge 2(y_0 - 1) \not\doteq 2(y_0' - 1)$ cannot be satisfied. Hence the insecurity formula is unsatisfiable which means that if the precondition $x > 0$ holds then the program satisfies the given NI policy.

Analyzing Programs with Method Contracts for Noninterference.

This problem is solved in [59] by using method contracts in a similar way loop specifications have been used. Instead of a loop invariant, the pre- and postconditions become part of the path conditions. The modifies clause again introduces fresh values to represent the symbolic value of program variables that might have been changed as a side effect of method invocation.

Let m be the method analyzed for secure information flow and assume it invokes method n. Let the contract of n be $(Pre_{\mathtt{n}}, Post_{\mathtt{n}}, Mod_{\mathtt{n}})$. In the case that each of the paths i, j contains exactly one method call for n we obtain:

$$\begin{aligned} Leak^{NI}(H, L, l, i, j) \ = \ & L \doteq L' \wedge pc_i \wedge pc_j[V_S'/V_S] \wedge \\ & Pre_{\mathtt{n}}^{wp} \wedge Post_{\mathtt{n}}^{wp} \wedge Pre_{\mathtt{n}}^{wp}[V_S'/V_S] \wedge Post_{\mathtt{n}}^{wp}[V_S'/V_S] \wedge f_i^l \not\doteq f_j^l[V_S'/V_S] \end{aligned} \quad (3.7)$$

where $V_S = Var \cup V_{Mod_{\mathtt{n}}}$ (analagous for the copies) and $Pre_{\mathtt{n}}^{wp}$, $Post_{\mathtt{n}}^{wp}$ are the weakest preconditions of $Pre_{\mathtt{n}}$, $Post_{\mathtt{n}}$ computed directly before and after method invocation, respectively. If method n returns a value, a fresh variable representing the return value of n is added to V_S. The return value can be referenced in $Post_{\mathtt{n}}$. The general case (no method call, different method calls, or more than one method call) is handled similarly as in the loop case.

Example 3.7. *We illustrate formula* (3.7) *with method* `run` *shown in Listing 3.2. We want to establish whether* `run` *is secure with respect to the policy* $\{h\} \not\rightsquigarrow \{l\}$. *We expect that it is insecure: the returned value of method* `calc` *depends on its parameter. In line 3,* `h` *is passed as argument, hence, the returned value depends on high input, but it is assigned to low variable* `l`.

We construct the insecurity formula: method `run` *invokes the recursive method* `calc`. *To analyse the information flow resulting from this invocation, we have to use a method contract for* `calc`, *because the recursion does not have a fixed bound.*[1] *Let* `calc`*'s contract be given as follows:*

$$\begin{aligned} Pre_{\texttt{calc}}\colon &\quad true \\ Post_{\texttt{calc}}\colon &\quad (x \leq 0 \rightarrow \mathit{result} \doteq 0) \wedge (x > 0 \rightarrow 2 * \mathit{result} \doteq x*(x+1)) \\ Mod_{\texttt{calc}}\colon &\quad \emptyset \end{aligned}$$

where `result` *refers to the return value and* $Mod_{\texttt{calc}}$ *is empty as* `calc` *does not change the state.*

Let `r` *be a program variable representing the return value of* `calc`. *To apply the contract for the invocation at line 3, we need to instantiate the above contract as follows:*

$$\begin{aligned} Pre^{wp}_{\texttt{calc}}\colon &\quad true \\ Post^{wp}_{\texttt{calc}}\colon &\quad (h+2 \leq 0 \rightarrow r \doteq 0) \wedge (h+2 > 0 \rightarrow 2 * r \doteq (h+2)*((h+2)+1)) \end{aligned}$$

The insecurity formula is then

$$\begin{aligned} &l_0 \doteq l'_0 \wedge (h_0 + 2 \leq 0 \rightarrow r_0 \doteq 0) \wedge (h_0 + 2 > 0 \rightarrow 2r_0 \doteq (h_0 + 2)(h_0 + 3)) \wedge \\ &(h'_0 + 2 \leq 0 \rightarrow r'_0 \doteq 0) \wedge (h'_0 + 2 > 0 \rightarrow 2r'_0 \doteq (h'_0 + 2)(h'_0 + 3)) \wedge r_0 \not\doteq r'_0 \end{aligned}$$

The formula is satisfiable, for example, with $l_0 = l'_0 = 10$, $h_0 = 1$, $h'_0 = 2$, $r_0 = 6$ *and* $r'_0 = 10$ *which means that method* `run` *is insecure.*

Listing 3.2: Recursive method call

```
1 public void run() {
2   h = h + 2;
3   l = calc(h);
4 }
5
6 private int calc (int x) {
7   if (x <= 0)
8     return 0;
9   else
10    return x + calc(x-1);
11 }
```

3.4.3 General Observations and Remarks

Using loop specifications or method contracts has one major drawback, namely, that not all models of a formula give rise to an actual information leak, or even worse, the insecurity formula of a secure program might become satisfiable. This case does not effect the soundness,

[1] Strictly speaking, the Java type **int** and stack size are bounded, but the bound is far too large to be feasible.

but triggers *false warnings*. The reason is that the specifications might be too weak and allow behaviours that are not possible in the actual program. These false warnings can be filtered out by actually running the generated *leak demonstrators* (Section 3.5). If the leak demonstrator fails to demonstrate the information leak, we know that our model was a spurious one. We can even start a feedback loop with a conflict clause which rules out the previously found model.

On the other hand, if loop or method specifications are not only too weak, but wrong in the sense that they exclude possible behaviour, then leaks might go undetected. As we are concerned with bug detection and not verification, this is not too serious as we do not claim to find all bugs. Nevertheless, incompleteness can be avoided by verifying the specifications using a program verification tool such as KeY [1].

3.5 Leak Demonstration

3.5.1 Leak Demonstration Program

The main idea of leak demonstration program is that an information flow leak can be exposed by comparing two program runs. To be more specific, a leak demonstration program basically runs the target program twice in two disjoint memories, using two initial states that coincide in their low part and differ in their high part. The low outputs of two runs can then be compared to check whether they differ. We call this program *leak demonstrator*. The definition of leak demonstrator is given in Definition 3.3.

Definition 3.3 (Leak demonstrator)**.** *Given a program p and an information flow policy X where X is either GNI (Definition 3.1) or $(\mathscr{D}, GNI)$ (Definition 3.2). Let $l \in L$ be a low variable w.r.t. the information flow policy X and l satisfies that its output value, obtained by running program p, depends only on the initial input values of all program variables of p w.r.t. this run. Then a leak demonstrator of p w.r.t. the information flow policy X and the low variable l , denoted by $LD_p^{X,l}$, is a program executing p twice using two initial program states σ^1, σ^2 that satisfy:*

1. $\exists v \in H.\ \sigma^1(v) \neq \sigma^2(v)$ *(H is the set of high variables)*
2. $\sigma^1 \simeq_{Var \setminus H} \sigma^2$ *(Var is the set of all program variables of p)*
3. $\neg[\![C]\!]_{\sigma^1} \vee \neg[\![C]\!]_{\sigma^2} \vee [\![e]\!]_{\sigma^1} = [\![e]\!]_{\sigma^1}$ *(C, e are defined in Definition 3.2)*

where the third condition is applied only for the case $X = (\mathscr{D}, GNI)$. $LD_p^{X,l}$ must guarantee that at the final states of two executions, denoted by σ^1_{out} and σ^2_{out} , the output values of l $(\sigma^1_{out}(l), \sigma^2_{out}(l))$ do not depend on the order of executions.

The first and the second condition of two initial states ensure that if the output values of the low variables observed in two program executions differ, such differentiation is caused by the different values of high variables. The third one guarantees that if two executions expose a leak, such a leak is not allowed by the conditional delimited release policy. The condition that the output value of low variable produced by each execution does not change when the order of two executions changes ensures that the low output of an execution only depends on its initial state. It requires that all sharing memories between two executions i.e. global or static variables, must be reconfigured before executing the program and cannot be changed during this execution by another one.

The leak demonstrator $LD_p^{X,l}$ can be considered as a test case that checks whether information flow policy X is violated by program p. The test oracle is simply the assertion $\sigma^1_{out}(l) = \sigma^2_{out}(l)$. If that assertion succeeds, the policy is respected and $LD_p^{X,l}$ is a *false warning*. Otherwise, $LD_p^{X,l}$ is correct, confirming the existence of information flow leak w.r.t. policy X.

Example 3.8. *Consider the program in Example 2.2:*

```
if ( h > 0 ) { l = 2; }
```

and the noninterference policy $\{h\} \not\rightsquigarrow \{l\}$. *The leak demonstrator using two initial states* σ^1, σ^2 *in which* $\sigma^1(h) = 1, \sigma^2(h) = -1, \sigma^1(l) = \sigma^2(l) = 1$ *exposes an information flow from* `h` *to* `l` *because at the final states of the two executions, the output values of* `y` *are different:* $\sigma^1_{out}(l) = 2, \sigma^1_{out}(l) = 1$. *However, if two initial states* σ^1, σ^2 *are defined by* $\sigma^1(h) = 1, \sigma^2(h) = 2, \sigma^1(l) = \sigma^2(l) = 1$ *then this leak demonstrator is a false warning because the output values of* `l` *at two final states are identical* ($\sigma^1_{out}(l) = \sigma^1_{out}(l) = 2$). □

It is worth noting that leak demonstrator $LD_p^{X,l}$ is dedicated only to the information flow policy X and cannot be used for other policies. For example, the first leak demonstrator in Example 3.8 is not applicable to the TCD policy ($\{$`h>0`, *true*, $\{$`l`$\}\}$, $\{h\} \not\rightsquigarrow_{GNI} \{l\}$), because the condition $[\![\texttt{h} > 0]\!]_{\sigma^1} = [\![\texttt{h} > 0]\!]_{\sigma^2}$ is not satisfied. In addition, if program p is changed, leak demonstrator $LD_p^{X,l}$ can change from a correct one to a false warning and vice versa. For instance, if we add the assignment "`l = 0;`" to the end of the program in Example 3.8, the correct leak demonstrator whose initial states are $\sigma^1(\texttt{h}) = 1, \sigma^2(\texttt{h}) = -1, \sigma^1(\texttt{l}) = \sigma^2(\texttt{l}) = 1$ becomes a false warning, because $\sigma^1_{out}(\texttt{l}) = \sigma^1_{out}(\texttt{l}) = 0$. This coincides with the fact that the program is secure after being modified.

3.5.2 Leak Demonstrator Generation

The leak detection approach, proposed in previous sections of this chapter, enables one to generate leak demonstrators. Algorithm 3.1 presents how to detect possible leaks of program p w.r.t. information flow policy X and generating leak demonstrators.

In a nutshell, for each pair of symbolic path i, j and each low variable l, we compose the insecurity formula $Leak^X(H, L, l, i, j)$ as defined in (3.3) and (3.4). Then, the formula is passed to an SMT solver to check its satisfiability. If it is satisfiable, one leak has been found and a concrete model M satisfying the insecurity formula is returned by the SMT solver. A leak demonstrator generated with M is used to set up for two initial states σ^1, σ^2 of two runs of p occurred in it. By iterating all possible ordered pairs of symbolic execution paths i, j ($i \leq j$) and low variable l, we can know the exact sinks of the leak (if it exists) and can detect specific *risky paths* (path might contribute to a leak).

Two initial states σ^1, σ^2 are set up using M as follows: for each program variable v, $\sigma^1(v)$ is assigned the value of variable v in M while $\sigma^2(v)$ is assigned v'. From the definition of insecurity formula $Leak^X(H, L, l, i, j)$ in (3.3) and (3.4), it can be seen that two initial states σ^1, σ^2 fulfill all requirements in Definition 3.3.

Data: SET_p: symbolic execution tree of program p, X: information flow policy (X is either GNI or $(\mathscr{D}, GNI)$)
Result: A set of leak demonstrators for p w.r.t. policy X

```
1  begin
2      LD ← ∅; // LD is the set of all leak demonstration programs
3      for i = 1; i < N_p; i++ do
4          // N_p is the number of symbolic paths of SET_p programs
5          for j = i; j ≤ N_p; j++ do
6              foreach l ∈ L do
7                  Build insecurity formula Leak^X(H, L, l, i, j);
8                  if Leak^X(H, L, l, i, j) is satisfiable then
9                      M ← concrete model satisfying Leak^X(H, L, l, i, j) ;
10                     LD_p^{X,l} ← GenLeakDemonstrator(M) ;
11                     LD ← LD ∪ LD_p^{X,l};
12                 end
13             end
14         end
15     end
16     return LD;
17 end
```

Algorithm 3.1: Leak detection and demonstrator generation

4 Automatic Secret Inference

Concerning information flow security, there are two common questions raised if one wants to judge whether a program is safe to be used: "are there any information flow leaks w.r.t a given policy?" and "if such leaks exist, how severe are they?". Chapter 3 presented a novel approach tackling the first question. The second one motivates many quantitative information flow analyses that aim to measure information leakage by means of a security metric. However, most of the approaches quantifying information flow leaks are static and do not show explicitly how the secret data of a program can be revealed throughout the information leaks in this program in practice.

This chapter proposes a novel approach combining static and dynamic analysis addressing the second question. To judge the severity of information flow leaks, we propose a model where the attacker can learn the secret by running program and observing the secret. This model is given in Section 4.1. Logical representation of the attacker's knowledge of a secret is explained in Section 4.2. Section 4.3 explains in detail the adaptive algorithm inferring high input. Finally, Section 4.4 describes how "good" low inputs can be generated to optimize the secret inference strategy.

4.1 Attacker Model and Overview

Given a deterministic program p, the set of variables of p is denoted by Var. Var is partitioned into two sets H, L that are sets of high and low variables respectively. We assume that the attacker knows the source code and can run the program multiple times. In our setting, the attacker can choose a value for L before each run and observe the value of a subset O of L after a program execution terminates. The attacker's aim is to infer the value of H via concrete runs, so called *experiments*. We assume that high variables are not modified by or in between concrete runs. We use $\overline{h}_s \in \mathbb{H}$ to refer to a secret, i.e. concrete (to us unknown) values of H.

Figure 4.1 shows an overview of our approach. First, the source code is analyzed statically by symbolic execution to identify execution paths, called risky paths, that might cause information leakage (directly or indirectly) from H to O. Based on this analysis a number of experiments is performed to infer the secret. An experiment is a program run with concrete input together with the outcome. To perform an experiment the algorithm selects suitable low input based on knowledge about risky execution paths and knowledge accumulated in previous runs. The algorithm terminates when one of the following conditions holds: (i) all secrets have been inferred unambiguously; (ii) it can be determined that no new knowledge can be inferred; (iii) a specified limit of concrete program runs is reached.

We give here the formal definition of risky path that is a symbolic execution path which might contribute to an information leakage.

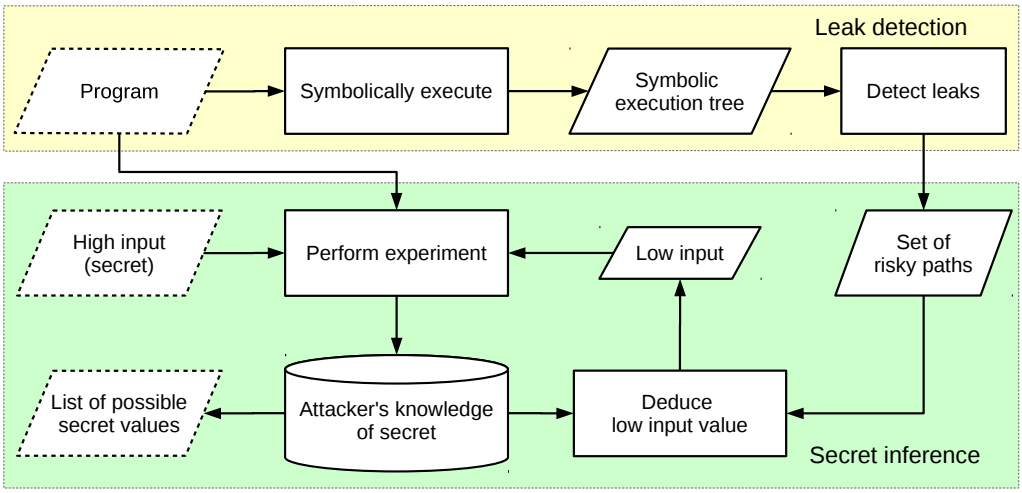


Figure 4.1: Structure of the algorithm to infer secrets

Definition 4.1 (Risky path)**.** *Let p be a program and N_p be the number of all symbolic paths of p. A symbolic path i ($1 \leq i \leq N_p$) is called a* risky path *for a generalized noninterference policy $H \not\rightarrow_{GNI} O$ iff*

$$\exists k.(1 \leq k \leq N_p \wedge Leak(H, O, i, k) \text{ is satisfiable})$$

where $Leak(H, O, i, k) = \bigvee_{v \in O} Leak^{GNI}(H, O, v, i, k)$ ($Leak^{GNI}(H, O, v, i, k)$ is defined in Section 3.2 (formula (3.3)). The set of all risky paths of p is denoted with $Risk_p$.

Listing 4.1: Running example program `rPC` for secret inference

```
1 if (l < 100) {
2   if (l == h)
3     l = 3;
4   else if (l < h)
5     l = 0;
6   else
7     l = -3;
8 } else
9   l = 2;
```

Example 4.1. *The SE tree of the program `rPC` in Listing 4.1 has four paths with path conditions $pc_1 = l < 100 \wedge l \doteq h$, $pc_2 = l < 100 \wedge l < h$, $pc_3 = l < 100 \wedge l > h$ and $pc_4 = l \geq 100$. We analyze this program with respect to noninterference policy $\{h\} \not\rightarrow \{l\}$ using the approach proposed in Chapter 3. It can be seen that all of four insecurity formulas $Leak(\{h\}, \{l\}, 4, 1)$, $Leak(\{h\}, \{l\}, 4, 2)$, $Leak(\{h\}, \{l\}, 4, 3)$, $Leak(\{h\}, \{l\}, 4, 4)$ are unsatisfiable, while the two formulas $Leak(\{h\}, \{l\}, 1, 2)$, $Leak(\{h\}, \{l\}, 2, 3)$ are satisfiable. According to Definition 4.1, we have the set of risky paths is $\{1, 2, 3\}$ while 4 is not a risky path of the given program.*

4.2 Knowledge Representation of High Input

We fix a program p, a noninterference policy $H \not\rightarrow L$, and a set $O \subseteq L$ of observable low variables. The concrete value sets $\mathbb{L}, \mathbb{H}, \mathbb{O}_D(\cdot)$ are as before.

To gain knowledge about a secret, a series of experiments is performed.

Definition 4.2 (Experiment)**.** *A pair* $\langle \overline{l}, \overline{o} \rangle$ *with* $\overline{l} \in \mathbb{L}$, $\overline{o} \in \mathbb{O}_D(\overline{l})$ *is called an* experiment *for* p*. By convention, we denote with* $\overline{h}_s$ *the high input value that was used in the run.*

Let $E = \{\langle \overline{l}_j, \overline{o}_j \rangle \mid 1 \leq j \leq m\}$ be a set of experiments for a program p. Symbolic execution of p yields a precise logical description of all reachable final states, see Section 3.1. Recall that N_p is the number of all feasible symbolic execution paths. For each symbolic execution path i, we obtain its path condition pc_i and the final symbolic values f_i^v of any program variable v. Let O' be an ordered set of fresh program variables such that for any $v \in O$ there is a corresponding $v' \in O'$ and the cardinality of O and O' is equal, i.e. $|O| = |O'|$. The formula

$$Info(L, H, O') = \bigvee_{1 \leq i \leq N_p} InfoPath_i(L, H, O') \tag{4.1}$$

where $InfoPath_i(L, H, O') = pc_i \wedge O' \doteq \overline{f_i^O}$ "records" the information about final values contained in a symbolic execution path. It is true whenever the variables in H, L, O' are assigned values $\overline{h}$, $\overline{l}$, $\overline{o}$ such that executing p in an initial state $\langle \overline{l}, \overline{o} \rangle$ terminates in a final state where the variables in O have values $\overline{o}$. For a concrete experiment $\langle \overline{l}, \overline{o} \rangle$ formula (4.1) is instantiated to

$$Info_{\langle \overline{l}, \overline{o} \rangle}(H) = Info(\overline{l}, H, \overline{o}) = Info(L, H, O')[\overline{l}, \overline{o} \,/\, L, O'] \tag{4.2}$$

This formula must be true at the time of running the experiment, because (i) the taken execution path must be contained in one of the symbolic execution paths, and (ii) the observed output values must be equal to those obtained by evaluating the symbolic values with the concrete initial values of the low and high variables.

We write $Info_{\langle \overline{l}, \overline{o} \rangle}(H)$ to emphasize that the truth value of the formula only depends on the assignment of concrete values to the program variables in H. The formula $Info_{\langle \overline{l}, \overline{o} \rangle}(H)$ constrains the possible high values and can be seen as the information about $\overline{h}_s$ that can be learned from experiment $\langle \overline{l}, \overline{o} \rangle$. The *knowledge* about $\overline{h}_s$ gained from all experiments in a set E is then

$$K^E(H) = K^{\emptyset}(H) \wedge \bigwedge_{\langle \overline{l}, \overline{o} \rangle \in E} Info_{\langle \overline{l}, \overline{o} \rangle}(H) \tag{4.3}$$

where $K^{\emptyset}(H)$ is the *initial knowledge* about $\overline{h}_s$ that is known before performing any experiment, for example, domain restrictions. If nothing is known about $\overline{h}_s$, then the initial knowledge $K^{\emptyset}(H)$ is simply *true*. The set of all models of $K^E(H)$ contains by construction also the actual secret $\overline{h}_s$ (a simple inductive argument with base case that $K^{\emptyset}(H)$ is satisfied by $\overline{h}_s$).

Example 4.2. *Consider again program* `rPC` *from Listing 4.1 with* `l` *as low variable and* `h` *as high variable. Initially, the knowledge about the value of* `h` *is its domain* $-2^{31} \leq h < 2^{31}$*. Assume that the value of* `h` *is* 10*. If we run this program with low input* $l = 5$*, because* $5 < 10$*, the (concrete) output value of* `l` *is* 0 *and* $\langle 5, 0 \rangle$ *is an experiment of the given program. The information about* `h` *gained by experiment* $\langle 5, 0 \rangle$ *is*

$$\begin{aligned} Info_{\langle 5,0 \rangle}(h) \;= {} & (5 < 100 \wedge 5 \doteq h \wedge 0 \doteq 3) \vee \\ & (5 < 100 \wedge 5 < h \wedge 0 \doteq 0) \vee \\ & (5 < 100 \wedge 5 > h \wedge 0 \doteq -3) \vee \\ & (5 \geq 100 \wedge 0 \doteq 2) \end{aligned}$$

$Info_{\langle 5,0\rangle}(h)$ can be simplified to equivalent formula $5 < h$. The knowledge of h after performing experiment $\langle 5,0\rangle$ is

$$K^{\{\langle 5,0\rangle\}}(h) = -2^{31} \leq h < 2^{31} \wedge 5 < h \equiv 5 < h < 2^{31}$$

We want to design a set of experiments that reduces, as much as possible, the number of possible concrete values for H that satisfy (4.3). The smaller this number is, the more we succeeded to narrow down the possible values for the secret. In particular, if only one possible value remains, we know the secret.

We recall here some notational conventions. The set of all values of a variable set X that satisfy a formula $\varphi(X)$ is denoted by $Sat(\varphi)$. Hence, $Sat(K^E(H))$ is the set of all values of H that satisfy $K^E(H)$. By convention we use $|S|$ to denote the cardinality of a set S.

Example 4.3 (Example 4.2 cont'd)**.** *We continue performing experiments on the program in Listing 4.1. Consider two experiment sets $X = \{\langle 5,0\rangle, \langle 3,0\rangle, \langle 8,0\rangle\}$, $Y = \{\langle 5,0\rangle, \langle 17,-1\rangle\}$. The knowledge about the secret input value of h that can be gained from X and Y is $K^X(\{h\}) = 8 < h < 2^{31}$ and $K^Y(\{h\}) = 5 < h < 17$, respectively. Even though $|X| > |Y|$, it is obvious that $|Sat(K^Y(\{h\}))| \ll |Sat(K^X(\{h\}))|$, hence the knowledge about the secret value of h obtained from Y is significantly higher than the one obtained from X.*

We want to accumulate maximal knowledge about a secret with as few experiments as possible. In particular, we do not want to perform experiments that do not create any knowledge gain. Avoiding redundant experiments is essential to achieve performance.

Definition 4.3 (Redundant experiment)**.** *An experiment $\langle \overline{l}, \overline{o}\rangle$ is called* redundant *for $K^E(H)$ if the following holds:*

$$\forall \overline{h}.(K^E(\overline{h}) \rightarrow Info_{\langle \overline{l},\overline{o}\rangle}(\overline{h}))$$

A redundant experiment $\langle \overline{l}, \overline{o}\rangle$ gains no new information about a secret $\overline{h}_s$ for knowledge $K^E(H)$, because $K^E(\overline{h}) \wedge Info_{\langle \overline{l},\overline{o}\rangle}(\overline{h}) \equiv K^E(\overline{h})$.

Example 4.4 (Example 4.2 cont'd)**.** *Experiment $\langle 5,0\rangle$ results in the knowledge $K^{\{\langle 5,0\rangle\}}(h) = 5 < h < 2^{31}$ (recall that the secret value of h is 10). Experiment $\langle 3,0\rangle$ is a redundant experiment for knowledge $K^{\{\langle 5,0\rangle\}}(h)$ because $Info_{\langle 3,0\rangle}(h) = 3 < h$ brings no new information about h, due to the fact that formula $5 < h < 2^{31} \rightarrow 3 < h$ is true for all possible values of h.* □

4.3 Algorithm for Inferring High Input

Algorithm 4.1 implements the core of our approach. The result is a logic formula that represents the accumulated knowledge about the high variables the algorithm was able to infer. The result can be used as input to an SMT solver or another model finder to compute concrete models representing possible secrets.

Algorithm 4.1 receives as input the program p, the symbolic execution result for p, i.e. p's SE tree together with all path conditions and symbolic values in the final symbolic execution state, the attacker's initial knowledge, etc. In particular, the formula $Info_{\langle \overline{l},\overline{o}\rangle}(H)$ can be computed.

Data: p: program to be attacked (with the high input already set); noninterference policy $H \not\rightarrow L$; $O \subseteq L$: observable low variables; $K^{\emptyset}(H)$: initial knowledge about H; $maxE$: maximum number of experiments
Result: $K^{E}(H)$: the accumulated knowledge about H obtained by executing the experiments E

```
1  begin
2      E ← ∅;
3      K ← K^∅(H);
4      while |E| < maxE do
5          (l̄, leakage) ← findLowInput(E, K);
6          if leakage > 0 then
7              execute p with low input l̄;
8              ō ← values of O when p terminates;
9              E ← E ∪ ⟨l̄, ō⟩;
10             K ← K ∧ Info_⟨l̄,ō⟩(H);
11             if |Sat_H(K)| = 1 then
12                 exit while;
13             end
14         else
15             exit while;
16         end
17     end
18     return K
19 end
```

Algorithm 4.1: Secret inference

First, the set of already performed experiments E is initialized with the empty set and the accumulated knowledge K is initialized with the initial knowledge of the attacker. Thereafter, the main loop of the algorithm is entered. At the beginning of each iteration K contains the accumulated knowledge of all experiments executed up to now, i.e. $K = K^{E}(H)$. At the beginning of each loop iteration the low input $\overline{l}$ for a new experiment is determined by method $findLowInput(E,K)$ based on the set of experiments E and the knowledge K accumulated so far. That method returns also a measure of the leakage expected to be observed by executing p with the provided low input. The method returns 0 as leakage only if all low input values would result in redundant experiments. In its most rudimentary implementation the method returns simply random values and a positive number for the leakage. We discuss more refined implementations in Section 4.4.

In case the expected leakage is positive (i.e. something new might be learned), program p is executed with the computed low input $\overline{l}$ and the set of experiments is extended by the pair $\langle \overline{l}, \overline{o} \rangle$ where $\overline{o}$ are the values of the observable variables when p terminates. In the next step we update the accumulated knowledge by adding the conjunct $Info_{\langle \overline{l}, \overline{o} \rangle}(H)$. Afterwards, we check whether the accumulated knowledge uniquely determines the values of the high variables. If this is the case we know the exact secret and return. Otherwise, we continue another loop

iteration until the maximal number of experiments *maxE* is reached. In case that the expected leakage is zero, no useful low input can be found any longer and the algorithm terminates.

4.4 Finding Optimal Low Inputs

This section discusses method *findLowInput* in detail and aims to provide more efficient implementations than the trivial one sketched above. We begin with the case that observable outputs do not depend on low inputs (Subsection 4.4.1). Some exploitations of risky paths and reachable paths to avoid potential redundant experiments as well as to simplify the knowledge computation are given in Subsection 4.4.2. Subsection 4.4.3 introduces an algorithm generating low input that maximizing the leakage measured by some general security metrics.

4.4.1 Low-independent Program

We consider programs whose observable output values only depend on high input values. In this case, two program runs that agree on high inputs and differ on low inputs at their initial states will produce the same observable output value at their final states. We call this program *low-independent*. A formal definition for low-independent program is given below:

Definition 4.4 (Low-independent program)**.** *Given program p, noninterference policy $H \not\rightarrow L$ and observable variables $O \subseteq L$, p is called* low-independent *w.r.t. policy $H \not\rightarrow L$ and observable variables O if and only if it enjoys generalized noninterference policy $L \not\rightarrow_{GNI} O$.*

In the context that the noninterference policy and the set of observable variables are defined unambiguously, we simply call the program low-independent. We will show that if p is low-independent, finding optimal low input for an experiment is unnecessary because all experiments bring the same knowledge of high input (the secret high input value involved in all experiments is $\overline{h}_s$).

