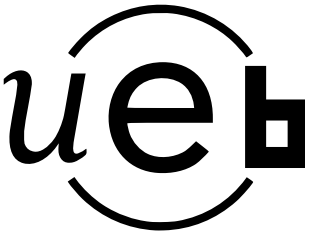
From qualitative to quantitative program analysis : permissive enforcement of secure information flow

Mounir Assaf

ANNÉE 2015



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Mounir ASSAF

préparée à l'unité de recherche EPC CIDRE
IRISA/INRIA/CentraleSupélec

From Qualitative to Quantitative Program Analysis: Permissive Enforcement of Secure Information Flow

Thèse soutenue à Rennes
le 06/05/2015

devant le jury composé de :

David PICHARDIE
Professeur à l'ENS Rennes/IRISA / *Président*

Jean GOUBAULT LARRECQ
Professeur à l'ENS Cachan/LSV/CNRS / *Rapporteur*

David SANDS
Professeur à Chalmers University of Technology / *Rapporteur*

Jérôme FERET
Chargé de Recherche 1ère classe, INRIA / *Examinateur*

Boris KÖPF
Assistant Research Professor, IMDEA Software Institute / *Examinateur*

Éric TOTEL
Professeur à CentraleSupélec/CIDRE / *Directeur de thèse*

Julien SIGNOLES
Ingénieur Chercheur au CEA LIST / *Co-directeur*

Frédéric TRONEL
Professeur Associé à CentraleSupélec/CIDRE / *Co-directeur*

Résumé

De nos jours, les ordinateurs sont omniprésents. Tous ces ordinateurs stockent et manipulent de l'information, parfois sensible, d'où l'intérêt de protéger et de confiner la dissémination de cette information. Les mécanismes de contrôle de flux d'information permettent justement d'analyser des programmes manipulant de l'information sensible, afin de prévenir les fuites d'information.

Les contributions de cette thèse incluent des techniques d'analyse de programmes pour le contrôle de flux d'information tant qualitatif que quantitatif. Les techniques d'analyse qualitatives permettent la détection et la prévention des fuites d'information. Les techniques quantitatives vont au-delà de la simple détection des fuites d'information, puisqu'elles permettent d'estimer ces fuites afin de décider si elles sont négligeables.

Nous formalisons un moniteur hybride de flux d'information, combinant analyse statique et dynamique, pour un langage supportant des pointeurs et de l'aliasing. Ce moniteur permet de mettre en œuvre une *propriété de non-interférence*, garantissant l'absence de fuites d'information sensible. Nous proposons aussi une transformation de programmes, qui permet de tisser la spécification du moniteur au sein du programme cible. Cette transformation de programme permet de mettre on œuvre la propriété de non-interférence soit par analyse dynamique en exécutant le programme transformé, ou par analyse statique en analysant le programme transformé grâce à des analyseurs statiques existants, sans avoir à les modifier.

Certains programmes, de par leur fonctionnement intrinsèque, font fuiter une quantité – jugée négligeable – d'information sensible. Ces programme ne peuvent donc se conformer à des propriétés telles que la non-interférence. Nous proposons une propriété de sécurité quantitative qui permet de relaxer la propriété de non-interférence, tout en assurant les même garanties de sécurité. Cette propriété de *secret relatif* est basée sur une mesure existante de quantification des flux d'information : la *min-capacité*. Cette mesure permet de quantifier les fuites d'information vis-à-vis de la probabilité qu'un attaquant puisse deviner le secret.

Nous proposons aussi des techniques d'analyse statique permettant de prouver qu'un programme respecte la propriété de *secret relatif*. *L'abstraction des cardinaux*, une technique d'analyse par interprétation abstraite, permet de sur-approximer la mesure de min-capacité pour des programmes acceptant des entrées publiques et confidentielles, mais n'affichant le résultat de leurs calculs qu'à la fin de leur exécution. *L'abstraction des arbres d'observation*, quant à

elle, supporte des programmes pouvant afficher le résultat de leurs calculs au fur et à mesure de leur exécution. Cette dernière analyse est paramétrée par l'abstraction des cardinaux. Elle s'appuie sur la combinatoire analytique afin de décrire les observations d'un attaquant et quantifier les fuites d'information pour prouver qu'un programme respecte la propriété de secret relatif.

Abstract

Computers have become widespread nowadays. All these computers store and process information. Often, some of this information is sensitive; hence the need to confine and control its dissemination. An important field in computer science, that is concerned about analysing programs in order to confine and control the release of sensitive information, is the information flow control field.

The contributions of this thesis include program analysis techniques for qualitative and quantitative information flow control. Qualitative techniques aim at detecting and preventing information leaks. Quantitative techniques go beyong the detection of information leaks, by estimating the leakage in order to decide whether it is negligeable.

We formalize a hybrid information flow monitor, combining both a static and a dynamic analysis, for a language supporting pointers and aliasing. This monitor prevents information leaks by enforcing a *non-interference security property*. We also propose a program transformation in order to inline the specification of our monitor into the target program. This program transformation enables the verification of non-interference either dynamically by executing the transformed program, or statically by analysing the transformed program using off-the-shelf static analysis tools.

Some programs inherently leak a negligible amount of sensitive information because of their functionality. Such programs cannot comply with security properties such as non-interference. Therefore, we also propose a quantitative security property in order to relax non-interference, while still providing the same security guarantees. This *relative secrecy* property is based on *min-capacity*, an existing measure of information leaks. This measure quantifies information leaks wrt. the probability of attackers guessing the secret.

We also propose static analysis techniques in order to prove that programs comply with *relative secrecy*. *The cardinal abstraction*, based on abstract interpretation, enables the over-approximation of min-capacity for programs that accepts both public and confidential inputs, but output the results of their computation only at the end of their execution. *The tree abstraction* builds on the cardinal abstraction in order to support programs that may output intermediate results of computation as well. This latter analysis relies on *analytic combinatorics* in order to describe attackers' observations and quantify information leaks. It also aims to prove the compliance of a program wrt. relative secrecy.

Remerciements

Je remercie David Sands et Jean Goubault-Larrecq d'avoir accepté de rapporter ma thèse. Merci aussi aux examinateurs, Boris Köpf et Jérôme Feret, ainsi qu'au président du jury, David Pichardie. Je vous suis reconnaissant pour tous vos commentaires et remarques qui, sans aucun doute, auront grandement contribué à améliorer la qualité de ce manuscrit.

Cette thèse, qui n'est que le prélude d'un long cheminement scientifique, n'aurait pas été possible sans l'accompagnement et le soutien, tant technique que moral, de mes encadrants Éric Totel, Frédéric Tronel et Julien Signoles. Je vous remercie chaleureusement d'avoir mobilisé un climat propice à la prise d'initiatives. Merci également d'avoir favorisé, nourri et entretenu un esprit critique et une libre pensée nécessaires à tout jeune chercheur. Vos conseils avisés continueront certainement de guider mes pas.

J'aimerai aussi souligner l'excellent accueil dont j'ai pu bénéficier à l'équipe CIDRE et au Laboratoire de Sûreté des Logiciels.

Finalement, je souhaiterai exprimer ma gratitude à ma famille, ainsi qu'à mes amis, pour leur soutien indéfectible et leurs encouragements tout au long de mon parcours.

Résumé étendu en français

De nos jours, les ordinateurs sont devenus très répandus. Ordinateurs personnels, téléphones intelligents, voitures connectées, cartes bancaires ainsi que les objets connectés sont omniprésents.

Tous ces ordinateurs stockent et traitent de l'information, parfois sensible. La dissémination de cette information sensible a donc besoin d'être contrôlée. Un domaine important des sciences informatiques, dont le but est de confiner ainsi que de contrôler la dissémination de cette information, est le domaine du *contrôle de flux d'information*.

Contrôle de flux d'information

La sécurité de l'information s'intéresse généralement à 3 différentes propriétés :

1. la confidentialité s'assure que les données sensibles ne sont divulguées qu'aux personnes autorisées,

2. l'intégrité s'assure que les données de confiance ne sont pas altérées, voire falsifiées,

3. la disponibilité assure le fonctionnement d'un système et garantit l'accès à l'information, sans interruptions.

Les propriétés de confidentialité et d'intégrité sont en général gérées grâce à des mécanismes de contrôles d'accès. Ces mécanismes vérifient que les utilisateurs, ou les programmes, demandant un accès à l'information ont les autorisations et permissions nécessaires pour y accéder. Cependant, ces mécanismes sont insuffisants car ils mettent en œuvre des politiques de sécurité à un niveau gros grain.

Supposez par exemple qu'une application demande les permissions nécessaires pour accéder à votre liste de contacts et à internet. Supposez aussi que ces permissions semblent être légitimes. Par exemple, cette

application a besoin d'accéder à la liste de contacts, puisque sa principale fonctionnalité est de fusionner automatiquement les entrées dupliquées. Cette application a aussi besoin d'accéder à internet, pour ses mises à jour. Si vous décidez de fournir à cette application les permissions qu'elle demande, pour pouvoir l'utiliser, vous ne pouvez garantir que l'application ne divulguera pas votre liste de contacts sur internet.

Dans un tel contexte, les mécanismes de *contrôle de flux d'information* interviennent afin de mettre en œuvre des politiques de sécurité à un niveau de granularité fin. Ces mécanismes permettent d'analyser et de suivre la manière dont l'information se propage au sein d'une application, afin de s'assurer que la propagation de l'information respecte une certaine politique de sécurité. Dans le cas de l'application mentionnée plus haut, la politique de sécurité qui nous intéresse est de s'assurer que les données confidentielles – le carnet d'adresses – ne sont pas divulguées sur des canaux publics – internet. Le contrôle de flux d'information permet de fournir à l'application les autorisations nécessaires à son fonctionnement, tout en garantissant que les flux d'information, c'est-à-dire la propagation de l'information, respectent cette politique de sécurité.

En général, les approches de contrôle de flux d'information sont qualitatives ou quantitatives.

Les techniques qualitatives d'analyse de programmes pour le contrôle de flux d'information s'intéressent à la détection ainsi qu'à la prévention des fuites d'information. Ces approches mettent en œuvre une *propriété de non-interférence*. La non-interférence [Coh77, GM82] est une propriété de sécurité qui formalise la non-dépendance des sorties publiques vis-à-vis des entrées confidentielles.

Au-delà de la détection des fuites d'information, les techniques quantitatives d'analyse de programmes pour le contrôle de flux d'information permettent d'estimer les fuites d'information, afin de décider si ces fuites sont négligeables ou pas. En effet, certains programmes tels les systèmes de vote ou les vérificateurs de mot de passe, font fuiter de l'information sensible de par leur fonctionnement intrinsèque. Ces programmes ne peuvent donc se conformer à une propriété de sécurité telle que la non-interférence.

Plan et Contributions

Cette thèse développe des techniques d'analyse de programmes pour le contrôle de flux d'information qualitatif et quantitatif. Le Chapitre 2 présente brièvement une revue de l'état de l'art du domaine du contrôle de flux d'information. Nous introduisons des propriétés de sécurité, tant qualitatives

que quantitatives, ainsi que des techniques d'analyse permettant de vérifier la conformité des programmes vis-à-vis de ces propriétés.

Les chapitres suivants décrivent les contributions principales de cette thèse. Ces contributions sont :

1. la formalisation d'un moniteur hybride de flux d'information pour un langage supportant des pointeurs et de l'aliasing. Ce moniteur, dénommé moniteur `PWhile`, combine analyse statique et analyse dynamique pour le suivi de flux d'information. Le moniteur `PWhile` se base sur les moniteurs sensibles aux flux d'information proposés par Le Guernic et al. [LGBJS06] ainsi que Russo et Sabelfeld [RS10], afin d'étendre le langage qu'ils supportent avec des pointeurs et de l'aliasing. Le moniteur `PWhile` est aussi prouvé correct. Il permet la mise en œuvre d'une propriété de non-interférence insensible à la terminaison (TINI). Le Chapitre 3 présente ce résultat.

2. une transformation de programmes qui permet de tisser le moniteur `PWhile` au sein du programme analysé. Cette transformation de programmes, décrite dans le Chapitre 4, préserve le comportement initial du programme analysé. Elle reproduit aussi exactement la sémantique du moniteur `PWhile`. Cette transformation de programme est aussi prouvée correcte. Elle permet la mise en œuvre de la non-interference (TINI) soit par analyse dynamique en exécutant le programme transformé, ou par analyse statique en analysant le programme transformé grâce à des analyseurs statiques existants, sans avoir à les modifier.

3. une propriété de sécurité quantitative qui permet de relaxer la propriété de non-interference (TINI), tout en assurant les mêmes garanties de sécurité. Cette propriété de sécurité, introduite dans le Chapitre 5, est une propriété de *secret relatif* [VS00]. Elle permet de garantir que la probabilité qu'un attaquant puisse deviner le secret, en un seul essai, est négligeable en la taille du secret, lorsque cet attaquant n'observe qu'une quantité polynomiale d'instructions de sortie. Nous étendons aussi cette propriété de sécurité au cas où l'attaquant a droit à plusieurs essais pour deviner le secret. Les travaux présentés dans ce chapitre s'appuient sur la *min-capacité*, une mesure de quantification des fuites d'information proposée par Smith [Smi09, Smi11]. Ces travaux s'appuient aussi sur les résultats proposés par Askarov et al. [AHSS08] concernant la quantification des fuites d'information dues aux programmes conformes vis-à-vis de TINI.