Theorem 4.1. *If program p is low-independent, then $K^E(H) \equiv K^{E'}(H)$ for any two non-empty sets of experiments E and E'.*

Proof. Let $\overline{l}_1, \overline{l}_2 \in \mathbb{L}$ be two arbitrary input values of L. Because p is low-independent, by Definitions 4.4 and 2.4 we have that the observable output values obtained by running p with two initial states $(L = \overline{l}_1, H = \overline{h}_s)$ and $(L = \overline{l}_2, H = \overline{h}_s)$ are identical. We denote this value by $\overline{o}$ $(\overline{o} = \{o_v | v \in O\})$. Theorem 4.1 can be trivially concluded if we can prove that $Info_{\langle \overline{l}_1, \overline{o} \rangle}(H) \equiv Info_{\langle \overline{l}_2, \overline{o} \rangle}(H)$. Assume that $Info_{\langle \overline{l}_1, \overline{o} \rangle}(H) \not\equiv Info_{\langle \overline{l}_2, \overline{o} \rangle}(H)$, there exists $\overline{h} \in \mathbb{H}$ such that $Info(\overline{l}_1, \overline{h}, \overline{o}) = true$ and $Info(\overline{l}_2, \overline{h}, \overline{o}) = false$, or vice versa. Without loss of generality, we assume that $\overline{h} \in Sat(Info(\overline{l}_1, H, \overline{o}))$ and $\overline{h} \notin Sat(Info(\overline{l}_2, H, \overline{o}))$. Because $Info(\overline{l}_1, \overline{h}, \overline{o}) = true$, from (4.1) we have $(\bigvee_{1 \le i \le N_p} pc_i[\overline{l}_1, \overline{h}/L, H] \wedge \overline{o} \doteq \overline{f_i^O}[\overline{l}_1, \overline{h}/L, H]) = true$, hence there exists $j \in [1, N_p]$ such that

$$(pc_j[\overline{l}_1, \overline{h} \ / \ L, H] \wedge \overline{o} \doteq \overline{f_j^O}[\overline{l}_1, \overline{h} \ / \ L, H]) = true \tag{4.4}$$

Because $Info(\overline{l}_2, \overline{h}, \overline{o}) = (\bigvee_{1 \le i \le N_p} pc_i[\overline{l}_2, \overline{h} \ / \ L, H] \wedge \overline{o} \doteq \overline{f_i^O}[\overline{l}_1, \overline{h} \ / \ L, H]) = false$, we have $\forall i \in [1, N_p].\ (pc_i[\overline{l}_2, \overline{h} \ / \ L, H] \wedge \overline{o} \doteq \overline{f_i^O}[\overline{l}_2, \overline{h} \ / \ L, H]) = false$. Let k be the symbolic execution

path describing the concrete one gained by executing program p with initial state $L = \bar{l}_2, H = \bar{h}$, we have $pc_k[\bar{l}_2, \bar{h} \ / \ L, H] = true$ that implies $(\overline{o} \doteq \overline{f_k^O}[\bar{l}_2, \bar{h} \ / \ L, H]) = false$. Consequently, we have

$$(pc_k[\bar{l}_2, \bar{h} \ / \ L, H] \wedge \overline{o} \neq \overline{f_k^O}[\bar{l}_2, \bar{h} \ / \ L, H]) = true \tag{4.5}$$

From (4.4) and (4.5) we have

$$pc_j[\bar{l}_1, \bar{h} \ / \ L, H] \wedge pc_k[\bar{l}_2, \bar{h} \ / \ L, H] \wedge \overline{f_j^O}[\bar{l}_1, \bar{h} \ / \ L, H] \neq \overline{f_k^O}[\bar{l}_2, \bar{h} \ / \ L, H] = true$$

which means that formula $Leak(L, O, j, k)$ is satisfied by $L = \bar{l}_1, L' = \bar{l}_2, H = H' = \bar{h}$. Hence generalized noninterference policy $L \not\rightsquigarrow_{GNI} O$ is violated which contradicts to the assumption that p is low-independent. Therefore, $Info_{\langle \bar{l}_1, \overline{o} \rangle}(H) \equiv Info_{\langle \bar{l}_2, \overline{o} \rangle}(H)$ and Theorem 4.1 is proven. □

A trivial corollary of Theorem 4.1 is that we need one and only one experiment with arbitrary low input value to achieve the maximum knowledge of high input if program p is low-independent. In this case, finding optimal low input is clearly unnecessary.

Example 4.5. *The following program*

```
if ( h > 0 ) { l = 2; } else {l = 1;}
```

is low-independent w.r.t. noninterference policy $\{h\} \not\rightsquigarrow \{l\}$ *and observable variable* `l` *because it enjoys the generalized noninterference policy* $\{l\} \not\rightsquigarrow_{GNI} \{l\}$. *The maximum knowledge of* `h` *is simply its sign and can be obtained by a single experiment with arbitrary input value of* `l`. *For instance, assume that the input value of* `h` *is* -5, *any experiment will return* 1 *as the output value of* `l`, *bringing* $h \leq 0$ *as the maximum knowledge of* `h`*'s input value.*

4.4.2 Exploiting Risky Paths and Reachable Paths

If program p is not low-independent, different low inputs might bring different knowledge of high input. We start with a set of experiments E ($|E| = m$) and the accumulated knowledge about the high inputs in the form of a logic formula $K^E(H)$. We assume the initial knowledge about secret $K^{\emptyset}(H)$ is correct ($\bar{h}_s$ satisfies $K^{\emptyset}(H)$), hence $\bar{h}_s$ also satisfies $K^E(H)$. Our aim is to find the low level input $\bar{l}_{m+1}$ for a new experiment that is most promising for a maximal knowledge gain. In this subsection we discuss how to avoid generation of low input that would lead to a redundant experiment.

Definition 4.5 (Path-matched low input)**.** *An input value* $\bar{l}$ *of* L *is called* i-matched *for* $K^E(H)$ *iff the formula* $\forall \bar{h}.\ (K^E(\bar{h}) \rightarrow pc_i[\bar{l}, \bar{h} \ / \ L, H])$ *holds.*

Intuitively, if a concrete low input $\bar{l}$ is i-matched for $K^E(H)$ and $K^E(H)$ is correct, then $pc_i[\bar{l}/L]$ holds for all possible values of high inputs.

Since all path conditions are mutually exclusive, we can conclude that if $\bar{l}$ is i-matched for $K^E(H)$ and $\langle \bar{l}, \overline{o} \rangle$ is the corresponding experiment, then $K^E(H) \wedge Info_{\langle \bar{l}, \overline{o} \rangle}(H) \equiv K^E(H) \wedge InfoPath_i(\bar{l}, H, \overline{o})$. If $K^E(H) = true$ then we have $Info_{\langle \bar{l}, \overline{o} \rangle}(H) \equiv InfoPath_i(\bar{l}, H, \overline{o})$.

Lemma 4.1. *If* $\bar{l}$ *is* i*-matched for* $K^E(H)$ *for some* $i \notin Risk$, *then any program run with input* $\bar{l}$ *leads to the same output* $\overline{o} \in \mathbb{O}_D(\bar{l})$ *for all high inputs* $\bar{h}$ *for which* $K^E(\bar{h})$ *holds.*

Proof. Consider two arbitrary concrete values $\overline{h}_1, \overline{h}_2 \in \mathbb{H}$ for which $K^E(\overline{h}_1)$ and $K^E(\overline{h}_2)$ hold. Let $\overline{o}_1, \overline{o}_2 \in \mathbb{O}_D(\overline{l})$ be output values observed in the final state of two program runs taking $(\overline{l}, \overline{h}_1)$ and $(\overline{l}, \overline{h}_2)$ as input values, respectively. We will prove that $\overline{o}_1 = \overline{o}_2$. Assume that $\overline{o}_1 \neq \overline{o}_2$, because $\overline{l}$ is i-matched and $K^E(\overline{h}_1)$, $K^E(\overline{h}_2)$ hold, we get as direct consequence of Definition 4.5 that $pc_i[\overline{l}, \overline{h}_1/L, H]$ and $pc_i[\overline{l}, \overline{h}_2/L, H]$ are true. This means that the two concrete runs with $(\overline{l}, \overline{h}_1)$ and $(\overline{l}, \overline{h}_2)$ as input values correspond to the symbolic execution path i, and hence, by the correctness of symbolic execution that $\overline{o}_1 = \overline{f_i^O}[\overline{l}, \overline{h}_1 \ / \ L, H]$ and $\overline{o}_2 = \overline{f_i^O}[\overline{l}, \overline{h}_2 \ / \ L, H]$. By assumption $\overline{o}_1 \neq \overline{o}_2$, i.e., $\overline{f_i^O}[\overline{l}, \overline{h}_1 \ / \ L, H] \neq \overline{f_i^O}[\overline{l}, \overline{h}_1 \ / \ L, H]$. Consequently, we have

$$pc_i[\overline{l}, \overline{h}_1/L, H] \wedge pc_i[\overline{l}, \overline{h}_2/L, H] \wedge \overline{f_i^O}[\overline{l}, \overline{h}_1 \ / \ L, H] \neq \overline{f_i^O}[\overline{l}, \overline{h}_1 \ / \ L, H] = true$$

Thus formula $Leak(H, O, i, i)$ (see Definition 4.1) is satisfied by $\overline{l}, \overline{h}_1, \overline{h}_2$, which means that i is a risky path. It contradicts the assumption of the lemma that i is not a risky path. Hence $\overline{o}_1 = \overline{o}_2$ and this lemma is proven. □

Theorem 4.2. *If $\overline{l}$ is i-matched for $K^E(H)$ and some $i \notin Risk$, then experiment $\langle \overline{l}, \overline{o} \rangle$ is redundant.*

Proof. By Lemma 4.1 we know that for a given $\overline{l} \in \mathbb{L}$ that is i-matched with $i \notin Risk$, all program runs produce the same output $\overline{o} \in \mathbb{O}$ for any high input $\overline{h} \in \mathbb{H}$ for which $K^E(\overline{h})$ holds.
Hence we have only to prove that

$$\forall \overline{h}.(K^E(\overline{h}) \rightarrow Info_{\langle \overline{l}, \overline{o} \rangle}(\overline{h})) \tag{4.6}$$

Let $\overline{h}_0$ be an arbitrary but fixed value in $\mathbb{H}$. We have to show that

$$K^E(\overline{h}_0) \rightarrow Info_{\langle \overline{l}, \overline{o} \rangle}(\overline{h}_0) \tag{4.7}$$

is true.

Case 1: If $K^E(\overline{h}_0)$ is false then (4.7) is trivially true (semantics of implication) and we are done.

Case 2: We can now assume that $K^E(\overline{h}_0)$ is true. Because $\overline{l}$ is i-matched (assumption of the theorem) we have $Info_{\langle \overline{l}, \overline{o} \rangle}(\overline{h}_0) \equiv InfoPath_i(\overline{l}, \overline{h}_0, \overline{o}) \equiv$

$$pc_i[\overline{l}, \overline{h}_0/L, H] \wedge \overline{o} \doteq \overline{f_i^O}[\overline{l}, \overline{h}_0 \ / \ L, H]$$

- The validity of $pc_i[\overline{l}, \overline{h}_0/L, H]$ follows directly from Definition 4.5 and our case assumption ($K^E(\overline{h}_0)$ holds).
- The second conjunct $\overline{o} \doteq \overline{f_i^O}[\overline{l}, \overline{h}_0 \ / \ L, H]$ is a direct consequence of the correctness of symbolic execution: The $\overline{o}$ are the result of running program p with input $\overline{l}, \overline{h}_0$, hence, given the correctness of symbolic execution the symbolic output values must evaluate to the same concrete values.

□

Theorem 4.2 has the following corollary:

Corollary 4.1. *InRisk*(L) *denotes the formula* $\exists \overline{h}.\left(K^E(\overline{h}) \wedge \bigwedge_{i \notin Risk} \neg pc_i[\overline{h}\,/\,H]\right)$. *If for some* $\overline{l} \in \mathbb{L}$ *the formula InRisk*$(\overline{l})$ *is false then the experiment* $\langle \overline{l}, \overline{o} \rangle$ *is redundant for* $K^E(H)$.

Example 4.6. *[Example 4.1 cont'd] According to Example 4.1, the SE tree of program rPC has four paths with path conditions* $pc_1 = l < 100 \wedge l = h$, $pc_2 = l < 100 \wedge l < h$, $pc_3 = l < 100 \wedge l > h$ *and* $pc_4 = l \geq 100$. *The set of risky paths is* $Risk = \{1, 2, 3\}$. *The fourth path is not a risky path as it does not contribute to any leak. We have* $InRisk(\{l\}) = \exists h.\neg(l \geq 100) \equiv l < 100$ *indicating that only low input values less than* 100 *may lead to any information gain.*

Definition 4.6 (Reachable path)**.** *An SE path* i *is called a* reachable path *for* $K^E(H)$ *iff the following formula is satisfiable:*

$$K^E(H) \wedge pc_i \tag{4.8}$$

R^E *denotes the set of all reachable paths for* $K^E(H)$.

Example 4.7. *(Example 4.6 cont'd) Assume the initial knowledge about the value of* h *is* $-2^{31} \leq h < 2^{31}$ *and the secret value of* h *is* 1000. *We execute the program in Listing 4.1 with* $l = 98$. *The execution terminates in a state where* l *has been set to* 0. *Using this experiment, we obtain as accumulated knowledge about* h: $-2^{31} \leq h < 2^{31} \wedge ((98 \doteq h \wedge 3 = 0) \vee (98 < h \wedge 0 \doteq 0) \vee (98 > h \wedge -3 \doteq 0)) \equiv 98 < h < 2^{31}$. *With this knowledge about* h, *the risky path 3 becomes unreachable because the formula* $98 < h < 2^{31} \wedge l < 100 \wedge l > h$ *is unsatisfiable.*

Theorem 4.3. *For all experiments* $\langle \overline{l}, \overline{o} \rangle$, *it holds that*

$$K^E(H) \wedge Info_{\langle \overline{l}, \overline{o} \rangle}(H) \equiv K^E(H) \wedge \bigvee_{i \in R^E} InfoPath_i(\overline{l}, H, \overline{o})$$

Proof. We can rewrite the definition of $Info_{\langle \overline{l}, \overline{o} \rangle}(H)$ ($\overline{o} = \{o_\nu | o_\nu \in \mathbb{O}_{\overline{l}}, \nu \in O\}$) (from (4.1) and (4.2)) simply to

$$\Big(\bigvee_{i \in R^E} InfoPath_i(\overline{l}, H, \overline{o})\Big) \vee \Big(\bigvee_{i \notin R^E \wedge 1 \leq i \leq N_p} InfoPath_i(\overline{l}, H, \overline{o})\Big)$$

To prove Theorem 4.3, we prove that for all $i \notin R^E$ and $1 \leq i \leq N_p$

$$K^E(H) \wedge InfoPath_i(\overline{l}, H, \overline{o}) \tag{4.9}$$

is unsatisfiable (i.e., equivalent to *false*).
Let $i_0 \notin R^E$ be an arbitrary but fixed unreachable path. $InfoPath_{i_0}(\overline{l}, H, \overline{o})$ is defined as

$$pc_{i_0}[\overline{l}/L] \wedge \overline{o} \doteq \overline{f_{i_0}^O}[\overline{l}/L]$$

Because $i \notin R^E$, by Definition 4.6, we have $K^E(H) \wedge pc_{i_0}$ is unsatisfiable, hence

$$K^E(H) \wedge InfoPath_{i_0}(\overline{l}, H, \overline{o})$$

is unsatisfiable. which proves (4.9) and therewith Theorem 4.3. □

Theorem 4.3 shows that all unreachable paths can be ignored while constructing the knowledge about $\overline{h_s}$. Moreover, it allows us to consider only reachable paths when deducing optimal low input, which we explain in the next sections.

4.4.3 Implementation of Method *findLowInput*

In case p is not a low-independent program, we want to extend E by adding a new experiment $\langle \bar{l}, \bar{o} \rangle$ to gain as much information about the secret as possible. The mission now is finding the optimal low input value $\bar{l}$. Quantitative information flow analysis gives us a hint: the low input value to be used should be one that maximizes the leakage quantified by means of a security metric. This idea is implemented in Algorithm 4.2.

Data: Set of performed experiments E, current knowledge $K^E(H)$
Result: $(\bar{l}, leakage)$: optimal low input value and corresponding leakage

```
1  begin
2      R^E ← findAllReachablePaths(K^E(H));
3      if |R^E| > 0 ∧ R^E ∩ Risk ≠ ∅ then
4          QLeak(L) ← appropriatly instantiated entropy formula;
5          l̄ ← findL2Maximize(QLeak(L));
6          if l̄ = null then
7              l̄ ← random value that does not appear in E;
8          end
9          leakage ← QLeak(l̄);
10     else
11         l̄ ← null;
12         leakage ← 0;
13     end
14 end
15 return (l̄, leakage)
```

Algorithm 4.2: Implementation of method *findLowInput*

Algorithm 4.2 shows the pseudo-code of method *findLowInput*. It computes the optimal low input values for a given leakage metric together with the computed leakage. First, the set of reachable paths R^E is determined by checking the reachability of all paths using formula (4.8). If no reachable paths exists or all reachable paths are not risky, the algorithm exits and returns 0 as leakage value (in those cases the low input values are irrelevant). Otherwise, the optimal low input values for the leakage metric are computed.

Here $QLeak(L)$ is one of $\mathtt{ShEL}_p(L)$, $\mathtt{GEL}_p(L)$, $\mathtt{MEL}_p(L)$ according to the chosen security metric. The low input values are determined by solving the optimization problem: $argmax_{\bar{l} \in \mathbb{L}} QLeak(\bar{l})$. The details of how to construct $QLeak(L)$ as well as how to find $\bar{l}$ maximizing $QLeak(L)$ are the content of Chapter 5.

5 Leakage Maximization with Low Input

This chapter discusses in detail how to generate optimal low input that maximizes information leakage. Section 5.1 introduces a novel approach that precisely quantifies information leakage with low inputs. Based on the result of Section 5.1, Section 5.2 provides some algorithms to find optimal low input. Drawbacks and applicable use cases of the approach are discussed in Section 5.3. Parts of this chapter are based on the technical report [46] of the author of this thesis.

5.1 Quantifying Leakage with Low Input

This section provides explicit representations for different information leakages measured from Section 2.2 as corresponding functions of low inputs. This approach employs symbolic execution and parametric counting and is based on the essential assumption that the knowledge $K^E(H)$ is correct.

5.1.1 Parametric Counting Function

We give here the formal definition and notational convention for parametric counting functions that will be used throughout this chapter.

Definition 5.1 (Parametric counting function)**.** *For a formula g, let V be the set of all program variables occurring in g and let $V = X \dot\cup Y$ be a partitioning. Function $\mathtt{C}_X[Y](g)$ is called* parametric counting function *iff it returns the number of assignments to the variables of X that satisfy g (i.e. the number of models) as a function of Y.*

Example 5.1. *Given $V = \{h, l\}$ and $g = 0 \leq h < 100 \wedge h \geq l \wedge 0 \leq l < 100$. Then the number of models of h satisfying g depends on l and can be determined for any value of l satisfying $0 \leq l < 100$ by $\mathtt{C}_{\{h\}}[\{l\}](g) = 100 - l$.*

5.1.2 Logic Characterization of Probability Distribution

The leakage quantification approach proposed in the following section makes use of parametric counting on logical formulas composed by means of symbolic execution. To incorporate the prior distribution of high input into counting formulas more conveniently, we give a formalization of probability distribution based on logic formulas. It can be seen as an equivalent characterization and an alternative of a *weight function* that is a popular encoding of probability distributions in the literature.

Given program p with H the set of high variables and $\mathbb{H}$ the set of all possible values of H, the distribution of high input is given in the form of a *weight function* $w : \mathbb{H} \mapsto \mathbb{Z}_{\geq 0}$ that assigns each possible high value a non-negative integer number that is its frequency of occurrence. The following definition gives a logic characterization of a weight function w. We restrict the definition to a set of high variables H.

Definition 5.2 (Logic-characterization of probability distribution)**.** *Given a set of high variables H whose value's distribution is defined by the weight function $w : \mathbb{H} \mapsto \mathbb{Z}_{\geq 0}$. A logic characterization of w, denoted by Δ_H^w is a tuple $\langle FO_H, \mu \rangle$ where $FO_H = \{\varphi_i(H) | i = 1 \ldots n\}$ is a set of logic formulas over H that satisfies following conditions:*

- $\forall i, j \in [1, n].\ i \neq j \Rightarrow Sat(\varphi_i(H)) \cap Sat(\varphi_j(H)) = \emptyset$
- $\forall \overline{h} \in \mathbb{H}.\ \exists i \in [1, n].\ \overline{h} \in Sat(\varphi_i(H))$

and $\mu : FO_H \mapsto \mathbb{Z}_{\geq 0}$ is a function mapping each formula $\varphi_i(H)$ to a non-negative integer number that satisfies:

- $\forall i \in [1, n].\ \forall \overline{h} \in Sat(\varphi_i(H)).\ w(\overline{h}) = \mu(\varphi_i(H))$
- $\forall i, j \in [1, n].\ i \neq j \Rightarrow \mu(\varphi_i(H)) \neq \mu(\varphi_j(H))$

$\mu(\varphi_i(H))$ is called weight value *of formula $\varphi_i(H)$.*

Intuitively, the set $\mathbb{H}$ is partitioned into subsets $\{Sat_H(\varphi_i(H)) | i = 1..n\}$ in which each partition contains all high inputs having the same frequency value determined by the weight function w. We call $\varphi_i(H)$ *partition formula* of partition i. It can be seen that the logic characterization of w preserves the weight function of all values of the sample space. We show that it can be used as an alternative for the weight function to define the distribution of high inputs by following lemma:

Lemma 5.1. *For any weight funtion w, there exists a logic characterization of w as defined in Definition 5.2.*

Proof. We prove Lemma 5.1 by pointing out how to construct a logic characterization for a weight function w on the joint domain $\mathbb{H}$ of the set of variables H. We start with an empty set FO_H then iterate all elements of $\mathbb{H}$. For each $\overline{h} \in \mathbb{H}$, if there is a formula $\varphi_i(H) \in FO_H$ that satisfies $w(\overline{h}) = \mu(\varphi_i(H))$ then we add a disjunct $H = \overline{h}$ to $\varphi_i(H)$, otherwise we add a new formula $\varphi_{|FO_H|+1}(H) \equiv H = \overline{h}$ to FO_H and assign $\mu(\varphi_{|FO_H|+1}(H)) = w(\overline{h})$. It is easy to see that $\langle FO_H, \mu \rangle$ satisfies all conditions given in Definition 5.2. □

From now on, we always use a logic characterization of weight functions to define the distribution of high input. Instead of using notation Δ_H^w, we simply use Δ_H to refer to the prior distribution of high input. Δ_H is called a *formula-based distribution* of a secret.

Example 5.2. *We consider program `rPC` in Listing 4.1 with the set of high variables $\{h\}$. The uniform distribution on input values of h can be characterized by formula-based distribution $\langle FO_h, \mu \rangle$ in which $FO_h = \{true\}$ and $\mu(true) = c$ ($c \in \mathbb{Z}^+$ is a positive integer number, e.g. 1). On the other hand, the formula-based distribution $\langle FO_h, \mu \rangle$ in which $FO_h = \{h \geq 0, h < 0\}$ and $\mu(h \geq 0) = 1$, $\mu(h < 0) = 2$ determines a non-uniform distribution, which defines that the probability that a number c is the input value of h is identical for all c being negative, and this probability is two times bigger than the probability that a non-negative number is the input value of h.*

The next section explains how to quantify the leakage with respect to a formula-based distribution of secret. In case of guessing entropy, the computation iterates over all possible values of the sample space in descending order of probability (see Definition 2.8). Therefore, to ease the computation and representation, with guessing entropy we use the sorted form of formula-based distribution defined as follows:

Definition 5.3 (Sorted formula-based distribution)**.** *A formula-based distribution* $\Delta_H = \langle FO_H, \mu \rangle$ *is called* sorted *if and only if it holds that the set* FO_H *is ordered and* $\mu(\varphi_1(H)) > \mu(\varphi_2(H)) > .. > \mu(\varphi_n(H))$ *with* $n = |FO_H|$.

5.1.3 Quantifying Leakage with Arbitrary Distribution of Secret

We show how to quantify the leakage given that the prior distribution of the input value of H is $\Delta_H = \langle FO_H, \mu \rangle$ ($FO_H = \{\varphi_1(H), \cdots, \varphi_n(H)\}$). We assume that an experiment set E has been conducted establishing $K^E(H)$ as the correct knowledge about H's value. The following theorems provide an iterative method to compute leakage using Shannon, guessing or min entropy as leakage metric. Because the approach takes prior distribution of high inputs into account, the channel-capacity metric, which is the maximum leakage over all prior distributions of high inputs, is not considered.

Shannon Entropy

Theorem 5.1. *Given the prior distribution of high inputs* $\Delta_H = \langle FO_H, \mu \rangle$ *where* $FO_H = \{\varphi_1(H), \ldots, \varphi_n(H)\}$*, the Shannon entropy-based leakage can be computed as follows:*

$$\mathtt{ShEL}_\mathtt{p}(L) = \mathtt{log}(S_E^{\Delta_H}) - \frac{1}{S_E^{\Delta_H}} \sum_{\overline{o} \in \mathbb{O}_D(L)} \Big(\sum_{i=1}^{n} c_i^{\overline{o}}(L)\mu(\varphi_i(H))\mathtt{log}(\sum_{i=1}^{n} c_i^{\overline{o}}(L)\mu(\varphi_i(H))) \Big) \tag{5.1}$$

where $\mathbb{O}_D(\overline{l})$ *is the set of all possible output value of* O *given* $\overline{l}$ *is input value of* L *and the input value of* H *satisfies* $K^E(H)$*, and*

$$\begin{aligned}
&S_E^{\Delta_H} = \sum_{i=1}^{n} c_i \mu(\varphi_i(H)) \\
&c_i = |Sat(K^E(H) \wedge \varphi_i(H))| \\
&c_i^{\overline{o}}(L) = \mathtt{C}_H[L](\mathtt{g}_i(L, H, \overline{o})),\ (1 \le i \le n) \\
&\mathtt{g}_i(L, H, O) = K^E(H) \wedge InRisk(L) \wedge (\bigvee_{j \in R^E} InfoPath_j(L, H, O)) \wedge \varphi_i(H)
\end{aligned}$$

in which O' *is defined as in Section 4.2.*

Proof. According to (2.1), we have

$$\mathtt{ShEL}_\mathtt{p}(L) = \mathscr{H}(O_{out}(L)) \underset{\text{Def. 2.6}}{=} - \sum_{\overline{o} \in \mathbb{O}_D(L)} P(O_{out}(L) = \overline{o}) \mathrm{log} P(O_{out}(L) = \overline{o}) \tag{5.2}$$

By definition, $O_{out}(L)$ is the value of O observed after running program with low input L and H_{in}. We have by the law of total probability

$$P(O_{out}(L) = \overline{o}) = \sum_{\overline{h} \in \mathbb{H}} P(H_{in} = \overline{h}) P(O_{out}(L) = \overline{o} | H_{in} = \overline{h}) \tag{5.3}$$

Under the assumption that $K^E(H)$ is correct, from Corollary 4.1 and Theorem 4.3, we only consider values of L that satisfy $InRisk(L)$ (to avoid redundant experiments) and take into account only reachable paths for $K^E(H)$. Because program p is deterministic, for any $\bar{l} \in \mathbb{L}$, we have:

$$P(O_{out}(\bar{l}) = \bar{o} | H_{in} = \bar{h}) = \begin{cases} 1, \text{ if } \bar{h} \in Sat(K^E(H) \wedge InRisk(\bar{l}) \wedge \bigvee_{i \in R^E} pc_i[\bar{l}/L] \wedge \bar{o} \doteq \overline{f_i^O}[\bar{l}/L]) \\ 0, \text{ otherwise} \end{cases} \tag{5.4}$$

Because H_{in} has formula-based distribution $\Delta_H = \langle FO, \mu \rangle$, we have:

$$P(H_{in} = \bar{h}) = \begin{cases} \frac{\mu(\varphi_i(H))}{S_E^{\Delta_H}}, & \text{if } \bar{h} \in Sat(K^E(H) \wedge \varphi_i(H)) \\ 0, & \text{otherwise} \end{cases} \tag{5.5}$$

where $S_E^{\Delta_H} = \sum_{i=1}^{n} |Sat(K^E(H) \wedge \varphi_i(H))| \mu(\varphi_i(H))$.

Recall that $g_i(L, H, \bar{o}) = K^E(H) \wedge InRisk(L) \wedge (\bigvee_{j \in R^E} pc_j \wedge \bar{o} \doteq \overline{f_j^O}) \wedge \varphi_i(H)$, from (5.3), (5.4) and (5.5) we have

$$\forall \bar{l} \in \mathbb{L}.\ P(O_{out}(\bar{l}) = \bar{o}) = \frac{\sum_{i=1}^{n} |Sat(g_i(\bar{l}, H, \bar{o}))| \mu(\varphi_i(H))}{S_E^{\Delta_H}} \underset{\text{Def. 5.1}}{=} \frac{\sum_{i=1}^{n} c_i^{\bar{o}}(\bar{l}) \mu(\varphi_i(H))}{S_E^{\Delta_H}}$$

hence

$$P(O_{out}(L) = \bar{o}) = \frac{\sum_{i=1}^{n} c_i^{\bar{o}}(L) \mu(\varphi_i(H))}{S_E^{\Delta_H}} \tag{5.6}$$

Equation (5.1) can be derived from (5.2) and (5.6) □

Intuitively, $S_E^{\Delta_H}$ is the weighted sum of all possible values of H that satisfy the current knowledge $K^H(H)$, while c_i is the weighted sum of all values correctly described by the current knowledge and belongs to partition characterized by distribution formula $\varphi_i(H)$. To compute Shannon entropy-based leakage for the case of uniform distribution, we derive the following corollary from Theorem 5.1:

Corollary 5.1. *If the prior distribution of high inputs Δ_H is uniform then*

$$\mathtt{ShEL}_{\mathtt{p}}(L) = log(S_E) - \frac{1}{S_E} \sum_{\bar{o} \in \mathbb{O}_D(L)} \left(\mathtt{C}_H[L](g(L, H, \bar{o})) log(\mathtt{C}_H[L](g(L, H, \bar{o}))) \right) \tag{5.7}$$

where $S_E = |Sat(K^E(H))|$ and $g(L, H, O') = K^E(H) \wedge InRisk(L) \wedge (\bigvee_{j \in R^E} InfoPath_j(L, H, O'))$ (O' is defined as in Section 4.2).

Proof. We show that formula (5.1) can be reduced to (5.7) in case of uniform distribution of high input. For the sake of simplicity, we characterize uniform distribution by $\Delta_H = \langle FO_H, \mu \rangle$ in which $FO_H = \{true\}$ and $\mu(true) = 1$. Then we have $S_E^{\Delta_H} = \sum_{i=1}^{n} |Sat(K^E(H) \wedge \varphi_i(H))| \mu(\varphi_i(H)) = |Sat(K^E(H))| = S_E$ and $\sum_{i=1}^{n} c_i^{\bar{o}}(L) \mu(\varphi_i(H)) = \mathtt{C}_H[L](g(L, H, \bar{o}))$. Thus we can obtain equation (5.7) from (5.1). □

Formula $g(L,H,O')$ captures a big-step operational semantics for program p: $g(\overline{l},\overline{h},\overline{o})$ describes the transition from initial state σ that $\sigma(L) = \overline{l}, \sigma(H) = \overline{h}$ to final state σ_{out} that $\sigma_{out}(O) = \overline{o}$, where $\overline{l}$ satisfies $InRisk(L)$ and $\overline{h}$ satisfies $K^E(H)$. Apart from g, formula $g_i(L,H,O')(i = 1 \cdots n)$, which is equivalent to $g(L,H,O') \wedge \varphi_i(H)$, enriches the constraint of the high input value by specifying that it also satisfies distribution formula $\varphi_i(H)$. Such transformation relations are characterized within $g(L,H,O')$ and $g_i(L,H,O')$ using symbolic execution that takes into account only reachable paths (w.r.t. $K^E(H)$) to reduce the complexity.

In a special case where the observable outputs depend only on the chosen symbolic execution path, but not on the actual values of the low or high variables, formula $g(L,H,O')$ (and thereby $g_i(L,H,O')$) can be simplified significantly. Going into detail, consider program p satisfying above condition, let j be a reachable path with path condition pc_j and symbolic output values $\overline{f_j^O}$. By assumptiom, the symbolic values in $\overline{f_j^O}$ are constants (i.e. independent of any program variables), so they can be evaluated to concrete values $\overline{o}_j$. We may assume that the output values for all SE paths $j \neq k$ differ, hence $\overline{o}_j \neq \overline{o}_k$ (otherwise, paths j, k are merged into one with path condition $pc_j \vee pc_k$). Further, $\mathbb{O}_D(L) = \{\overline{o}_j | j \in R^E\}$, because we only consider reachable paths. Taking both observations together, we conclude that for all $j,k \in R^E$ with $j \neq k$ the formula $InfoPath_j(L,H,\overline{o}_k)$ is equivalent to false and $InfoPath_j(L,H,\overline{o}_j)$ simplifies to pc_j. We use this to simplify the definitions of g in Corollary 5.1:

$$g(L,H,\overline{o}_j) \equiv K^E(H) \wedge InRisk(L) \wedge pc_j$$

Guessing Entropy

Theorem 5.2. *Given the prior distribution of high inputs $\Delta_H = \langle FO_H, \mu \rangle$ where $FO_H = \{\varphi_1(H), \ldots, \varphi_n(H)\}$. If Δ_H is sorted, the guessing entropy-based leakage can be computed as follows:*

$$\mathtt{GEL}_{\mathrm{p}}(L) = \sum_{i=1}^{n} \frac{\mu(\varphi_i(H))}{S_E^{\Delta_H}} \Big(\frac{c_i + 1}{2} + \sum_{j=1}^{i-1} c_j\Big) c_i - \sum_{\overline{o} \in \mathbb{O}_D(L)} \Big(\sum_{i=1}^{n} \frac{\mu(\varphi_i(H))}{S_E^{\Delta_H}} \Big(\frac{c_i^{\overline{o}}(L) + 1}{2} + \sum_{j=1}^{i-1} c_j^{\overline{o}}(L)\Big) c_i^{\overline{o}}(L) \Big) \quad (5.8)$$

where c_i, $S_E^{\Delta_H}, c_i^{\overline{o}}(L)$ are defined as in Theorem 5.1.