4. une analyse statique, prouvée correcte, calculant le nombre cardinal de valeurs que les variables peuvent prendre. Cette analyse statique permet de quantifier les fuites d'information, grâce à une sur-approximation de la min-capacité [Smi09, Smi11] pour des programmes déterministes qui peuvent accepter des entrées publiques et confidentielles. Cependant, cette analyse statique ne supporte que des programmes qui n'affichent le résultat de leurs calculs qu'à la fin de leur exécution. Cette analyse statique, dénommée l'abstraction des cardinaux, est décrite dans le Chapitre 6.

5. une analyse statique paramétrée par l'abstraction des cardinaux, qui permet de quantifier les fuites d'information pour des programmes déterministes pouvant afficher les résultats de leur calculs au fur et à mesure de leur execution. Cette analyse statique, dénommée l'abstraction des arbres d'observation, calcule une *spécification régulière* qui décrit les valeurs qu'un attaquant peut observer. En s'appuyant sur la combinatoire analytique [FS09], la spécification régulière calculée est traduite en une *fonction génératrice*. Nous dérivons ensuite des conditions suffisantes sur la fonction génératrice afin de prouver qu'un programme est conforme à la propriété de secret relatif, introduite dans le Chapitre 5. Ce résultat est décrit dans le Chapitre 7.

Finalement, le Chapitre 8 conclut et présente les travaux futurs.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

Computers have become widespread nowadays. Personal computers, smart phones, connected cars, credit cards as well as wearable gadgets are omnipresent.

All these computers store and process information. Some of this information is sensitive. Therefore, its dissemination needs to be controlled. An important field in computer science, that is concerned about confining and controlling the dissemination of information, is the *information flow control* field.

1.1 Information Flow Control

Information security can be characterized by 3 different properties:

1. confidentiality: ensuring sensitive data does not leak to unauthorized users

2. integrity: preventing unauthorized users from tampering with trusted data

3. availability: guaranteeing access to data or services, without disruptions.

The two first properties, namely confidentiality and integrity, are in general managed through access control mechanisms. For instance, prior to accessing any piece of data, such mechanisms ensure that the user or application querying information does indeed have the required authorizations or permissions. However, such mechanisms are insufficient since they enforce security properties at a coarse-grained level.

Assume for instance that an application requires permissions to access both the user's contact list and the internet. Assume also that both permission requests seem legitimate. For instance, the application needs access to the contact list since its principal functionality is merging duplicate entries automatically. Additionally, this application needs access to the internet as well, in order to check for updates. If the user grants the application both required authorizations in order to be able to use it, then all bets are off. Indeed, access control mechanisms cannot guarantee that the application, once granted both authorizations, will not leak the user's contact list over the internet for instance.

In such a scenario, *information flow control* mechanisms step in, in order to enforce fine-grained security policies by analysing the way information propagates in an application. In the aforementioned example, the security policy of interest requires preventing confidential data – the address book – from leaking into public data – the internet. In a nutshell, such mechanisms allow the application to access both the contact list and the internet. Yet, they also guarantee that information flows, namely the propagation of information, are legal wrt. the user's security policy.

In general, information flow control approaches can be classified as either qualitative or quantitative.

On the one hand, program analysis techniques for qualitative information flow aim at detecting and preventing information leaks. These approaches usually enforce a *non-interference property*. Non-interference [Coh77, GM82] is a baseline security property formalizing the non-dependence of public outputs on confidential inputs.

On the other hand, quantitative information flow techniques go beyond the detection of information leaks, by estimating the leakage in order to decide whether it is negligible. Intuitively, some applications such as password checkers or voting systems, inherently leak a small amount of sensitive information because of their functionality. Therefore, such applications cannot comply with non-interference.

1.2 Contributions and Outline

This thesis develops program analysis techniques for qualitative and quantitative information flow. Due to the large body of research in both qualitative and quantitative information flow control, we present a brief overview of the literature in Chapter 2. We introduce both qualitative and quantitative security properties, as well as program analysis techniques aimed at proving these security properties.

The subsequent chapters describe the main contributions of this thesis. These contributions are:

1. a formalization of a hybrid information flow monitor for an imperative language supporting pointers and aliasing. This monitor, dubbed the `PWhile` monitor, builds on the flow-sensitive hybrid monitors proposed by Le Guernic et al. [LGBJS06] and Russo and Sabelfeld [RS10], in order to extend the supported language with pointers and aliasing. We also prove the soundness of the `PWhile` monitor wrt. TINI. Chapter 3 introduces this result. The material presented in this chapter draws on two previously published conference papers [ASTT13b, ASTT13a].

2. a program transformation that inlines the `PWhile` monitor. This inlining approach, described in Chapter 4, preserves the initial behaviour of target programs, and reproduces exactly the semantics of the `PWhile` monitor. To the best of our knowledge, this program transformation is the first proven sound inlining approach for dynamic monitors handling pointers. The material presented in this chapter draws on a previously published conference paper [ASTT13b].

3. a quantitative security property aimed at relaxing TINI, while still providing the same security guarantees. This security property, introduced in Chapter 5, is a relative secrecy (RS) property [VS00]. It ensures that the probability of attackers guessing the secret in one try is negligible in the size of the secret, when attackers observe only a polynomial amount of outputs. We also extend this security property to account for the probability of polynomial time attackers guessing the secret in the case of k-try attacks. This chapter builds on *min-capacity*, a quantitative information flow metrics introduced by Smith [Smi09, Smi11], as well as Askarov et al.'s [AHSS08] results on the amount of leakage allowed by TINI.

4. a sound static analysis computing the cardinal number of values variables may take. This static analysis enables the over-approximation of min-capacity [Smi09, Smi11] for deterministic programs that accept both confidential and public inputs, but only output the result of their computation at the end of their execution. This static analysis, called the cardinal abstraction, is based on abstract interpretation [CC77]. It is presented in Chapter 6.

5. a static analysis built on top of the cardinal abstraction, in order to quantify information flow for deterministic programs that may output

intermediate steps of computation. This static analysis, called the tree abstraction, computes a *regular specification* describing attackers' observations. By relying on the framework of analytic combinatorics [FS09], the computed regular specification translates into a *generating function*. We then derive sufficient conditions on the generating function in order to prove that a program is secure wrt. RS, the quantitative security property introduced in Chapter 5. These results are described in Chapter 7. To the best of our knowledge, this is the first analysis aimed at quantifying information flow wrt. min-capacity, for programs that may output intermediate steps of computation.

Finally, Chapter 8 concludes and discusses future work.

Chapter 2

Information Flow

Denning's seminal work [Den76, DD77] in information flow control proposes a static analysis taking place during a program compilation. This analysis certifies that a program cannot leak confidential data. Denning also distinguishes different kinds of information flows that arise in a program:

1. *explicit information flows* are due to assignments. For instance, Listing 2.1 illustrates a program assigning variable `secret` to variable `public`. Therefore, this assignment generates an explicit information flow from variable `secret` to variable `public`.

```
1 // explicit flow from variable secret to public
2 public := secret ;
```

Listing 2.1: Explicit information flows

2. *implicit information flows* are due to branching instructions whose guard depends on confidential data. For instance, consider the example program in Listing 2.2. If attackers have access to variable `public` after the execution of this program, they may deduce information about variable `secret`. In general, we assume attackers know the source code of analysed programs, and have perfect deduction capabilities. Therefore, the assignment to variable `public` inside a conditional whose guard depends on variable `secret` generates an implicit information flow from variable `secret` to variable `public`.

3. *covert information flows* are due to covert channels leaking information. For instance, attackers may leverage timing information, power consumption or electromagnetic leaks to learn sensitive information.

```
1 // implicit flow from variable secret to public
2 if (secret > 0) {
3   public := 1;
4 }
5 else {
6   public := 0;
7 }
```

Listing 2.2: Explicit information flows

These covert information flows are the most difficult to mitigate, since addressing them often requires a refined knowledge of the underlying system and hardware. These covert channels fall beyond the scope of this thesis.

Denning's seminal work provides the bases for the *qualitative information flow* field, aimed at the detection of information leaks. The prolific literature in this field eventually lead to noteworthy tools, among which are Jif [Mye99, MNZZ01] and Flow Caml [PS02, Sim03].

Denning [Den82] also pioneers the *quantative information flow* field, aimed at estimating the amount of information leaks rather than just detecting them. Indeed, Denning defines the existence of an information flow by the existence of a reduction in uncertainty. Intuitively, if knowledge of a variable `y` reduces the initial uncertainty about variable `x`, then there exists an information flow from variable `x` to variable `y`. Measured by Shannon entropy [Sha48], this reduction of uncertainty also provides an estimation of information leaks.

The next sections provide a brief overview of the secure information flow literature. We introduce both qualitative and quantitative security properties, as well as program analysis techniques aimed at proving these security properties.

2.1 Qualitative Information Flow

2.1.1 Non-interference

Goguen and Meseguer [GM82] propose a general automaton approach to model secure systems. They propose a non-interference security policy in order to confine information flows at the system level. The non-interference property Goguen and Meseguer propose ensures that "one group of users,

using a certain set of commands, has no effect on what a second group of users can see". This concept of non-interference also goes back to Cohen's [Coh77] work on strong dependency. Intuitively, a variable y strongly depends on variable x if changing the values of variable x is mirrored by changes in the values of y.

Volpano et al. [VIS96] introduce a variant of non-interference suited for reasoning at the language level, for deterministic programs. Under their definition of security, a program is secure if any two terminating executions, differing only on confidential inputs, yield the same public outputs. Intuitively, this property ensures that while the public inputs are fixed, changing the confidential inputs does not affect the observations attackers make, provided that the program terminates. Thus, this definition guarantees that the public outputs are indeed non-interferent with – independent of – the confidential inputs. This definition of non-interference is also known as termination-insensitive non-interference (TINI) [SM03], since it ignores informations leaks due to the observation of termination or divergence of a program run.

Volpano and Smith [VS97] also introduce a second variant of non-interference, accounting for covert informations leaks due to non-termination in deterministic programs. A program is secure if any two executions, differing only on confidential inputs, either both diverge or both terminate and yield the same public outputs. This variant of non-interference is also known as termination-sensitive non-interference (TSNI) [SM03].

2.1.2 Static Analyses

Volpano et al. [VIS96] also propose a type system enforcing TINI, for a deterministic imperative language. They prove the soundness of their type system, proving that a well-typed program is secure wrt. TINI. Their type system prevents explicit information flows from leaking sensitive information by ensuring that no confidential expression gets assigned to a public variable. Additionally, it also prevents implicit information flows from leaking sensitive information by ensuring that no assignment to a public variable occurs inside a branching instruction depending on a confidential guard.

Volpano and Smith [VS97] also improve their type system in order to prevent information leaks due to termination. They propose a sound type system enforcing TSNI rather than TINI, by improving their previous type system [VIS96] and enforcing additional requirements. Specifically, in addition to preventing both explicit and implicit information leaks, their type system also ensures that guards of loop instructions do not depend

on confidential data, and loop instructions do not occur inside conditionals whose guard depend on confidential data.

Hunt and Sands [HS06] improve Volpano and Smith's type system [VIS96] with flow-sensitivity, in order to enforce TINI. Flow-sensitive type systems allow the security level mapped to each variable to change between different points of the program. Hunt and Sands' flow-sensitive type system is more permissive than Volpano and Smith's type system, in the sense that a larger set of non-interferent programs are well-typed, thus proven secure wrt. TINI.

Non-interference is not a safety property [McL94, Sch00], since it defines a relation over two program executions. However, Barthe et al. [BDR04, BDR11] propose self-composition as a mechanism to reduce non-interference of a program P to a safety property over a program P', derived from P. This technique encodes variants of non-interference in program logics, in order to leverage tools and theories traditionally aimed at safety properties, such as model checking, as well as automatic and interactive theorem proving.

While most of the initial work in qualitative information flow focuses on static analysis techniques to prove non-interference, recent approaches also investigate dynamic approaches enforcing non-interference. These dynamic approaches have the advantage of reasoning on single execution paths, rather than the whole program paths. Therefore, untrusted programs can still execute while monitored, without any change affecting their behaviour as long as they do not attempt to leak sensitive information.

2.1.3 Dynamic analyses

Le Guernic et al. [LGBJS06] propose a hybrid information flow monitor, combining both a static and dynamic analysis, to track information flows for a deterministic `While` language. Their approach, based on edit automata [LBW05], modifies program outputs to default values or suppresses them whenever such outputs may leak sensitive information. Le Guernic et al. also prove the soundness of their monitor wrt. TINI. Moreover, they prove that their monitor also preserves executions of well-typed programs wrt. Volpano and Smith's [VIS96]. Le Guernic [LG07] also improves the precision of the initial monitor by relying on a context-sensitive static analysis, and propose an extension [LG08] of the initial monitor to concurrent programs.

Russo and Sabelfeld [RS10] formalize a hybrid information flow monitor for a deterministic `While` language with outputs. They also prove the soundness of their monitor wrt. TINI, as well as its permissiveness wrt. Hunt and Sands's [HS06] flow-sensitive type system. Russo and Sabelfeld also prove that a purely dynamic monitor – a monitor that has neither knowledge

of commands in non-executed alternative paths, nor knowledge of commands ahead of their execution –, cannot guarantee soundness wrt. non-interference while being at least as permissive as Hunt and Sands flow-sensitive type system.

Austin and Flanagan [AF09] propose a purely dynamic monitor for a λ-calculus language with references. Their monitor supports a limited flow-sensitivity since it implements a conservative no-sensitive upgrade policy [Zda02, Section 3.2]; their monitor stops the execution when the monitored programs tries to assign a public reference in a context depending on confidential data. Thus, their monitor is proven sound wrt. TINI without having to rely on prior static analyses.

Austin and Flanagan [AF10] also enhance their monitor by a permissive-upgrade approach; their monitor labels public data that is assigned in a confidential security context as partially leaked, then forbids sensitive operations on these data – such as branching and dereferencing – by stopping the monitored program. They also propose to dynamically infer the program points where sensitive operations on partially leaked data occur, in order to label such data as private. Interestingly, this "privatization inference" approach eventually enforces TINI without halting the monitored program, provided that all program points, where sensitive operations on partially leaked data occur, are already inferred.

Devriese and Piessens [DP10] propose "secure multi-execution" in order to monitor information leaks in a program. This technique is also known as "shadow executions" [CLVS08]. They propose to run multiple copies of the same program in parallel, once for each security level. Intuitively, each copy of the target program have a security clearance and can only access the inputs having a lower security clearance. Otherwise, inputs with higher security level are replaced with default values. Additionally, each copy of the target program can only output to channels having the same security clearance. Devriese and Piessens prove the soundness of their approach wrt. TSNI for a `While` language with inputs and outputs. They also prove that their approach eliminates covert timing channels, provided that (1) the scheduling strategy ensures high security copies of the program wait for low security ones whenever they access low inputs, and (2) each running copy of the target program never waits for executions having a higher security level. They also implement their approach for the Spidermonkey javascript engine.

2.2 Quantitative Information Flow

2.2.1 Information Flow Metrics

Clark, Hunt and Malacaria [Cla05] build on Denning's [Den82] work in order to propose a measure of information leakage, based on Shannon entropy [Sha48] and mutual information. In particular, they prove in the deterministic setting that the leakage of a program can be measured by the conditional Shannon entropy of the observable outputs, knowing the public inputs. They also prove that this latter quantity equals zero in the case of a deterministic program iff. the program is non-interferent.

Clarkson et al. [CMS09, CS10] propose a new information flow measure, coined as information belief. This measure reasons on attackers' belief, namely the probability distribution of the confidential input that attackers may assume. Clarkson et al. also model how attackers revise their belief after observing outputs of a program. The revised attackers' belief as well as the initial belief provide a way to measure the improvement in the accuracy of attackers' belief. Clarkson et al. also propose to measure this improvement in accuracy by relying on relative entropy [CT06], a pseudo-metric defined over probability distributions.

Smith [Smi09] notes that Shannon entropy and mutual information are unsuitable metrics for estimating information flow leakage. Indeed, Shannon entropy of a random variable can be arbitrarily high despite being highly vulnerable to being guessed in one try [ES13]. Smith then proposes the use of min-entropy as a measure for quantifying information flow. This measure estimates the probability that attackers guess the confidential inputs in one try, after observing a program run. Additionally, Smith proves that the maximum leakage measured either with min-entropy (namely, min-capacity) or Shannon entropy (namely, capacity) coincide for deterministic programs, and conjecture that min-capacity upper-bounds capacity for probabilistic programs [Smi11].

Braun, Chatzikokolakis and Palamidessi [BCP09] propose an additive notion of information leakage based on the probability that attackers guess the wrong confidential inputs. They also investigate how to compute supremums for their additive notion as well as for min-entropy [Smi09]. Particularly, they prove that the supremum for min-entropy is reached when the confidential inputs are uniformly distributed [BCP09, Smi11].

Alvim et al. [ACPS12] propose a generalization of min-entropy using gain functions. Gain functions model a variety of scenarios for attackers, such as the advantage attackers gain from guessing part of the secret or making a set of guesses. They also prove in the probabilistic setting that

min-capacity is an upper bound on both Shannon capacity and capacity defined using gain functions. Hence, min-capacity offers a way to abstract from a priori distributions of the confidential inputs and to bound various entropy-based information flow metrics.

2.2.2 Static Analyses

Clark et al. [CHM07] propose the first type system to quantify information flow for a deterministic imperative language. Their definition of information leakage [Cla05] relies on mutual information and Shannon entropy. They also model an attacker that controls public inputs, but is not allowed to observe intermediate steps of computation.

Backes, Köpf and Rybalchenko [BKR09] propose an analysis that iteratively synthesizes equivalence relations over the confidential inputs of a program. Their approach computes approximations of various quantitative information flow measures, such as min-entropy. They also compute approximations of Shannon entropy by enumerating each equivalence class over the confidential inputs. These metrics enable modelling different scenarios depending on the appropriate one.

Köpf and Rybalchenko [KR10] propose an analysis based on both static and dynamic techniques to compute over-approximations as well as under-approximations of information flow leakage. Their approach computes over-approximations (resp. under-approximations) of the set of reachable states by abstract interpretation (resp. by model checking). This enables them to derive upper and lower bounds on the remaining uncertainty measured by min-entropy. Additionally, they also propose a randomization technique based on sampling in order to compute tight bounds on Shannon remaining uncertainty, without the requirement of enumerating the set of reachable states and estimating the size of their pre-images.

Meng and Smith [MS11] propose to quantify information flow for straight-line deterministic programs by synthesizing bit patterns over the final values. Their approach is similar to Backes, Köpf and Rybalchenko [BKR09], although they adopt a low level approach. By counting the number of solutions satisfying the computed patterns, they are able to provide bounds on capacity. Their approach derives precise bounds of the leakage for low level computations over variables. They leave for future work the calculation of precise bounds over capacity for programs accepting public inputs.

Köpf and Rybalchenko [KR13] propose to rely on self-compositions [BDR04, BDR11] of a program to take into account public inputs. This promising approach simulates through self-composition the leakage under an actual k-try scenario. Attackers can supply k public inputs, then observe k runs of

a program on the supplied inputs. They can hence refine their knowledge accordingly.

2.2.3 Dynamic Analyses

Mardziel et al. [MMHS11] propose an analysis to dynamically enforce knowledge-based policies. This approach maintains a model of attackers' knowledge by tracking and updating attackers' information beliefs [CMS09]. Their monitor actually performs a static analysis on-the-fly, relying on the framework of abstract interpretation and probabilistic abstract polyhedra. Their monitor then decides to answer a query over sensitive data only provided that attackers' knowledge does not exceed a certain threshold.

Besson, Bielova and Jensen [BBJ13] propose a hybrid monitor aimed at protecting users against web tracking. Their approach computes a symbolic representation of attackers' knowledge in order to quantify information leaks wrt. self-information [Eck10]. Their monitor is also a generic one that enables modelling different hybrid monitors parametrized by static analyses and aimed at enforcing non-interference. They also prove that the derived monitors are more permissive than the state-of-the-art flow-sensitive monitors [LGBJS06, RS10], thanks to the combination of a constant propagation and a more precise analysis of implicit information flows.

Part I

Qualitative Information Flow

Chapter 3

Information Flow Monitor

In this chapter, we formalize the `PWhile` monitor, an information flow monitor for an imperative language supporting pointers.

We introduce the semantics of `PWhile`, an imperative language supporting pointers and aliasing. We also formalize the `PWhile` information flow monitor. The `PWhile` monitor is flow-sensitive. It relies on a prior static analysis in order to soundly track information flows. We also prove that the `PWhile` monitor prevents information leaks, by proving its soundness wrt. non-interference.

3.1 `PWhile` language

This section introduces the syntax and the semantics of the `PWhile` language. `PWhile` extends an imperative `While` language [Win93] with pointers and aliasing support. Its semantics is inspired by the Clight language [BL09] that formalizes a large subset of the C language.

3.1.1 Syntax of `PWhile`

Figure 3.1 describes the abstract syntax of our language `PWhile`. It is an imperative language handling pointers ($ptr(\tau)$) and basic types (κ) such as integers. Expressions have no side effects. `PWhile` supports aliasing, but no pointer arithmetic: binary operators do not take pointers as arguments.

3.1.2 Operational Semantics

The semantics of `PWhile` draws on the semantics of Clight [BL09], a language formalized in the context of the CompCert verified compiler for C

Types:	$\tau ::= \kappa$	(integers)
	$\mid ptr(\tau)$	(pointers)
Expressions:	$a ::= n$	(constants)
	$\mid id$	(variables)
	$\mid uop\ a$	(unary operators)
	$\mid a_1\ bop\ a_2$	(binary operators)
	$\mid *a$	(pointer dereferencing)
	$\mid \&a$	(address of)
Instructions:	$c ::= skip$	(empty instruction)
	$\mid a_1 := a_2$	(assignment)
	$\mid c_1;\ c_2$	(sequence)
	$\mid if\ (a)\ c_1\ else\ c_2$	(conditional)
	$\mid while\ (a)\ c$	(loop)
Declarations:	$dcl ::= (\tau\ id;)^*$	
Programs:	$P ::= dcl;\ c$	

Figure 3.1: Abstract syntax of `PWhile`

programs [Ler09]. The semantics of `PWhile` is a big-step semantics [Kah87], also known as natural semantics. This semantics considers an environment $E \in Var \rightharpoonup Loc$ that maps variables to statically allocated locations as well as a memory $M \in Loc \rightharpoonup \mathbb{V}$ mapping locations to values of type τ.

$l \in Loc$	(a memory location)
$v \in \mathbb{V}$	(a value of type τ)
$E \in Var \rightharpoonup Loc$	(bijection from variables to static locations)
$M \in Loc \rightharpoonup \mathbb{V}$	(map from locations to values)
$E \vdash c, M \Rightarrow M'$	(evaluation of an instruction c)
$E \vdash a, M \Leftarrow l$	(left-value evaluation of expression a)
$E \vdash a, M \Rightarrow v$	(right-value evaluation of expression a)

Figure 3.2: Environments, memories and judgement rules

Figure 3.2 introduces the judgement rules of `PWhile` semantics. The

evaluation of an instruction `c` in an environment E and a memory M yields a new memory M' representing the effect of instruction `c` on the memory M. Additionally, the evaluation of an expression is either a left-value evaluation (denoted by $\Leftarrow$) or a right-value evaluation (denoted by $\Rightarrow$):

1. left-value evaluations yield the location (address) where an expression is stored, whereas
2. right-value evaluations provide the content of an expression.

Both left-value and right-value evaluations of expressions modify neither environments nor memories since expressions are side-effect-free.

3.1.3 Semantics of Expressions

Figure 3.3 defines the semantics of expressions. All expressions can occur as right-values, but only variables `id` and dereferenced expressions `*a` can occur as left-values.

$$(LV_{ID})\ \frac{E(id) = l}{E \vdash id, M \Leftarrow l} \qquad (LV_{MEM})\ \frac{E \vdash a, M \Rightarrow ptr(l)}{E \vdash *a, M \Leftarrow l}$$

(a) Left-value evaluation rules

$$(RV_{CONST})\ E \vdash n, M \Rightarrow n \qquad (RV)\ \frac{E \vdash a, M \Leftarrow l \qquad M(l) = v}{E \vdash a, M \Rightarrow v}$$

$$(RV_{REF})\ \frac{E \vdash a, M \Leftarrow l}{E \vdash \&a, M \Rightarrow ptr(l)} \qquad (RV_{UOP})\ \frac{E \vdash a, M \Rightarrow v \qquad uop\ v = v'}{E \vdash uop\ a, M \Rightarrow v'}$$

$$(RV_{BOP})\ \frac{E \vdash a_1, M \Rightarrow v_1 \qquad E \vdash a_2, M \Rightarrow v_2 \qquad v_1\ bop\ v_2 = v}{E \vdash a_1\ bop\ a_2, M \Rightarrow v}$$

(b) Right-value evaluation rules

Figure 3.3: Semantics of `PWhile` expressions

The left-value evaluation of an expression determines the location where the expression is stored. The left-value evaluation of a variable `id` yields the

statically allocated location that environment E maps to `id` (rule LV_{ID}). The left-value evaluation of a dereferenced expression `*a` evaluates expression `a` as a right-value in order to determine the content of `a`. If the right-value of `a` –its content– is a pointer value $ptr(l)$, then the left-value evaluation of `*a` results in the location l (rule LV_{MEM}).

The right-value evaluation of an expression provides the content of that expression. The right-value evaluation of a constant results in its value (rule RV_{CONST}). In order to determine the right-value of expression `a`, the semantics first determines the location l of expression `a` through a left-value evaluation, then yields the value $M(l)$ that the memory M maps to the location l (rule RV). If the left-value evaluation of an expression `a` yields a location l, the semantics defines the right-value of the referenced expression `&a` as the pointer value $ptr(l)$ (rule RV_{REF}). The right-value evaluation of expression `uop a` is the result of the unary operator `uop` applied to the right-value of expression `a` (rule RV_{UOP}). Similarly, the right-value evaluation of expression $\texttt{a}_1$ `bop` $\texttt{a}_2$ is the binary operator `bop` applied to both right-values of $\texttt{a}_1$ and $\texttt{a}_2$ (rule RV_{BOP}).

3.1.4 Semantics of Instructions

Instructions evaluate in an environment E and a memory M, in order to result in a new memory M'. Figure 3.4 introduces the semantics of instructions.