Proof. By Definition 5.2, the set $Sat(K^E(H))$ is partitioned into subsets $\{Sat(K^E(H) \wedge \varphi_i(H)) | i = 1 \ldots n\}$. Recall that $S_E = |Sat(K^E(H))|$ is the number of all possible values of H satisfying $K^E(H)$, it can be seen that $S_E = \sum_{i=1}^{n} c_i$ (c_i as in Theorem 5.1). By Definition 2.8 we have

$$\mathscr{G}(H_{in}) = \sum_{1 \leq k \leq S_E} k \cdot P(H_{in} = \overline{h_k}) \quad (5.9)$$

Consider an arbitrary partition defined by formula $\varphi_i(H)$ and arbitrary k satisfying that $\overline{h_k} \in Sat(K^E(H) \wedge \varphi_i(H))$. From (5.5) we have $P(H_{in} = \overline{h_k}) = \frac{\mu(\varphi_i(H))}{S_E^{\Delta_H}}$. Because Δ_H is sorted, we can see that k must range from $1 + \sum_{j=1}^{i-1} c_j$ to $\sum_{j=1}^{i} c_j$. Therefore we have from (5.9):

$$\mathscr{G}(H_{in}) = \sum_{1 \le k \le S_E} k \cdot P(H_{in} = \overline{h_k}) = \sum_{i=1}^{n} \frac{\mu(\varphi_i(H))}{S_E^{\Delta_H}} \sum_{j=c_1+\cdots+c_{i-1}+1}^{c_1+\cdots+c_{i-1}+c_i} j$$
$$= \sum_{i=1}^{n} \frac{\mu(\varphi_i(H))}{S_E^{\Delta_H}} \left(\frac{c_i + 1}{2} + \sum_{j=1}^{i-1} c_j\right) c_i \quad (5.10)$$

Pay attention that if $i = 1$ then $\sum_{j=1}^{i-1} j = \sum_{j=1}^{i-1} c_j = 0$ by convention. According to Definition 2.8 we also have

$$\mathscr{G}(H_{in}|O_{out}(L)) = \sum_{\overline{o} \in \mathbb{O}_D(L)} P(O_{out}(L) = \overline{o}) \mathscr{G}(H_{in}|O_{out}(L) = \overline{o})$$

where

$$\mathscr{G}(H_{in}|O_{out}(L) = \overline{o}) = \sum_{1 \le k \le S_E} k \cdot P(H_{in} = \overline{h_k}|O_{out}(L) = \overline{o})$$

Because $P(H_{in} = \overline{h_k}|O_{out}(L) = \overline{o}) \underset{\text{Bayes}}{=} \frac{P(O_{out}(L)=\overline{o}|H_{in}=\overline{h_k})P(H_{in}=\overline{h_k})}{P(O_{out}(L)=\overline{o})}$ we have

$$\mathscr{G}(H_{in}|O_{out}(L)) = \sum_{\overline{o} \in \mathbb{O}_D(L)} \sum_{1 \le k \le S_E} k \cdot P(O_{out}(L) = \overline{o}|H_{in} = \overline{h_k}) P(H_{in} = \overline{h_k}) \quad (5.11)$$

from (5.4) and (5.5) we have

$$\forall \bar{l} \in \mathbb{L}.\ P(O_{out}(\bar{l}) = \overline{o}|H_{in} = \overline{h_k}) P(H_{in} = \overline{h_k}) = \begin{cases} \frac{\mu(\varphi_i(H))}{S_E^{\Delta_H}}, & \text{if } \overline{h_k} \in Sat(\mathrm{g}_i(\bar{l}, H, \overline{o})) \\ 0, & \text{otherwise} \end{cases} \quad (5.12)$$

Similar to computing $\mathscr{G}(H_{in})$, from (5.11) and (5.12) we have

$$\mathscr{G}(H_{in}|O_{out}(L)) = \sum_{\overline{o} \in \mathbb{O}_D(L)} \sum_{i=1}^{n} \frac{\mu(\varphi_i(H))}{S_E^{\Delta_H}} \left(\frac{c_i^{\overline{o}}(L) + 1}{2} + \sum_{j=1}^{i-1} c_j^{\overline{o}}(L)\right) c_i^{\overline{o}}(L) \quad (5.13)$$

Theorem 5.2 is proven by combining (2.8), (5.10) and (5.13). □

In case the prior distribution of high inputs is uniform, the following corollary shows the simplified version of guessing entropy-based leakage

Corollary 5.2. *If the prior distribution of high inputs Δ_H is uniform then*

$$\mathtt{GEL}_{\mathrm{p}}(L) = \frac{S_E + 1}{2} - \frac{1}{2S_E} \sum_{\overline{o} \in \mathbb{O}_D(L)} \left(\mathtt{C}_H[L](g(L,H,\overline{o}))(\mathtt{C}_H[L](g(L,H,\overline{o})) + 1)\right) \quad (5.14)$$

Where S_E and $\mathrm{g}(L,H,O')$ are defined as in Corollary 5.1.

Proof. For the sake of simplicity we assume that $FO_H = \{true\}$ and $\mu(true) = 1$. We have $n = 1$, $S_E^{\Delta_H} = S_E = c_1$, and $c_1^{\overline{o}}(L) = \mathtt{C}_H[L](\mathrm{g}(L,H,\overline{o}))$. In this case equation (5.8) is reduced to (5.14). □

Theorem 5.3. *Given the prior distribution of high inputs $\Delta_H = \langle FO_H, \mu \rangle$ where $FO_H = \{\varphi_1(H), \ldots, \varphi_n(H)\}$, for any $\bar{l} \in \mathbb{L}$ as low input value, the min entropy-based leakage can be computed as follows:*

$$\mathtt{MEL}_{\mathtt{p}}(\bar{l}) = \mathtt{log}\left(\frac{\sum_{\bar{o} \in \mathbb{O}_D(\bar{l})} \mu(\varphi_{x_{\bar{o}}}(H))}{\mu(\varphi_x(H))}\right) \tag{5.15}$$

where $x \in [1, n]$ satisfies

- $Sat(K^E(H) \wedge \varphi_x(H)) \neq \emptyset$
- $\forall i \in [1, n] \wedge i \neq x.\ Sat(K^E(H) \wedge \varphi_i(H)) \neq \emptyset \Rightarrow \mu(\varphi_i(H)) < \mu(\varphi_x(H))$

and $x_{\bar{o}} \in [1, n]$ satisfies

- $Sat(\mathrm{g}_{x_{\bar{o}}}(\bar{l}, H, \bar{o})) \neq \emptyset$
- $\forall i \in [1, n] \wedge i \neq x_{\bar{o}}.\ Sat(\mathrm{g}_i(\bar{l}, H, \bar{o})) \neq \emptyset \Rightarrow \mu(\varphi_i(H)) < \mu(\varphi_{x_{\bar{o}}}(H))$

where $\mathrm{g}_i(L, H, O')$ is defined as in Theorem 5.1

Proof. By Definition 2.7 we have

$$\mathscr{V}(H_{in}) = \max_{\bar{h} \in \mathbb{H}} P(H_{in} = \bar{h}) = \frac{\mu(\varphi_x(H))}{S_E^{\Delta_H}} \tag{5.16}$$

where x is defined as in Theorem 5.3. We also have from Definition 2.7

$$\forall \bar{l} \in \mathbb{L}.\ \mathscr{V}(H_{in} | O_{out}(\bar{l})) = \sum_{\bar{o} \in \mathbb{O}_D(\bar{l})} P(O_{out}(\bar{l}) = \bar{o}) \max_{\bar{h} \in \mathbb{H}} P(H_{in} = \bar{h} | O_{out}(\bar{l}) = \bar{o})$$

We have $P(H_{in} = \bar{h} | O_{out}(\bar{l}) = \bar{o}) \underset{\text{Bayes}}{=} \frac{P(O_{out}(\bar{l}) = \bar{o} | H_{in} = \bar{h}) P(H_{in} = \bar{h})}{P(O_{out}(\bar{l}) = \bar{o})}$, hence

$$\forall \bar{l} \in \mathbb{L}.\ \mathscr{V}(H_{in} | O_{out}(\bar{l})) = \sum_{\bar{o} \in \mathbb{O}_D(\bar{l})} \max_{\bar{h} \in \mathbb{H}} \big(P(O_{out}(\bar{l}) = \bar{o} | H_{in} = \bar{h}) P(H_{in} = \bar{h})\big) \tag{5.17}$$

From (5.12) and (5.17) we have

$$\forall \bar{l} \in \mathbb{L}.\ \mathscr{V}(H_{in} | O_{out}(\bar{l})) = \sum_{\bar{o} \in \mathbb{O}_D(\bar{l})} \frac{\mu(\varphi_{x_{\bar{o}}}(H))}{S_E^{\Delta_H}} \tag{5.18}$$

where $x_{\bar{o}}$ is defined as in Theorem 5.3. From (2.3), (5.16) and (5.18) we conclude

$$\forall \bar{l} \in \mathbb{L}.\ \mathtt{MEL}_{\mathtt{p}}(\bar{l}) = \mathtt{log}\left(\frac{\sum_{\bar{o} \in \mathbb{O}_D(\bar{l})} \mu(\varphi_{x_{\bar{o}}}(H))}{\mu(\varphi_x(H))}\right)$$

□

The computation of min entropy-based leakage for the case of non-uniform distribution does not rely on parametric counting as the two other metrics. This leads to a different technique to find the optimal value of low input that will be explained in Section 5.2. If the distribution Δ_H is uniform, it is easy to see that $\forall \bar{l} \in \mathbb{L}.\ \mathtt{MEL}_{\mathtt{p}}(\bar{l}) = \log(\frac{\sum_{\bar{o}\in\mathbb{O}_D(\bar{l})}\mu(\varphi_{x_{\bar{o}}}(H))}{\mu(\varphi_x(H))}) = \log(\sum_{\bar{o}\in\mathbb{O}_D(\bar{l})} 1) = \log(|\mathbb{O}_D(\bar{l})|)$. This result coincides to Theorem 2.2. In this case, parametric counting can be used to compute min entropy-based leakage.

Theorem 5.4. *If the prior distribution of high inputs Δ_H is uniform then*

$$\mathtt{MEL}_{\mathtt{p}}(L) = \mathtt{log}(\mathtt{C}_{O'}[L](\exists \overline{h}.\ \mathrm{g}(L, \overline{h}, O'))) \tag{5.19}$$

where formula $\mathrm{g}(L, H, O')$ is defined as in Corollary 5.1.

Proof. According to (2.4), the min entropy-based leakage of a deterministic program p with uniform distribution of H_{in} can be computed as below:

$$\mathtt{MEL}_{\mathtt{p}}(L) = \mathtt{log}|\mathbb{O}_D(L)|$$

Under the assumption that the attacker's knowledge of H is correct, we have for any low input $\bar{l} \in \mathbb{L}$, $\bar{o} \in \mathbb{O}_D(\bar{l})$ if and only if there exists a concrete value $\overline{h}_0 \in Sat_H(K^E(H))$ such that program p taking $\overline{h}_0$ and $\bar{l}$ as high and low input, respectively, produces observable output $\bar{o}$. Thus we have

$$\forall \bar{l} \in \mathbb{L}.\ \mathtt{MEL}_{\mathtt{p}}(\bar{l}) = \mathtt{log}|\mathbb{O}_D(\bar{l})| = \mathtt{log}|Sat(\exists \overline{h}.K^E(\overline{h}) \wedge \bigvee_{i\in R^E} pc_i[\bar{l}, \overline{h}/L, H] \wedge O' \doteq \overline{f_i^O}[\bar{l}, \overline{h}/L, H])|$$
$$= \mathtt{log}|Sat(\exists \overline{h}.\mathrm{g}(\bar{l}, \overline{h}, O'))|$$

Hence by Definition 5.1, $\mathtt{MEL}_{\mathtt{p}}(L) = \mathtt{log}(\mathtt{C}_{O'}[L](\exists \overline{h}.g(L, \overline{h}, O')))$. □

5.1.4 Example

Consider program `rPC` in Listing 4.1 with noninterference policy $\mathtt{h} \not\rightsquigarrow \mathtt{l}$, Example 4.6 gives us the set of risky paths $Risk = \{1, 2, 3\}$ with path conditions $pc_1 = \mathtt{l} < 100 \wedge \mathtt{l} \doteq \mathtt{h}$, $pc_2 = \mathtt{l} < 100 \wedge \mathtt{l} < \mathtt{h}$, $pc_3 = \mathtt{l} < 100 \wedge \mathtt{l} > \mathtt{h}$. Assume that `h`,`l` are integer. We first show how to compute information leakage as a function of low input if the input value of `h` has uniform distribution. We assume that the initial knowledge about `h` is that it is a 32 bits integer number, thus $K^{\emptyset}(\mathtt{h}) = -2^{31} \leq \mathtt{h} < 2^{31}$. The number of the values of `h` that satisfy $K^{\emptyset}(\mathtt{h})$ is $S_{\emptyset} = |Sat(-2^{31} \leq \mathtt{h} < 2^{31})| = 2^{32}$. To avoid redundant experiments, we know already that `l` must be chosen such that $\mathtt{l} < 100$ (= $InRisk(\mathtt{l})$). From the symbolic output values, we obtain $\mathbb{O}_{\{\mathtt{l}\}} \subseteq \{3, 0, -3\}$ and:

$$\begin{aligned}
\mathrm{g}(\mathtt{l}, \mathtt{h}, 3) &= -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} \doteq \mathtt{l} \\
\mathrm{g}(\mathtt{l}, \mathtt{h}, 0) &= -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} > \mathtt{l} \\
\mathrm{g}(\mathtt{l}, \mathtt{h}, -3) &= -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} < \mathtt{l} \\
\mathrm{g}(\mathtt{l}, \mathtt{h}, \mathtt{l}') &= -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \\
&\quad \big((\mathtt{l} \doteq \mathtt{h} \wedge \mathtt{l}' \doteq 3) \vee (\mathtt{l} < \mathtt{h} \wedge \mathtt{l}' \doteq 0) \vee (\mathtt{l} > \mathtt{h} \wedge \mathtt{l}' \doteq -3)\big)
\end{aligned}$$

where $\mathtt{l}'$ is a new program variable representing the final value of $\mathtt{l}$. Model counting (we used the tool Barvinok [116]) yields the following functions:

$$\mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}(\mathtt{l},\mathtt{h},3)) = \begin{cases} 1, & \text{if} -2^{31} \leq \mathtt{l} < 100 \\ 0, & \text{otherwise} \end{cases}$$

$$\mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}(\mathtt{l},\mathtt{h},0)) = \begin{cases} 2^{31}-1-\mathtt{l}, & \text{if } -2^{31} \leq \mathtt{l} < 100 \\ 0, & \text{if } \mathtt{l} \geq 100 \\ 2^{32}, & \text{otherwise} \end{cases}$$

$$\mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}(\mathtt{l},\mathtt{h},-3)) = \begin{cases} 2^{31}+\mathtt{l}, & \text{if } -2^{31} \leq \mathtt{l} < 100 \\ 0, & \text{otherwise} \end{cases}$$

$$\mathtt{C}_{\{\mathtt{l}'\}}[\mathtt{l}](\exists \mathtt{h}.\mathtt{g}(\mathtt{l},\mathtt{h},\mathtt{l}')) = \begin{cases} 3, & \text{if } -2^{31} < \mathtt{l} < 100 \\ 2, & \text{if } \mathtt{l} = -2^{31} \\ 1, & \text{otherwise} \end{cases}$$

Shannon entropy-based leakage, computed by using Corollary 5.1, is as follows:

$$\mathtt{ShEL}_{\mathtt{rPC}}(\mathtt{l}) = \begin{cases} 32 - \frac{(2^{31}-1-\mathtt{l})\mathtt{log}(2^{31}-1-\mathtt{l})+(2^{31}+\mathtt{l})\mathtt{log}(2^{31}+\mathtt{l})}{2^{32}}, & \text{if } -2^{31} \leq \mathtt{l} < 100 \\ 0, & \text{otherwise} \end{cases} \tag{5.20}$$

Corollary 5.2 gives us the guessing entropy-based leakage:

$$\mathtt{GEL}_{\mathtt{rPC}}(\mathtt{l}) = \begin{cases} \frac{2^{32}+1}{2} - \frac{(2^{31}-1-\mathtt{l})(2^{31}-\mathtt{l})+(2^{31}+\mathtt{l})(2^{31}+1+\mathtt{l})}{2^{33}}, & \text{if } -2^{31} \leq \mathtt{l} < 100 \\ 0, & \text{otherwise} \end{cases} \tag{5.21}$$

By Theorem 5.4, the min entropy-based leakage is computed as follows:

$$\mathtt{MEL}_{\mathtt{rPC}}(\mathtt{l}) = \begin{cases} \mathtt{log}(3), & \text{if } -2^{31} < \mathtt{l} < 100 \\ 1, & \text{if } \mathtt{l} = -2^{31} \\ 0, & \text{otherwise} \end{cases} \tag{5.22}$$

Now we illustrate the leakage for the case of non-uniform distribution. Assume that the secret input has prior distribution described by formula-based distribution $\Delta_{\mathtt{h}} = \langle FO_{\mathtt{h}}, \mu \rangle$ defined as belows:

$$FO_{\mathtt{h}} = \{\varphi_1(\mathtt{h}), \varphi_2(\mathtt{h})\}$$
$$\varphi_1(\mathtt{h}) = \mathtt{h} < 0 \qquad \mu(\varphi_1(\mathtt{h})) = 2$$
$$\varphi_2(\mathtt{h}) = \mathtt{h} \geq 0 \qquad \mu(\varphi_2(\mathtt{h})) = 1$$

We illustrate how to compute information leakage w.r.t. given non-uniform prior distribution of high input. Recall that $\mathbb{O}_{\{\mathtt{l}\}} \subseteq \{3, 0, -3\}$, we have:

$$\mathtt{g}_1(\mathtt{l},\mathtt{h},3) = -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} \doteq \mathtt{l} \wedge \mathtt{h} < 0$$
$$\mathtt{g}_1(\mathtt{l},\mathtt{h},0) = -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} > \mathtt{l} \wedge \mathtt{h} < 0$$
$$\mathtt{g}_1(\mathtt{l},\mathtt{h},-3) = -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} < \mathtt{l} \wedge \mathtt{h} < 0$$
$$\mathtt{g}_2(\mathtt{l},\mathtt{h},3) = -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} \doteq \mathtt{l} \wedge \mathtt{h} \geq 0$$
$$\mathtt{g}_2(\mathtt{l},\mathtt{h},0) = -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} > \mathtt{l} \wedge \mathtt{h} \geq 0$$
$$\mathtt{g}_2(\mathtt{l},\mathtt{h},-3) = -2^{31} \leq \mathtt{h} < 2^{31} \wedge \mathtt{l} < 100 \wedge \mathtt{h} < \mathtt{l} \wedge \mathtt{h} \geq 0$$

Model counting provides us with the following functions:

$$c_1^3(\mathtt{l}) = \mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}_1(\mathtt{l},\mathtt{h},3)) = \begin{cases} 1, & \text{if } -2^{31} \le \mathtt{l} < 0 \\ 0, & \text{otherwise} \end{cases}$$

$$c_1^0(\mathtt{l}) = \mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}_1(\mathtt{l},\mathtt{h},0)) = \begin{cases} -1-\mathtt{l}, & \text{if } -2^{31} \le \mathtt{l} < -1 \\ 2^{31}, & \text{if } \mathtt{l} < -2^{31} \\ 0, & \text{otherwise} \end{cases}$$

$$c_1^{-3}(\mathtt{l}) = \mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}_1(\mathtt{l},\mathtt{h},-3)) = \begin{cases} 2^{31}, & \text{if } 0 < \mathtt{l} < 100 \\ \mathtt{l}+2^{31}, & \text{if } -2^{31} \le \mathtt{l} \le 0 \\ 0, & \text{otherwise} \end{cases}$$

$$c_2^3(\mathtt{l}) = \mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}_2(\mathtt{l},\mathtt{h},3)) = \begin{cases} 1, & \text{if } 0 \le \mathtt{l} < 100 \\ 0, & \text{otherwise} \end{cases}$$

$$c_2^0(\mathtt{l}) = \mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}_2(\mathtt{l},\mathtt{h},0)) = \begin{cases} 2^{31}-1-\mathtt{l}, & \text{if } 0 \le \mathtt{l} < 100 \\ 2^{31}, & \text{if } \mathtt{l} < 0 \\ 0, & \text{otherwise} \end{cases}$$

$$c_2^{-3}(\mathtt{l}) = \mathtt{C}_{\{\mathtt{h}\}}[\mathtt{l}](\mathtt{g}_2(\mathtt{l},\mathtt{h},-3)) = \begin{cases} \mathtt{l}, & \text{if } 0 < \mathtt{l} < 100 \\ 0, & \text{otherwise} \end{cases}$$

We have $c_1 = |Sat(-2^{31} \le \mathtt{h} < 0)| = 2^{31}, c_2 = |Sat(0 \le \mathtt{h} < 2^{31})| = 2^{31}$, $S_{\emptyset}^{\Delta_\mathtt{h}} = 2c_1 + c_2 = 3 \cdot 2^{31}$. By Theorem 5.1, the Shannon entropy-based leakage of program rPC is

$$\mathtt{ShEL}_{\mathtt{rPC}}(\mathtt{l}) = \begin{cases} \log(3\cdot 2^{31}) - \frac{(2^{31}-1-\mathtt{l})\log(2^{31}-1-\mathtt{l})+(\mathtt{l}+2^{32})\log(\mathtt{l}+2^{32})}{3\cdot 2^{31}}, & \text{if } 0 < \mathtt{l} < 100 \\ \log(3\cdot 2^{31}) - \frac{(2^{31}-1)\log(2^{31}-1)+2^{32}\log(2^{32})}{3\cdot 2^{31}}, & \text{if } \mathtt{l} = 0 \\ \log(3\cdot 2^{31}) - \frac{2^{31}\log(2^{31})+2(2^{31}-1)\log(2(2^{31}-1))}{3\cdot 2^{31}}, & \text{if } \mathtt{l} = -1 \\ \log(3\cdot 2^{31}) - \frac{(2^{31}-2(\mathtt{l}+1))\log(2^{31}-2(\mathtt{l}+1))+2(\mathtt{l}+2^{31})\log(2(\mathtt{l}+2^{31}))+2}{3\cdot 2^{31}}, & \text{if } -2^{31} \le \mathtt{l} < -1 \\ 0, & \text{otherwise} \end{cases} \tag{5.23}$$

It is easy to see that $\Delta_\mathtt{h}$ is a sorted formula-based distribution. By Theorem 5.2, the guessing entropy-based leakage is

$$\mathtt{GEL}_{\mathtt{rPC}}(\mathtt{l}) = \begin{cases} \frac{5\cdot 2^{31}+3}{6} - \frac{2+(2^{31}-\mathtt{l})(2^{31}-\mathtt{l}-1)+2^{32}(2^{31}+1)+\mathtt{l}(2^{32}+\mathtt{l}+1)}{3\cdot 2^{32}}, & \text{if } 0 < \mathtt{l} < 100 \\ \frac{5\cdot 2^{31}+3}{6} - \frac{1+2^{30}(2^{31}-1)+2^{31}(2^{31}+1)}{3\cdot 2^{31}}, & \text{if } \mathtt{l} = 0 \\ \frac{5\cdot 2^{31}+3}{6} - \frac{2+2^{30}(2^{31}+1)+2^{31}(2^{31}-1)}{3\cdot 2^{31}}, & \text{if } \mathtt{l} = -1 \\ \frac{5\cdot 2^{31}+3}{6} - \frac{2+\mathtt{l}(\mathtt{l}+1)+2^{30}(2^{31}+1)-2^{31}(\mathtt{l}+1)+(\mathtt{l}+2^{31})(\mathtt{l}+2^{31}+1)}{3\cdot 2^{31}}, & \text{if } -2^{31} \le \mathtt{l} < -1 \\ 0, & \text{otherwise} \end{cases} \tag{5.24}$$

To compute min entropy-based leakage, we need to find x, x_3, x_0, x_{-3} (see Theorem 5.3). We have $x = 1$ because $Sat(K^{\emptyset}(\mathtt{h}) \wedge \varphi_1(\mathtt{h})) = Sat(-2^{31} \le \mathtt{h} < 0) \neq \emptyset$. The value of x_3, x_0, x_{-3}

depends on the value of $\mathtt{l}$. Assume that 10 is the input value of $\mathtt{l}$, we have $x_3 = x_0 = 2, x_{-3} = 1$ because two formulas $-2^{31} \leq \mathtt{h} < 0 \wedge \mathtt{h} = 10$ and $-2^{31} \leq \mathtt{h} < 0 \wedge \mathtt{h} > 10$ are unsatisfiable while the set $Sat(-2^{31} \leq \mathtt{h} < 0 \wedge \mathtt{h} < -10)$ is not empty. The min entropy-based leakage for the case $\mathtt{l} = 10$ is $\mathtt{MEL}_{\mathtt{rPC}}(10) = \mathtt{log}(\frac{1+1+2}{2}) = 1$. Choosing another input value for $\mathtt{l}$, i.e. -1, might result in a different leakage value: we have $x_3 = x_{-3} = 1, x_0 = 2$ because $-2^{31} \leq \mathtt{h} < 0 \wedge \mathtt{h} > -1$ is unsatisfiable (recall that $\mathtt{h}$ is integer) while $-2^{31} \leq \mathtt{h} < 0 \wedge \mathtt{h} = -1$ can be satisfied. Thus we have the min entropy-based leakage when $\mathtt{l} = -1$ is $\mathtt{MEL}_{\mathtt{rPC}}(-1) = \mathtt{log}(\frac{2+1+2}{2}) = \mathtt{log}(2.5)$ that is larger than $\mathtt{MEL}_{\mathtt{rPC}}(10)$.

5.2 Finding Low Input Maximizing Leakage

5.2.1 Leakage Computed using Parametric Counting

Finding optimal low input is equivalent to the optimization problem $argmax_{\bar{l} \in \mathbb{L}} QLeak(\bar{l})$ in which $QLeak(L)$ can be $\mathtt{ShEL}_{\mathtt{p}}(L)$, $\mathtt{GEL}_{\mathtt{p}}(L)$ or $\mathtt{MEL}_{\mathtt{p}}(L)$. If the leakage can be represented by means of a parametric counting function of low input as shown in Theorems 5.1, 5.2, and 5.4, the low input maximizing such leakage can be determined by solving an optimization problem. The objective function of this problem is derived directly from the function calculating the corresponding leakage, and the constraints of this problem are synthesized from the conditions of parametric counting results. By definition, the values of $S_E^{\Delta_H}$ and S_E are constant in the sense that they depend only on the current knowledge $K^E(H)$ and the distribution Δ_H while they do not depend on low input. Hence, in cases of Shannon or guessing entropy, maximizing the leakage can be reduced to minimizing the sum expressions in their corresponding functions.

The result of parametric counting, as shown by the examples in Section 5.1.4, is usually a collection of ordered pairs of a counting result and its corresponding condition. Hence, solving the optimization problem $argmax_{\bar{l} \in \mathbb{L}} QLeak(\bar{l})$ might require to consider a number of aggregated conditions, each of which corresponds to an unique objective function. To be more specific, let's assume that to compute $QLeak(L)$, we need to consider a set of parametric counting function $\{\mathtt{C}_H[L](\mathtt{cf}_i(L,H)) | i = 1 \cdots ncf\}$. For the case of Shannon or guessing entropy, the number of parametric counting formulas ncf is $|\mathbb{O}_D(L)| \times |FO_H|$, while there is only one parametric counting formula to quantify min entropy-based leakage provided that Δ_H is uniform (see Theorem 5.4). We assume that, counting the number of H satisfying $\mathtt{cf}_i(L,H)$ returns a set of pairs $\{\langle \mathtt{cr}_j^i(L), \mathtt{cc}_j^i(L)\rangle | j = 1 \cdots n_i\}$, in which the function $\mathtt{cr}_j^i(L)$ is the counting result for the case L satisfies condition $\mathtt{cc}_j^i(L)$. Hence, the set $\{\mathtt{C}_H[L](\mathtt{cf}_i(L,H)) | i = 1 \cdots ncf\}$ can be determined by a list of indices $\{d_1, \cdots, d_{ncf}\}$ so that $\mathtt{C}_H[L](\mathtt{cf}_i(L,H)) = \mathtt{cr}_{d_i}^i(L)$ and the corresponding aggregated condition is $\bigwedge_{i=1}^{ncf} \mathtt{cc}_{d_i}^i(L)$. If a condition $\bigwedge_{i=1}^{ncf} \mathtt{cc}_{d_i}^i(L)$ is satisfiable, the list $\{d_1, \cdots, d_{ncf}\}$ is called a *candidate indexing list*. Let S be the set of all possible candidate indexing lists, the optimal value of L is found by iterating through S. The process of finding optimal low input that uses parametric counting is shown in Algorithm 5.1.

To find all possible candidate indexing lists, a simple backtracking algorithm can be used. In a nutshell, we start with $i = 1, k = 1$, if $\mathtt{cc}_k^i(L) \wedge \bigwedge_{j=1}^{i-1} \mathtt{cc}_{d_j}^j(L)$ is satisfiable, we assign d_i by k and go further with $i + 1$, otherwise we increase the value of k by 1. If we cannot assign d_i by $k \in [1, n_i]$, we go back to choose another value for d_{i-1} and start over the process. Whenever

Data: Ordered set of parametric counting formulas $\{\mathtt{cf}_i(L,H)|i=1\cdots ncf\}$ with corresponding counting results $\{\langle \mathtt{cr}^i_j(L), \mathtt{cc}^i_j(L)\rangle | j=1\cdots n_i\}$, *metric*: leakage metric (Shannon, guessing, min entropy)
Result: $(\bar{l}, leakage)$: optimal low input value and corresponding leakage

```
1  begin
2      S ← find all possible candidate indexing lists;
3      l̄ ← random value;
4      leakage ← 0;
5      foreach {d_1, ··· , d_ncf} ∈ S do
6          Obj(L) ← createObjectiveFunction({cr^1_{d_1}(L), ··· , cr^ncf_{d_ncf}(L)}, metric);
7          Constraint(L) ← ⋀_{i=1}^{ncf} cc^i_{d_i}(L);
8          l̄_opt ← solve optimization problem (Obj(L), Constraint(L));
9          leakage_tmp ← computeLeakage(l̄_opt, metric);
10         if leakage < leakage_tmp then
11             l̄ ← l̄_opt;
12             leakage ← leakage_tmp;
13         end
14     end
15     return (l̄, leakage);
16 end
```

Algorithm 5.1: Finding low input maximizing leakage using parametric counting

d_{ncf} is determined, a candidate indexing list has been found and we continue to find another one until all possible combinations are explored.