Instruction skip does not modify the input memory M (rule *Skip*). The evaluation of a sequence $\texttt{c}_1$`;` $\texttt{c}_2$ of two instructions starts with the evaluation of the first instruction $\texttt{c}_1$. When this first evaluation terminates, its outcome memory M_1 determines the input memory in which the second instruction $\texttt{c}_2$ evaluates in order to yield the output memory M_2 (rule *Comp*). Assignment instructions evaluate the left-hand side expression $\texttt{a}_1$ as a left-value to determine the location l_1 to update. They also determine the value v_2 written to that location by evaluating the right-hand side expression $\texttt{a}_2$ as a right-value (rule *Assign*). The semantics then stores the value v_2 at location l_1, producing the output memory M'. Conditional instructions evaluate the conditional guard as a right-value to determine which branch should be executed through the predicates $istrue$ or $isfalse$. The semantics then defines the resulting memory as the evaluation outcome of the executed branch (rules If_{tt} and If_{ff}). Similarly, loop instructions evaluate the guard as a right-value to determine if the loop body should execute. If the guard evaluates to false, the execution results in the same input memory (rule $While_{ff}$). Otherwise, when the guard evaluates to true,

$$(Skip)\ E \vdash skip, M \Rightarrow M$$

$$(Comp)\ \frac{E \vdash c_1, M \Rightarrow M_1 \qquad E \vdash c_2, M_1 \Rightarrow M_2}{E \vdash c_1; c_2, M \Rightarrow M_2}$$

$$(Assign)\ \frac{E \vdash a_1, M \Leftarrow l_1 \qquad E \vdash a_2, M \Rightarrow v_2 \qquad M' = M[l_1 \mapsto v_2]}{E \vdash a_1 := a_2, M \Rightarrow M'}$$

$$(If_{tt})\ \frac{E \vdash a, M \Rightarrow v \qquad istrue(v) \qquad E \vdash c_1, M \Rightarrow M_1}{E \vdash if\ (a)\ c_1\ else\ c_2, M \Rightarrow M_1}$$

$$(If_{ff})\ \frac{E \vdash a, M \Rightarrow v \qquad isfalse(v) \qquad E \vdash c_2, M \Rightarrow M_2}{E \vdash if\ (a)\ c_1\ else\ c_2, M \Rightarrow M_2}$$

$$(While_{tt})\ \frac{E \vdash a, M \Rightarrow v \qquad istrue(v) \qquad E \vdash c, M \Rightarrow M' \qquad E \vdash while\ (a)\ do\ c, M' \Rightarrow M''}{E \vdash while\ (a)\ do\ c, M \Rightarrow M''}$$

$$(While_{ff})\ \frac{E \vdash a, M \Rightarrow v \qquad isfalse(v)}{E \vdash while\ (a)\ do\ c, M \Rightarrow M}$$

Figure 3.4: Semantics of `PWhile` instructions

the evaluation of the loop body results in a new memory M' in which the loop instruction is re-evaluated (rule $While_{tt}$).

3.2 `PWhile` Monitor

This section formalizes an information flow monitor for the `PWhile` language. This monitor tracks the security level of data manipulated by programs. As defined in Section 3.1, the `PWhile` semantics describes operations on value memories M. In order to define our monitor semantics, we introduce additional security memories mapping each location to a security level. We then extend the `PWhile` semantics to track information flows by describing the additional operations on security memories.

3.2.1 Information Flow Lattice

Information flow control mechanisms enforce security policies such as "confidential data must not leak into public data". A lattice is a simple yet general structure that describes such an information flow policy. For instance, the two-point lattice in Figure 3.5 describes the above policy, where $High$ (resp. Low) is a security level representing confidential data (resp. public data).

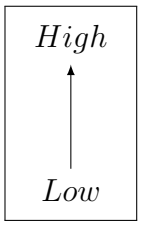


Figure 3.5: A simple information flow lattice

Indeed, this lattice defines a partial order relation $\sqsubseteq$ over security levels Low and $High$ such that $Low \sqsubseteq High$. Hence, it explicitly allows only :

1. Low data to flow to $High$ data since $Low \sqsubseteq High$, and
2. Low data to flow to Low data by reflexivity of the partial order relation $\sqsubseteq$, and finally
3. $High$ data to flow to $High$ data by reflexivity of the partial order relation $\sqsubseteq$.

In general, a lattice $(\mathbb{S}; \sqsubseteq, \sqcup, \sqcap)$ of information flow authorizes data having a security level $s_1 \in \mathbb{S}$ to flow into data having a security level $s_2 \in \mathbb{S}$ if and only if $s_1 \sqsubseteq s_2$. Similarly, a merge of two pieces of data d_1 and d_2 having respective security levels s_1 and s_2 can flow into data d having a security level $s \in \mathbb{S}$ if and only if :

1. d_1 can flow into d ($s_1 \sqsubseteq s$), and
2. d_2 can flow into d ($s_2 \sqsubseteq s$).

Thus, a merge of data d_1 and data d_2 can flow into data d if and only if $s_1 \sqcup s_2 \sqsubseteq s$. Stated otherwise, if data d is the result of merging both data d_1 and d_2 having respective security levels s_1 and s_2, then data d should be considered as having at least a security level equal to $s_1 \sqcup s_2$.

Information flow lattices simplify greatly the mechanisms enforcing flow policies. Indeed, the join operator offers a straightforward way of computing

the security level of data that results from the combination of multiple other data. Additionally, lattices are general enough to offer a great power of expressiveness, since every flow policy can be represented as an information flow lattice if the flow policy satisfies 2 conditions [Den76, DDG76]:

- its set of security levels is finite, and
- the authorized flow relation is transitive and reflexive.

3.2.2 Operational Semantics

Let us assume a lattice $(\mathbb{S}; \sqsubseteq, \sqcup, \sqcap)$ of security levels describing an information flow policy. We denote by *public* the bottom element of the lattice $\mathbb{S}$. Let us also introduce additional security memories $\Gamma \in Loc \rightharpoonup \mathbb{S}$ mapping locations to security levels. Intuitively, just as $M(l)$ represents the value of data stored at location l, $\Gamma(l)$ represents the security level of that data.

The judgement rules of `PWhile` extend to three new rules involving security memories and security levels, as summarized by Figure 3.6. Both judgement rules for expressions take as input an additional security memory Γ. They also result in an additional security level s. Judgement rules for instructions take as input an additional security memory Γ as well as an initial security context $\underline{pc}$, but they only yield an additional security memory Γ'. Since the monitor semantics is a big-step semantics, there is no need to output a new security context $\underline{pc}$; this security context remains unchanged after the evaluation of each instruction.

$(\mathbb{S}; \sqsubseteq, \sqcup, \sqcap)$,	(a lattice describing a flow policy)
$s \in \mathbb{S}$	(a security level)
$\underline{pc} \in \mathbb{S}$	(a security context)
$\Gamma \in Loc \rightharpoonup \mathbb{S}$	(a mapping from locations to security levels)
$E \vdash a, M, \Gamma \Leftarrow l, s$	(left-value evaluation of an expression)
$E \vdash a, M, \Gamma \Rightarrow v, s$	(right-value evaluation of an expression)
$E \vdash c, M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$	(evaluation of instructions)

Figure 3.6: Monitor semantics judgement rules

3.2.3 Non-Interference

The `PWhile` monitor must ensure that programs are secure with respect to an information flow policy described by a lattice $\mathbb{S}$. Technically, it prevents programs from leaking sensitive information by enforcing a termination-insensitive non-interference (TINI) security property.

In the case of the two-point lattice of Figure 3.5, a program is secure with respect to TINI if two arbitrary terminating executions differing only on $High$ inputs deliver the same Low outputs. Assuming that attackers observe only inputs and outputs that are tagged with the security level Low, TINI ensures that attackers learn no information on confidential inputs tagged as $High$. Indeed, since this property ensures that changing the $High$ inputs does not influence the observations attackers make, it guarantees that these observations do not depend on $High$ inputs.

In order to formalize TINI for the `PWhile` monitor, Definition 1 introduces an equivalence relation over memories M up to a security level s: two memories are s-equivalent if they are equal for the set of locations l whose security label $\Gamma(l)$ is at most s.

Definition 1 (Equivalence relation $\sim^s_\Gamma$)**.** *For all security memories* Γ*, for all security labels* $s \in \mathbb{S}$*, for all memories* M_1 *and* M_2*:*

M_1 *and* M_2 *are* s*-equivalent* ($M_1 \sim^s_\Gamma M_2$) *if and only if*

$$\forall l \in Loc, \Gamma(l) \sqsubseteq s \implies M_1(l) = M_2(l).$$

TINI, in the general case of a lattice $\mathbb{S}$, states that attackers only having access to data up to a security level s cannot gain any knowledge of inputs whose security levels are higher than s. Definition 2 formalizes the above intuition.

Definition 2 (Termination-insensitive non-interference)**.**
For all commands c*, environments* E*, security memories* Γ*, memories* M_1*,* M_1'*,* M_2*,* M_2'*, and security levels* s*,* $\underline{pc} \in \mathbb{S}$*, such that*

$$E \vdash c, M_1, \Gamma, \underline{pc} \Rightarrow M_1', \Gamma_1' \text{ and } E \vdash c, M_2, \Gamma, \underline{pc} \Rightarrow M_2', \Gamma_2'$$

then:

$$M_1 \sim^s_\Gamma M_2 \implies \Gamma_2' = \Gamma_1' = \Gamma' \text{ and } M_1' \sim^s_{\Gamma'} M_2'.$$

This definition assumes two arbitrary terminating runs of a command c on input memories M_1 and M_2, that yield respective output memories M_1' and M_2'. If both input memories are s-equivalent, Definition 2 states that:

1. the output memories M_1' and M_2' are also s-equivalent, thus guaranteeing that command c does not leak data having a security level above s, and

2. the output security memories Γ_1' and Γ_2' are equal, thus guaranteeing that the behaviour of the monitor itself does not leak information.

3.2.4 Semantics of Expressions

Expressions in `PWhile` are side-effect-free. Hence, expressions evaluation do not modify initial security contexts described by security memories Γ. Thus, both left-value and right-value evaluation of expressions only yield an additional security level.

The monitor semantics of expressions take into account all information flows that are produced during the evaluation of an expression as either a left-value or a right-value. Intuitively, the `PWhile` semantics read values from a memory M in order to evaluate an expression. Hence, the resulting security level computed by the monitor semantics for expressions is a merge of all the security levels corresponding to the values that are read from memory M. Consider for example an environment E and two variables `x` and `y` whose statically allocated locations are respectively denoted by $l_x \triangleq E(x)$ and $l_y \triangleq E(y)$. Assume also a memory $M \triangleq \{l_x \mapsto ptr(l_y), l_y \mapsto v_y\}$ where variable `x` points to variable `y`. What values do the semantics read from memory M in order to compute the right-value – the content – of expression `*x`?

$$RV\ \frac{LV_{MEM}\ \dfrac{RV\ \dfrac{LV_{ID}\ \dfrac{E(x) = l_x}{E \vdash x, M \Leftarrow l_x} \qquad M(l_x) = ptr(l_y)}{E \vdash x, M \Rightarrow ptr(l_y)}}{E \vdash *x, M \Leftarrow l_y} \qquad M(l_y) = v_y}{E \vdash *x, M \Rightarrow v_y}$$

Figure 3.7: The initial computation of the right-value of expression `*x` by the `PWhile` semantics

Figure 3.7 illustrates the computations involved in the right-value evaluation of expression `*x`. As these computations show, the semantics reads two values from memory M:

- the value $M(l_x)$ in order to determine the location where x points to, and finally

- the value $M(l_y)$ in order to determine the right-value of expression `*x`.

Assuming a security memory $\Gamma \triangleq \{l_x \mapsto s_x, l_y \mapsto s_y\}$ describing the security levels associated to the content of memory M, the monitor semantics should account for both these information flows by computing a security level equal to $s_x \sqcup s_y$. Figure 3.8 illustrates the monitor semantics computations involved in the right-value evaluation of expression `*x`. Note that the computed security level s is in fact equal to $s_x \sqcup s_y$, which accounts for both information flows that are produced by reading both values $M(l_x)$ and $M(l_y)$. Indeed,

$$\dfrac{\dfrac{\dfrac{E(x) = l_x}{E \vdash x, M \Leftarrow l_x, public}\, LV_{ID} \qquad M(l_x) = ptr(l_y)}{E \vdash x, M \Rightarrow ptr(l_y), s'}\, RV \quad \Gamma(l_x) = s_x \qquad s' = public \sqcup s_x}{E \vdash *x, M \Leftarrow l_y, s'}\, LV_{MEM}$$

$$\dfrac{\dfrac{M(l_y) = v_y}{E \vdash *x, M \Rightarrow v_y, s}\, RV \quad \Gamma(l_y) = s_y \qquad s = s' \sqcup s_y}{E \vdash *x, M, \Gamma \Rightarrow v_y, s}\, RV$$

Figure 3.8: The computation of the right-value of expression `*x` by the monitor semantics

the left-value evaluation of expression `*x` yield a security level taking into account pointer-induced information flows due to dereferencing variable `x`. Additionally, the right-value evaluation of expression `*x` yields a security level taking into account both pointer-induced information flow and explicit information flow due to reading the value $M(l_y)$. Moreover, the left-value evaluation of `x` yields the bottom element of the lattice $\mathbb{S}$ – denoted by *public* –, since it involves no dereferences.

Figure 3.9 introduces the monitor semantics for expressions. Left-value evaluations of variables `id` involve no dereferences. Therefore, they result in the least security level of lattice $\mathbb{S}$, denoted by *public* (rule LV_{ID}). The right-value of expression `a` determines the location of expression `*a`. Hence, the security level of the right-value of expression `a` defines the security level of the left-value of expression `*a` (rule LV_{MEM}).