Whenever S is found, the process of finding optimal low input and corresponding maximum leakage begins. The maximum leakage value is instantiated with 0 and low input is generated randomly. For each candidate indexing list in S, an optimization problem is constructed using this list. Solving this problem returns a value of L that is used to compute the corresponding leakage. Current low input value and maximum leakage are assigned by the new found values if the current maximum leakage is smaller than one computed using the new low input. Algorithm 5.1 guarantees that the generated low input is the optimal value in the sense that it maximizes the leakage computed using corresponding metric.

Example 5.3. *Consider program* `rPC` *in Listing 4.1 with noninterference policy* $\mathtt{h} \not\rightarrow \mathtt{l}$ *(*$\mathtt{l}, \mathtt{h}$ *are integer), we aim to find the value of low input that maximizes the information leak throughout the execution of* `rPC`*. A closer inspection of the program reveals the following: as long as our only knowledge about* $\mathtt{h}$ *is that its value is within an interval* $[a,b]$ *and all values in the range* $[a,b]$ *have the same likelihood (uniform distribution), then choosing the arithmetic middle* $\frac{b+a}{2}$ *for the input value of* $\mathtt{l}$ *is the best choice. The initial knowledge about* $\mathtt{h}$ *is that its value is between* -2^{31} *and* $2^{31}-1$*, hence, the best choice is* 0 *or* -1*. We show that the solution computed* automatically *by our algorithm comes close to this inspection.*

Section 5.1.4 gives details about the computation of information flow leak in program `rPC`*. For the case of uniform distribution, the Shannon, guessing and min entropy-based leakage is computed by equations* (5.20)*,* (5.21) *and* (5.22) *respectively. From* (5.22) *we see that the maximum leak-*

age measured by the min entropy-based metric is $\log(3)$ *for all values of low input in the range* $(-2^{31}, 100)$. *This restricts the choice of a suitable value for* l *only slightly. Computation of the maximal leakage for the Shannon and guessing entropy-based metrics requires more effort. To find* l *maximizing* $\mathrm{ShEL}_{\mathrm{rPC}}(l)$ *we have to solve an optimization problem:*

$$\begin{array}{ll} \underset{l}{\textit{minimize}} & (2^{31}-1-l)\log(2^{31}-1-l)+(2^{31}+l)\log(2^{31}+l) \\ \textit{subject to} & -2^{31} \leq l < 100 \end{array}$$

and for $\mathrm{GEL}_{rPC}(l)$ *the problem:*

$$\begin{array}{ll} \underset{l}{\textit{minimize}} & (2^{31}-1-l)(2^{31}-l)+(2^{31}+l)(2^{31}+1+l) \\ \textit{subject to} & -2^{31} \leq l < 100 \end{array}$$

Using the optimizers Bonmin[1] *and* Couenne[2] *with default settings, we get as result* $l = 0$ *for the second problem, which meets our intuition. However, these optimizers only return an approximate result* $l = 3$ *for the first problem. Nevertheless, the Shannon entropy-based leakages computed with* $l = 0$ *and* $l = 3$ *are very close and approximately* 1, *i.e.* 1 *bit of* h *is revealed. For this program, the Shannon and guessing entropy based-metric perform significantly better than the min entropy-based metric. In both cases their successive application generates a series of experiments that performs binary search to uncover the secret.*

Now we consider the case that the prior distribution of h *is non-uniform. We continue with the distribution given in Section 5.1.4 defining that the probability that* h *has negative value is twice as high as the non-negative case. This distribution brings an intuition that we should choose a negative value for* l. *We consider the case Shannon entropy is used to measure the leakage. From* (5.23), *we have* $\mathrm{ShEL}_{\mathrm{rPC}}(0) \approx 0.9182958391$ *and* $\mathrm{ShEL}_{\mathrm{rPC}}(-1) \approx 0.9182958444$. *It is clear that* $\mathrm{ShEL}_{\mathrm{rPC}}(-1) > \mathrm{ShEL}_{\mathrm{rPC}}(0)$. *To find the optimal value of* l, *we still need to solve the two following optimization problems:*

$$\begin{array}{ll} \underset{l}{\textit{minimize}} & (2^{31}-1-l)\log(2^{31}-1-l)+(l+2^{32})\log(l+2^{32}) \\ \textit{subject to} & 0 < l < 100 \end{array}$$

and

$$\begin{array}{ll} \underset{l}{\textit{minimize}} & (2^{31}-2(l+1))\log(2^{31}-2(l+1))+2(l+2^{31})\log(2(l+2^{31})) \\ \textit{subject to} & -2^{31} \leq l < -1 \end{array}$$

then compute the leakages using values returned by the optimizer and compare them (also to $\mathrm{ShEL}_{\mathrm{rPC}}(-1)$*) to decide what value is the best choice. The solution of the first problem is* $l = 1$ *with corresponding leakage* $\mathrm{ShEL}_{\mathrm{rPC}}(1) \approx 0.9182958389$. *For the second problem, the optimizer* Bonmin *yields the solution* $l = -536871000$ *with corresponding leakage* $\mathrm{ShEL}_{\mathrm{rPC}}(-536871000) \approx 1.0000000099$. *Hence the optimal value of* l *is* -536871000 *that meets our intuition. The computation for the case of guessing entropy-based leakage using optimizer* Bonmin *brings the same result that the optimal value of* l *is* -536871000.

[1] `http://www.coin-or.org/Bonmin`

[2] `https://projects.coin-or.org/Couenne`

5.2.2 Max-SMT Approach for Min Entropy-Based Leakage

If the prior distribution of high inputs is uniform, the parametric counting approach can be used to find low inputs maximizing min entropy-based leakage, thanks to Theorem 5.4. The general computation of min entropy-based leakage for an arbitrary prior distribution of high input is given in Theorem 5.3. Take a closer look into equation (5.15), we can see that $\mu(\varphi_x(H))$ depends only on the experiment set E and the formula-based distribution Δ_H. Intuitively, $\mu(\varphi_x(H))$ is the highest frequency value of all high values satisfying the current knowledge $K^E(H)$.

To find $\mu(\varphi_x(H))$, we simply iterate over all $\varphi_i(H) \in FO_H$ and choose among satisfiable formulas $K^E(H) \wedge \varphi_i(H)$ $(i = 1, \ldots, n)$ the formula $K^E(H) \wedge \varphi_x(H)$ where the frequency value $\mu(\varphi_x(H))$ is highest.

Because $\mu(\varphi_x(H))$ does not depend on L, to maximize $\mathtt{MEL}_{\mathrm{p}}(L)$ we only need to find $\bar{l}$ so that $\sum_{\overline{o} \in \mathbb{O}_D(\bar{l})} \mu(\varphi_{x_{\overline{o}}}(H))$ maximizes. We introduce a Max-SMT approach to solve this problem.

The Max-SMT problem [95] is an extension of the SMT problem that concentrates on optimization. Generally, a Max-SMT problem can be defined as follows: given a set of pairs $\{(C_i, w_i) | i = 1, \cdots, m)\}$ where each C_i is a formula and w_i is a positive number that is the *weight* of C_i, find an assignment A that maximizes (or equivalently minimizes) the sum of the weights of all true clauses in A. Max-SMT problems can be solved by a number of Max-SMT solvers such as Z3 [41], Yices [50], CVC4 [14], VeryMax [27].

According to Theorem 5.3, for each low input $\bar{l}$, to compute min entropy-based leakage $\mathtt{MEL}_{\mathrm{p}}(\bar{l})$, we have to iterate over all possible output values (the set $\mathbb{O}_D(\bar{l})$). For each $\overline{o} \in \mathbb{O}_D(\bar{l})$, we have to find the probability partition $x_{\overline{o}}$ where $\mu(\varphi_{x_{\overline{o}}}(H))$ is the probability of the most likely input value of H that satisfies the current knowledge $K^E(H)$. The set $\{x_{\overline{o}} | \overline{o} \in \mathbb{O}_D(\bar{l})\}$ is determined only by $\bar{l}$ and we denote it by $X^{\bar{l}}$. Finding $\bar{l}$ maximizing the leakage is equivalent to choosing $\bar{l}$ to maximize the sum of weights $\sum_{i \in X^{\bar{l}}} \mu(\varphi_i(H))$ under the condition that all formulas of the set $\{\mathrm{g}_i(\bar{l}, H, \overline{o}) | i \in X^{\bar{l}}\}$ are satisfiable. This inspires a Max-SMT approach to find the optimal value of L. The basic idea is to encode the optimization problem $argmax_{\bar{l} \in \mathbb{L}}(\mathtt{MEL}_{\mathrm{p}}(\bar{l}))$ as a Max-SMT problem using the weights of partition formulas as the weights of corresponding SMT clauses and then, let the Max-SMT solver find the value of L that maximizes the sum of weights.

Algorithm 5.2 lays out the approach using a Max-SMT solver to find the optimal low input and the corresponding maximum min entropy-based leakage. For each observable output value $\overline{o}_j$ and each partition formula $\varphi_i(H) \in FO_H$, a Max-SMT clause $\mathrm{g}_i(L, H_j, \overline{o}_j)$ with the weight $\mu(\varphi_i(H))$ is built. Here H_j is a set of fresh variables obtained by renaming all variables in H, e.g. $H_j = \{v_j | v \in H\}$. Afterwards, the generated Max-SMT problem is passed to a Max-SMT solver to find the solution. That solver returns an assignment A for $L, H_1, \cdots, H_m$. The value of L given by A, denoted by $\bar{l}$, is the optimal value of L in the sense that $\mathtt{MEL}_{\mathrm{p}}(\bar{l})$ is the maximum min entropy-based leakage. Along with A, the Max-SMT solver also returns a set D_A of true clauses in A. After iterating over all clauses in D_A to compute the sum of the weights SW_A, the min entropy-based leakage is computed as the logarithm based 2 of the fraction $\frac{SW_A}{\mu(\varphi_x(H))}$.

Theorem 5.5. *Algorithm 5.2 yields the optimal low input and the maximum min entropy-based leakage of program p w.r.t. formula-based distribution* Δ_H.

Proof. Because the assignment A is found by a Max-SMT solver, the sum of the weights of all true clauses in A must be a maximum, which means that there exists no assignment B satisfying

Data: Set of performed experiments E, set of reachable paths R^E, knowledge of secret $K^E(H)$, formula-based distribution of secret $\Delta_H = \langle FO_H, \mu \rangle$ with $|FO_H| = n$, set of possible observable output $\mathbb{O} = \{\overline{o}_1, \overline{o}_2, .., \overline{o}_m\}$, $\mu(\varphi_x(H))$: the highest frequency value of all high input satisfying $K^E(H)$

Result: $(\overline{l}, leakage)$: optimal low input value and corresponding min entropy-based leakage

```
1  begin
2      foreach φ_i(H) ∈ FO_H do
3          foreach ō_j ∈ 𝕆 do
4              Build Max-SMT clause with weight μ(φ_i(H)): C_ij :: g_i(L, H_j, ō_j);
5          end
6      end
7      Solve {(C_ij, μ(φ_i(H)))|1 ≤ i ≤ n, 1 ≤ j ≤ m} by Max-SMT solver;
8      A ← assignment found by Max-SMT solver;
9      l̄ ← the value assigned for L given by A;
10     D_A ← set of all true clauses in A;
11     SW_A ← 0;
12     foreach C_ij ∈ D_A do
13         SW_A ← SW_A + μ(φ_i(H));
14     end
15     leakage ← log(SW_A / μ(φ_x(H)));
16     return (l̄, leakage);
17 end
```

Algorithm 5.2: Finding low input maximizing leakage using Max-SMT solver

that $SW_B > SW_A$. Recall that $\overline{l}$ is the value of L given by A, we denote the value of $H_1, \ldots, H_m$ defined in A by $\overline{h}_1, \ldots, \overline{h}_m$, respectively. To prove Theorem 5.5, we will prove that the leakage computed by Algorithm 5.2 is $\mathtt{MEL}_{\mathtt{p}}(\overline{l})$ (i) and it is actually the maximum min entropy-based leakage (ii).

To prove (i), we will prove that $SW_A = \sum_{\overline{o} \in \mathbb{O}_D(\overline{l})} \mu(\varphi_{x_{\overline{o}}}(H))$. Consider an arbitrary value $\overline{o}_j \in \mathbb{O}$, we will prove that if $\overline{o}_j \in \mathbb{O}_D(\overline{l})$ then there exists one and only one i that C_{ij} is true in A, otherwise C_{ij} is false for all $i \in [1, n]$. Assume that $\forall i \in [1, n]$. $\mathrm{g}_i(\overline{l}, \overline{h}_j, \overline{o}_j) = false$, because $\overline{o}_j \in \mathbb{O}_D(\overline{l})$, then there exists $\overline{h} \in |Sat(K^E(H))|$ such that program p runs with input values $\overline{l}$ and $\overline{h}$ of L and H respectively and returns $\overline{o}_j$ as output value of O. Because $\overline{l} \in Sat(InRisk(L))$ we have $\mathrm{g}_k(\overline{l}, \overline{h}, \overline{o}_j) = true$ where k satisfies $\varphi_k(\overline{h}) = true$. Consider assignment A_1 that is identical to A except that the value of H_j is $\overline{h}$ not $\overline{h}_j$, we can see that all clauses that are true in A also hold in A_1, furthermore C_{kj} is true in A_1 not in A. Hence we have $SW_{A_1} = SW_A + \mu(\varphi_k(H)) > SW_A$, which means that SW_A is not maximal (contradiction). Hence, there exists such i that C_{ij} is true in A if $\overline{o}_j \in \mathbb{O}_D(\overline{l})$. Assume that $\exists k \neq i$ satisfying that C_{kj} is also true in A, we have $\varphi_i(\overline{h}_j) = true$ and $\varphi_k(\overline{h}_j) = true$ leading to $Sat(\varphi_i(H)) \cap Sat(\varphi_k(H)) \neq \emptyset$ that contradicts Definition 5.2. Hence, we have if $\overline{o}_j \in \mathbb{O}_D(\overline{l})$ then there exists one and only one true clause C_{ij}. In case of $\overline{o}_j \notin \mathbb{O}_D(\overline{l})$, the formula $\mathrm{g}(\overline{l}, H, \overline{o}_j)$ is unsatisfiable, hence C_{ij} is false for all $i \in [1, n]$. Consequently, (i) will be proven if we can prove that: $\forall C_{ij} \in D_A.\ i = x_{\overline{o}_j}$ ($x_{\overline{o}_j}$ is defined as in Theorem 5.3).

Consider an arbitrary clause $C_{ij} \in D_A$, because $g_i(\bar{l}, \bar{h}_j, \bar{o}_j) = true$, we have $Sat(g_i(\bar{l}, H, \bar{o}_j)) \neq \emptyset$. Now proving (i) is equivalent to proving that $\nexists k$. $Sat(g_k(\bar{l}, H, \bar{o}_j)) \neq \emptyset \wedge \mu(\varphi_k(H)) > \mu(\varphi_i(H))$. Let's assume that such k exists, because $g_k(\bar{l}, H, \bar{o}_j)$ is satisfiable, $g_k(\bar{l}, H_j, \bar{o}_j)$ can also be satisfied by a value $\bar{h}_j'$ of H_j. Consider assignment A_2 that is identical to A except that the value of H_j is $\bar{h}_j'$ not $\bar{h}_j$. Analog to above, we have all clauses true in A are also true in A_2 and vice versa, except that $C_{ij} \in D_A \wedge C_{ij} \notin D_{A_2} \wedge C_{kj} \in D_{A_2} \wedge C_{kj} \notin D_A$. Therefore, $SW_{A_2} = SW_A + \mu(\varphi_k(H)) - \mu(\varphi_i(H))$. Because $\mu(\varphi_k(H)) > \mu(\varphi_i(H))$, we have $SW_{A_2} > SW_A$ (contradiction), hence (i) is proven.

Now we prove (ii). Assume that $\exists \bar{l}' \neq \bar{l}$ satisfying that $\texttt{MEL}_\texttt{p}(\bar{l}') > \texttt{MEL}_\texttt{p}(\bar{l})$, for each $\bar{o}_j \in \mathbb{O}_D(\bar{l}')$, we have $Sat(g_{x_{\bar{o}_j}}(\bar{l}', H, \bar{o}_j)) \neq \emptyset$ where $x_{\bar{o}_i}$ is defined as in Theorem 5.3. Let $\bar{h}_j' \in \mathbb{H}$ be a value of H such that $g_{x_{\bar{o}_j}}(\bar{l}', \bar{h}_i', \bar{o}_j) = true$. Consider assignment A' in which L is assigned $\bar{l}'$ and H_j is assigned $\bar{h}_j'$ if $\bar{o}_j \in \mathbb{O}_D(\bar{l}')$ or a random value otherwise. As above we have $\texttt{MEL}_\texttt{p}(\bar{l}') = \frac{SW_{A'}}{\mu(\varphi_x(H))}$, thus from $\texttt{MEL}_\texttt{p}(\bar{l}') > \texttt{MEL}_\texttt{p}(\bar{l})$ we have the result $SW_{A'} > SW_A$ (contradiction). Hence, (ii) is proven that brings the correctness of Theorem 5.5. □

Example 5.4. *We continue the example in Section 5.1.4 to find an input value of* `l` *maximizing min entropy-based leakage of program* `rPC` *(Listing 4.1), given that the prior distribution of* `h` *is the same as the non-uniform distribution defined in Section 5.1.4 as follows:*

$$FO_h = \{\varphi_1(h), \varphi_2(h)\}$$
$$\varphi_1(h) = h < 0 \qquad \mu(\varphi_1(h)) = 2$$
$$\varphi_2(h) = h \geq 0 \qquad \mu(\varphi_2(h)) = 1$$

We also have the set of observable output values $\mathbb{O}_{\{l\}} \subseteq \{3, 0, -3\}$. *Using Algorithm 5.2 we obtain the following Max-SMT clauses:*

$$C_{11} = g_1(l, h_1, 3) = -2^{31} \leq h_1 < 2^{31} \wedge l < 100 \wedge h_1 \doteq l \wedge h_1 < 0 \ : weight = 2$$
$$C_{12} = g_1(l, h_2, 0) = -2^{31} \leq h_2 < 2^{31} \wedge l < 100 \wedge h_2 > l \wedge h_2 < 0 \ : weight = 2$$
$$C_{13} = g_1(l, h_3, -3) = -2^{31} \leq h_3 < 2^{31} \wedge l < 100 \wedge h_3 < l \wedge h_3 < 0 \ : weight = 2$$
$$C_{21} = g_2(l, h_1, 3) = -2^{31} \leq h_1 < 2^{31} \wedge l < 100 \wedge h_1 \doteq l \wedge h_1 \geq 0 \ : weight = 1$$
$$C_{22} = g_2(l, h_2, 0) = -2^{31} \leq h_2 < 2^{31} \wedge l < 100 \wedge h_2 > l \wedge h_2 \geq 0 \ : weight = 1$$
$$C_{23} = g_2(l, h_3, -3) = -2^{31} \leq h_3 < 2^{31} \wedge l < 100 \wedge h_3 < l \wedge h_3 \geq 0 \ : weight = 1$$

Look at the above Max-SMT problem, we can see that the maximum sum of the weights of true clauses is 6 *and can only be obtained with an assignment in which* C_{11}, C_{12}, C_{13} *are true. Hence the value of* `l` *must belong to the range* $(-2^{31}, -1)$. *A Max-SMT Solver (here we use Z3) returns a concrete model with* $l = -2147450800$. *This model maximizes the sum of weight values computed via three true clauses* C_{11}, C_{12}, C_{13}. *The maximum min entropy-based leakage of program* `rPC` *therefore is* $\texttt{log}(\frac{2+2+2}{2}) = \texttt{log}(3)$.

5.3 Discussion

In the two previous sections of this chapter, we have introduced an approach quantifying entropy-based leakages and finding optimal low inputs. The proposed approach is precise

w.r.t. a correct and complete symbolic execution tree. However, it has some drawbacks that might limit the applicability for real world programs. We discuss in this section such limitations, point out applicable conditions and show some directions to overcome those drawbacks.

5.3.1 The Set $\mathbb{O}_D(L)$

The computation of entropy-based leakages requires to enumerate all possible output values of O, except for the case that the security metric is based on min entropy and the secret's distribution is uniform.

Determining all possible output values of O is a tough task that might be very expensive. A quick estimation can be made by using symbolic execution. The basic idea is deriving the set $\mathbb{O}_D(L)$ from the set of symbolic observable output values $\overline{f_i^O}$. If they (i) either depend only on the chosen SE path, but not on the actual values of the low or high variables (i.e. each SE path assigns only constant values to the observable variables), (ii) or the output values depend only on the low input (i.e. for a specific concrete low input, their concrete value can be determined by evaluating the corresponding symbolic value f), then determining $\mathbb{O}_D(L)$ is significantly cheaper, because the cardinality of the set of possible observable outputs is bounded by the number of reachable paths.

Section 5.1.3 demonstrates how the set $\mathbb{O}_D(L)$ is determined and how formula g is simplified for case (i). We illustrate here the case (ii). For the sake of simplicity, we assume that the set O is a singleton, i.e. $O = \{o\}(o \in L)$, the extension for the case $|O| > 1$ is straightforward.

Consider the set of reachable paths R^E, assume that there exists a path $j \in R^E$ satisfying that output value of o given at path x depends on and only on low input, while other reachable paths yield constant output values of o. Because the symbolic value f_i^o $(i \neq x)$ is constant, it can be evaluated to a concrete value, denoted by o_i. By assumption, f_x^o depends only on L, it can be represented by a function of L, denoted as $f_x(L)$. Without loss of generality, we assume that $o_i \neq o_j$ for all different reachable paths i and j $(i \neq x \wedge j \neq x)$. Let $\bar{l}$ be an arbitrary, concrete value of L, the set of observable output values $\mathbb{O}_D(\bar{l})$ becomes either $\{o_i | i \in R^E \wedge i \neq x\}$ if there exists a path $j \in R^E$ and $j \neq x$ such that $o_j = f_x(\bar{l})$, or $\{o_i | i \in R^E \wedge i \neq x\} \cup \{f_x(\bar{l})\}$ otherwise. To find an optimal low input, the leakage needs to be computed w.r.t. both cases and the condition must be incorporated into formula $g(L, H, \{o\}')$. Consider the first case, for each concrete ouput o_i $(i \neq x)$, formula g in Corollary 5.1 becomes

$$g(L, H, \{o_i\}) \;=\; K^E(H) \wedge InRisk(L) \wedge \big(pc_i \vee (pc_x \wedge o_i \doteq f_x(L))\big)$$

Formula g is simplified in the second case as following:

$$g(L, H, \{o_i'\}) \;=\; K^E(H) \wedge InRisk(L) \wedge pc_i \wedge \bigwedge_{i \in R^E \setminus \{x\}} o_i \neq f_x(L)$$

where $i \in R^E$ and $o_i' = o_i$ if $i \neq x$ and $o_i' = f_x(L)$ if $i = x$.

If the program satisfies (i) or (ii), computation of entropy-based leakages might also be considerably cheaper, because not only the number of observable outputs is bounded by the number of reachable paths, but also formulas g and g_i that are supplied to the parametric counting tool are much simpler. However, if the program does not fall into either (i) or (ii), finding all elements of the set $\mathbb{O}_D(L)$ might be very expensive and even infeasible in practice. In this case,

using Shannon entropy or guessing entropy as security metric might become inapplicable. The same for min entropy if the high input has non-uniform probability distribution. However, if this distribution is uniform, computing min entropy-based leakage does not suffer from the same problem, because it merely requires to estimate the *cardinality* of the observable output values.

5.3.2 Parametric Counting

Another limitation of our approach comes from parametric model counting, that computes the number of assignments to the set of integer variables X that satisfy formula $g(X,Y)$ in the form of a function of Y, denoted by $\mathtt{C}_X[Y](g(X,Y))$. Generally, this problem is intractable, however if formula $g(X,Y)$ fulfills some special conditions then it becomes feasible. One of them is that if $g(X,Y)$ is an integer linear arithmetic formula, then computing $\mathtt{C}_X[Y](g(X,Y))$ can be reduced to counting the number of integer points in parametric and non-parametric polytopes for which efficient approaches (and tools) exist [116].

The limitation of parametric counting approach requires that formulas $g(L,H,O')$ and $g_i(L,H,O')$ are linear. This can only be fulfilled if all of the path conditions pc_i, the symbolic observable output values $\overline{f}_i^O$ and the partition formulas $\varphi_j(H)$ are also linear. Even for a linear formula, if it contains many disjuncts, then the counting tool may return many different results along with respective conditions. This increases computational overhead significantly. Such limitation, indeed, restricts the scalability of our approach. To overcome this problem, approximation techniques that accept "good enough" low input instead of optimal one need to be devised.

On the other hand, The Max-SMT based approach for the case of min entropy does not suffer from the same problem with parametric counting, because many Max-SMT solvers support nonlinear arithmetic. Moreover, not only integer but also other types, such as String and arrays, can be supported by Max-SMT solvers. This equips Max-SMT approach the ability to work with a wider class of programs.

5.3.3 Optimization Tool

Even if the leakage can be computed by using parametric counting, it might be difficult to find optimal low input. The reason is that the parametric counting-based approach requires to solve a number of non-linear optimization problems. Those problems can be too expensive to solve, even by the most powerful optimization tool. Moreover, most non-linear optimizers use a combination of several heuristic and approximation techniques that could bring not an actual optimal low input value but an approximate one (that is shown in Example 5.3). Although this approximation is "good enough" in many cases, in some other cases it might fail to help the attacker learn the secret via corresponding experiment efficiently.

One naive solution to deal with the drawback of approximate optimization results is to use a combination of several different optimizers instead of a single one. The idea is that different tools might be better for different problems. Hence, in the implementation, we use several tools to solve a single problem and we recompute the values of objective functions with the low inputs returned by them and choose the best one. Although this solution generally generates a "better" low input than using only one optimizer, it obviously reduces the performance. It could

be relaxed by using a heuristic approach to decide what optimizer should be used to solve a specific problem. This is out of the scope of this thesis and is left for future work.

6 Implementation and Experiments

Chapter 3, Chapter 4 and Chapter 5 of this thesis have proposed a number of approaches to detect, demonstrate and exploit information flow leaks in programs. Those approaches have been fully implemented in a prototype tool, namely *KeY Exploit Generation*[1] (KEG) . This chapter describes the KEG tool and evaluates KEG using a collection of micro benchmarks. The main architecture, work-flow and usage of KEG are given in Section 6.1. Section 6.2 illustrates the work-flow of KEG with a focus on the leak demonstrator and the secret inference program. Section 6.3 presents the evaluation results. Parts of this chapter are based on previous formal publications [47, 48] and a technical report [46] of the author of this thesis.

6.1 The KEG Tool

6.1.1 Architecture

KEG is implemented on top of the KeY system [1]. KEG can be used to detect information flow leaks in sequential Java programs, to generate demonstrators for detected leaks, and to exploit information flow leaks to infer program's secrets. Its top-level architecture is shown in Figure 6.1.

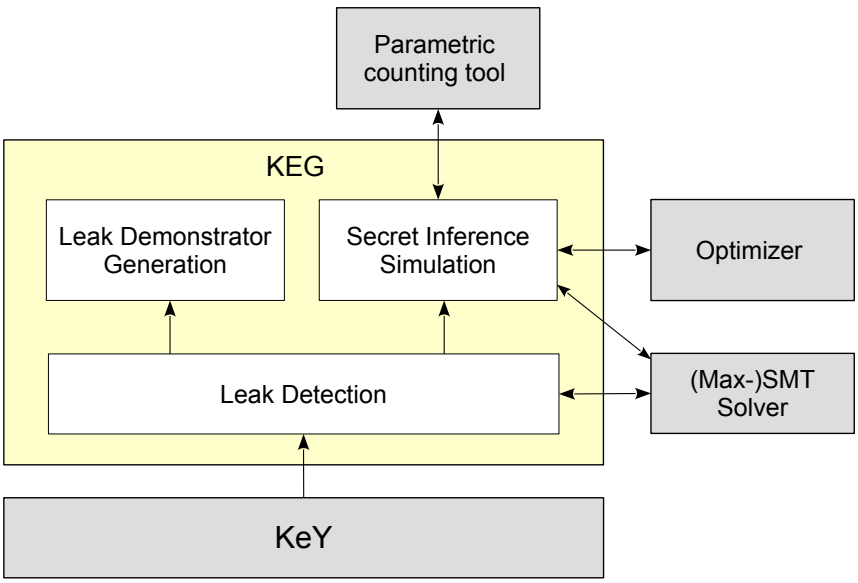


Figure 6.1: Top-level architecture of KEG

KEG comprises three main modules implementing its main features. Information flow leaks of insecure Java programs, i.e. programs that do not adhere to a specified information flow policy, are discovered within module *Leak Detection*. Information flow leaks are detected by using an

[1] www.se.tu-darmstadt.de/research/projects/albia/download/exploit-generation-tool

SMT solver to check *insecurity formulas* characterizing the violation of a given information flow policy. Such formulas are composed using the symbolic execution tree (SET) of a program. The SET is generated by the symbolic execution engine of KeY. Concrete models for satisfying insecurity formulas are found by an SMT solver. These models are then used to generate *leak demonstrators* (KEG module *Leak Demonstrator Generation*) as JUnit test cases for insecure Java programs. The leak detection also indicates a set of symbolic *risky paths* (path might contribute to a leak) that can be exploited by module *Secret Inference Simulation*.

Module *Secret Inference Simulation* conducts simulated attacks to demonstrate how detected leaks can be exploited to guess the secret (the input values of high variables). A simulation basically runs the insecure program multiple times using low input values generated automatically by KEG and (secret) high input values. Those runs, so called *experiments*, build up a logic characterization of the secret as the attacker's *knowledge*. The set of *risky paths* is used to generate low inputs for the program runs. Those values are found with the help of *optimizers* or *Max-SMT solvers*. KEG employs a *parametric counting tool* to quantify the leakage. The counting tool can also be used to calculate the number of possible high inputs that satisfy the knowledge synthesized from experiments.

6.1.2 Workflow

Figure 6.2 outlines KEG's workflow. The input is a Java program annotated with information flow policies and necessary JML specifications. KEG checks each method `m` of the input program with class-level noninterference policies and method-level declassification policies (if they are specified) to which `m` has to adhere. Possibly, `m` is annotated with loop specifications. Contracts of methods called by `m` are provided by the user if they are not to be inlined during symbolic execution. KEG expects the information flow policy to be present in the source code inside specifically marked Java comments. For each method `m` to be checked, first, `m` is symbolically executed (using KeY) to obtain a complete symbolic execution tree which can then be queried for the method's path conditions and the final symbolic values of the program locations modified by `m`. Second, using the path conditions and symbolic final values, instances of the insecurity formulas described in Sections 3.3 and 3.4 are generated. In a third step, these formulas are passed to a model finder (currently we use the SMT solver Z3 [41]). If a model for the insecurity formula has been found, it is used to determine the initial states of two runs which exhibit a forbidden information flow. KEG outputs the leak demonstrator as a self-contained JUnit test, which can then be included into a regression test suite. The generated leak demonstrator executes two runs (one for each initial state) and inspects the final states to detect a leak in the form of an assertion. In this way the conjuncts in the insecurity formulas that contain the inequality over symbolic final values can be viewed as an automatically synthesized test oracle.

An on-demand secret inference simulation is launched after the leak detection finishes. It simulates an adaptive attack where the attacker actively choses the low inputs and runs program multiple times to guess the values of high inputs (the secret). KEG generates an attack simulation setup that includes: an attack program conducting *experiments* on method `m` (it sets the initial state, runs `m` and prints out the outputs); two input files storing high input values and low input values; a file to specify the prior probability distribution of secret. The initial knowledge of the attacker about the secret is specified in the precondition of `m`. High input values are chosen by the user before starting simulation. During the simulation, those values are unchanged. Besides high input values, the user can also define the high input distribution,

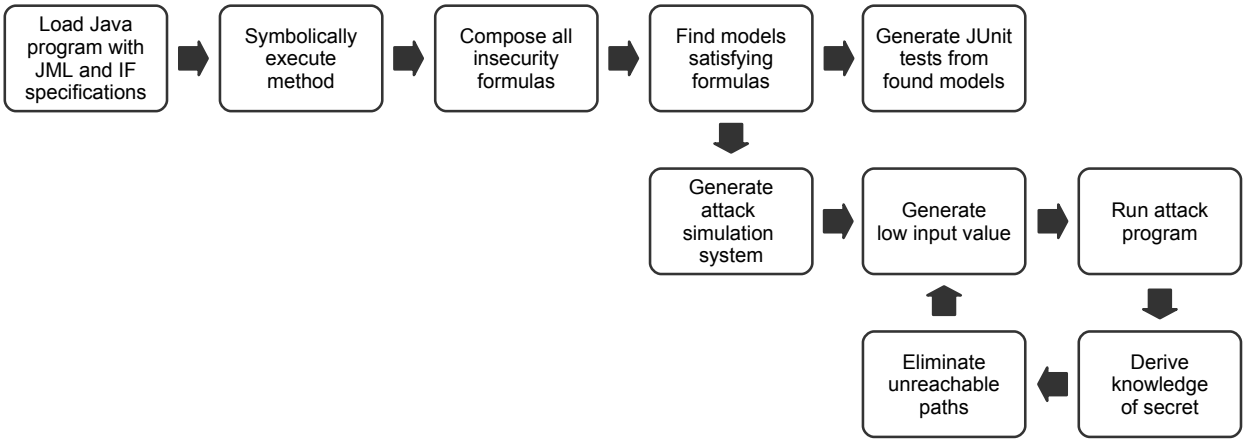


Figure 6.2: The workflow of KEG

decide the security metric to be used and fix the maximum number of experiments to be carried out.

As soon as the attack simulation's fixtures are set, KEG generates optimal low input values maximizing the leakage measured by the chosen security metric. Then the attack program executes method `m` using high and low inputs given in corresponding input files. Using observed outputs, KEG derives a new knowledge about the secret. Furthermore, all unreachable paths with respect to the new knowledge are found and marked so that they can be ignored in subsequent steps. The simulation halts and the result is reported if one of the followings happens: the number of experiments exceeds the limit, the whole secret is inferred, or all low inputs lead to a redundant experiment. If the secret inference simulation finishes, KEG with the help of an SMT solver tries to provide all possible values of secrets satisfying achieved knowledge. Otherwise, new low input values for the next experiment are generated using updated knowledge about the secret and method `m` is executed again.

6.1.3 Implementation Features

By default, KEG uses Z3 [41] to solve composed insecurity formulas and Max-SMT problems. Nevertheless, all (Max-)SMT solvers that support SMT-LIB can be easily integrated.

Information flow policies supported by KEG are noninterference (generalized version) and declassification (targeted conditional delimited information release). KEG allows the user to supply those policies by explicitly specifying them as source code comments. Noninterference is defined as a class-level policy that affects all methods of a class. Declassification, on the other hand, is specific to a method and specified as part of a JML method contract. KEG introduces some new keywords to specify declassification.

The symbolic execution engine of KeY enables KEG to use loop invariants and method contracts to deal with unbounded loops and recursive method calls. The user can flexibly choose whether loop invariants and/or method contracts are used. Similar to KeY, KEG supports a rich subset of Java, including primitive types, objects types and arrays. Class inheritance, object creation and aliasing are also supported. However, the secret inference feature is restricted to primitive types and objects. To test the value of variables that have a reference type for equality, KEG uses the approach proposed in [18].

The crucial aspect in the secret inference approach is finding optimal low input. KEG outputs the optimization problems as AMPL [53] specifications in case the leakage is quantified by means of parametric counting. This makes it possible to use all optimizers that support AMPL. Currently, KEG uses a combination of two open source optimizers, *Bonmin* and *Couenne*, as well as the commercial optimizer *Local Solver* [21].

For model counting KEG uses Barvinok [116] that only supports parametric polytopes. This restricts the use of Barvinok to programs whose path conditions and symbolic output values are linear. However, this restriction does not affect other features of KEG, including leak detection (and generation of code demonstrating the leakage). Nonlinear path conditions and symbolic values also do not restrict the use of Max-SMT solvers for finding optimal low inputs.

6.1.4 Usage

Specifying Information Flow Policies

Noninterference

(Generalized) noninterference policies are class-level and can be annotated directly as Java comments. A noninterference policy consists of two elements: *source* (high variables) and *sink* (low variables). It prohibits all information flow from the source to the sink. The syntax to specify a noninterference policy is as below

```
/*! sink | source ; !*/
```

where `sink` and `source` are two sets of memory locations in the program, i.e. fields, array elements.

KEG allows more than one noninterference policy to be defined. The syntax is given as below

```
/*!
sink_1 | source_1 ;
sink_2 | source_2 ;
...
sink_n | source_n ;
!*/
```

If there is more than one noninterference policy being specified in a program, KEG tries to find information leaks in the annotated program w.r.t. the specified policies.

Declassification

KEG supports targeted conditional delimited information release for declassification. A declassification policy is specified within the JML method contract specification of a method and affects only that method. The syntax for targeted conditional delimited release is as follows:

```
@escapes (released expression E) [\if condition C ] [\to destinations D ];
```

The escape hatch expression is defined after the keyword **`escapes`**. If the declassification condition and specific targets are given, they are specified after the keywords **`\if`** and **`\to`** respectively. The above syntax defines following declassification policy: secret information (defined by a class-level noninterference policy) can be leaked through expression `E` to destinations `D` (set of memory locations) if condition `C` is satisfied before the execution of the method.

Defining the Set of Observable Variables

By default, the set of low variables L, defined by a generalized noninterference policy $H \not\rightsquigarrow_{GNI} L$, is also the set of observable variables. KEG allows one to restrict the set of observable variables by specifying the set of observable variable O within the method contract of a method (affecting only this method). The syntax is as follows:

```
@observes varName_1,...,varName_n ;
```

Supplying Prior Probability Distribution of High Input

Secret inference simulation requires the prior probability distribution of high input. The user can define this distribution in an input file. This file is generated by KEG with the default content "`true: 1`" that specifies the uniform prior distribution for high input. Non-uniform distributions can be specifed as:

```
formula_1 : weight_1 ;
formula_2 : weight_2 ;
...
formula_n : weight_n ;
```

where $\texttt{formula}_i$ is a formula on set of high variables and $\texttt{weight}_i$ is a positive integer number that is the weight value of all high values satisfying $\texttt{formula}_i$. All formulas must be pairwise exclusive and they must describe all possible values of high variables.

6.2 Workflow Illustration

6.2.1 Leak Demonstrator Generation

We illustrate the specification for information flow policies and the leak demonstrator generation process in KEG by a simple Java program in Listing 6.1.

Class `Simple` declares three integer typed fields `l`, `x`, `y` as well as a method called `magic` which assigns a value to `l` depending on the sign of field `x`.

The information flow policy of the class is {x, y} $\not\rightsquigarrow$ {l} and specified in a comment starting with "`/*!`" in Line 4. Variables `x`, `y` are implicitly declared as high variables and `l` as a low variable. This strict noninterference policy is relaxed in line 6 for method `magic` by providing a targeted conditional release specification consisting of an escape hatch `x*y`, the target `l` and the condition `x>-1`.

Running KEG on the above example produces a symbolic execution tree consisting of two paths; one for each branch of the conditional statement. KEG generates for each unique pair of these paths the corresponding insecurity formulas and passes these to an SMT solver. Only one of the three generated insecurity formulas is satisfiable. The following model is found by Z3:

Listing 6.1: Example program to illustrate leak demonstrator generation of KEG

```
1 public class Simple {
2   public int l;
3   private int x, y;
4   /*! l | x y ; !*/
5
6   /*@ escapes (x*y) \to l \if x>-1; @*/
7   public void magic() {
8     if (x>0) {
9       l=x*y;
10    } else {
11      l=0;
12    }
13  }
14 }
```

Insecurity Formula (in SMT-LIB syntax)	Model
(let ((a!1 (not (and (> x_1 (- 1)) (> x_2 (- 1)))))) (and (>= x_1 1) (<= x_2 0) (or (not (= x_1 x_2)) (not (= y_1 y_2))) (= l_1 l_2) (not (= (* y_1 x_1) 0)) (or a!1 (= (* x_1 y_1) (* x_2 y_2)))))	x_1: 1 x_2: -1 y_1: 1 y_2: -1 l_1: 0 l_2: 0

The model describes two runs: The first run, labelled with 1, starts in an initial state with `x`, `y` and `l` initialized with 1, 1 and 0, respectively. We identify variable `v` in execution X by `v_X`. The second run, labelled with 2, has initial values -1, -1, and 0. The first run enters the then-branch of the conditional, the second one does not.

As the value of `l` is altered in the first run when executing the then-branch but not by the second run, where it remains 0, a leak is detected (the leaked information is the sign of field `x`). KEG generates exactly one *leak demonstrator*, which is output as a well-structured and human readable JUnit test. The leak demonstrator program is depicted in Listing 6.2. KEG outputs the leak demonstrator exactly as shown, i.e. pretty printed and structured with comments. We have only renamed a few fields to increase readability further.

The two initial states for the two runs of method `magic` are set up in lines 6–14 and 23–31. To ensure that the runs do not interfere, two instances of class `Simple` are created (lines 6 and 23). In general, more work needs to be invested to ensure independent runs (for instance, if static members are modified by a run). This can be achieved by running the program on two different Java Virtual Machines and by querying those for the required information (this is currently not supported by KEG).

Before invoking method `magic`, the initial values of all fields and method parameters are assigned. In Listing 6.2, the initial value of each field `l`, `x`, `y` of both runs are stored in the corresponding variables `l_1`, `x_1`, `y_1` (lines 7–9) and `l_2` , `x_2`, `y_2` (lines 24–26). These initial values are taken from the counter example produced by the SMT solver. To assign values

Listing 6.2: JUnit test as leak demonstrator program

```
1 @Test
2 public void test_magic_l_0 ()
3 throws NoSuchFieldException, SecurityException,
4 IllegalArgumentException, IllegalAccessException {
5   /* Prepare for execution 1 */
6   Simple s1 = new Simple();
7   int l_1 = 0;
8   int x_1 = 1;
9   int y_1 = 1;
10
11  /* Configure variable: s1 */
12  setFieldValue(s1,"l",l_1);
13  setFieldValue(s1,"x",x_1);
14  setFieldValue(s1,"y",y_1);
15
16  /* Perform execution 1 */
17  s1.magic();
18
19  /* Get the value of low variable l after execution 1 */
20  int l_out_1 = ((Integer)getFieldValue(s1,"l")).intValue();
21
22  /* Prepare for execution 2 */
23  Simple s2 = new Simple();
24  int l_2 = 0;
25  int x_2 = -1;
26  int y_2 = -1;
27
28  /* Configure variable: s2 */
29  setFieldValue(s2,"l",l_2);
30  setFieldValue(s2,"x",x_2);
31  setFieldValue(s2,"y",y_2);
32
33  /* Perform execution 2 */
34  s2.magic();
35
36  /* Get the value of low variable l after execution 2 */
37  int l_out_2 = ((Integer)getFieldValue(s2,"l")).intValue();
38
39  assertNotNull(l_out_1);  assertNotNull(l_out_2);
40  /*
41  * assert that the value of low variable l is not changed after performing
42  * two executions
43  */
44  assertTrue(l_out_1 == l_out_2);
45 }
```

to the fields of an object, the auxiliary method `setFieldValue` makes use of Java's reflection framework.

After each run (at line 17 and line 34), the concrete output value of `l` is extracted (line 20 and 37). Line 44 asserts that the output values of `l` observed at the end of each run are equal. The JUnit test throws an assertion failure exception, if the output values of the two runs are different and thus an actual leak happened. If no exception is thrown then the counter example was spurious, for instance, due to a too weak loop specification. Tests that do not result in an assertion failure can then be omitted from our test suite.

6.2.2 Secret Inference Simulation

If a target program is insecure w.r.t. a noninterference policy, KEG can perform a secret inference simulation on an insecure method. After the user selected the insecure method, KEG generates a Java program to run an attack on this method. With class `Simple` in Listing 6.1, there is only one method (`magic`) that violates noninterference policy $\{x,y\} \not\rightsquigarrow \{l\}$. The attack program for method `magic` is depicted in Listing 6.3, with few changes in variable names to increase readability.

The program in Listing 6.3 executes method `magic` and prints out the output value of `l`. One instance of class `Simple` is created to invoke method `magic` at line 4. Before executing `magic`, the initial values of all fields of `s` (`l, x, y`) are set up by low and high inputs read from corresponding input files (lines 16 - 22). Then method `magic` is invoked (line 25) and the output value of `l` is extracted (line 28) and printed out (line 29).

The high input, i.e. the input values of `x` and `y`, is chosen by the user and does not change during the secret inference simulation. On the other hand, the low input, i.e. the input value of `l`, is generated automatically by KEG to maximize the potential leakage and is updated before each run. With method `magic`, KEG automatically detects that it is low-independent w.r.t. noninterference policy $\{x, y\} \not\rightsquigarrow \{l\}$ (Definition 4.4), hence KEG simply generates a random value to assign to `l` and runs the attack program only once. Finally, the knowledge of high input is computed and given to the user in the form of a formula on `x,y`. For example, if the user chooses 2 and 3 as the input value of `x` and `y` respectively, with an arbitrary input value of `l`, the output value of `l` is always 6 and KEG returns the formula $xy \doteq 6$ as the full knowledge of `x` and `y`. With this knowledge, KEG offers $(1,6),(6,1),(2,3),(3,2),(-1,-6),(-6,-1),(-2,-3),(-3,-2)$ as the possible input values of `x` and `y`.

We give another example for the secret inference simulation using a program that is not low-independent. We use a relaxed password checker program, namely `RelaxPC`, that is presented in Listing 6.4. This program is modified from the running example program in Listing 4.1 by simply removing the condition statement checking whether `l` is smaller than 100. `RelaxPC` can be seen as a relaxed version of a typical password checker, in which `h` is the secret password and `l` is the guess of the user. This program not only returns whether an user's guess is correct, but also tells the user whether her guess is smaller than the password (here password and guess are integer numbers).

Class `RelaxPC` declares two long integer typed fields `l`, `h` as well as a method called `check` which assigns a value to `l` depending on the comparison result between `l` and `h`. The class-level information flow policy is $\{h\} \not\rightsquigarrow \{l\}$ and specified at line 4. Method `check` is supplied by a precondition requiring that `h` is a 32-bits integer number, with the value is bounded in the half-open interval $[-2^{31}, 2^{31})$. The precondition $-2^{31} \leq h < 2^{31}$ is used as the initial knowledge of

Listing 6.3: Attack program to perform method `magic` and return the output value of `l`

```
1  public  class Simple_magic_SecretInferSimulator {
2    public static void main (String[] args)
3    throws NoSuchFieldException, SecurityException, IllegalArgumentException,
       IllegalAccessException, NumberFormatException, IOException {
4      Simple s = new Simple();
5      Integer x = new Integer(0);
6      Integer y = new Integer(0);
7      Integer l = new Integer(0);
8
9      Map<String, Object> mapObj = new HashMap<String,Object>();
10     mapObj.put("x", x);
11     mapObj.put("y", y);
12     mapObj.put("l", l);
13
14     String highInputFile = "...";
15     String lowInputFile = "...";
16     readInputValuesFromFile(highInputFile,mapObj);
17     readInputValuesFromFile(lowInputFile,mapObj);
18
19     /* Configure variable: s */
20     setFieldValue(s,"x",((Integer)mapObj.get("x")).intValue());
21     setFieldValue(s,"y",((Integer)mapObj.get("y")).intValue());
22     setFieldValue(s,"l",((Integer)mapObj.get("l")).intValue());
23
24     /* Perform execution */
25     s.magic();
26
27     /* Get the value of low variable l after execution */
28     int l_out = ((Integer)getFieldValue(s,"l")).intValue();
29     System.out.println("l: " + l_out);
30   }
31   ...
32 }
```

Listing 6.4: Relaxed Password Checker (`RelaxPC`)

```
1 public class RelaxPC{
2   private long h;
3   public long l;
4   /*! l | h ; !*/
5
6   /*@ requires h >= -2147483648 && h < 2147483648; @*/
7   public void check(){
8     if (l == h)
9       l = 3;
10    else if (l < h)
11      l = 0;
12    else
13      l = -3;
14  }
15 }
```

the attacker about `h`. The output value of `l` is either 3, 0 or -3 depending on whether $\texttt{l} = \texttt{h}$, $\texttt{l} < \texttt{h}$ or $\texttt{l} > \texttt{h}$ respectively.

Running KEG on program `RelaxPC` with respect to the noninterference policy specified at line 4, KEG detects three information leaks and generates three leak demonstrators for them. If the user wants to perform a secret inference simulation, KEG creates an attack program for method `check`. KEG detects that method `check` is not low-independent w.r.t. the policy $\{\texttt{h}\} \not\rightsquigarrow \{\texttt{l}\}$, hence beside the initial value of `h`, KEG requires the user to choose a maximal number of experiments to be carried out. It also generates a file in which the distribution of `h`'s input value is specified. The default uniform distribution is given as "*true* : 1" . This file can also be modified by the user to supply an arbitrary formula-based distribution. Then the user is asked to choose a leakage metric among Shannon, guessing and min entropy. This metric is used in finding optimal input values for `l`.

Let's assume that the user has fixed already the setting: the input value of `h` is 1000, the maximum number of experiments is 32, the distribution of `h`'s input values is uniform and the leakage metric is Shannon entropy. By using the combination of three optimization tools *Local Solver*, *Bonmin* and *Couenne*, KEG produces 3 as low input and writes it to low input file. Then the attack program is performed within a subprocess created by KEG. The output value of `l` obtained by running the attack program using 3 as input value of l is 0. Based on that, KEG synthesizes $3 < \texttt{h} < 2^{31}$ as the knowledge about `h`'s input value. From this new knowledge, KEG deduces 1073740000 as the input value of `l` for the next experiment and changes the content of low input file by the new optimal value. Then KEG continues to infer the input value of `h` by running the attack program again. With this setting, KEG successfully discovers 1000 as the input value of `h` after 31 experiments.

6.3 Experiments

We separately evaluate the main features of KEG (leak detection/demonstrator generation and secret inference) using a collection of micro benchmarks.

6.3.1 Leak Detection and Demonstrator Generation

We performed experiments on a set of programs[2] to evaluate the leak detection and demonstrator generation of KEG. Table 6.1 shows the aggregated results. All experiments were done on an Intel Core i7-4702HQ processor with JVM setting -Xmx4096m.

Table 6.1: Benchmark statistics of leak detection and demonstrator generation

File name	Analyzed Method	#L/MI	Policy (NI/D)	S/I	T_L (ms)	T_{SE} (ms)	T_{MF} (ms)	T_{Tot} (ms)	#LD/FW
Mul	product	0 / 0	D	I	4187	847	1188	6266	1 / 0
Mul_StrongLI	product	1 / 0	D	I	4275	1746	1211	7274	1 / 0
Mul_WeakLI	product	1 / 0	D	I	4214	1909	1293	7463	2 / 1
Mul_WrongLI	product	1 / 0	D	I	4397	1678	1169	7285	0 / 0
Comp_StrongMC	doWork	0 / 1	NI	I	4181	1491	2278	7995	3 / 0
Comp_WeakMC	doWork	0 / 1	NI	I	4217	1383	2417	8065	3 / 3
Comp_WrongMC	doWork	0 / 1	NI	I	4182	1395	2275	7887	0 / 0
Company	calculate	1 / 1	NI	I	4283	2496	1990	8816	3 / 0
ExpList	magic	0 / 0	NI	I	4178	1911	2535	8668	1 / 0
ExpLinkedList	magic	0 / 4	NI	I	4229	4690	6564	15526	2 / 0
ExpArrayList	magic	0 / 5	NI	I	4230	8975	11505	24752	3 / 0
ArrSearch	search	1 / 0	D	S	4199	2934	2400	9568	0 / 0
ArrMax	findMax	1 / 0	NI	I	4215	3584	963	8804	1 / 0
ArrMin	findMin	1 / 0	D	S	4746	3128	983	8925	0 / 0
ArrSum	calcSum	1 / 0	D	S	5481	2504	788	8846	0 / 0

#(L/MI/LD/FW): nr of **L**oops/**M**ethod **I**nvocations/generated **L**eak **D**emonstrators/**F**alse **W**arnings
NI/D: **N**on-**I**nterference/**D**eclassification, S/I: **S**ecure/**I**nsecure
T_X: Time for **L**oading/**S**ymbolic **E**xecution/**M**odel **F**inding/**Tot**al

Concerning the runtime performance: A significant amount is spent for parsing the program, this can be reduced by parser optimizations, for example, by using a hand-coded version instead of a generated parser. Model finding time can be further optimized by performing simple techniques like symmetry reduction, learning and caching, all of which have not yet been implemented. Another factor is the programming language Java whose optimizations are performed at runtime and, hence, code that is run only few times will not be optimized at all.

A few observations concerning the benchmarks: For the examples `Mul` and `Comp`, we analyzed the effect of loop and method specifications in case of strong, weak and wrong specifications (*filename*_Strong/Weak/Wrong_LI/MC). As expected, with sufficiently strong specifications, all insecure paths could be precisily identified and only actual leak demonstrators were generated. Weak specifications over-approximate the behaviour, leading to false positives, while wrong specifications can prevent to analyze all possible behaviours and some existing leaks were missed. The analysis of method `search` in classes `ArrSearch`, `ArrMin`, `ArrSum` identified the method correctly as secure with respect to the specified declassification policy and generated no leak demonstrators.

[2] `www.se.tu-darmstadt.de/fileadmin/user_upload/Group_SE/Tools/KEG/experiments.zip`

6.3.2 Secret Inference

Uniform Distribution

The secret inference approach implemented in KEG is evaluated on a collection of insecure programs[3] under the assumption that the distribution of the high input is uniform. We also assume that for any program the attacker knows nothing about the secret except that it is a 32 bit integer number. Loop specifications and method contracts are supplied for programs containing unbounded loops and recursive method invocations. KEG has been configured to terminate its attack when it was able to infer the values of the high variables, the maximum achievable knowledge has been reached (there is no way to avoid redundant low input), or the number of experiments exceeded the limit of 32. The evaluation was performed on a Macbook Pro Retina late 2013 (2.67GHz Processor, 8GiB RAM). The results are shown in Table 6.2.

Table 6.2: Benchmark statistics of secret inference w.r.t. uniform distribution of high input

File name	#SP /RP	High input	Shannon entropy #RB/E	Shannon entropy T(s)	Min entropy #RB/E	Min entropy T(s)	Guessing entropy #RB/E	Guessing entropy T(s)
PassChecker	2/2	2135451222	10^{-8}/32	159	10^{-8}/32	13.3	10^{-8}/32	139.3
RelaxPC	4/3	-1208665253	32/31	31.7	11.1/32	6.9	32/31	29.4
MultiLows	6/3	395444738	32/20	22.6	32/30	14.3	32/22	24.3
ODependL	4/3	-13484756	1/1	1.8	1/4	1.8	1/1	1.6
ODependL	4/3	95464630	32/31	39	13/32	13.7	32/31	33.1
ODependLH	6/5	-941087637	n/a	n/a	32/1	1.5	n/a	n/a
ODependLH	6/5	23269332	n/a	n/a	1/1	1.2	n/a	n/a
LoopPlus	3/2	-552256949	n/a	n/a	1/1	0.4	n/a	n/a
LoopPlus	3/2	1707132530	n/a	n/a	32/1	0.9	n/a	n/a
EWallet	3/2	692935244	n/a	n/a	20.9/1	0.6	n/a	n/a

#(SP/RP): nr of **S**ymbolic **P**aths/**R**isky **P**aths
#(RB/E): nr of **R**evealed **B**its/necessary **E**xperiments
T(s): Time for experiments (seconds)

Table 6.2 shows that using min entropy to guide experiment generation is in most cases the fastest option, but it lags often behind the other entropies regarding the amount of inferred information, because it considers merely the number of output values. The Shannon and guessing entropy-based metrics can only be used for analysing the programs `PassChecker`, `RelaxPC`, `MultiLows`, and `ODependL`, because only those fall into the class of programs characterized in Section 5.3.1. For these programs (exception `PassChecker`) the Shannon and guessing entropy-based metrics turn out to be very effective. Both reveal almost 1 bit per experiment.

Figure 6.3 compares for program `RelaxPC` the number of bits revealed after each experiment for each of the supported metrics and with a simple exhaustive brute force attack (the latter could be lucky and hit the secret in one of the first 32 attempts). For this program we can see that the Shannon and guessing entropy-based metrics perform best, extracting almost one bit per experiment and reveal the complete secret after 31 steps. The amount of bits revealed by

[3] `www.se.tu-darmstadt.de/research/projects/albia/download/secret-inferring/`

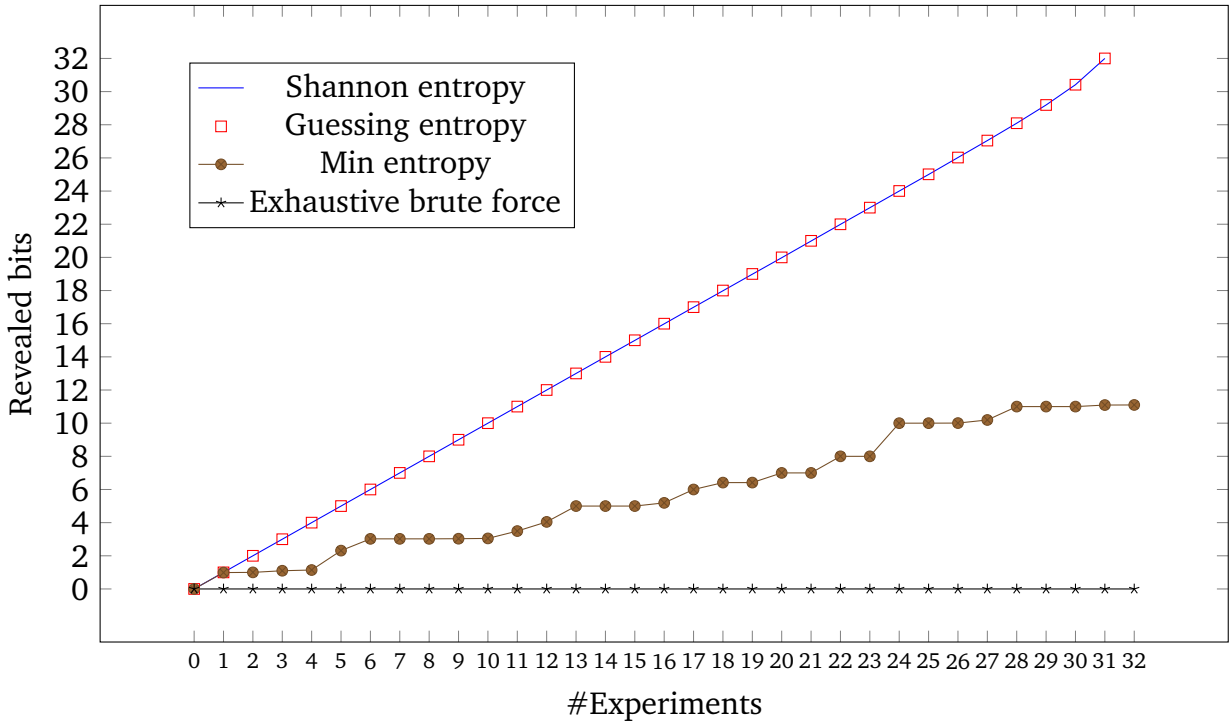


Figure 6.3: Bits revealed per experiment on `RelaxPC` with uniform distribution of high input

using min entropy-based metric, although is less than one using Shannon or guessing entropy, is still significant better than a brute force attack. The reason is that, with a knowledge $a \leq \mathtt{h} \leq b$, the optimization tools tend to yield a number that is close to $\frac{a+b}{2}$ as the optimal value of `l` (that maximizes Shannon or guessing entropy-based leakages), whereas the Max-SMT solver only returns a value in the range (a, b). Consequently, using Shannon or guessing entropy leads to a binary search on the range $[a, b]$.

The program `PassChecker` is a simple password checker, leaking only whether the given input is equal to the secret or not. The amount of leakage does not depend on the low input and all entropy-based approaches perform equally bad as random experiments or exhaustive brute-force attacks.

For programs whose observable output depends on high variables (`ODependLH`, `LoopPlus` and `EWallet`), Shannon entropy and guessing entropy are practically infeasible as the range of observable values is too large. However, min entropy is still applicable and quite effective as well, as it leads to the generation of low input for execution paths on which the observable output depends on the high input. Observe that `LoopPlus` and `EWallet` contain unbounded loops and recursive method calls.

The programs `ODependL`, `ODependLH` and `LoopPlus` witness the fact that successful secret inference may also depend on the values of high variables. The reason is that in these programs the high variable influences the taken symbolic execution path and the final output values, which renders the set of reachable paths value-dependent on high variables. Hence, the quality of the generated experiments depends as well on the high variables.

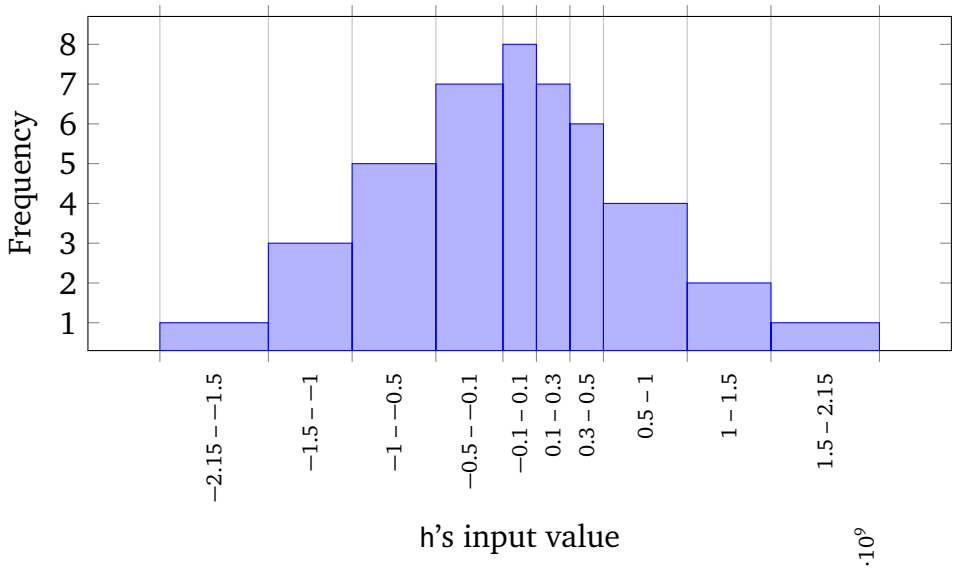


Figure 6.4: `RelaxPC`: Non-uniform prior distribution of `h`'s input value

Non-Uniform Distribution

Intuitively, if the attacker can make a good estimation about the secret's distribution, she can optimize her guessing strategy based on such a distribution to learn the secret more efficiently. In order to confirm this intuition, we conduct a secret inference simulation on program `RelaxPC` (Listing 6.4) using a non-uniform distribution for the input value of `h` as follows:

$$\mu(\mathsf{h} < -1.5 \cdot 10^9 \vee \mathsf{h} \geq 1.5 \cdot 10^9) = 1$$
$$\mu(\mathsf{h} \geq -1.5 \cdot 10^9 \wedge \mathsf{h} < -10^9) = 3$$
$$\mu(\mathsf{h} \geq -10^9 \wedge \mathsf{h} < -5 \cdot 10^8) = 5$$
$$\mu((\mathsf{h} \geq -5 \cdot 10^8 \wedge \mathsf{h} < -10^8) \vee (\mathsf{h} \geq 10^8 \wedge \mathsf{h} < 3 \cdot 10^8)) = 7$$
$$\mu(\mathsf{h} \geq -10^8 \wedge \mathsf{h} < 10^8) = 8$$
$$\mu(\mathsf{h} \geq 3 \cdot 10^8 \wedge \mathsf{h} < 5 \cdot 10^8) = 6$$
$$\mu(\mathsf{h} \geq 5 \cdot 10^8 \wedge \mathsf{h} < 10^9) = 4$$
$$\mu(\mathsf{h} \geq 10^9 \wedge \mathsf{h} < 1.5 \cdot 10^9) = 2$$

Recall that $\mu(f(x))$ defines the frequency of all values in the domain of x that satisfy formula $f(x)$. For example, with the above distribution, all input values of `h` that are in the range $[-10^8, 10^8)$ have the frequency value 8. This distribution is depicted in Figure 6.4, where we restrict that the value of `h` to be in the range $[-2^{31}, 2^{31})$, which is also the initial knowledge of the attacker about the input value of `h`, similar to other small benchmarks used for secret inference.