The right-value of constants `n` has the least security level of lattice $\mathbb{S}$, since attackers are assumed to know the source code of analysed programs (rule RV_{CONST}). The location l of an expression `a` as well as the value $M(l)$ stored in that location determine the right-value of `a`. Hence, the

$$LV_{ID}\ \frac{E(id) = l}{E \vdash id, M, \Gamma \Leftarrow l, public} \qquad LV_{MEM}\ \frac{E \vdash a, M, \Gamma \Rightarrow ptr(l), s}{E \vdash *a, M, \Gamma \Leftarrow l, s}$$

(a) Left-value evaluation rules

$$RV_{CONST}\ E \vdash n, M, \Gamma \Rightarrow n, public$$

$$RV\ \frac{E \vdash a, M, \Gamma \Leftarrow l, s_l \qquad M(l) = v \qquad s_r = \Gamma(l) \qquad s = s_l \sqcup s_r}{E \vdash a, M, \Gamma \Rightarrow v, s}$$

$$RV_{REF}\ \frac{E \vdash a, M, \Gamma \Leftarrow l, s}{E \vdash \&a, M, \Gamma \Rightarrow ptr(l), s}$$

$$RV_{UOP}\ \frac{E \vdash a, M, \Gamma \Rightarrow v, s \qquad uop\ v = v'}{E \vdash uop\ a, M, \Gamma \Rightarrow v', s}$$

$$RV_{BOP}\ \frac{\begin{array}{c} E \vdash a_1, M, \Gamma \Rightarrow v_1, s_1 \qquad E \vdash a_2, M, \Gamma \Rightarrow v_2, s_2 \\ v_1\ bop\ v_2 = v \qquad s_1 \sqcup s_2 = s \end{array}}{E \vdash a_1\ bop\ a_2, M, \Gamma \Rightarrow v, s}$$

(b) Right-value evaluation rules

Figure 3.9: Information flow monitor big-step semantics of expressions

right-value evaluation of `a` merges both `a`'s left-value security level and `a`'s content security level $\Gamma(l)$, in order to take into account both pointer-induced information flow and explicit information flow (rule RV). The left-value of expression `a` determines the right-value of expression `&a`. Hence, the security level of the right-value of `&a` is the same as the security level of the left-value of `a` (rule RV_{REF}). As for unary operators, the security level of the right-value of `a` defines the security level of `uop a` (rule RV_{UOP}). Similarly, for binary operators the merge of both security levels of the right-values of $\texttt{a}_1$ and $\texttt{a}_2$ defines the security level of the right-value of $\texttt{a}_1$ `bop` $\texttt{a}_2$ (rule RV_{BOP}).

In order to prove the soundness of the semantics rules for expressions, we prove a result that is similar to Definition 2 of TINI: the evaluation of an expression in two s-equivalent memories yield the same value provided that its security level is less than or equal to s. In fact, we do prove a

slightly more general property by taking advantage of the symmetry of both evaluations and introducing a partial order relation over security memories Γ in Definition 3. According to the definition of this partial order relation $\sqsubseteq_s$, if attackers who have a security clearance lesser than s are allowed to read the content of location l in the first run ($\Gamma_1(l) \sqsubseteq s$), then they are also allowed to read the content of location l in the second run ($\Gamma_2(l) \sqsubseteq \Gamma_1(l)$). Stated otherwise, the security memory Γ_2 of the second run is less restrictive than the security memory Γ_1 of the first run, up to security label s.

Definition 3 (Less restrictive up to label s ($\sqsubseteq_s$))**.**
For all $s \in \mathbb{S}$, for all Γ_1, Γ_2,

Γ_2 *is less restrictive than* Γ_1 *up to security label* s *($\Gamma_2 \sqsubseteq_s \Gamma_1$) iff :*

$$\textit{for all } l \in \mathit{Loc}, \Gamma_1(l) \sqsubseteq s \implies \Gamma_2(l) \sqsubseteq \Gamma_1(l).$$

Lemma 1 and Corollary 1 state that the semantics of expressions, presented in Figure 3.9, is sound. This proof, presented in Appendix A.1, is by induction on the evaluation rules of expressions.

Lemma 1 (L-value evaluation in s-equivalent memories)**.**
For all environments E, for all memories M_1, M_2, for all security memories Γ_1, Γ_2, for all security levels $s \in \mathbb{S}$, such that $\Gamma_2 \sqsubseteq_s \Gamma_1$ and $M_1 \sim^s_{\Gamma_1} M_2$, for all $a \in \mathit{Exp}$ such that $E \vdash a, M_1, \Gamma_1 \Leftarrow l_1, s_1$ and $E \vdash a, M_2, \Gamma_2 \Leftarrow l_2, s_2$ then

$s_1 \sqsubseteq s \implies l_1 = l_2$ *and* $s_2 \sqsubseteq s_1$.

Lemma 1 proves that the left-value evaluation of expressions in two s-equivalent memories yields equal values if the output security label of the first run is lesser than s. Additionally, note that if both input security memories Γ_1 and Γ_2 are equal, then the left-value evaluation of expressions in two s-equivalent memories also yield two equal security labels provided that one of the output security labels is lesser than s. The proof of this lemma is by induction on the left-value evaluation of expressions. All proofs of this paper can be found in the accompanying appendix. Corollary 1 formalizes the same results for the right-value evaluation of expressions: the right-value evaluation in two s-equivalent memories yield the same value if the resulting security label is lesser than s.

Corollary 1 (R-value evaluation in s-equivalent memories)**.**
For all environments E, for all memories M_1, M_2, for all security memories Γ_1, Γ_2, for all security levels $s \in \mathbb{S}$, such that $\Gamma_2 \sqsubseteq_s \Gamma_1$ and $M_1 \sim^s_{\Gamma_1} M_2$, for all $a \in \mathit{Exp}$ such that $E \vdash a, M_1, \Gamma_1 \Rightarrow v_1, s_1$ and $E \vdash a, M_2, \Gamma_2 \Rightarrow v_2, s_2$ then

$s_1 \sqsubseteq s \implies v_1 = v_2$ *and* $s_2 \sqsubseteq s_1$.

3.2.5 Semantics of Instructions

The evaluation of instructions takes as input a security memory Γ as well as a security context $\underline{pc}$. The security memory Γ describes security levels corresponding to the values stored in memory M, whereas the security context $\underline{pc}$ describes the context of execution for instructions.

```
1 if (user_input == secret){
2     y := 1;
3 }
4 else {
5     z :=1;
6 }
7 x := 1;
```

Listing 3.1: Implicit information flows

Implicit flows. The label $\underline{pc}$ represents a security label that is attached to the program counter. Consider the example program in Listing 3.1 where the security label of variable `secret` is $High$. If attackers have access to either variables `y` or `z` at the end of this program, they may deduce information about variable `secret`. Hence, there are two information flows; a first one from `secret` to `y` and a second one from variable `secret` to `z`. Both these implicit information flows are due to assignments that occur inside a conditional depending on $High$ variables. Therefore, the monitor maintains a security label – denoted by $\underline{pc}$ – that accounts for these implicit dependencies.

Listing 3.2 is an annotated version of Listing 3.1 illustrating how the `PWhile` monitor handles the security context $\underline{pc}$. When a conditional instruction is about to be executed, the monitor evaluates the security level of the conditional guard, then propagates this security label to the current security context $\underline{pc}$. Consequently, the monitor evaluates either the then-branch or the else-branch in the newly computed security context $\underline{pc}'$ in order to account for implicit information flows occurring inside conditional branches. The annotations in Listing 3.2 assume that the conditional guard evaluates to true. Hence, since the assignment at Line 2 occurs in a high security context $\underline{pc}'$, the monitor labels variable `y` as $High$ in order to account for the implicit flow from `secret` to `y`. Note that after the conditional instruction – at Line 7 –, the monitor lowers the security context to the old label $\underline{pc}$, since execution of these instructions does not depend

on the conditional guard anymore. This operation is transparent in the monitor semantics thanks to a big-step semantics that simplifies greatly the evaluation rules.

```
0 // initialised to pc = Low
1 if (user_input == secret){ // pc' = pc ⊔ High
2    y := 1; // evaluated in pc'
3 }
4 else {
5    z := 1; // not executed
6 }
7 x := 1; // evaluated in pc
```

Listing 3.2: The monitor handling of the security context for Listing 3.1

Hybrid information flow monitor. The `PWhile` monitor is a hybrid monitor combining both dynamic and static analyses techniques in order to soundly track information flows. Consider for instance the previous example program in Listing 3.1 where variable `secret` is $High$. Assuming that the conditional guard evaluates to true, a purely dynamic monitor is unaware of information flows due to the else-branch since the control flow goes only through the then-branch. Therefore, a dynamic monitor misses the implicit flow involving variable `z`; an attacker may learn information on variable `secret` by having access to variable `z`. This program would leak information unless the monitor can rely on a static analysis of the non-executed branches in order to label `z` as $High$ and prevent attackers from reading its value.

The `PWhile` monitor relies on the result of a static analysis denoted by S_P. This static analysis provides the set of locations possibly written by instructions of the analysed program, in order to guarantee the monitor soundness. Listing 3.3 illustrates how the monitor accounts for implicit flows that are due to non-executed branches. Assuming that the conditional guard evaluates to true, the monitor evaluates the then-branch in a $High$ security context $\underline{pc'}$. Before leaving the then-branch, the monitor also propagates the security context $\underline{pc'}$ to the set of locations that may be written in the else-branch. Hence, the monitor tags the location l_z of variable `z` with a $High$ security label in order to account for the implicit information flow from `secret` to `z`.

```
1 if (user_input == secret){
2     y := 1; // evaluated in pc'
3 } // propagating pc' to the set S_P(z := 1) of locations
4 else {
5     z := 1; // not executed
6 }
7 x := 1; // evaluated in pc
```

Listing 3.3: The monitor handling of implicit flows for Listing 3.1

Pointer-induced information flows. The static analysis S_P also proves useful to account for information flows in the presence of pointers. Consider for example the program in Listing 3.4. This program is semantically equivalent to the previous example in Listing 3.1. Therefore, the `PWhile` monitor should label at least variables `y` and `z` as *High* in order to prevent attackers from learning sensitive information about variable `secret`. Let us first validate this intuition; may attackers learn sensitive information if they have access to either variable `y` or `z`?

```
0 // Initialised to pc = Low
1 if (user_input == secret){ // pc' = High
2     p := &y;
3 }
4 else {
5     p := &z;
6 }
7 x := 1;
8 *p := 1;
```

Listing 3.4: Pointer-induced information flows

If attackers know the value of variable `y`, they may learn that the conditional guard evaluates to true (resp. false) when `y` is equal to 1 (resp. different from 1). Hence, attackers may learn that the sensitive variable `secret` is either equal to or different from the user-supplied input. This proves that there is indeed an information flow from variable `secret` to variable `y`. Similarly, there is also an information flow from variable `secret` to variable `z`. Consequently, two questions come to mind. How can we explain both these information flows? And how can the `PWhile` monitor account for these information flows?

First of all, both assignments to variable p at Lines 2 and 5 occur in a *High* context since the conditional guard depends on variable secret. Hence, there is an implicit information flow from variable secret to variable p. Additionally, the assignment at Line 8 reads the value of variable p in order to dereference it and determine which location need to be updated. Hence, there is also an information flow from variable p to expression ⋆p. Finally, since ⋆p can point to both variables y and z, this explains by transitivity the information flow from variable secret to both variables y and z.

Note that the assignment to expression ⋆p actually generates a pointer-induced information flow from p to all variables that expression ⋆p may point to. Even when the conditional evaluates to true – thus, p points to y –, the fact that the assignment at Line 8 does not modify variable z can itself leak information. This behaviour is indeed similar to implicit information flows due to non-executed branches. Therefore, similarly to the treatment of implicit flows, the PWhile monitor relies on the static analysis S_P in order to soundly account for these pointer-induced information flows.

```
0 // Initialised to pc = Low
1 if (user_input == secret){ // pc' = High
2    p := &y; // p gets labelled as High
3 } // propagating pc' to the set S_P(p := &z) of locations
4 else {
5    p := &z; // not executed
6 }
7 x := 1; // evaluated in pc
8 *p := 1; // propagating the label of p to the set
9 // S_P(*p := 1) of locations.
```

Listing 3.5: The monitor handling of pointer-induced flows for Listing 3.4

Listing 3.5 illustrates how the monitor handles these pointer-induced information flows for the example program in Listing 3.4. Assuming the conditional guard evaluates to true, the PWhile monitor labels p as *High* at Line 2 since this assignment occurs in a *High* security context. Then, before leaving the then-branch, the monitor propagates the security context $\underline{pc'}$ to the locations that are modified in the else-branch by relying on the static analysis S_P; the security label of p does not change since the monitor already tagged it as *High*. At Line 7, the monitor lowers the security context to the old label $\underline{pc}$ since the control flow does not depend on the conditional instruction anymore. At Line 8, the monitor propagates the label of variable

p to variable y since the write to y involves first reading the value of p in order to dereference it; hence, the monitor labels variable y as *High*. Additionally, the PWhile monitor also propagates the label of p to the set $S_P(*p := 1)$ of locations that may be written by the assignment in order to account for all pointer-induced information flows. Consequently, the monitor labels variables y and z as *High* in order to prevent information leaks due to pointer-induced information flows.