We perform KEG on the program `RelaxPC` using different input values of `h`. Those values are chosen as follows: For each range of `h`'s values depicted in Figure 6.4, we generate 20 random values. For each input value of `h` and each security metric (Shannon, guessing and min entropy) we perform a secret inference simulation by KEG. We fix 10 as the limit for the number of experiments for all simulations.

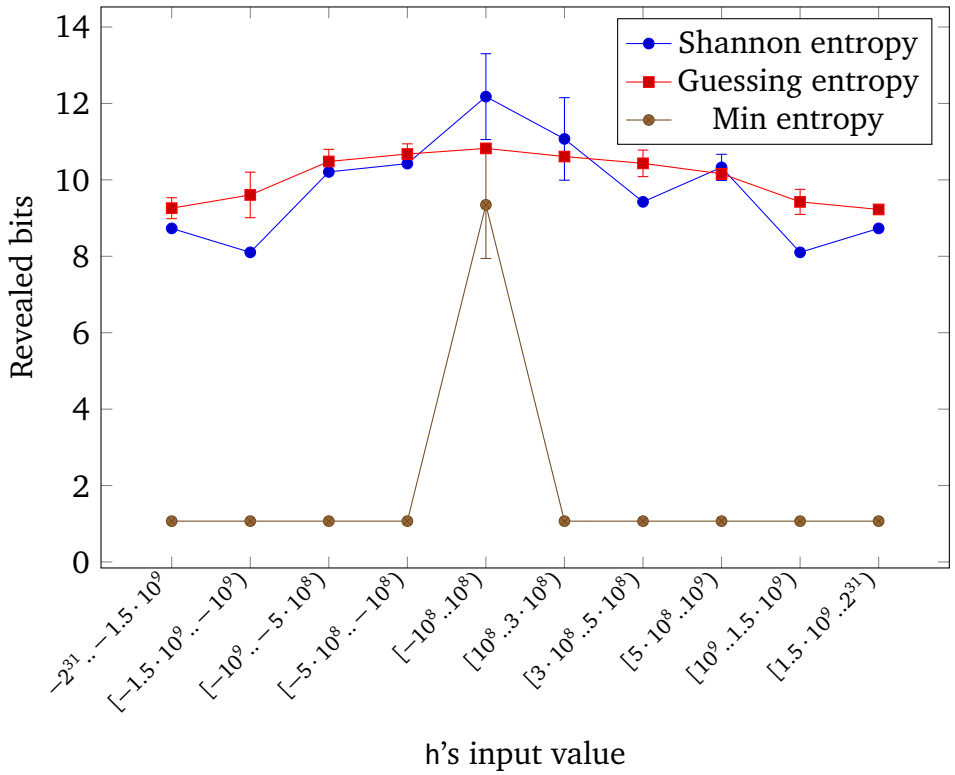


Figure 6.5: `RelaxPC`: The average and standard deviation of revealed bits of `h`'s value after 10 experiments using non-uniform distribution

For each set of simulations that use the same security metric and take `h`'s input values in the same range, we compute the average and standard deviation of bits revealed after 10 experiments as well as the average and standard deviation of time consumptions. The result is aggregated in Figure 6.5 for bits revealed and Figure 6.6 for time consumption. All secret inference simulations are executed on the same computer used also for the experiments in the uniform distribution case.

Figure 6.5 shows that in general, the higher the frequency of the input value of `h` is, the more amount of bits are revealed. In particular, if the input values of `h` are in the range that has the highest frequency value, the average amount of revealed bits is maximal for all three security metrics. This amount, in the case of Shannon entropy, is significantly higher than the amount of bits extracted by using Shannon and guessing entropy under the assumption that the input value of `h` has uniform distribution (almost 10 bits, see Figure 6.3). However, if the actual input value of `h` has a low frequency, the number of bits that can be revealed is usually lower than 10 bits. Figure 6.5 also shows that the fluctuations of revealed bits obtained by using guessing entropy are considerably smaller than those in the case of using Shannon entropy.

The amount of revealed bits in the case of min entropy is always less than that of Shannon or guessing entropy. In case the input value of `h` is in the range that has the highest frequency, the amount of discovered bits using min entropy is significantly higher than the amount in the uniform distribution case (where min entropy is also used). However, with values that are not in the highest frequency range, using min entropy turns to be much less efficient, with only about 1 bit is discovered. The reason is that, with the given non-uniform distribution, the Max-SMT solver Z3 always provides a number in the range $(-10^8, 10^8)$ as the optimal low input.

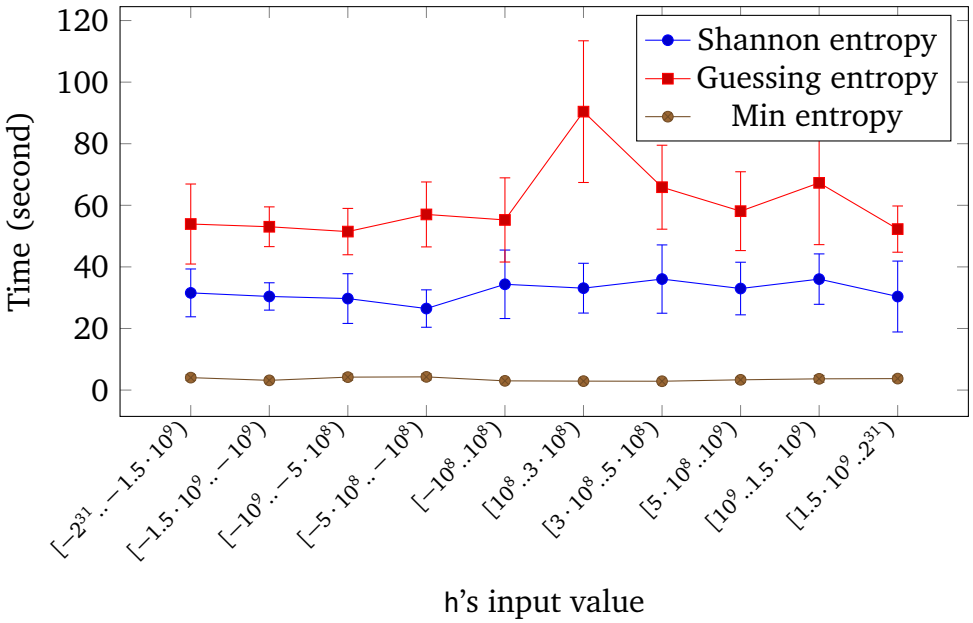


Figure 6.6: `RelaxPC`: The average and standard deviation of time consumptions value after 10 experiments using non-uniform distribution

Consequently, if the actual secret value is not in that range, the low input generated by using min entropy does not help much to infer the secret.

The aggregated result in Figure 6.5 confirms the intuition that if the attacker's assumption about the probability distribution of the secret is close to the actual distribution, she can learn the secret more efficiently, no matter what security metric she uses.

Concerning the runtime overhead, Figure 6.6 shows that using guessing entropy is the most expensive choice, while using min entropy is significantly cheaper than two other metrics. Along with the results in Table 6.2, it can be seen that the approach using a Max-SMT solver to find optimal low inputs is much cheaper than the approach using parametric counting and optimization tool.

7 Electronic Voting Case Study

In this chapter, the KEG tool is employed to check the confidentiality of individual ballots on an electronic voting system. The main aim of this case study is to showcase the leak detection and demonstrator generation function of KEG. Section 7.1 briefly introduces *sElect*, a real-world e-voting system on which the case study is based. The case study is twofold. We show that ballot confidentiality can be established with a proper declassification policy in Section 7.2. Another approach to establish vote privacy is using a privacy game, that is introduced in Section 7.3. The analysis in Section 7.3 is performed with a combination of KEG and a loop invariant generation tool, that constitute a scheme for *fully automatic logic-based* information flow analysis. Section 7.4 concludes the chapter with insightful discussions. This chapter is based on two publications [48, 49] of the author of this thesis.

7.1 Electronic Voting System *sElect*

Electronic voting (e-voting) has been widely used for years in various kinds of election due to its undeniable advantages: auditable, transparent, accurate, faster result, etc. However, those benefits rely mostly on a well-designed, error-free e-voting system, that is very difficult to achieve in practice due to the high complexity of such a system and the many (usually contradictory) requirements that have to be taken into account. Designing an e-voting system requires to balance between *usability*, *simplicity* and *security* that greatly differ on kinds of election. Moreover, verifying e-voting systems to ensure that they satisfy some essential requirements e.g. *ballot privacy* or *coercion-resistance* is still a very challenging task.

sElect [77] is a remote electronic voting system implemented in Java that allows voters to vote over the Internet. It is a lightweight web-based system being designed towards simplicity and security, and targeting low-risk elections (elections within clubs and associations). Its main characteristics are fully automated verification, voter-based verification and simple cryptography and design.

The protocol of sElect includes four main phases. The election begins in the *setup phase* where all election parameters (list of candidates, list of eligible voters, time frame, etc) are fixed and public/private keys are generated. The *voting phase* starts after the *setup phase* completes, in which the voters make a choice and let their *voter supporting devices* (VSDs) cast the ballots. Here the VSD could be simply the voter's web browser. The ballots are encrypted by VSDs and sent to the authentication server. For each ballot, a *verification code* is also generated. When the *voting phase* is over, the list of cypher texts (encrypted ballots and verification code) is passed to the mix servers, where they are validated and decrypted to compute the final result. This phase is called *mixing phase*. The final phase is the *verification phase*: after the final result has been published on the bulletin board, the voters can verify that their ballots were counted correctly. The verification can be carried out by both the voters and their VSDs (the verification performed by VSDs is fully automated).

Like any other electronic voting systems, sElect rises a plethora of security issues like *integrity*, *verifiability* and *coercion-resistance*. In this case study, we focus on the *confidentiality* of individ-

ual ballots, i.e. we aim to detect all possible information flow leaks from individual ballots to the public result. Even though being defined as a lightweight e-voting system, the complexity of sElect is beyond the power of KEG, mainly because of its distributed architecture and cryptography deployment. The key that enables KEG as well as other logic-based information flow analysis/verification tools to be used is simplifying sElect. The simplified version must be simple enough to be analyzed by the tools while it still can be used to guarantee the ballots' privacy under some specific assumptions. The simplified version of sElect and the ballots' privacy checking are detailed in the next two sections of this chapter.

7.2 Ballot Confidentiality with Declassification

Because the final result is aggregated from all individual ballots, it is obvious that there is an information flow from each ballot to the result. This information flow is crucial for any e-voting system, hence using a noninterference policy to enforce ballots privacy is insufficient. In this section, we establish ballots confidentiality for a simplified version of sElect using declassification policy. For the case study, we adapted the electronic voting system presented in [56, 106], which is based on sElect.

7.2.1 Simplified E-Voting System

Vote confidentiality in sElect is guaranteed by the use of cryptography. Cryptographic algorithms and protocols are based on advanced mathematical theories (and even unproven assumptions about the hardness of an underlying mathematical problem). This makes it infeasible to verify sElect using current information flow analysis tools.

To be able to analyse such systems, Küsters et al. [76] proposed a solution using *ideal encryption*. In a nutshell, their solution removes the encryption component from the system and enables the use of information flow analysis tools, which can now be run on the "simpler" program. The authors prove that information flow analysis results for the transformed program keep their validity for the original system.

Grahl [56] and Scheben [106] (their works are synthesized in [1, chapter 18]) apply the formal verification tool KeY on an e-voting system that contains the essential components of sElect. They focus on formal verification of vote confidentiality and integrity. Even though their e-voting program is not distributed and does not contain complex features, i.e. cryptography and networking, its verification requires considerable effort and user interaction.

Figure. 7.1 shows a UML class diagram of the e-voting system used in our case study[1], which is based on the implementation presented in [56]. We redesigned, but did not simplify, the system slightly to be able to show the capabilities of KEG, in particular, its support for information erasure policies. It consists of five classes: `VotingServer`, `CountingServer`, `Voter`, `Message` and `Result`. The voting protocol is as follows: First, voters (class `Voter`) register and obtain a unique identifier from the voting server (class `VotingServer`). Then, they send their vote to the server using a message (class `Message`) composed of the voter's identifier and vote. Voters are not allowed to change their vote once cast, even if the voting is still ongoing. The voting server receives the messages sent by the voters and forwards the ballots to the counting server (class `CountingServer`). Once all voters have cast their vote, the counting server computes the

[1] www.se.tu-darmstadt.de/research/projects/albia/download/e-voting-declassification-erasure

election result and returns it to the voting server. The result is then published. The counting server must not keep any ballots after the election result has been computed.

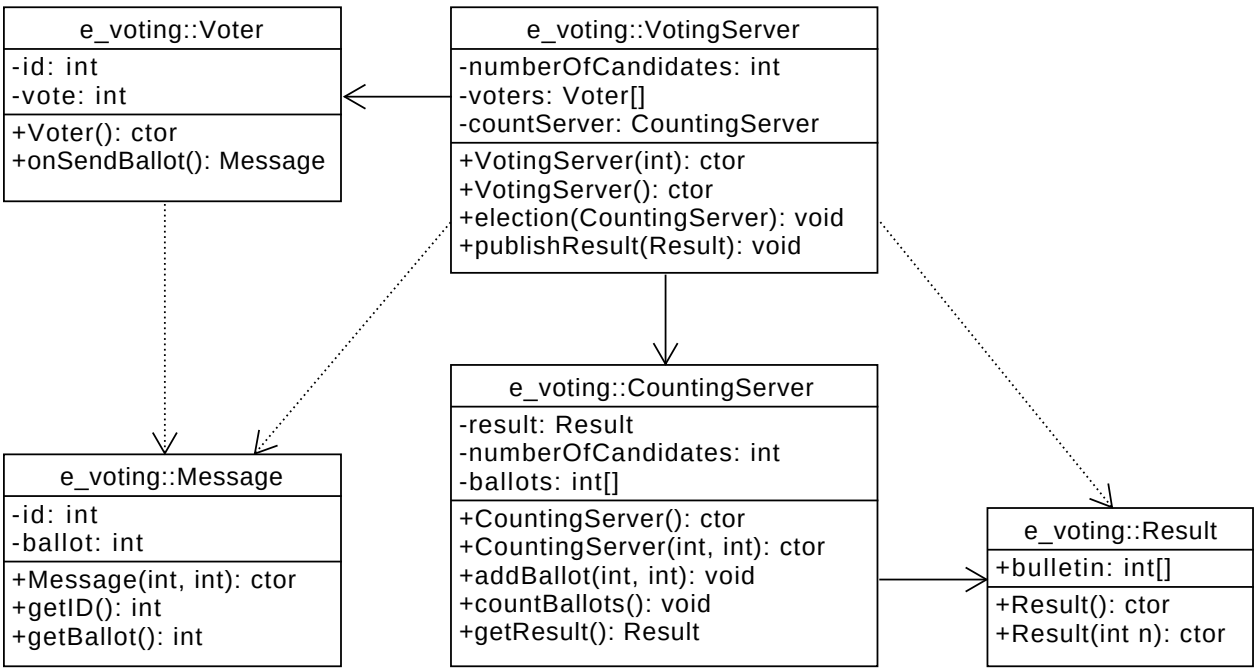


Figure 7.1: UML class diagram of the e-voting system in the case study

Listing 7.1: Class `VotingServer`

```
1  public class VotingServer {
2    private int numberOfCandidates;
3    private Voter[] voters;
4    private CountingServer countServer;
5    ...
6    public void election(){
7      for (int i=0; i<voters.length; i++){
8        Message message = voters[i].onSendBallot();
9        countServer.addBallot(i, message.getBallot());
10     }
11     countServer.countBallots();
12     publishResult(countServer.getResult());
13   }
14 }
```

Class `VotingServer` (Listing 7.1) is responsible for the overall election process which is coordinated by method `election`. Each voter is queried for her/his ballot, which is then passed on to the counting server (lines 7–10). After all voters cast their vote, the counting server starts counting all ballots (line 11) and computes the election's result. Finally, the voting server publishes the computed election result (line 12).

The relevant methods of class `CountingServer` are shown in Listing 7.2. Its field `result` keeps the bulletin, which is the aggregated result of all ballots counted so far. Counting must only

Listing 7.2: Class `CountingServer`

```
1 public class CountingServer {
2   private Result result;
3   private int numberOfCandidates;
4   private int[] ballots;
5   /*! result ballots numberOfCandidates | ballots ; !*/
6   ...
7   public void addBallot(int idx, int vote) { ballots[idx] = vote; }
8
9   /*@ requires numberOfCandidates>0 && ballots.length>0; @*/
10  public void countBallots() {
11    result = new Result(numberOfCandidates);
12    for (int i=0; i<ballots.length; i++)
13      if (ballots[i] >= 0 && ballots[i] < numberOfCandidates) {
14        result.bulletin[ballots[i]]++;
15      }
16  }
17
18  public Result getResult() { return result; }
19 }
```

take place, if there is at least one candidate and one ballot. This assumption is specified in the precondition of method `countBallots` at line 9. Line 5 specifies the generalized noninterference policy

$$\{\texttt{ballots}\} \not\rightsquigarrow_{GNI} \{\texttt{result}, \texttt{ballots}, \texttt{numberOfCandidates}\}$$

which disallows any information flow from `ballots` to any other field, including `ballots` itself. If an object or array appears in a noninterference policy, then KEG also includes in the policy all of its fields or elements, respectively. The election protocol described above requires the deletion of all ballots once `countBallots` finished computing the result. Hence, the *GNI* policy contains the field `ballots` on both sides to enforce the field's erasure as well as the erasure of all elements of the referred array.

To be able to analyse method `countBallots` with respect to secure information flow, KEG needs to reason about the flow of information within the (unbounded) loop iterating over all ballots. Hence, a correct and sufficiently strong loop specification is required. Listing 7.3 shows one possible loop invariant which guarantees that all ballots are counted correctly. It is quite simple and expresses (a) that the loop counter `i` stays within valid bounds and (b) that all ballots up to `i` have been correctly counted. The assignable clause (line 6) states that the loop may modify all elements of the array `result.bulletin` and the loop counter.

7.2.2 Checking Noninterference and Declassification

To analyse class `CountingServer` with respect to the *GNI* policy defined above we ran KEG on a Macbook Pro Retina late 2013 (2.6 GHz Intel Core i5 processor, 8 GiB RAM, Mac OS X 10.11.5).

Listing 7.3: Loop invariant in method `countBallots`

```
1 /*@ loop_invariant
2 @   i>=0 && i<=ballots.length &&
3 @     (\forall int k; k >= 0 && k < numberOfCandidates;
4 @         result.bulletin[k] == (\sum int j; 0 <= j && j < i;
5 @                                     (k==ballots[j]?1:0)));
6 @ assignable result.bulletin[*], i;
7 @*/
```

By default, KEG analyses all public methods of a class. After 63 seconds KEG finished its analysis and generated seven exploits showcasing different violations of the specified *GNI*. Listing 7.4 shows one of the generated leak demonstrators exposing an information leak from `ballots` to `result` in method `countBallot`.

Running the generated leak demonstrators results in assertion failures for all of them which means that genuine leaks were found. Looking closer at the generated exploit in Listing 7.4, we see that both initial states (lines 5–13 and lines 18–25) are identical except for the content of the `ballots` array whose entries are set to 2006 in the first initial state and to 0 in the second one. Consequently, the assertion failure in line 31 must be the result of an information flow from `ballots` (or its contents) to the aggregated result `result.bulletin`.

This is not very surprising, because the outcome of an election depends on the votes. We have to fix our policy by allowing some information to be leaked to the election result, namely, the aggregated number of votes per candidate. To specify this we use the TCD policy introduced in Sect. 3.3 and relax the noninterference policy of method `countBallots` by adding an escape hatch expression:

```
@ escapes
@  (\seq_def int i; 0; numberOfCandidates;
@            (\num_of int j;
@                     0<=j && j<ballots.length; i==ballots[j]));
```

The escape hatch expression uses JML's **`\seq_def`** constructor to define a sequence whose i-th element is equal to the number of votes cast for the i-th candidate. This means that the program is allowed to leak the number of votes for each candidate. For each candidate `i`, the comprehension expression **`\num_of`** returns the number `j` of indices between 0 (inclusive) and `ballots.length` (exclusive) that satisfy the Boolean expression `i==ballots[j]`.

After changing the policy we rerun KEG on `CountingServer`. It finds six leak demonstrators in ca. 50 seconds, one less than before. Running the exploits results again in assertion failures for all which means they are genuine. But inspection of the generated leak demonstrators shows that none of them indicates an information flow from votes to the result. This is a strong hint that no such leak exists (by proving the loop invariant and method contracts this can even be verified). Therefore, the problem must lie elsewhere. By similar reasoning as above, it can be easily seen that the test cases fail, because the information erasure policy `ballots` $\not\rightsquigarrow_{GNI}$ `ballots` is violated by methods `countBallot`, `addBallot`, and `getResult`. Inspecting the source code reveals that the ballots are not erased at all.

Listing 7.4: Leak demonstrator for class `CountingServer`

```
1  public class TestCountingServer extends TestCase {
2    @Test
3    public void test_countBallots_result_bulletin_0()
4    throws NoSuchFieldException, ... {
5      /* Prepare for execution 1 */
6      CountingServer cs_1 = new CountingServer();
7      e_voting.Result cs_result_1 = new e_voting.Result();
8      ...
9      int cs_numberOfCandidates_1 = 2006;
10     int[] cs_ballots_1 = new int[1];
11     ...
12     /* Configure variable: cs_ballots_1 */
13     for (int i=0; i<cs_ballots_1.length; i++) cs_ballots_1[i] = 2006;
14     /* Perform execution 1 */
15     cs_1.countBallots();
16
17     /* Prepare for execution 2 */
18     CountingServer cs_2 = new CountingServer();
19     e_voting.Result cs_result_2 = new e_voting.Result();
20     ...
21     int cs_numberOfCandidates_2 = 2006;
22     int[] cs_ballots_2 = new int[1];
23     ...
24     /* Configure variable: cs_ballots_2 */
25     for (int i=0; i<cs_ballots_2.length; i++) cs_ballots_2[i] = 0;
26     /* Perform execution 2 */
27     cs_2.countBallots();
28     ...
29     /* assert that the value of low variable cs_result_bulletin
30     * is not changed after performing two executions */
31     Assert.assertArrayEquals(cs_result_bulletin_out_1,
32     cs_result_bulletin_out_2);
33   }
34 }
```

Let us now fix these issues one method at a time. We decide that method `countBallot` is responsible to erase the individual ballots once it computed the result. To do this, we add the new private method `clearBallots` to class `CountingServer`

```
private void clearBallots() { ballots = new int[0]; }
```

and add an invocation of `clearBallots` as a final statement to `countBallots`. Now those exploits related to `countBallots` pass without assertion failure.

We turn to method `getResult`. The leak demonstrators related to it still fail, because the method does not erase any information about the ballots. If the method were only called after `countBallots` terminates, this would actually be fine, because `countBallots` erases all information. This shows that one has to be extremely careful when refactoring security-critical code: seemingly harmless rearrangements can introduce subtle leaks.

One can argue in favor of defensive programming and simply add a call to `clearBallots` to `getResult`. Another strategy is to weaken the security policy. We discuss this possibility now for method `addBallot`.

Assume we fixed `getResult`, then the remaining failing test cases are related to method `addBallot`. This method is used to collect the individual votes before the result is computed. The solution to erase the ballots is not applicable, because method `countBallots` needs this information. Instead, we decide to alter the information flow policy for this method by adding the escape hatch expression

```
@ escapes ballots;
```

as a local method annotation. This "deactivates" the information erasure requirement, but still enforces the noninterference part of the policy. We run KEG now on the corrected version of `CountingServer`. KEG finishes without generating any leak demonstrators after 20 seconds.

7.3 Ballot Confidentiality with Privacy Game

In this section we present another approach to establish the confidentiality of votes that relies on a privacy game. In fact, using the privacy game was in part motivated by the need to avoid declassification. We perform the case study on verifying the privacy property of sElect by proving the noninterference property of a simplified, ideal Java counterpart[2]. Moreover, the case study is carried out using a combination of KEG and a loop invariant generation tool to achieve *full automation* while still keeping high precision.

7.3.1 Fully Automatic Logic Based Approach

The logic-based information flow analysis approach proposed in Chapter 3 of this thesis requires that specifications that are necessary for information flow analysis, i.e. loop invariants and method contracts, must be supplied by the user. This is usually a tough task and requires a considerable effort. In this section we demonstrate an approach that reduces the workload of the user towards obtaining a fully automatic analysis of information flow for Java programs. The fundamental idea is to integrate KEG with specification generation techniques [73, 101, 118] to reduce the need of user-specified loop invariants and method contracts.

[2] `www.se.tu-darmstadt.de/research/projects/albia/download/e-voting-case-study/`

In the paper [57] and [1, Chapter 6], an approach to generate loop invariants for unbounded loops is proposed. The needed loop invariants are automatically generated by abstraction techniques, including array abstraction with symbolic pivots, based on abstract interpretation [36] of partitions in an array. Loop invariants are generated without user interaction by repeated symbolic execution of the loop body and abstraction of modified variables or, in the case of arrays, the modified array elements. The invariant generation provides loop invariants which are often precise enough to be used in information leak detection. The approach has been implemented based on the KeY system. The tool can output loop invariants for respective unbounded loops in the form of JML specifications.

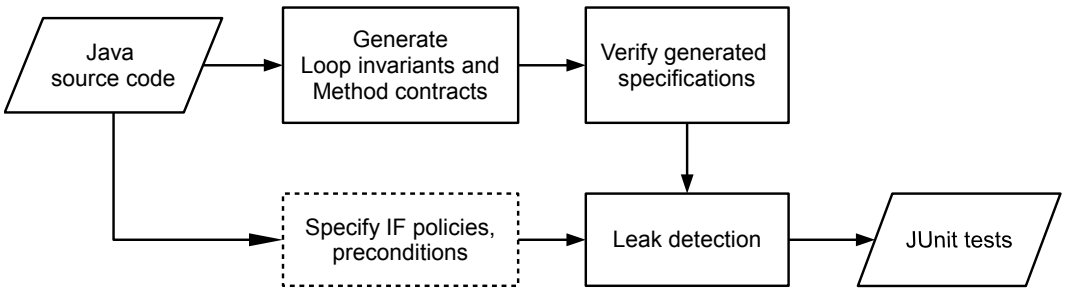


Figure 7.2: Fully automatic leak detection for Java programs

Figure 7.2 shows the combination of the two tools to automatically detect information leaks in a Java program. The solid border rectangle boxes represent automatic actions performed by our tools, while the dashed border one is for manual action done by the user. If the Java program contains unbounded loops and/or recursive method calls, the specification generator is activated to generate corresponding specifications and insert them into the original source code. Generated specifications are also verified by a verification tool, here we use the theorem prover KeY. Finally, the specified program is automatically analysed w.r.t. user-defined information flow policies and other specifications (usually preconditions) using KEG to create JUnit tests helping to demonstrate discovered leaks as well as serving for regression tests.

7.3.2 From Privacy to Noninterference

Our case study is a modified, extended version of the e-voting case study introduced in [74, 75]. In order to prove the cryptographic privacy of the votes of honest voters, the authors constructed a cryptographic privacy game formulated in Java. In that game, the environment (the adversary) can provide two vectors $\overline{c_0}$ and $\overline{c_1}$ of choices of voters such that the two vectors yield the same result according to the counting function, otherwise the game is stopped immediately. Afterwards, the voters vote according to $\overline{c_b}$, where b is a secret bit. The adversary tries to distinguish whether the voters voted according to $\overline{c_0}$ or to $\overline{c_1}$. If they succeed, the cryptographic privacy property is broken. By defining this game, instead of proving the cryptographic privacy property of the complex e-voting system sElect, the authors of [75] prove the noninterference property of its ideal simplified counterpart, which states that there is no information flow from secret bit b to the public result on the bulletin board. It states that if the voting machine computes the result correctly, then this result is independent of whether the voters voted according to $\overline{c_0}$ or $\overline{c_1}$.

We re-implement the simplified version of the e-voting system in [74] by a slightly more complicated version in which the system can handle an arbitrary number of candidates rather

than only two. Listing 7.5 depicts the core of our case study program that includes two classes: `Result` (that has been proposed in Section 7.2.1) wraps the result of the election and `SimplifiedEvoting` reproduces the privacy game mentioned in [75]. Class `Result` has one public integer array field `bulletin`, where `bulletin[i]` stores the number of votes for candidate i. Class `SimplifiedEVoting` has the following fields: a private logic variable `secret` as the secret bit, an integer variable `n` representing the number of candidates indexed by n consecutive integer number from 0 to $n-1$; two integer arrays `votesX`, `votesY` as two vectors of votes supplied by the adversary, where each array's element i is an integer number j (ideally $0 \leq j \leq n-1$) which mean that voter i votes for candidate j; and finally the public variable `Result` that can be observed by the adversary. Method `privacyGame` of class `SimplifiedEvoting` mimics the process that the result is computed using one of two vectors of votes based on the value of the secret bit. Method `compute` of class `SimplifiedEvoting` computes the result of the election using the corresponding vector of votes passed as its parameter. Line 7 is the noninterference policy claiming that there is no information flow from `secret` to `result`. To deal with this object-sensitive noninterference policy, we implement the approach introduced in [18]. We experiment using our approach on two versions of `compute`: one is a correct implementation, while the other is faulty.

The precondition of method `privacyGame` is depicted in Listing 7.6, enforcing that two vectors of votes (`votesX` and `votesY`) have the same size and produce the same result before `privacyGame` is executed. It also makes sure that the number of candidates is greater than 1 and every single vote belongs to one of those candidates.

7.3.3 Leak Detection for Correct Implementation

We first show the result of our approach for the correct implementation of method `compute` as shown in Listing 7.7. It is identical to method `countBallots` of class `CountingServer` in Listing 7.2, except that the vector of votes to be counted is a parameter of `compute` rather than a field. This helps to count both vectors of votes (`votesX` and `votesY`) conveniently.

To check the security of the method `privacyGame`, it is necessary to generate the loop specifications for the loop commands executed in method `compute`. This is performed by employing the approach in the paper [57] that has been implemented in KeY. Firstly, method `privacyGame` is symbolically executed by the KeY tool. The input file is shown in Listing 7.8.