Propagating only the security label of p to the set $S_P(*p := 1)$ of locations at Line 8 is sufficient to track information flows soundly. Yet, it does not guarantee the monitor correctness [LG08] : the behaviour of the monitor itself can leak information. In fact, at Line 8, the PWhile monitor also propagates the security context $\underline{pc}$ to the set $S_P(*p := 1)$ of locations that may be written by the assignment `*p := 1`. *The following example in Listing 3.6 illustrates the problem by assuming that the monitor only propagates the security level of* p *to the set* $S_P(*p := 1)$ *of locations.*

Monitor correctness. Consider the example program in Listing 3.6, where only variable secret is *High*. Let us suppose that the results of the static analysis S_P are imprecise, by assuming that the computed set of written locations at Line 8 is $\{l_y, l_z\}$, where l_y (resp. l_z) stands for the location of variable y (resp. the location of variable z). This assumption is likely, since the conditional guard at Line 1 is an opaque predicate that always evaluates to true and the static analysis S_P might not find out that the else-branch of the conditional at Line 1 is unreachable.

```
1  if ((x² + x) mod 2 == 0) {
2    p := &y;
3  }
4  else {
5    p := &z;
6  }
7  if (user_input == secret) {
8    *p := 1; // assumption: S_P(*p := 1) = {l_y, l_z}
9  }
10 else {
11   skip;
12 }
```

Listing 3.6: The monitor handling of pointer-induced flows for Listing 3.4

Listings 3.7 and 3.8 illustrate how the `PWhile` monitor handles the program in Listing 3.6, depending on the evaluation result of the conditional guard at Line 7. Since the conditional at Line 1 manipulates only *Low* variables, all variables but variable `secret` are still *Low* before entering the conditional at Line 7. Thus, let us compare the results of the `PWhile` monitor in both cases past this program point.

Listing 3.7 assumes the conditional guard at Line 7 evaluates to true. Since this conditional guard depends on variable `secret`, the monitor creates a new security context $\underline{pc}''$ that is *High*. At the assignment of Line 8 in the then-branch, the monitor labels variable `y` as *High* since `*p` points to `y`, and the assignment to `y` occurs in a *High* security context $\underline{pc}''$. Then, the monitor propagates the security level *Low* of the right-value of `p` to the set $S_P(*p := 1)$ of locations; the security level of variable `y` is still *High* whereas the security level of variable `z` is *Low*. Then, before leaving the then-branch, the `PWhile` monitor propagates $\underline{pc}''$ to the set of locations that may be written in the else-branch. Since this set is empty, no security levels change. Hence, both variables `secret` and `y` end up being labelled as *High*, whereas the other variables are *Low*.

```
0  // Initialised to pc = Low
1  if ((x^2 + x) mod 2 == 0) { // pc' = Low
2    p := &y;
3  }
4  else {
5    p := &z; // not executed
6  }  // secret is High, whereas p,y, and z are Low
7  if (user_input == secret) { // pc'' = High
8    *p := 1; // y gets labelled as High
9  }
10 else {
11   skip; // not executed
12 }
13 // secret and y are High, whereas p, x and z are Low
```

Listing 3.7: The monitor handling for Listing 3.6 (case 1)

Listing 3.8 assumes the conditional at Line 7 evaluates to false. The monitor evaluates the security level of the conditional guard, then creates a new security context $\underline{pc}''$ that is *High*. At the `skip` instruction of Line 11, the monitor modifies no security levels. Then, before leaving the else-branch, the `PWhile` monitor propagates $\underline{pc}''$ to the set $S_P(*p := 1)$ of locations

```
0  // Initialised to pc = Low
1  if ((x^2 + x) mod 2 == 0) {
2    p := &y; // p gets labelled as low
3  }
4  else {
5    p := &z; // not executed
6  }  // secret is High, whereas p,y, and z are Low
7  if (user_input == secret) {  // pc'' = High
8    *p := 1; // not executed
9  }
10 else {
11   skip;
12 }
13 // secret, y and z are High, whereas p, x are Low
```

Listing 3.8: The monitor handling for Listing 3.6 (case 0)

that may be written in the then-branch. Therefore, the monitor labels both variables y and z as *High*. Hence, variables secret, y and z end up being labelled as *High*, whereas the other variables are *Low*.

Note that in both cases illustrated by Listings 3.7 and 3.8, the resulting security level for variable z is different. Hence, the behaviour of the monitor itself can leak information since attackers can deduce which branch is executed by looking up the security level of variable z. In order to mitigate this, the PWhile monitor ensures that variable z is also labelled as *High* in the case of Listing 3.7, when the conditional guard at Line 7 evaluates to true. The monitor achieves this by propagating the security context $\underline{pc''}$ – in addition to the security level of p – to the set $S_P(*p := 1)$ of locations that may be written by the assignment at Line 8.

Formal semantics. Figure 3.10 introduces the information flow monitor semantics for instructions. This semantics is a big-step one as it is more convenient to reason about information flow properties. Indeed, since a big-step semantics describes the behaviour of a term in terms of the behaviour of its subterms, the control flow is explicit. Therefore, it is easier to formalize implicit information flows by relying on a big-step semantics. In particular, there is no need to pinpoint immediate post-dominators of conditional branches in order to formalize the semantics of implicit information flows.

Instructions skip modify neither the input value memory nor the input security memory (rule *Skip*). Assignment instructions a_1 := a_2 read the

$$(Skip)\ E \vdash skip, M, \Gamma, \underline{pc} \Rightarrow M, \Gamma$$

$$(Assign)\ \frac{\begin{array}{c} E \vdash a_1, M, \Gamma \Leftarrow l_1, s_1 \qquad E \vdash a_2, M, \Gamma \Rightarrow v_2, s_2 \\ s = s_1 \sqcup s_2 \sqcup \underline{pc} \qquad s' = s_1 \sqcup \underline{pc} \qquad M' = M[l_1 \mapsto v_2] \\ \Gamma'' = \Gamma[l_1 \mapsto s] \qquad \Gamma' = update(a_1 = a_2, s', \Gamma'') \end{array}}{E \vdash a_1 := a_2, M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'}$$

$$(Comp)\ \frac{E \vdash c_1, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma_1 \qquad E \vdash c_2, M_1, \Gamma_1, \underline{pc} \Rightarrow M_2, \Gamma_2}{E \vdash c_1; c_2, M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2}$$

$$(If_{tt})\ \frac{\begin{array}{c} E \vdash a, M, \Gamma \Rightarrow v, s \qquad istrue(v) \qquad \underline{pc}' = s \sqcup \underline{pc} \\ E \vdash c_1, M, \Gamma, \underline{pc}' \Rightarrow M_1, \Gamma_1 \qquad \Gamma_1' = update(c_2, \underline{pc}', \Gamma_1) \end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma_1'}$$

$$(If_{ff})\ \frac{\begin{array}{c} E \vdash a, M, \Gamma \Rightarrow v, s \qquad isfalse(v) \qquad \underline{pc}' = s \sqcup \underline{pc} \\ E \vdash c_2, M, \Gamma, \underline{pc}' \Rightarrow M_2, \Gamma_2 \qquad \Gamma_2' = update(c_1, \underline{pc}', \Gamma_2) \end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2, M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2'}$$

$$(While_{tt})\ \frac{\begin{array}{c} E \vdash a, M, \Gamma \Rightarrow v, s \qquad istrue(v) \\ \underline{pc}' = s \sqcup \underline{pc} \qquad E \vdash c, M, \Gamma, \underline{pc}' \Rightarrow M', \Gamma' \\ E \vdash while\ (a)\ c, M', \Gamma', \underline{pc} \Rightarrow M'', \Gamma'' \end{array}}{E \vdash while\ (a)\ c, M, \Gamma, \underline{pc} \Rightarrow M'', \Gamma''}$$

$$(While_{ff})\ \frac{\begin{array}{c} E \vdash a, M, \Gamma \Rightarrow v, s \qquad isfalse(v) \\ \underline{pc}' = s \sqcup \underline{pc} \qquad \Gamma' = update(c, \underline{pc}', \Gamma) \end{array}}{E \vdash while\ (a)\ c, M, \Gamma, \underline{pc} \Rightarrow M, \Gamma'}$$

$$update(c, s, \Gamma) \triangleq \begin{cases} \Gamma(l) & \forall l \notin S_P(c) \\ \Gamma(l) \sqcup s & \forall l \in S_P(c) \end{cases}$$

Figure 3.10: Information flow monitor big-step semantics of instructions

left-value of $\mathtt{a}_1$ in order to determine the location to update. They also read the right-value of $\mathtt{a}_2$ in order to determine the value to write. Hence, assignments generate an information flow from both the left-value of $\mathtt{a}_1$ and the right-value of $\mathtt{a}_2$ toward the value that is stored. Additionally, assignments occur in an execution context $\underline{pc}$ capturing implicit flows. Consequently, the monitor accounts for these three information flow by merging the security levels s_1, s_2 and $\underline{pc}$ that corresponds respectively to the left-value of $\mathtt{a}_1$, the right-value of $\mathtt{a}_2$ and the program counter. The monitor semantics then maps the computed security label to the written location l_1. Additionally, the monitor relies on the *update* operator in order to propagate the security context $\underline{pc}$ and the security level s_1 to the set $S_P(a_1 := a_2)$ of locations that may be written by the assignment $\mathtt{a}_1$ `:=` $\mathtt{a}_2$, in order to account for pointer-induced information flows (rule $Assign$).

The operator $update(c, s, \Gamma)$ updates the security memory Γ by modifying only the values mapped to the set $S_P(c)$ of locations that may be written by instruction c. It propagates the security level s to these values.

For a sequence $c_1;\ c_2$ of two instructions, the monitor evaluates the first instruction c_1, then uses both output memories M_1 and Γ_1 as an input to evaluate the second instruction c_2. The monitor evaluates both instructions in the same security context $\underline{pc}$ since execution of c_2 does not depend on conditionals occurring in c_1 – recall the monitor semantics is a big-step semantics – (rule $Comp$).

Conditional instructions generate implicit information flows from the conditional guard to the locations that are modified in either the then-branch or the else-branch. Hence, the monitor semantics create a new security context $\underline{pc'}$ by merging the old security context $\underline{pc}$ and the security level s of the conditional guard. If the conditional guard evaluates to true, the monitor executes the then-branch in the new security context $\underline{pc'}$ in order to account for implicit information flows in this branch. Additionally, the monitor also propagates the security context $\underline{pc'}$ to the set $S_P(c_2)$ of locations that may be written by the else-branch. Thus, the monitor also accounts for implicit flows due to the non-executed branch (rules If_{tt} and If_{ff}).

Loop instructions generate implicit information flows from the loop guard to the locations that may be written in the loop body. Hence, the monitor creates a new security context $\underline{pc'}$ by merging the old security context $\underline{pc}$ and the security level s of the loop guard. If the loop guard evaluates to true, the monitor evaluates the loop body in the new security context $\underline{pc'}$ (rule $While_{tt}$). Otherwise, if the loop guard evaluates to false, the monitor propagates the new security context $\underline{pc'}$ to the location that may be written by the loop body (rule $While_{ff}$).

Soundness. Theorem 1 proves that the `PWhile` monitor is sound with respect to TINI. It ensures the monitor semantics soundly tracks information flows produced by the monitored program. Moreover, it also guarantees that the behaviour of the monitor itself does not leak information.

Theorem 1 (Soundness)**.**
The `PWhile` monitor semantics is sound with respect to TINI as introduced in Definition 2.

The proof of Theorem 1 can be found in Appendix A.2. This proof is by induction on the evaluation judgement of instructions. It relies on both Lemma 1 and Corollary 1.

3.3 Related Work

Our `PWhile` information flow monitor is inspired by the monitor proposed by Le Guernic et al. [LGBJS06] as well as Russo and Sabelfeld [RS10]. It is a flow-sensitive hybrid monitor relying on a prior static analysis computing the set of locations that may be modified by an instruction. Thus, it enjoys the same permissive properties as Le Guernic et al.'s monitor as well as Russo and Sabelfeld's monitor. Our monitor also extends the language supported by both previous monitors to include pointers. Although the presented `PWhile` monitor does not support output instructions, this extension is straightforward as we illustrate in [ASTT13a]. We also make the same choice as Le Guernic et al., by formalizing the semantics of our monitor as a big-step semantics. This choice simplifies greatly the handling of implicit information flows.

Moore and Chong [MC11] also extend Russo and Sabelfeld's monitor with dynamically allocated references, allowing different sound memory abstractions. In our semantics, we use the most precise instantiation of their memory abstraction where each concrete location correspond to one abstract location. While Moore and Chong argue that it is undecidable in the general case to determine which locations might be updated by an instruction, we believe that, for the sake of permissiveness, it is necessary to be as precise as possible at least for the set of finite statically allocated locations. Moore and Chong also propose a static analysis determining when their monitor can stop tracking the security level of some variables that pose no security threats. This static analysis offers the opportunity to reduce the runtime overhead, while still guaranteeing the soundness of their monitor.

3.4 Summary

We have formalized in this chapter a flow-sensitive information flow monitor for the `PWhile` language, an imperative language supporting both pointers and aliasing. Our hybrid monitor relies on a prior static analysis computing the set of locations that may be written by instructions of the analysed program. Indeed, Russo and Sabelfeld [RS10] prove that a flow-sensitive dynamic monitor must rely on a prior static analysis in order to guarantee its soundness. We also prove the soundness of our monitor with respect to TINI, as introduced in Definition 2. Our monitor semantics ignores diverging runs since it is inspired by a simple version of the Clight big-step semantics stripped of coinduction [BL09]. As pointed by Le Guernic et al. [LGBJS06], this is not problematic when dealing with TINI because we ignore non-termination covert channels.

We investigate in the next chapter how such an information flow monitor can be implemented.

Chapter 4

Inlining Approach

Formalizing the `PWhile` monitor semantics and proving its soundness is a first step toward provable secure information flow. Implementing such a monitor is a second step.

One approach to implement such a security monitor consists in modifying the target runtime environment. Such a modification can be achieved by enhancing virtual machines with security checks in the case of interpreted languages such as the Java bytecode or JavaScript. Additionally, in the case of compiled languages such as the C language or OCaml, this approach may involve redesigning the underlying hardware in order to enhance it with monitoring capabilities.

A second approach consists in modifying the target application in order to instrument its source code with security checks. This inlining approach is how we choose to tackle this problem, since it has the benefits of being agnostic related to the underlying runtime environment; the instrumented application can be compiled ahead of time, interpreted or even "simulated" by a static analyser.

4.1 Security Labels

Inlining an information flow monitor consists in defining a program transformation that instruments the target program in order to insert security checks. This program transformation maps initial variables of the target program to shadow variables that account for security levels. It also generates instructions involving shadow variables in order to track information flows according to the `PWhile` monitor semantics.

```
1  label secret_label :=1;
2  int secret; // variable secret  is tagged as High
3  label user_input_label := 0 ;
4  int user_input;
5  label x_label := 0 ;
6  int x;
7  label y_label := 0 ;
8  int y;
9  label z_label := 0 ;
10 int z;
11 label pc0 := 0 ;
12 label pc1 := pc0 or user_input_label or secret_label
13 if (user_input == secret){
14  y_label := 0 or pc1 ;
15  y := 1;
16  z_label:=z_label or pc1;//Else-branch's implic. flows
17 }
18 else {
19  z_label := 0 or pc1 ;
20  z :=1;
21  y_label:=y_label or pc1;//Then-branch's implic. flows
22 }
23 x_label := 0 or pc0;
24 x := 1;
```

Listing 4.1: Inlining the program in Listing 3.1

Integer variables. Listing 4.1 illustrates the inlining approach for the example program in Listing 3.1. This program transformation introduces a shadow variable for each integer variable of the initial program. It also initialises each shadow variable according to the security level of the initial variable it represents. For instance, the example program in Listing 4.1 assumes initially that only variable `secret` is *High*, whereas the other variables are *Low*. As an implementation choice, this example implements the type `label` as an integer type; the integer 1 (resp. the integer 0) represents the security level *High* (resp. the security level *Low*). It also implements the union operator $\sqcup$ as the logical operator `or`. This representation is indeed a sound implementation of the two-point lattice presented in Figure 3.5 since it preserves its lattice structure. Indeed, merging any security label with itself yields the same label. Additionally, merging *Low* and *High* yields

High (`0 or 1` yields `1`).

This program transformation also introduces a variable `pc0` – at Line 11 – representing the initial security context in order to account for implicit flows. Since the program runs initially in a *Low* security context, `pc0` is initialised to *Low*. At Line 12, the program transformation creates a new security context variable `pc1` for the conditional at Line 13. `pc1`'s value is the merge of both the initial context `pc0` and the security level `user_input_label or secret_label` of the conditional guard. Then, according to the `PWhile` monitor semantics, the program transformation assigns to the security label of variable `y` the value of `0 or pc1` since `y` is assigned a constant (having a security level *Low*) in a security context depending on `pc1`. Furthermore, at Line 16, the program transformation relies on the static analysis S_P to account for implicit flows due to the else-branch; it propagates the security context variable `pc1` to the security label `z_label` of variable `z`. Finally, at Line 23, the security label `x_label` gets assigned the value of `0 or pc0` since variable `x` is assigned a constant in a security context that only depends on `pc`.

Pointers. In the case of programs handling pointers, the program transformation needs to handle aliasing in a correct way. Consider for example the programs in Listings 4.2 and 4.4. At the end of both programs, the value of variable `x` is read and assigned to variable `z`. Thus, the inlining approach should manage to read the security label of variable `x` at the end of both programs, whether accessed directly through `x` or through dereferencing the pointer `p`.

The program transformation handles such aliasing relations between variables by assigning multiple shadow variables to variables of type pointer. For instance, since `p` is a pointer to an integer, the program transformation maps it to two shadow variables:

1. a variable `p_label` of type label that represents the security level of `p` itself, and

2. a pointer variable `*p_label_d1` of type pointer to a type label, that represents the security level of `*p`.

Hence, when the program in Listing 4.4 creates an aliasing relation by assigning `&x` to `p` at Line 4, the program transformation also reproduces the same aliasing relation for the corresponding security labels: it assigns the address `&x_label` of `x`'s label to the label `p_label_d1` of `*p`, as illustrated by Line 9 of the inlined program at Listing 4.5. This way, the program transformation can access the security label of `x` simply by dereferencing

```
1 int x;
2 int z;
3 z := x;
```

Listing 4.2: Direct access to variable x

```
1 label x_label ;
2 int x;
3 label z_label := 0 ;
4 int z;
5 z_label := x_label ;
6 z := x;
```

Listing 4.3: Inlining of Listing 4.2

```
1 int x;
2 int z;
3 int *p;
4 p := &x;
5 z := *p;
```

Listing 4.4: Accessing variable x through dereferencing

```
1 label x_label ;
2 int x;
3 label z_label ;
4 int z;
5 label p_label ;
6 label *p_label_d1 ;
7 int *p;
8 p_label := 0;
9 p_label_d1 := &x_label ;
10 p := &x;
11 z_label := p_label or
*p_label_d1;
12 z := *p;
```

Listing 4.5: Inlining of Listing 4.4

p_label_d1 at Line 11. Note that both inlined versions presented at Listings 4.3 and 4.5 do compute the same security level for variable z, whether the program accesses it directly or through dereferencing the pointer p. A constant propagation can make this fact even more explicit by optimizing the term p_label or *p_label_d1 to *p_label_d1 at Line 11 of Listing 4.5, since the value of p_label is constant and equal to 0. In general, the program transformation does not attempt to do such optimizations in the resulting programs in order to stick to the PWhile monitor semantics and keep the proof of soundness less cumbersome. Such optimizations can still be performed afterwards by the compiler or a dedicated optimizing program transformation.

Shadow variables. Henceforth, we extend the `PWhile` language in order to formalize the monitor inlining approach. We denote by τ_s a new type representing security labels. We also extend the range of memories M to $\mathbb{V} \cup \mathbb{S}$ to include security levels of a lattice $\mathbb{S}$. Moreover, we extend binary operators `bop` to include the union operator $\sqcup$ of the lattice $\mathbb{S}$.

The approach that consists in mapping pointer variables to multiple shadow variables generalizes to any pointer. The number of shadow variables mapped to a variable `x` depends on how many times such variable `x` can be dereferenced. Definition 4 introduces such a quantity as the depth of a variable `x`. For instance, an integer variable has a depth of 0 since it cannot be dereferenced, whereas a pointer to an integer has a depth of 1.

Definition 4 (Depth $\mathscr{D}(x)$ of variable `x`)**.**
Let τ_x be the type of variable `x`. The depth of `x` is defined as:

$$\mathscr{D}(x) = \mathscr{D}(\tau_x) = \begin{cases} 0 & \text{if } \tau = \kappa \\ 1 + \mathscr{D}(\tau') & \text{if } \tau_x = ptr(\tau'). \end{cases}$$

Definition 5 introduces a bijection Λ that maps initial variables of the target program to their shadow variables. We denote by $Var(P)$ the set of initial variables of the target program P. The range Var' of the mapping Λ and the set $Var(P)$ of initial variables must be disjoint in order for the program transformation to preserve the behaviour of target programs. Furthermore, the mapping Λ must be injective to soundly reproduce the semantics of the `PWhile` monitor.

Definition 5 (Bijection Λ)**.**
Let $Var(P)$ be the set of initial variables of the target program, and $Var' \subset Var \setminus Var(P)$ be a set of shadow variables.

$\Lambda : \{(x, k) : x \in Var(P)$ and $k \in [0, \mathscr{D}(x)]\} \to Var'$ is a bijection mapping each initial variable `x` to exactly $\mathscr{D}(x)+1$ shadow variables, denoted Λ`(x, k)`, such that Λ`(x, k)` has a type $ptr^{(k)}(\tau_s)$ for all $k \in [0, \mathscr{D}(x)]$.

The bijection Λ maps an initial variable `x` of the target program to exactly $\mathscr{D}(x) + 1$ shadow variables. By convention, we let $ptr^{(0)}(\tau)$ denote the type τ. For instance, assuming that variable `x` has a type $ptr(ptr(\kappa))$, then `x` has a depth of 2. Consequently, the bijection Λ maps `x` to 3 shadow variables:

1. Λ`(x, 0)` which has a type τ_s of a security label, and

2. Λ`(x, 1)` which has a type $ptr(\tau_s)$ of a pointer to security label, and

Table 4.1: The mapping Λ used in Listing 4.5

Λ(x,0)	x_label
Λ(z,0)	z_label
Λ(p,0)	p_label
Λ(p,1)	p_label_d1

3. Λ(x,2) which has a type $ptr(ptr(\tau_s))$ of a pointer to pointer to security label.

Intuitively, Λ(x,0) yields the security level of x, whereas ⋆Λ(x,1) yields the security level of ⋆x, and finally, ⋆⋆Λ(x,2) yields the security level of ⋆⋆x.

If you recall the example program in Listing 4.4 and its inlined version in Listing 4.5 for instance, Table 4.1 illustrates the mapping Λ that is used.

4.2 Expressions

This section introduces operators that compute the security label of expressions in terms of shadow variables. These operators reproduce exactly the PWhile monitor semantics for expressions.

The monitor semantics presented in Figure 3.9 include inductive rules for left-value evaluations and right-value evaluations of expressions. Thus, we inductively define two operators denoted by $\mathcal{L}_L$ and $\mathcal{L}_R$ that respectively compute the security label resulting from a left-value and a right-value evaluation. Additionally, the right-value evaluation rule *RV* in Figure 3.9(a) accesses the security memory Γ. Since inlining a monitor consists in encapsulating the security memory Γ in the value memory M of values – in broad outline –, accesses to the security memory Γ in the semantics are replaced by accesses to memory M in the inlined program. Hence, we also introduce a third operator denoted by $\mathcal{L}$ that represents the accesses to memory M through shadow variables.

In order to facilitate the notations, we introduce a syntactic sugar that extends the bijection Λ to all expressions that may occur in a left-value position. For instance, the syntactic sugar Λ(⋆x,0) denotes the shadow security label of expression ⋆x. Thus, desugaring Λ(⋆x,0) results in the expression ⋆Λ(x,1) which has a type τ_s. Likewise, the syntactic sugar Λ(⋆x,1) denotes a pointer to the shadow security label of the expression resulting from dereferencing one more time expression ⋆x. Thus, desugaring

Λ`(*x,1)` results in the expression $\star\Lambda$`(x,2)` which has a type $ptr(\tau_s)$. Definition 6 introduces this extension of the mapping Λ. As a special case, we also denote by $\star^{-1}$`x` the referencing expression `&x`, such that Λ`(&x,1)` also denotes a pointer to the shadow security label of the expression resulting from dereferencing one more time the right-value expression `&x`. Thus, desugaring Λ`(&x,1)` results in `&`Λ`(x,0)`.

Definition 6 (Extension of the bijection Λ)**.**
For all $x \in Var(P)$, for all $r \in [-1, \mathscr{D}(x)]$, for all $k \in [0, \mathscr{D}(*^r x)]$, then the bijection Λ extends to left-value expressions as well as the referencing expression as follows:

$$\Lambda(*^r x, k) \triangleq *^r \Lambda(x, k + r).$$

$$\mathcal{L}_L(id) \triangleq public \qquad \mathcal{L}_L(*a) \triangleq \mathcal{L}_R(a)$$

(a) Inlining left-value evaluation rules

$$\mathcal{L}_R(n) \triangleq public \qquad \mathcal{L}_R(uop\ a) \triangleq \mathcal{L}_R(a) \qquad \mathcal{L}_R(\&a) \triangleq \mathcal{L}_L(a)$$

$$\mathcal{L}_R(a_1\ bop\ a_2) \triangleq \mathcal{L}_R(a_1) \sqcup \mathcal{L}_R(a_2) \qquad \mathcal{L}(a) \triangleq \Lambda(a, 0)$$

$$\mathcal{L}_R(a) \triangleq \mathcal{L}_L(a) \sqcup \mathcal{L}(a) \text{ (if } a \text{ is a left-value)}$$

(b) Inlining right-value evaluation rules

Figure 4.1: Operators $\mathcal{L}_L$, $\mathcal{L}_R$ and $\mathcal{L}$

Figure 4.1 introduces the definition of the three operators $\mathcal{L}_L$, $\mathcal{L}_R$ and $\mathcal{L}$. The operator $\mathcal{L}_L$ – introduced in Figure 4.1(a) – defines the security label of left-values in terms of shadow variables, whereas the operator $\mathcal{L}_R$ – introduced in Figure 4.1(b) – defines the security label of right-values. In the `PWhile` monitor semantics presented in Figure 3.9(a), the left-value evaluation of variables `id` yields the security level *public* (rule LV_{ID}), so does $\mathcal{L}_L(id)$. Similarly, $\mathcal{L}_L(*a)$ accounts for the security level of expression $\star$`a` evaluated as a left-value. Hence, $\mathcal{L}_L(*a)$ yields $\mathcal{L}_R(a)$ which accounts for the security level of expression `a` evaluated as a right-value, according to the monitor semantics (rule LV_{MEM}). Likewise, the operator $\mathcal{L}_R$ reproduces the `PWhile` monitor semantics of the right-value evaluation rules presented in Figure 3.9(b). For instance, let us assume a variable `x` of type pointer to

an integer ($ptr(\kappa)$). Figure 4.2 illustrates the computation of the security label resulting from the right-value evaluation of expression `*x`.

$$
\begin{aligned}
\mathcal{L}_R(*x) &= \mathcal{L}_L(*x) \sqcup \mathcal{L}(*x) && \text{By definition of } \mathcal{L}_R(a)\\
&= \mathcal{L}_R(x) \sqcup \mathcal{L}(*x) && \text{By definition of } \mathcal{L}_L(*a)\\
&= \mathcal{L}_L(x) \sqcup \mathcal{L}(x) \sqcup \mathcal{L}(*x) && \text{By definition of } \mathcal{L}_R(a)\\
&= public \sqcup \Lambda(x,0) \sqcup \Lambda(*x,0) && \text{By definition of } \mathcal{L}_L(id) \text{ and } \mathcal{L}(a)\\
&= \Lambda(x,0) \sqcup *\Lambda(x,1) && \text{By definition of the extension } \Lambda
\end{aligned}
$$

Figure 4.2: The security label resulting from the right-value evaluation of expression `*x`

4.3 Aliasing Invariant

The inlining approach maintains an aliasing invariant in order to correctly handle pointers. Essentially, if a pointer `p` points to an integer variable `x`, the program transformation ensures that the shadow variable Λ`(p,1)` points to the shadow variable Λ`(x,0)`. This way, whenever the inlined program reads an integer through `*p` or `x`, it also reads the same security label through either `*`Λ`(p,1)` or Λ`(x,0)`.