This first step symbolically executes the loop 7 times in total, opens 105 side proofs and needs 148 seconds on a i5-3210M CPU with 6 GB RAM. As the implementation is not optimized for speed, we suppose that it is possible to generate the invariants in significantly less time. The output of the symbolic execution is, besides the proof tree, a file named SimplifiedEVoting.java.mod.0, which contains the Java file with the annotations. In Listing 7.9 the result of the loop invariant generation for the loop in method `compute` is depicted. The invariant is generated by calling the method `compute`, not by calling the method `privacyGame`, because the loop invariant generation is *local*, in the sense that it produces invariants valid under a given precondition. Calling `privacyGame` would produce two invariants, one for each branch, which must be combined using the splitting condition distinguishing them. This may lose precision because the splitting condition may be not fully known, thus the generating call should be to the method containing the loop.

Listing 7.5: Privacy game establishing ballots confidentiality

```
1  public class SimplifiedEVoting {
2    private boolean secret;
3    public Result result;
4    int n; //number of candidates
5    int[] votesX, votesY;
6
7    /*! result | secret ; !*/
8
9    private Result compute(int[] votes){
10     /* implementation of compute  */
11   }
12
13   /*@requires ... @*/
14   public void privacyGame(){
15     if(secret)
16       result = compute(votesX);
17     else
18       result = compute(votesY);
19     }
20   }
21
22   public class Result {
23     public int[] bulletin;//result of votes
24     public Result(int n) {
25       if(n>0)
26         this.bulletin = new int[n];
27       else
28         this.bulletin = null;
29     }
30 }
```

Listing 7.6: Precondition as JML specification of method `privacyGame`

```
1 /*@requires votesX!=null && votesY != null
2     && (votesX.length == votesY.length) && (votesX.length>0) && n>=2
3     && (\forall int j; j>=0 && j<votesX.length;
4     votesX[j]>=0 && votesX[j]<n && votesY[j]>=0 && votesY[j]<n)
5     && (\forall int i; 0 <= i && i < n;
6       (\sum int j; 0 <= j && j < votesX.length; (votesX[j]==i ? 1 : 0))==
7       (\sum int j; 0 <= j && j < votesY.length; (votesY[j]==i ? 1 : 0)));
8     @ diverges true;
9 @*/
```

Listing 7.7: Correct implementation of method `compute`

```
1  private Result compute(int[] votes){
2    Result rs = new Result(n);
3    for(int i = 0; i< votes.length; i++){
4      if(votes[i]>=0 && votes[i]<n)
5        rs.bulletin[votes[i]] = rs.bulletin[votes[i]] + 1;
6    }
7    return rs;
8  }
```

Listing 7.8: Input file `input.key`

```
1  \javaSource ".";
2
3  \programVariables{
4    SimplifiedEVoting2 vt;
5    int[] v;
6    Result2 r;
7  }
8
9  \problem{
10   vt != null &
11   vt.<created> = TRUE &
12   vt.<inv> &
13   vt.n >0 &
14   v != null &
15   v.<created> = TRUE &
16   wellFormed(heap) &
17   (\forall int i; (i >=0 & i <v.length -> (v[i] >= 0 & v[i] <vt.n))) &
18   r = null
19   ->
20   \[{
21     r = vt.compute(v);
22   }\](r != null)
23 }
```

The automatically generated loop invariant in Listing 7.9 is longer and more complex than the loop invariant defined in Listing 7.3. How ever, both loop invariants have the same strength in the sense that all ballots up to the loop counter `i` are guaranteed to be counted correctly.

Listing 7.9: Annotated `SimplifiedEVoting.java.mod.0`

```
1  //@ghost int iter = 0;// AUTO_GENERATED BY KeY
2  /*@ // AUTO_GENERATED BY KeY
3  loop_invariant
4    rs.bulletin == rs.bulletin
5    && i >= 0
6    && i == (\sum int q; 0 <= q & q <iter;1 + 0)
7    &&( \forall int j_27;
8      (   0 <= j_27 & j_27 < rs.bulletin.length
9        ==>   (\sum int q_1; 0 <= q_1 & q_1 <iter; (votes[q_1] == j_27)
10           ? (1 + 0)
11           : (0))
12     == rs.bulletin[j_27]))
13   && (iter >= 0 & iter * 1 == i)
14   && i - votes.length <= 0
15   &&( \forall int j_28;
16     (j_28 < i & 0 <= j_28 ==> votes[j_28] >= 0)
17     );
18   assignable
19   rs.bulletin[*],i;
20 @*/
21
22   for(int i = 0; i< votes.length; i++){
23     //@set iter = iter + 1;// AUTO_GENERATED BY KeY
24     if(votes[i]>=0 && votes[i]<n)
25       rs.bulletin[votes[i]] = rs.bulletin[votes[i]] + 1;
26   }
27   return rs;
28 }
```

In the next step, the file SimplifiedEVoting.java.mod.0 is renamed to SimplifiedEVoting.java and used as input for the KEG tool. KEG finished checking the program w.r.t noninterference policy in 41 seconds on the same system without finding any information flow leak.

7.3.4 Leak Detection for Faulty Implementation

Now we change the implemenation of method `compute` slightly, such that it ignores the first element in the vector of votes when calculating the result. It is obviously an incorrect implementation, in that two vector of votes `votesX`, `votesY` can produce two different results even if the precondition of method `privacyGame` holds. The faulty implementation is given in Listing 7.10.

Listing 7.10: Faulty implementation of method `compute`

```
1  private Result compute(int[] votes){
2    Result rs = new Result(n);
3    for(int i = 1; i< votes.length; i++){ //omit votes[0]
4      if(votes[i]>=0 && votes[i]<n)
5        rs.bulletin[votes[i]] = rs.bulletin[votes[i]] + 1;
6    }
7    return rs;
8  }
```

For this method, the loop invariant generation opens 86 side proofs, executes the loop 7 times in total and needs 161 seconds on a i5-3210M CPU with 6 GB RAM.

The KEG tool finishes checking method `privacyGame` calling the faulty implementation of `compute` in 145 seconds and finds a leak. It reports that there is an implicit information flow leak caused by two different symbolic execution paths branched by the value of `secret`. Using precondition of method `privacyGame` as in Figure 7.6, KEG generates input values for `votesX` and `votesY` in order to demonstrate the leak as follows:

array	element at index								
	0	1	2	3	4	5	6	7	8
votesX	1	2	2	1	1	0	0	1	0
votesY	2	1	1	1	1	0	0	0	2

It is easy to see that the generated values of `votesX` and `votesY` bring the same election result by using the correct version of `compute`, however the results computed by the faulty method `compute` differ. This helps the attacker infer the value of bit `secret` and break the privacy property of the e-voting system.

7.4 Discussion

We chose the simplified e-voting system as case study for our approach for the following reasons: (i) its noninterference property has been verified using a hybrid approach [75] that is not automatic and requires the program to be modified; (ii) it is a sequential Java program having complex features of real-life object oriented programs such as reference types, arrays and object creation; and (iii) the program requires complex specifications containing comprehension sum that challenge both the specification generation tool and the KEG tool.

In contrast to [56, 106] our main interest is not to formally verify that the given program is secure, but to detect *and* to demonstrate the existence of leaks. The case study shows that information erasure can be represented as a generalized noninterference policy and be actually checked by KEG in practice.

We showed that KEG can be applied to object-oriented programs with unbounded loops. KEG was able to generate leak demonstrators that demonstrated violations of the specified information flow policy. The leak demonstrators assisted in identifying and fixing the existing leaks. The fixes were validated by checking that the generated leak demonstrators passed and that KEG was not able to generate any new ones. Except for the provision of the specifications, the approach does not require any user interaction. Specifically, no expert knowledge in logic or theorem proving is required.

Some words on scalability for real-world programs. Our approach is contract-based and thus only one (or very few methods) need to be considered at one time. It is also possible to restrict the analysis to critical modules and thus to reduce the number of required additional specifications like loop invariants. In addition, the approach is more about bug finding than verification, so even simple contracts and loop invariants are useful.

Comprehension expressions like sum, max and min are usually not natively supported by SMT solvers. KEG uses the SMT solver Z3 to solve insecurity formulas. While Z3 is very powerful, it does not natively support comprehension expressions. KEG treats sum in a similar way to the approach proposed in [80], where each sum is translated into a self-contained function characterized by its axioms. The original implementation for the translation of sum (and other comprehension expressions such as max and min) binds each expression to a corresponding function that has two parameters describing the interval. For example, consider the following sum expression in JML syntax:

```
(\sum int i; 0 <= i && i < votes.length; votes[i])
```

This can be translated into a function call `sum_0(0, votes.length-1)`, where `sum_0` is characterized by the following axioms:

$$
\begin{aligned}
&\forall\, x, y \in \{0, 1, .., \texttt{votes.length} - 1\} : \\
&\quad x > y \Rightarrow \texttt{sum\_0}(x, y) = 0 \wedge \\
&\quad x = y \Rightarrow \texttt{sum\_0}(x, y) = \texttt{votes}[x] \wedge \\
&\quad x < y \Rightarrow \texttt{sum\_0}(x, y) = \texttt{votes}[x] + \texttt{sum\_0}(x + 1, y)
\end{aligned}
$$

This translation approach is simple but versatile and can be used for all types of comprehension expressions. The drawback of this approach is that it does not support quantification, i.e. if sum is nested in a universal expression (as shown at lines 3 - 7 in Figure 7.6). To solve this problem, we tailor a new translation approach for sum if it is quantified. We extend the generated sum functions with a parameter representing the quantified variable. For example, following quantified clause in the precondition shown in Figure 7.6:

```
(\forall int i; 0 <= i && i < n;
(\sum int j; 0<=j && j<votesX.length; (votesX[j]==i?1:0))==
(\sum int j; 0<=j && j<votesY.length; (votesY[j]==i?1:0)))
```

can be translated into following expression:

$$
\begin{aligned}
&\forall\, i \in \{0, 1, .., n - 1\} : \\
&\quad \texttt{sum\_1}(0, \texttt{votesX.length} - 1, i) = \texttt{sum\_2}(0, \texttt{votesY.length} - 1, i)
\end{aligned}
$$

The corresponding axioms chracterising `sum_1` and `sum_2` are also added into the insecurity formula. Although this approach allows quantifying over sum expressions (also other comprehensions), it is not suitable for all instances of sum and brings considerable extra workload for the SMT solver. We do believe that there is no one-size-fits-all method translating comprehension expressions to SMT first order formulas that exists and it is necessary to optimize the translation w.r.t. each specific case.

8 PIN Integrity Check Case Study

This chapter showcases the secret inference approach proposed in this thesis along a case study. The software to be analyzed implements an integrity check for Personal Identification Number (PIN) assured by an Automated Teller Machine (ATM). The case study originated in [71] and has been used in [8] to evaluate an approach to transform the problem of quantitative information flow analysis (QIF) w.r.t. non-uniform distribution into the problem of QIF for uniform case. The problem of PIN integrity checking is briefly introduced in Section 8.1. Section 8.2 outlines the program modeling the PIN integrity check in detail. We explain how KEG can be used to extract the value of the PIN in Section 8.3. The case study is concluded with some discussion given in Section 8.4.

8.1 PIN Integrity Check Problem

This section describes the PIN integrity check problem proposed in [8] that in turn is picked up from literature [22, 23, 71, 82]. In a nutshell, when a customer requests a service from an Automated Teller Machine (ATM), she inputs her Personal Identification Number (PIN) that will be sent to the bank to verify. If the ATM is not connected directly to the bank, the PIN must be transmitted through a number of intermediate switches, that might be compromised by the attacker.

To maintain the secrecy (and integrity) of the PIN during the transaction, the ATM will send the PIN as an encrypted PIN block. This block, as described in the ISO standard, is constructed by XORing each digit of the PIN with a digit of the customer's Personal Account Number (PAN). Each digit of PIN is represented by 4 bits [22]. Before sending the PIN block (PIN$\oplus$PAN) to the next switch, it is encrypted using symmetric cryptography. Each pair of adjacent switches shares a common transport key that is used to encrypt and decrypt the PIN block transfered between them.

At each switch, the encrypted PIN block is decrypted and the PIN is extracted by XORing the decrypted PIN block with a given account number. Then the integrity checking is performed to inspect whether the PIN has been changed during the transaction. After that, the PIN block is encrypted and sent to the next switch. Those tasks are performed within the Hardware Security Modules (HSMs) that are tamper-resistant cryptographic devices [71].

The HSM performs an integrity check by simply checking whether all PIN digits are < 10. In case the PIN is not changed, this check will succeed if the given account number is the customer's PAN.

Although HSMs are designed towards protecting the communication keys and the PINs even if the switch is compromised, they fail to fulfill this goal because the attacker can actually learn the value of the PIN by observing the outcome of the PIN integrity check [34]. The reason is that the PIN integrity check protocol allows an arbitrary account number PAN$'$ to be used, hence the HSM will reveal whether PIN$\oplus$PAN$\oplus$PAN$'$ is a valid PIN or not. Because the value of PAN is not secret, we can assume that the attacker knows that value. If the attacker can compromise the switch, she is able to set a value for PAN$'$ and observe the outcome of the integrity checking

(whether all digit of PINs are < 10). Hence, by choosing different values for PAN$'$ and observing the output of the PIN integrity check, the attacker can learn some information about the PIN.

Because the attacker knows the value of PAN, the integrity check can be considered as a process that takes an input m and reports whether all digits of PIN$\oplus m$ are < 10. Here m can be chosen by the attacker.

8.2 PIN Integrity Check Program

In this case study we only consider the case of a one-digit PIN (PIN ranges from 0 to 9). To enable the KEG tool to be used, we adapt the PIN integrity check for one digit PINs in [8] by a Java program, namely `PINIntegrityCheck`, that is outlined in Listing 8.1.

Listing 8.1: PIN Integrity check program

```
1  public class PINIntegrityCheck {
2    public int m;
3    private int PIN;
4    /*! m | PIN ; !*/
5    /*@ requires 0 <= PIN && PIN <= 9; @*/
6    public void check(){
7      if((m>15)|(m<0))
8        m=-1;
9      else{
10       if((PIN ^ m)<10)
11         m=1;
12       else
13         m=0;
14     }
15   }
16 }
```

Class `PINIntegrityCheck` consists of two integer variables: `PIN` representing the PIN and `m` representing the XOR mask value that can be chosen by the attacker. To keep our setup simple, we assume that the attacker observes the result of PIN integrity checking from the output value of `m`. Therefore, in program `PINIntegrityCheck`, `PIN` is a high variable and `m` is a low variable. The noninterference policy `PIN` $\not\rightsquigarrow$ `m` is specified at line 4.

Method `check` models the PIN integrity checking process. Because PIN is a 4-bit number, the mask value to be used also has four bits. Thus the value of `m` should be in the range $[0, 15]$. In program `PINIntegrityCheck`, this is enforced by a branch condition at line 7.

The PIN integrity check is carried out by simply XORing `PIN` with `m` and compare the result with 10 (line 10). If the check succeeds, the output value of `m` is assigned by 1 (line 11), otherwise 0 (line 13). Before performing the integrity check, the attacker already knows that PIN is an integer number in the range $[0, 9]$. This initial knowledge is specified as the precondition of method `check` at line 5.

8.3 Learning a PIN's value by Performing PIN Integrity Check

To show how the value of a PIN value can be revealed by the PIN integrity check, we will perform KEG on the program `PINIntegrityCheck` with the intend to learn the input value of `PIN`. For each value of `PIN` (ranging from 0 to 9), KEG carries out an adaptive attack (as explained in Chapter 4) to obtain the maximum information about this value. KEG starts with the initial knowledge $0 \leq \texttt{PIN} \leq 9$ that is specified as the precondition of method `check` (Listing 8.1, line 5). The input value of the mask `m` is generated to maximize information leakage that is measured by one of three security metrics based on Shannon, guessing and min entropy.

First, we check how the value of PIN can be learned in case the attacker assumes that the value of PIN has uniform distribution, i.e. the probability $P(\texttt{PIN} = x)$ is $\frac{1}{10}$ for all $0 \leq x \leq 9$. Figure 8.1 shows the number of experiments needed to obtain the maximal knowledge about `PIN`'s value w.r.t. each possible `PIN`'s value from 0 to 9. It can be seen that the number of experiments needed in case the leakage metric is Shannon entropy is the same for all possible PINs as when using guessing entropy. Using min entropy as leakage metric gives different results. Nevertheless, for all three metrics, KEG needs at least two experiments and at most three experiments to achieve the maximal knowledge about the `PIN`'s value.

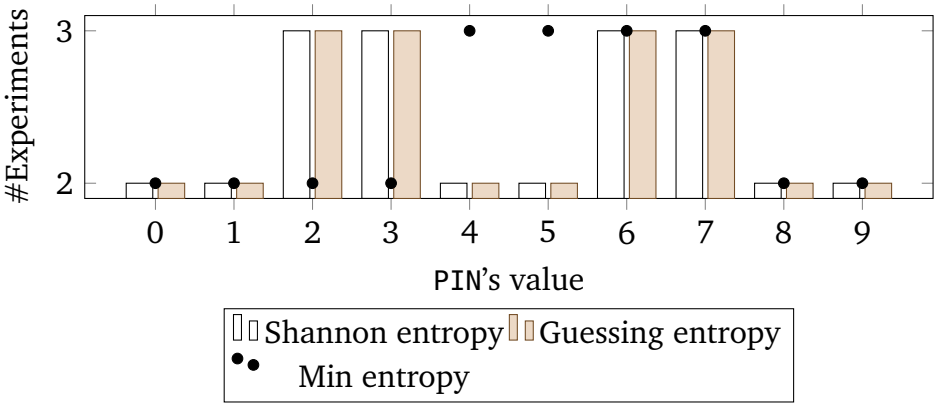


Figure 8.1: Number of experiments needed to achieve maximum knowledge of PIN with uniform PIN distribution

Whenever KEG reaches the maximum knowledge about the `PIN`'s value, it provides a list of concrete values that satisfies the obtained knowledge. Taking a closer look into such concrete values, we see that KEG always provides one of five pairs of the `PIN`'s values: $[0,1]$, $[2,3]$, $[4,5]$, $[6,7]$, $[8,9]$. This coincides with the fact that there does not exist an `m` satisfying that $(\texttt{m} \oplus \texttt{PIN}_1 < 10) \neq (\texttt{m} \oplus \texttt{PIN}_2 < 10)$ where the pair $[\texttt{PIN}_1, \texttt{PIN}_2]$ is one of five above pairs. We call such pairs *indistinguishable pairs*. Figure 8.1 also shows that with respect to each leakage metric, the number of experiments needed to achieve the maximum knowledge of `PIN`'s value for the case the value is $\texttt{PIN}_1$ is equal to the case the value is $\texttt{PIN}_2$ for all indistinguishable pairs $[\texttt{PIN}_1, \texttt{PIN}_2]$.

Now we inspect how the PIN's value can be learned assuming the PINs are non-uniformly distributed (and this distribution is known by the attacker). First, we consider the distribution given in [8]

$$w(\texttt{PIN}) = \begin{cases} 1, & \text{if } 6 \leq \texttt{PIN} \leq 9 \\ 2, & \text{if } 0 \leq \texttt{PIN} \leq 5 \end{cases}$$

The number of experiments needed to achieve the maximum knowledge about a PIN w.r.t. each PIN's value and each leakage metric is shown in Figure 8.2.

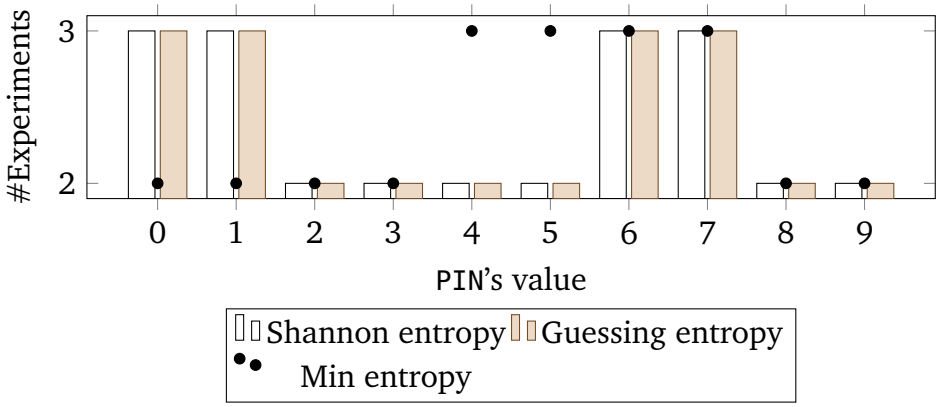


Figure 8.2: Number of experiments needed to achieve maximum knowledge of PIN with non-uniform PIN distribution: $\{\mu(0 \leq \texttt{PIN} \leq 5) = 2, \mu(6 \leq \texttt{PIN} \leq 9) = 1\}$

As shown in Figure 8.2, the number of experiments that KEG needs to provide an indistinguishable pair is almost identical to the case when the PIN's value has uniform distribution, except for two pairs $[0,1]$ and $[2,3]$ where Shannon or guessing entropy is used. Overall, KEG still needs from 2 to 3 experiments to obtain the maximum knowledge of the PIN's value. In this case, the knowledge about the distribution of the PIN's values does not help the attacker to learn the PIN's value more effectively.

We consider another distribution in which the probabilities of the PIN's values are more different than in distribution above. It is given as follows:

$$w(\texttt{PIN}) = \begin{cases} 1, & \text{if } 0 \leq \texttt{PIN} \leq 8 \\ 10, & \text{if } \texttt{PIN} = 9 \end{cases}$$

This distribution states that 9 is the most likely value of the PIN while other values have significantly lower probability. We perform KEG on the program `PINIntegrityCheck` using the new PIN distribution. The number of experiments needed to reach the maximal knowledge about the PIN is depicted in Figure 8.3.

Figure 8.3 shows that for the case PIN=9 (the greatest likelihood), KEG obtains the maximum knowledge about the PIN's value by only one experiment, in case the leakage metric is Shannon or min entropy. The same holds for the case PIN = 8 because $[8,9]$ is an indistinguishable pair. On the other hand, when Shannon or min entropy is used, for some other PIN values whose likelihood is relatively small, KEG needs up to 4 experiments to obtain maximum knowledge about the PIN's value. However, the polarized distribution of PIN's values does not affect the number of needed experiments in case guessing entropy is used, that is identical to the case that the attacker assumes that PIN's value has uniform distribution.

8.4 Discussion and Remark

The PIN integrity check problem has been used in the papers [8, 71]. In these papers, adaptive attacks for a PIN integrity check program are mentioned, but they only consider the uniform

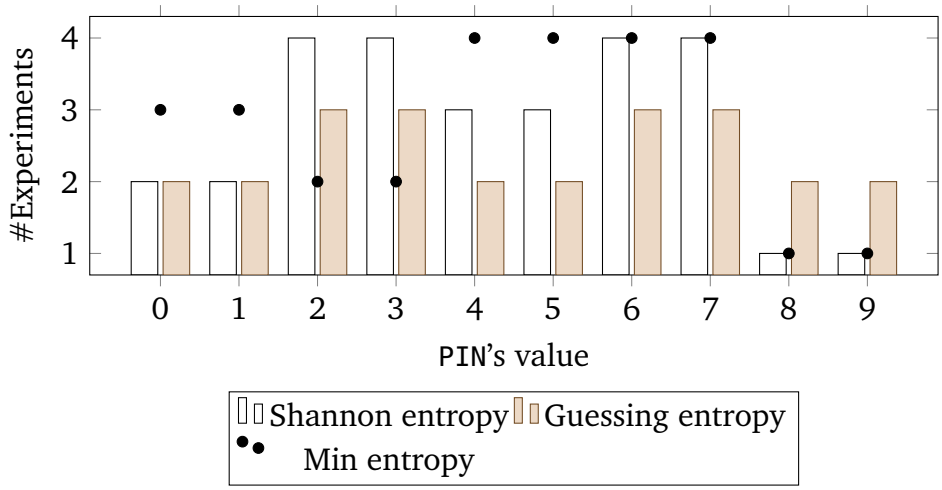


Figure 8.3: Number of experiments needed to achieve maximum knowledge of PIN with non-uniform PIN distribution: $\{\mu(0 \leq \texttt{PIN} \leq 8) = 1, \mu(\texttt{PIN} = 9) = 10\}$

distribution [71] or they reduce the analysis of a non-uniform distribution to the uniform one by using an auxiliary program as the distribution generator [8]. This case study addresses an adaptive attack performed on a PIN integrity check program that directly uses non-uniform distribution of PIN's values. The case study shows that in case of a uniform distribution of the PIN, KEG helps the attacker to learn the maximum information of a PIN's value in at most 3 experiments. This result is identical to the result in [8], that is obtained by using the optimal attack strategy proposed in [70].

In the real world, the distribution of the value of PIN and other passwords is often non-uniform [24], in which some values, e.g. "`1234`" or "`password`", have a higher probability than others. The case study confirms an intuition that the attacker can exploit her knowledge about the distribution of PIN's values to learn the PIN's value more efficiently: the more the attacker knows about the PIN distribution, the less experiments she needs to conduct to achieve the maximum knowledge about the PIN's value.

Program `PINIntegrityCheck`, although small in the size, poses a surprising challenge for the KEG tool due to the XOR operator. KEG uses parametric counting to derive optimization problems to find optimal low input w.r.t. Shannon or guessing entropy-based security metrics. Unfortunately, XOR is a bitwise not integer arithmetic operator, hence to the best of our knowledge, it is not supported by an integer parametric counting tool like Barvinok. We solve this problem by using a bounded translation that converts an XOR expression into an equivalent expression using the conditional operator. For example, assume a and b are 2-bit integer numbers, $a \oplus b$ can be translated to $((a\%2 \doteq b\%2)?0 : 1) + 2((a/2 \doteq b/2)?0 : 1)$. However, the conditional operator is also not supported by the Barvinok tool, thus we have to further split it into two disjuncts (true-case and false-case). Hence the number of disjuncts is exponential in the number of conditional operators. For the case study program `PINIntegrityCheck`, both the PIN and the XOR mask are 4-bits integer value, hence each XOR operator leads to four conditional expressions and consequently, 16 disjuncts in the parametric counting formula. For the case of 4-digits PIN (PIN has 16 bits), it leads to $16^4 = 65536$ disjuncts that overwhelm the parametric counting tool Barvinok.

Luckily, the min entropy as a security metric and the Max-SMT based approach do not suffer from the same above problem about the XOR operator. The Max-SMT problem used to find

optimal low input can be encoded using the bit vector theory that supports XOR. We examine KEG with the case of 4-digits PIN and a PIN distribution having 16 partitions, KEG successfully determines the optimal mask value and the corresponding maximum min entropy-based leakage in a few seconds. It turns out that min-entropy and Max-SMT could be a promising approach applicable for large programs.

9 Related Work

This chapter compares our work with related work in the literature in two main areas: leak detection and demonstrator generation (Section 9.1), quantitative information flow analysis and secret inference (Section 9.2). The content of this chapters is inherited and revised from previous publications/technical reports of the author of this thesis [46, 47, 48, 49].

9.1 Related to Leak Detection and Demonstrator Generation

Our approach to detect information leaks and generate leak demonstrators is based on self-composition [17, 38, 39]. Paper [39] addresses also declassification. Its authors observe that in their formalization it is possible to express and verify that a program is insecure. Our formalization of insecurity uses this observation. Our leak detection and demonstrator generation approach follows techniques that were first explored in automatic test generation. In particular, we build on work presented in [2, 40, 51, 68], where symbolic execution is used as a means to generate test cases for functional properties.

Deductive approaches to information flow analysis [18, 107] are fully precise and at the same time can flexibly express various information flow properties beyond the policies presented in this paper. The verification process is not fully automatic, however, and non-trivial interactions with a theorem prover are required. This restricts usability of these approaches seriously. In [92] higher-order logic is used to express information flow properties for object-oriented programs, which is highly expressive, but imposes even higher demands on user expertise.

Pairs of symbolic execution paths to improve the efficiency of self-composition have been independently introduced in [97] to check programs for noninterference. However, that paper focuses on checking noninterference and does not support declassification. Unbounded loops and recursive methods are not handled either.

In [115], leaks are inferred automatically and expressed in a human-readable security policy language, helping programmers to decide whether the program is secure or not, however, they cannot give concrete counter examples that could suggest further corrections. Counter examples can be used not only to generate executable exploits as in our approach, but also to refine declassification policies by quantifying the leakage [9, 11]. However, none of these approaches provides a solution for unbounded loops and recursion.

ENCoVer [10] uses epistemic logic and makes use of symbolic execution (concolic testing) to check noninterference for Java programs. In [88], the authors proposed a tool which checks that a C program is secure with respect to noninterference. It transforms the original program and makes use of dynamic symbolic execution to analyze the program's information flow. Both tools check loops and recursive method invocations only up to a fixed depth.

Type-based approaches to information flow like [63, 91, 103, 117] and those based on dependency graphs [54, 55] distinguish themselves by their high performance and ability to check large systems. Their common drawbacks are a lack of precision with a resulting high number of false positives and restrictions on the syntactic form of programs.

None of the logic-based and type-based approaches to noninterference analysis mentioned above does generate leak demonstrators from a failed proof or analysis. Our work does not intend to replace these approaches, but is intended to be used complementary, just like testing complements formal verification.

9.2 Related to Quantitative Information Flow Analysis and Secret Inference

An information-theoretic model for an adaptive side-channel attack is proposed in [70]. The idea of the attacker strategy is to choose at each step the query that minimizes the remaining entropy. Even though this greedy heuristic attack strategy is similar to our guided experiments approach, it requires to enumerate all possible queries to choose the best one, which is rather expensive. Our approach differs in the sense that we quantify the potential leakage as a function of low input, and hence, we can make use of many available, efficient optimization tools to find the optimal input value. Furthermore, our work takes into account non-uniform distribution of secret, while the paper [70] only considers uniform case.

Pasareanu et al. [96] propose a non-adaptive side-channel attack to find low input that maximizes the amount of leaked information. To find optimal low input, they solve a number of Max-SMT problems whose formalisation is based on path conditions and user-defined cost models. In contrast to our approach, only path conditions are considered, but not symbolic states. Hence, they cannot measure leakage caused by explicit information flow. The authors of [60] define a *quantitative policy* which specifies an upper bound for permitted information leakage. The model checker CBMC is used to generate low input that triggers a violation of the policy. Both of [60, 96] use channel capacity as their leakage metric which is the worst case over all *prior distributions* over high inputs. Low input is generated with the aim to maximize the number of equivalence classes on high inputs. The size of the individual class is not taken into account. Hence, they are less precise than our approach that takes into account prior distributions. Their generated low input often is not the optimal one: for example, in case of Listing 6.4, we are able to generate a sequence of low inputs for $\mathtt{l}$, each of which extracts nearly 1 bit of information, allowing to find the exact secret after 31 experiments. Their approach can only return a single, *arbitrary* input for $\mathtt{l} \in (-2^{31}, 2^{31})$, hence, using it for an attack would not perform better than brute force (see discussion in Section 6.3.2). Both approaches require a bound on the number of loop iterations or the recursion depth, whereas we can make use of specifications to deal with unbounded loops and recursion.

Low input as a parameter of quantitative information flow (QIF) analysis is also addressed in [94, 119]. In [119], the authors only analyze the bounding problem of QIF for low input, but do not provide a method to determine a bound for the leakage. The authors of [94] model the program with low input as a set of information channels, where each channel corresponds to a specific value of low input. While considering that the leakage depends on low input, they do not discuss how to find the input maximizing the leakage.

Symbolic execution as a static analysis technique is used in several quantitative information flow analyses [69, 98]. In [69] a precise quantitative information flow analysis based on calculating cardinalities of equivalent classes is presented. The approach is first applied for uniform distribution and extended for the case of non-uniform one. The author assumes an optimally chosen set of experiments, but does not describe how to construct such a set.

The authors of [32, 33] model the attacker's knowledge about the secret as belief and show how to update the knowledge after each experiment. In [9], the authors briefly discuss the cor-

relation between the set of experiments and an attacker's knowledge about the secret. However, none of these papers describes how to construct an optimal experiment set that maximizes the leakage. Other approaches in quantitative information flow [81, 86, 98, 110] do not address low input in their analyses and consider only channel capacity with the same drawbacks as discussed earlier.

Finding low input maximizing the leakage is investigated in few works, but they either only use channel-capacity as leakage metric [60, 96] or only take uniform distribution of high input into account [70]. On the other hand, information leakage with non-uniform secret distribution has been studied in many works [3, 4, 8, 25, 26, 69, 89], but none of them quantifies the leakage as an explicit function of low input or shows how to find low input maximizing the leakage. Details of those works will be discussed in the next paragraphs.

Backes et. al [8] propose a technique representing non-uniform distribution in terms of a generator program that received uniform distribution input and produces output according to the desired distribution. With this technique they can reduce a problem of QIF w.r.t. non-uniform distribution to a problem of QIF w.r.t. uniform distribution. However, their approach does not take low input into account.

Mu and Clark [89] introduce an interval-based abstraction method that transforms a domain equipped by an arbitrary, explicit non-uniform distribution to a set of partitions each of which has uniform distribution. It allows them to mitigate the restriction of their previous approach [90] that requires an explicit representation of the probability distribution thereby being hard to apply for a large state space. Both works are based on a distribution transformer semantics and compute the upper bound on the leakage. While allowing quantifying information leakage w.r.t. non-uniform distribution, those works only deal with a simple while-language and do not take into account low input.

Gain function [3] and *worth-assignment* [4] are introduced to extend and generalize current security metrics. Those metrics not only enrich the security's semantics but also affect the attacker's strategies in guessing a secret. Boreale and Pampaloni [26] prove that the maximum information leakage over attacking strategies under generic leakage function and an adaptive attacker can be expressed in term of a Bellman equation that can be used to compute the optimal finite strategies recursively. In [25], the authors model the gain function by a pair of *cost* and *reward* assigned for each set of secret corresponding to an attacker's guess. By using Bayes decision theory, the optimal action that maximizes the overall expected gain can be derived. All of those approaches, although dealing with arbitrary secret distribution, only consider the program as the channel between secret input and observable output, and hence the role of low input is omitted.

10 Conclusion and Future Work

10.1 Conclusion

Although many information flow analysis approaches have been proposed, there are still some remaining challenges: i) achieving full automation and precision in static qualitative information flow analysis, ii) path coverage and the requirement of runtime infrastructure in dynamic qualitative information flow analysis, iii) quantifying leakage with low input and non-uniform distribution of secrets in quantitative information flow analysis. In addition, to the best of our knowledge, exploiting information flow leaks to infer the secret have not been investigated very much in language-based information flow analysis. This thesis has complemented other existing approaches in the field of language-based information flow analysis by introducing a number of novel approaches combining static and dynamic analysis. Such approaches allow one to detect, exploit and judge the severity of information flow leaks in programs. We summarize the work of this thesis in following paragraphs.

To detect (as much as possible) information leaks in a program w.r.t. an information flow policy, this thesis proposed a static approach based on self-composition and symbolic execution. We formalize the violations of a given information flow policy in the form of *insecurity formulas* composed from pairs of symbolic execution paths of the program. SMT solvers are used to check the satisfiability of such formulas to discover possible information leaks.

Unlike other logic-based approaches, the leak detection approach proposed in this thesis does not aim to prove that a program is secure/insecure, but tries to find all potential leaks. Under the assumption that the symbolic execution tree of a target program is correct and finite, our approach is fully automatic and precise: it can formally conclude whether the program is secure without any user interactions. For the case of an infinite symbolic execution tree, due to unbounded loops or recursive method calls, our approach uses program specifications, i.e. loop invariants or method contracts, for composing insecurity formulas. This allows the approach to be fully precise (if needed) but also to abstract (and allow for false positives) in exchange for a higher degree of automation and simpler specifications. The approach supports several information flow security policies, namely, noninterference, delimited information release and information erasure.

To verify detected leaks and avoid false alarms, we introduced the concept of *leak demonstrator*. A leak demonstrator is a program exposing an information flow leak in the target program and terminates with an assertion failure if such a leak has been detected. A leak demonstrator can be seen as a test case for secure information flow where the test oracle is the assertion that the low outputs of two runs are equal. A leak demonstrator confirms an *actual* leak if this assertion fails, otherwise it is a false alarm. One nice thing about leak demonstrators is that they do not only confirm information leaks and filter false alarms, but that they can also be used for regression testing to avoid the reintroduction of a fixed leak.

To judge the severity of information leaks, this thesis proposed an approach that exploits detected leaks to infer the secrets of a program. It basically conducts an adaptive attack in which

the attacker tries to infer the secret by performing a series of *experiments*. Each experiment is carried out as follows: the attacker chooses the input values of low variables, then runs the program (initialized by secret high input and chosen low input), and observes the output values. The execution of a set of experiments accumulates *knowledge* about a secret in the form of a first order formula that constrains the possible values of the secret. The presented approach combines static and dynamic aspects into a hybrid analysis. Leak detection and quantification of a leak are achieved by static analysis, while the knowledge accumulation depends on concrete program runs. The approach allows developers to judge the severity of an information leak by measuring how much information is leaked and by actually demonstrating that (some parts of) the secret can be inferred.

The secret inference approach is designed to be optimal: it aims to extract a maximum of information about the secret with as few experiments as possible. Within this thesis, several optimizations have been investigated that allow one to avoid redundant experiments and even to determine when the knowledge accumulation is saturated. We explained in detail how to generate optimal low inputs for the experiments in two steps: (i) quantifying low input-parameterized leakages and (ii) solving the resulting optimization problems. The approach supports the security metrics Shannon, guessing and min entropy. It also supports arbitrary prior distributions of high input values. Symbolic execution and parametric model counting are used to compute a leakage representation as a function of low inputs. The resulting optimization problems are solved with help of (non-linear) optimization tools or Max-SMT solvers. Although there are still some drawbacks in scalability, the proposed approaches improve significantly upon the current state-of-the-art.

The proposed approaches have been implemented and integrated into the tool *KeY Exploit Generation* (KEG). The KEG tool is built on top of the deductive verification system KeY and makes use of KeY as symbolic execution engine. This symbolic execution framework allows KEG to use program specifications, e.g. loop invariants and method contracts to support unbounded loops and recursive method calls. Except for floating points, reflection and generic types, KEG supports full sequential Java, i.e. primitive and reference types (objects and arrays), object creation and aliasing. The information flow policies are specified as part of the source code embeded as special Java comments. Besides class-level (generalized) noninterference policies, we introduced new JML-style keywords to embed method-level declassification policies into method contracts.

With a Java program annotated with information flow policies and other auxiliary specifications i.e. method contracts and loop invariants, KEG can *automatically* detect all possible information leaks for which corresponding leak demonstrators as JUnit tests are generated. To infer as much information as possible about a program's secret data, KEG generates an executable Java program, which performs an adaptive attack. KEG allows the user to define the prior distribution and to specify the maximal number of experiments to be carried out. KEG uses the parametric counting tool *Barvinok*, the Max-SMT solver Z3 and a three optimizers: *Local Solver*, *Bonmin*, *Couenne* to quantify the leak and find optimal low input.

The approaches proposed in this thesis as well as the KEG tool have been applied to a collection of micro benchmarks. KEG proved to be successful in leak detection and leak demonstrator generation. Further, it showed that using its secret inference capabilities allowed to accumulate knowledge about the secret data of a program more efficiently than with a brute force attack.

Besides micro benchmarks, KEG was used in two case studies known from the literature. The first one considers an e-voting system that challenges the leak detection and demonstra-

tor generation approach. We used KEG to check a simplified e-voting program adapted from the real-world e-voting system sElect. The aim was to find all possible information leaks and to demonstrate them by executing the generated leak demonstrators. The e-voting case study showed that KEG is able to deal with relatively complicated programs including unbounded loops, objects and arrays. Moreover, we also showed that KEG can be integrated with a specification generation tool to obtain both high precision and full automation. The second case study used KEG to infer secret PIN numbers by observing the output of a PIN integrity checking program. The PIN integrity checking program was adapted from real-world system. The case study showed that KEG can make use of assumptions about prior distribution to infer secrets significantly more efficient than when just assuming a uniform distribution (if the assumed prior distribution is close enough to the acutal one).

In conclusion, this thesis contributes to the field of language-based information flow analysis with a number of precise semi-automatic approaches that allow one to detect, exploit and judge the severity of information flow leaks in programs. Those approaches have been implemented in the KEG tool and evaluated by a number of small benchmarks. Case studies showcasing the practical application of KEG have also been provided.

10.2 Future Work

Path explosion is a well-known bottle neck for all symbolic execution-based techniques, including the approach proposed in this thesis. Using (correct and strong) program specifications in leak detection can significantly reduce the total number of symbolic paths. However, manually supplying necessary specifications could be a tough task. The case study in Section 7.3 shows that specifications automatically generated by a specification generation approach [57, 118] can be used effectively by our approach (implemented in the KEG tool) for detecting information flow leaks. Because the methods in [57, 118] have been implemented in the KeY system, it is feasible to integrate them with KEG to achieve full automation in a transparent manner: required specifications will be automatically generated for leak detection and secret inference on demand.

One promising approach for mitigating path explosion is state merging. In [108], the authors proposed an approach that makes use of an abstract domain lattice to merge two symbolic states satisfying some specific conditions. This approach has also been implemented in KeY from which KEG can benefit, at least when using the *If-Then-Else* merging technique. Further research about utilizing information flow polices i.e. noninterference or delimited information release to optimize the state merging could be interesting and worth to come up with.

The approach finding optimal low inputs in this thesis is precise, but has some limitations in scalability as discussed in Section 5.3. Examples and small benchmarks for secret inference approach proposed in this thesis show that generally, “good enough” low input is still very efficient for finding the secret. To make the secret inference approach feasible for real world programs, it is crucial to devise some approximation and heuristic techniques to obtain a good tradeoff between precision and scalability.

The complexity of high inputs distribution is another source of computational overhead of leakage quantification. The paper [8] proved that the leakage is robust with respect to small variations in the distribution of high inputs. Simplifying supplied high inputs distribution, therefore, seems to be a promising way to enhance the performance of secret inference.

At the moment, our secret inference approach only consider the scenario that the attacker does not revise his assumption about the prior distribution of high inputs after observing the outputs. It could be very interesting to incorporate the concept of *belief* [32, 33] into our approach: the attacker's knowledge about the secret could be not only the constraint of the secret but also its prior distribution that can be updated after each experiment. This enables us to model a more general attack scenario that can capture a practical use case: the attacker does not completely trust his assumption about the secret's distribution and needs to revise it using the observed values.

On the other hand, in practice the attacker might try to infer only the worthiest part of a secret instead of the whole secret value. Incorporating *gain function* [3] and *worth-assignment* [4] seems to be a potential dimension to extend and generalize our secret inference approach to capture that attack scenario.

Bibliography

[1] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification—The KeY Book: From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, Dec. 2016.

[2] E. Albert, M. Gomez-Zamalloa, and G. Puebla. PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 25–28. ACM Press, 2010.

[3] M. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring Information Leakage Using Generalized Gain Functions. In *Computer Security Foundations Symp. (CSF), 2012 IEEE 25th*, pages 265–279, June 2012.

[4] M. S. Alvim, A. Scedrov, and F. B. Schneider. When Not All Bits Are Equal: Worth-Based Information Flow. In M. Abadi and S. Kremer, editors, *Principles of Security and Trust*, volume 8414 of *LNCS*, pages 120–139. Springer, 2014.

[5] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 134–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] J. M. Anderson. Why we need a new definition of information security. *Computers & Security*, 22(4):308 – 313, 2003.

[7] A. Askarov, S. Chong, and H. Mantel. Hybrid Monitors for Concurrent Noninterference. In C. Fournet, M. W. Hicks, and L. Viganò, editors, *IEEE 28th Computer Security Foundations Symp., CSF, Verona, Italy*, pages 137–151. IEEE Computer Society, 2015.

[8] M. Backes, M. Berg, and B. Köpf. Non-Uniform Distributions in Quantitative Information-Flow. In B. S. N. Cheung, L. C. K. Hui, R. S. Sandhu, and D. S. Wong, editors, *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, pages 367–375. ACM, 2011.

[9] M. Backes, B. Köpf, and A. Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *2009 30th IEEE Symposium on Security and Privacy*, pages 141–153, May 2009.

[10] M. Balliu, M. Dam, and G. Le Guernic. ENCoVer: Symbolic Exploration for Information Flow Security. In *25th IEEE Computer Security Foundations Symposium*, pages 30–44. IEEE CS, 2012.

[11] A. Banerjee, R. Giacobazzi, and I. Mastroeni. What You Lose is What You Leak: Information Leakage in Declassification Policies. *Electron. Notes Theor. Comput. Sci.*, 173:47–66, Apr. 2007.

[12] A. Banerjee and D. A. Naumann. Stack-based Access Control and Secure Information Flow. *J. Funct. Program.*, 15(2):131–177, 2005.

[13] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[14] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[15] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. *Secure Multi-execution Through Static Program Transformation*, pages 186–202. FMOODS'12/FORTE'12. Springer-Verlag, Berlin, Heidelberg, 2012.

[16] G. Barthe, J. M. Crespo, and C. Kunz. Relational Verification Using Product Programs. In *Proc. of the 17th Intl. Conf. on Formal Methods*, FM'11, pages 200–214. Springer, 2011.

[17] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In *Proc. of the 17th IEEE Workshop on Computer Security Foundations*, CSFW '04, pages 100–114. IEEE CS, 2004.

[18] B. Beckert, D. Bruns, V. Klebanov, C. Scheben, P. H. Schmitt, and M. Ulbrich. Information Flow in Object-Oriented Software. In G. Gupta and R. Peña, editors, *Logic-Based Program Synthesis and Transformation: 23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers*, pages 19–37, Cham, 2014. Springer International Publishing.

[19] B. Beckert and R. Hähnle. Reasoning and Verification. *IEEE Intelligent Systems*, 29(1):20–29, Jan.–Feb. 2014.

[20] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer, 2007.

[21] T. Benoist, B. Estellon, F. Gardi, R. Megel, and K. Nouioua. LocalSolver 1.x: a black-box local-search solver for 0-1 programming. *4OR*, 9:299–316, 2011.

[22] O. Berkman and O. M. Ostrovsky. The Unbearable Lightness of PIN Cracking. In S. Dietrich and R. Dhamija, editors, *Financial Cryptography and Data Security: 11th International Conference, FC 2007, and 1st International Workshop on Usable Security, USEC 2007, Scarborough, Trinidad and Tobago, February 12-16, 2007. Revised Selected Papers*, pages 224–238, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[23] M. Bond and P. Zieliński. Decimalisation table attacks for PIN cracking. Technical Report 560, University of Cambridge - Computer Laboratory, February 2003.

[24] J. Bonneau. *Guessing human-chosen secrets*. PhD thesis, University of Cambridge, May 2012.

[25] M. Boreale and F. Corradi. Searching secrets rationally. *International Journal of Approximate Reasoning*, 69:133 – 146, 2016.

[26] M. Boreale and F. Pampaloni. Quantitative information flow under generic leakage functions and adaptive adversaries. *Logical Methods in Computer Science*, 11(4), 2015.

[27] M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Compositional Safety Verification with Max-SMT. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 33–40, Austin, TX, 2015. FMCAD Inc.

[28] D. Bruns, H. Q. Do, S. Greiner, M. Herda, M. Mohr, E. Scapin, T. Truderung, B. Beckert, R. Küsters, H. Mantel, and R. Gay. Security in E-Voting. Poster presented at the 36th IEEE Symposium on Security and Privacy (S&P), 2015.

[29] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[30] P. Cerný, K. Chatterjee, and T. A. Henzinger. The Complexity of Quantitative Information Flow Problems. In *CSF*, pages 205–217. IEEE Computer Society, 2011.

[31] D. Clark, S. Hunt, and P. Malacaria. A Static Analysis for Quantifying Information Flow in a Simple Imperative Language. *J. Comput. Secur.*, 15(3):321–371, 2007.

[32] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in Information Flow. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18), Aix-en-Provence, France*, pages 31–45. IEEE Computer Society, 2005.

[33] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 17(5):655–701, 2009.

[34] J. Clulow. The Design and Analysis of Cryptographic Application Programming Interfaces for Security Devices. Master's thesis, University of Natal, Durban, South Africa, 2003.

[35] E. S. Cohen. Information Transmission in Sequential Programs. *Foundations of Secure Computation*, pages 297–335, 1978.

[36] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[37] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Provably correct inline monitoring for multithreaded Java-like programs. *Journal of Computer Security*, 18(1):37–59, 2010.

[38] A. Darvas, R. Hähnle, and D. Sands. A Theorem Proving Approach to Analysis of Secure Information Flow. In R. Gorrieri, editor, *Workshop on Issues in the Theory of Security*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

[39] Á. Darvas, R. Hähnle, and D. Sands. A Theorem Proving Approach to Analysis of Secure Information Flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing: Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005. Proceedings*, pages 193–209, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[40] J. de Halleux and N. Tillmann. Parameterized Unit Testing with Pex. In B. Beckert and R. Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *LNCS*, pages 171–181. Springer, 2008.

[41] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[42] F. Del Tedesco, S. Hunt, and D. Sands. A Semantic Hierarchy for Erasure Policies. In S. Jajodia and C. Mazumdar, editors, *Information Systems Security: 7th Intl. Conf., ICISS, Kolkata, India*, pages 352–369. Springer, 2011.

[43] D. E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, May 1976.

[44] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[45] D. Devriese and F. Piessens. Noninterference through Secure Multi-execution. In *31st IEEE Symp. on Security and Privacy, S&P, Berleley/Oakland, USA*, pages 109–124. IEEE Computer Society, 2010.

[46] Q. H. Do, R. Bubel, and R. Hähnle. Inferring Secrets by Guided Experiments. Technical report, TU Darmstadt, Feb. 2017. Available at `https://www.se.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_SE/Publications/ALBIA/SecretInference.pdf`.

[47] Q. H. Do, R. Bubel, and R. Hähnle. Exploit Generation for Information Flow Leaks in Object-Oriented Programs. In H. Federrath and D. Gollmann, editors, *ICT Systems Security and Privacy Protection*, volume 455 of *IFIP Advances in Information and Communication Technology*, pages 401–415. Springer, 2015.

[48] Q. H. Do, R. Bubel, and R. Hähnle. Automatic detection and demonstrator generation for information flow leaks in object-oriented programs. *Computers & Security*, 67:335 – 349, 2017.

[49] Q. H. Do, E. Kamburjan, and N. Wasser. Towards Fully Automatic Logic-Based Information Flow Analysis: An Electronic-Voting Case Study. In F. Piessens and L. Viganò, editors, *Principles of Security and Trust, 5th Intl. Conf., POST, Eindhoven, The Netherlands*, volume 9635 of *LNCS*, pages 97–115. Springer, 2016.

[50] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 737–744, Cham, 2014. Springer International Publishing.

[51] C. Engel and R. Hähnle. Generating Unit Tests from Formal Proofs. In B. Meyer and Y. Gurevich, editors, *Proc. of Tests and Proofs*, volume 4454 of *LNCS*, pages 169–188. Springer, 2007.

[52] R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[53] D. M. Gay. The AMPL Modeling Language: An Aid to Formulating and Solving Optimization Problems. In M. Al-Baali, L. Grandinetti, and A. Purnama, editors, *Numerical Analysis and Optimization: NAO-III, Muscat, Oman, January 2014*, pages 95–116, Cham, 2015. Springer International Publishing.

[54] J. Graf, M. Hecker, and M. Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Proc. of the 6th Working Conf. on Programming Languages*, LNI 215, pages 123–138. Springer, Feb. 2013.

[55] J. Graf, M. Hecker, M. Mohr, and G. Snelting. Tool Demonstration: JOANA. In F. Piessens and L. Viganò, editors, *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9635 of *Lecture Notes in Computer Science*, pages 89–93. Springer Berlin Heidelberg, 2016.

[56] D. Grahl. *Deductive Verification of Concurrent Programs and its Application to Secure Information Flow for Java*. PhD thesis, Karlsruhe Institute of Technology, Oct. 2015.

[57] R. Hähnle, N. Wasser, and R. Bubel. Array Abstraction with Symbolic Pivots. In E. Ábrahám, M. Bonsangue, and E. B. Johnsen, editors, *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *LNCS*, pages 104–121. Springer, 2016.

[58] C. Hammer and G. Snelting. Flow-sensitive, Context-sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. *Int. J. Inf. Secur.*, 8(6):399–422, Oct. 2009.

[59] M. Hentschel, R. Hähnle, and R. Bubel. Visualizing Unbounded Symbolic Execution. In M. Seidl and N. Tillmann, editors, *Tests and Proofs: 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings*, pages 82–98, Cham, 2014. Springer International Publishing.

[60] J. Heusser and P. Malacaria. Quantifying Information Leaks in Software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 261–269, New York, NY, USA, 2010. ACM.

[61] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[62] C. Hsieh, E. A. Unger, and R. A. Mata-Toledo. Using program dependence graphs for information flow control. *Journal of Systems and Software*, 17(3):227 – 232, 1992.

[63] S. Hunt and D. Sands. On Flow-Sensitive Security Types. In *ACM SIGPLAN Notices*, volume 41, pages 79–90. ACM, 2006.

[64] S. Hunt and D. Sands. Just Forget it – The Semantics and Enforcement of Information Erasure. In *Programming Languages and Systems. 17th European Symposium on Programming, ESOP 2008*, number 4960 in LNCS, pages 239–253. Springer Verlag, 2008.

[65] P. V. Huong, T. M. Tuan, D. Q. Huy, L. H. Trang, V. T. Nhan, N. N. Binh, T. A. Hoang, and V. Q. Dung. Some Approaches to Nôm Optical Character Recognition. *VNU Journal of Science, Natural Sciences and Technology*, 24(3S):90–99, October 2008.

[66] D. Q. Huy, T. A. Hoang, and N. N. Binh. Extending Crest with Multiple SMT Solvers and Real Arithmetic. In *Proceedings of the 2010 Second International Conference on Knowledge and Systems Engineering*, KSE '10, pages 183–187, Washington, DC, USA, 2010. IEEE Computer Society.

[67] M. Jaskelioff and A. Russo. Secure Multi-execution in Haskell. In E. Clarke, I. Virbitskaite, and A. Voronkov, editors, *Perspectives of Systems Informatics: 8th International Andrei Ershov Memorial Conference, PSI 2011, Novosibirsk, Russia, June 27-July 1, 2011, Revised Selected Papers*, pages 170–178, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[68] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

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

[70] B. Köpf and D. Basin. An Information-theoretic Model for Adaptive Side-channel Attacks. In *Proc. of the 14th ACM Conf. on Computer and Communications Security*, CCS '07, pages 286–296. ACM, 2007.

[71] B. Köpf and D. Basin. Automatically Deriving Information-theoretic Bounds for Adaptive Side-channel Attacks. *J. Comput. Secur.*, 19(1):1–31, Jan. 2011.

[72] B. Köpf and A. Rybalchenko. Approximation and Randomization for Quantitative Information-Flow Analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 3–14, Washington, DC, USA, 2010. IEEE Computer Society.

[73] L. Kovács. Symbolic Computation and Automated Reasoning for Program Analysis. In E. Ábrahám and M. Huisman, editors, *Integrated Formal Methods, 12th Intl. Conf., IFM, Reykjavik, Iceland*, volume 9681 of *LNCS*, pages 20–27. Springer, 2016.

[74] R. Küsters, T. Truderung, B. Beckert, D. Bruns, J. Graf, and C. Scheben. A Hybrid Approach for Proving Noninterference and Applications to the Cryptographic Verification of Java Programs. In *Grande Region Security and Reliability Day 2013*, 2013. Extended Abstract.

[75] R. Küsters, T. Truderung, B. Beckert, D. Bruns, M. Kirsten, and M. Mohr. A Hybrid Approach for Proving Noninterference of Java Programs. In C. Fournet and M. Hicks, editors, *28th IEEE Computer Security Foundations Symposium*, 2015.

[76] R. Küsters, T. Truderung, and A. Vogt. Verifiability, Privacy, and Coercion-Resistance: New Insights from a Case Study. In *32nd IEEE Symp. on Security and Privacy, S&P, Berkeley, CA, USA*, pages 538–553, 2011.

[77] R. Küsters, J. Müller, E. Scapin, and T. Truderung. sElect: A Lightweight Verifiable Remote Voting System. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 341–354, June 2016.

[78] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[79] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. JML Reference Manual, 2013. draft, revision: 2344.

[80] K. R. M. Leino and R. Monahan. Reasoning About Comprehensions with First-order SMT Solvers. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 615–622, New York, NY, USA, 2009. ACM.

[81] P. Malacaria and H. Chen. Lagrange Multipliers and Maximum Information Leakage in Different Observational Models. In *Proc. of the 3rd ACM SIGPLAN Workshop on Prog. Languages and Analysis for Security*, PLAS '08, pages 135–146. ACM, 2008.

[82] M. Mannan and P. C. van Oorschot. Weighing Down "The Unbearable Lightness of PIN Cracking". In G. Tsudik, editor, *Financial Cryptography and Data Security: 12th International Conference, FC 2008, Cozumel, Mexico, January 28-31, 2008. Revised Selected Papers*, pages 176–181, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[83] H. Mantel and H. Sudbrock. Types vs. PDGs in Information Flow Analysis. In E. Albert, editor, *Logic-Based Program Synthesis and Transformation: 22nd International Symposium, LOPSTR 2012, Leuven, Belgium, September 18-20, 2012, Revised Selected Papers*, pages 106–121, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[84] J. L. Massey. Guessing and entropy. In *Proceedings of 1994 IEEE International Symposium on Information Theory*, pages 204–, Jun 1994.

[85] S. McCamant and M. D. Ernst. Quantitative Information Flow As Network Flow Capacity. *SIGPLAN Not.*, 43(6):193–205, June 2008.

[86] Z. Meng and G. Smith. Calculating Bounds on Information Leakage Using Two-bit Patterns. In *Proc. of the ACM SIGPLAN 6th Workshop on Prof. Languages and Analysis for Security*, PLAS '11, pages 1:1–1:12. ACM, 2011.

[87] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, Oct. 1992.

[88] D. Milushev, W. Beck, and D. Clarke. Noninterference via Symbolic Execution. In H. Giese and G. Rosu, editors, *Formal Techniques for Distributed Systems: Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, pages 152–168, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[89] C. Mu and D. Clark. An Interval-based Abstraction for Quantifying Information Flow. *Electronic Notes in Theoretical Computer Science*, 253(3):119 – 141, 2009.

[90] C. Mu and D. Clark. Quantitative Analysis of Secure Information Flow via Probabilistic Semantics. In *Proceedings of the The Forth International Conference on Availability, Reliability and Security, ARES 2009, March 16-19, 2009, Fukuoka, Japan*, pages 49–57. IEEE Computer Society, 2009.

[91] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. of 26th ACM Symp. on Principles of Programming Languages*, pages 228–241, 1999.

[92] A. Nanevski, A. Banerjee, and D. Garg. Verification of Information Flow and Access Control Policies with Dependent Types. In *Proc. of the 2011 IEEE Symp. on Security and Privacy*, SP '11, pages 165–179. IEEE CS, 2011.

[93] J. Newsome, S. McCamant, and D. Song. Measuring Channel Capacity to Distinguish Undue Influence. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 73–85, New York, NY, USA, 2009. ACM.

[94] T. M. Ngo and M. Huisman. Quantitative Security Analysis for Programs with Low Input and Noisy Output. In *Proc. of the 6th Intl. Symp. on Engineering Secure Software and Systems*, volume 8364 of *ESSoS 2014*, pages 77–94. Springer, 2014.

[95] R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, pages 156–169, Berlin, Heidelberg, 2006. Springer-Verlag.

[96] C. S. Pasareanu, Q. Phan, and P. Malacaria. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal*, pages 387–400. IEEE Computer Society, 2016.

[97] Q.-S. Phan. Self-composition by Symbolic Execution. In A. V. Jones and N. Ng, editors, *Imperial College Computing Student Workshop*, volume 35 of *OASIcs*, pages 95–102. Schloss Dagstuhl, 2013.

[98] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu. Symbolic Quantitative Information Flow. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[99] V. R. Pratt. Semantical Considerations on Floyd-Hoare Logic. In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*, pages 109–121. IEEE Computer Society, 1976.

[100] W. Rafnsson and A. Sabelfeld. Secure Multi-execution: Fine-Grained, Declassification-Aware, and Transparent. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, CSF '13, pages 33–48, Washington, DC, USA, 2013. IEEE Computer Society.

[101] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Sci. Comput. Program.*, 64(1):54–75, 2007.

[102] A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 186–199, Washington, DC, USA, 2010. IEEE Computer Society.

[103] A. Sabelfeld and A. C. Myers. A Model for Delimited Information Release. In K. Futatsugi, F. Mizoguchi, and N. Yonezaki, editors, *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2003.

[104] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[105] A. Sabelfeld and D. Sands. Declassification: Dimensions and Principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[106] C. Scheben. *Program-level Specification and Deductive Verification of Security Properties*. PhD thesis, Karlsruhe Institute of Technology, 2014.

[107] C. Scheben and P. H. Schmitt. Verification of Information Flow Properties of Java Programs without Approximations. In *Formal Verification of Object-Oriented Software*, volume 7421 of *LNCS*, pages 232–249. Springer, 2012.

[108] D. Scheurer, R. Hähnle, and R. Bubel. A General Lattice Model for Merging Symbolic Execution Branches. In *Proceedings of the 18th International Conference on Formal Engineering Methods (ICFEM)*, number 10009 in Lecture Notes in Computer Science, pages 57–73. Springer, 2016.

[109] C. E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, Oct 1949.

[110] G. Smith. On the Foundations of Quantitative Information Flow. In *Proc. of the 12th Intl. Conf. on Foundations of Software Science and Computational Structures*, FOSSACS '09, pages 288–302. Springer, 2009.

[111] G. Smith. Quantifying Information Flow Using Min-Entropy. In *8th Intl. Conf. on Quantitative Evaluation of Systems*, QEST 2011, pages 159–167. IEEE Computer Society, 2011.

[112] T. Terauchi. A Type System for Observational Determinism. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF '08, pages 287–300, Washington, DC, USA, 2008. IEEE Computer Society.

[113] T. Terauchi and A. Aiken. Secure Information Flow as a Safety Problem. In C. Hankin and I. Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.

[114] N. Tillmann and J. de Halleux. Pex–White Box Test Generation for .NET. In B. Beckert and R. Hähnle, editors, *Tests and Proofs: Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, pages 134–153, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[115] J. A. Vaughan and S. Chong. Inference of Expressive Declassification Policies. In *Proc. of the 2011 IEEE Symp. on Security and Privacy*, pages 180–195. IEEE CS, 2011.

[116] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica*, 48(1):37–66, 2007.

[117] D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2):167–187, 1996.

[118] N. Wasser. Generating Specifications for Recursive Methods by Abstracting Program States. In X. Li, Z. Liu, and W. Yi, editors, *Dependable Software Engineering: Theories, Tools, and Applications: First International Symposium, SETTA 2015, Nanjing, China, November 4-6, 2015, Proceedings*, pages 243–257, Cham, 2015. Springer International Publishing.

[119] H. Yasuoka and T. Terauchi. On Bounding Problems of Quantitative Information Flow. *J. Comput. Secur.*, 19(6):1029–1082, 2011.