Definition 7 introduces the aliasing relation stating that two expressions are aliased in a memory M if and only if they both evaluate to the same left-value in memory M. For instance, if a pointer `p` points to an integer variable `x` in a memory M, then `*p` and `x` are aliased in memory M ($*p \sim^M_{lval} x$).

Definition 7 (Aliasing equivalence relation $\sim^M_{lval}$)**.**
$a_1 \sim^M_{lval} a_2$ iff. for every environment E, and all locations l_1, l_2 such that $E \vdash a_1, M \Leftarrow l_1$ and $E \vdash a_2, M \Leftarrow l_2$, then $l_1 = l_2$.

Definition 8 introduces the aliasing invariant that the inlining approach must satisfy. Intuitively, it states that initial variables of the target program are aliased if and only if their shadow variables are also aliased. It also states that one pair of shadow variables is aliased if and only if all shadow variables are aliased.

Definition 8 (Aliasing invariant $\Omega(M)$)**.**
For all environments E, for all memories M, the aliasing invariant is defined as the predicate $\Omega(M)$ such that:

$$\begin{aligned}\Omega(M) \triangleq \forall x, y \in Var(P), \forall r \in [0, \mathscr{D}(y)]\,, then \\ x \sim^{M}_{lval} *^{r} y \\ \iff \forall k \in [0, \mathscr{D}(x)]\,, \Lambda(x,k) \sim^{M}_{lval} \Lambda(*^{r} y, k) \\ \iff \exists k \in [0, \mathscr{D}(x)]\,, \Lambda(x,k) \sim^{M}_{lval} \Lambda(*^{r} y, k).\end{aligned}$$

4.4 Instructions

This section formalizes the inlining approach as a program transformation, denoted by T. This program transformation relies on Definition 4.1 of the operators $\mathcal{L}_L$, $\mathcal{L}_R$ and $\mathcal{L}$ which express security labels of expressions in terms of shadow variables.

The program transformation T is parametrized by a security context variable denoted by `pc`. This variable represents the security context denoted by $\underline{pc}$ in the `PWhile` monitor semantics. The program transformation rules $(T[\![c, pc]\!] \mapsto c')$, introduced in Figure 4.3, take as input a target instruction `c` as well as a security context variable `pc`, then yield a new command `c'` that corresponds to the inlined version.

The program transformation does not modify instructions `skip` according to the `PWhile` monitor semantics. As for a sequence $\texttt{c}_1\texttt{;}\ \texttt{c}_2$ of two instructions, the program transformation parametrizes the inlining of both instructions with the same security context variable, since the `PWhile` semantics evaluates both instructions in the same security context.

For assignments $\texttt{a}_1\ \texttt{:=}\ \texttt{a}_2$, the program transformation updates the security label mapped to $\texttt{a}_1$. This security label, determined by $\Lambda\texttt{(a}_1\texttt{,0)}$, gets assigned the union of three security labels according to the `PWhile` monitor semantics: the security label $\mathcal{L}_L\texttt{(a}_1\texttt{)}$ of the left-value of $\texttt{a}_1$, the security label $\mathcal{L}_R\texttt{(a}_2\texttt{)}$ of the right-value of $\texttt{a}_2$, and the security context variable `pc`. Additionally, Assignment instructions may create new aliasing relations when $\texttt{a}_1$ and $\texttt{a}_2$ are of type pointer $(\mathscr{D}(a_1) = \mathscr{D}(a_2) \geq 1)$. In that case, the program transformation also reproduces these aliasing relations for the corresponding shadow variables. Thus, $\Lambda\texttt{(a}_1\texttt{,k)}$ gets assigned $\Lambda\texttt{(a}_2\texttt{,k)}$ for all strictly positive k such that these shadow variables $\Lambda\texttt{(a}_1\texttt{,k)}$ and $\Lambda\texttt{(a}_1\texttt{,k)}$ are well-defined $(k \leq \mathscr{D}(a_1))$. Moreover, the program transformation also reproduces the semantics of the *update* operator by propagating to the set of locations that may be modified by the assignment two security labels: the security label $\mathcal{L}_L\texttt{(a}_1\texttt{)}$ and the security context variable `pc`. Note that the program transformation relies on the inverse of environment E, denoted by E^{-1}, in order to find the set of variables that correspond to the

$$T[skip, pc] \mapsto skip \qquad T[c_1;\ c_2, pc] \mapsto T[c_1, pc];\ T[c_2, pc]$$

$$T[a_1 := a_2, pc] \mapsto$$

$$\begin{cases} \Lambda(a_1, 0) := \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup pc; \\ \Lambda(a_1, k) := \Lambda(a_2, k); \forall k \in [1, \mathscr{D}(a_1)] \\ \Lambda(E^{-1}(l), 0) := \Lambda(E^{-1}(l), 0) \sqcup \mathcal{L}_L(a_1) \sqcup pc; \forall l \in S_P(a_1 := a_2) \\ a_1 := a_2; \end{cases}$$

$$T[if\ (a)\ c_1\ else\ c_2, pc] \mapsto$$

$$\begin{cases} pc' = \mathcal{L}_R(a) \sqcup pc; \\ if\ (a)\ \{ \\ \quad T[c_1, pc'] \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_2) \\ \}\ else\ \{ \\ \quad T[c_2, pc']; \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_1) \\ \} \end{cases}$$

$$T[while\ (a)\ c, pc] \mapsto \begin{cases} while\ (a)\ \{ \\ \quad pc' = \mathcal{L}_R(a) \sqcup pc; \\ \quad T[c, pc']; \\ \} \\ pc' = \mathcal{L}_R(a) \sqcup pc; \\ \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c) \end{cases}$$

Figure 4.3: Program transformation rules.

set $S_P(a_1 := a_2)$ of locations that may be modified by the assignment. This inverse E^{-1} is well defined since all statically allocated locations have only one corresponding declared variable.

For conditional instructions, the program transformation creates a new security context variable `pc'`. Then the program transformation parametrizes both the then-branch and the else-branch's inlining by the new security context `pc'`. Additionally, the program transformation also inlines the *update* operator for both branches to account for implicit information flows occurring in non-executed branches.

For loop instructions, the program transformation creates a new security context variable `pc'`. This security context variable has to be updated each time the loop guard is evaluated since the loop guard's security level can be modified in the loop body. Therefore, the security context variable is updated twice; right after entering the loop body and right after exiting the loop instruction. The new security context variable `pc'` also parametrizes the inlining of the loop body. Moreover, the program transformation inlines the *update* operator at the loop exit.

4.5 Soundness

The inlining approach preserves the behaviour of initial target programs. Moreover, it soundly monitors information flows with respect to the `PWhile` monitor semantics, thus with respect to Non-interference as introduced in Definition 2.

The inlining approach ensures that transformed programs are behaviourally equivalent to the initial target programs. Let $E|_{Var(P)}$ be the restriction of environment E to the set $Var(P)$ of the target program's initial variables. Let $M|_{Loc(P)}$ (resp. $\Gamma|_{Loc(P)}$) also be the restriction of memory M (resp. of the security memory Γ) to the set $Loc(P)$ of the target program's initial locations. Then, Theorem 2 states that two terminating runs of a target program P and its inlined version $T(P)$ in input memories that are equal for the initial locations $Loc(P)$, must result in equal memories for those same locations. The proof of this theorem is by induction on the evaluation of instructions. It relies on the fact that the program transformation T introduces only assignments handling shadow variables. Thus, these additional assignments modify neither values nor security levels that are mapped to the set $Loc(P)$ of initial locations.

Theorem 2 (Initial semantics preservation)**.**
For all instructions c, for all environments E, for all memories M, for all security memories Γ*, for all security contexts* $\underline{pc}$ *and variables pc such that:*

$$E|_{Var(P)} \vdash c, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc} \Rightarrow M_1, \Gamma_1 \text{ and}$$
$$E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2$$

$$\text{Then, } M_2|_{Loc(P)} = M_1 \text{ and } \Gamma_2|_{Loc(P)} = \Gamma_1.$$

In order to prove the main soundness theorem, we first introduce Lemma 2 that states the program transformation maintains the aliasing invariant of Definition 8. The proof is by induction on the evaluation of instructions.

Lemma 2 (Maintaining the aliasing invariant)**.**
For all environments E, for all instructions c, for all memories M, for all security memories Γ, for all security context $\underline{pc}$, and variable pc such that:

$$E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$$

then, the program transformation maintains the aliasing invariant of Definition 8:

$$\Omega(M) \implies \Omega(M').$$

The main theorem affirms that the inlining approach reproduces exactly the semantics of the `PWhile` monitor. Theorem 3 proves that the program transformation maintains an invariant $\Upsilon(E, M, \Gamma)$ stating that the right-value evaluation of shadow variables in environment E and memory M yield exactly the security levels in the security memory Γ. The proof of this theorem, by induction on the evaluation of instructions, relies on the aliasing invariant in Lemma 2.

Theorem 3 (Sound monitoring of information flows)**.**
For all environments E, for all memories M, for all security memories Γ, the invariant $\Upsilon(E, M, \Gamma)$ is defined as the following predicate:

$$\Upsilon(E, M, \Gamma) \triangleq \forall x \in Var(P), \forall k \in [0, \mathscr{D}(x)], \text{ then}$$

$$E \vdash *^k x, M \Leftarrow l_{xk} \text{ and } \Gamma(l_{xk}) = s_{xk}$$
$$\implies E \vdash *^k \Lambda(x, k), M \Rightarrow s_{xk}.$$

Hence, for all instructions c, for all memories M, for all security memories Γ, for all security context $\underline{pc}$ and variable pc such that $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$ and $E \vdash pc, M \Rightarrow \underline{pc}$, the following result holds :

$$\Upsilon(E, M, \Gamma) \implies \Upsilon(E, M', \Gamma').$$

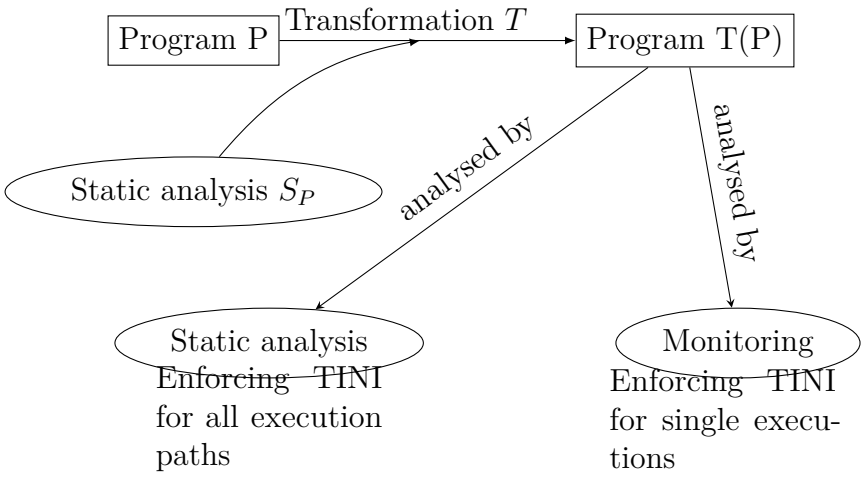


Figure 4.4: Non-interference verification using the program transformation T

TINI verification. Figure 4.4 illustrates how the inlining approach can be used to verify termination-insensitive non-interference (TINI). Once the `PWhile` monitor is inlined into a target program P, the inlined version $T(P)$ can be used to verify TINI either dynamically or statically.

Assuming the implementation of security labels and their join operator, running the self-monitoring program $T(P)$ enforces TINI dynamically – actually, this is a hybrid approach since the monitor relies on a prior static analysis S_P – for single execution paths. This dynamic approach has the advantage of being permissive since it ignores possible unsecure paths that are not executed. It also enables dynamic loading of security policies [LGBJS06], taking into account eventual updates.

Additionally, the transformation T also enables the verification of TINI by static analysis: for instance, off-the-shelf abstract interpretation tools can compute an over-approximation of $T(P)$ semantics for all execution paths, without implementing new abstract domains. While still being more permissive than traditional type systems, such an approach freezes the enforced security policy. Yet, it enhances our confidence in the analysed program since it ensures that all execution paths are secure with respect to TINI. It also completely lifts the burden of runtime overhead.

4.6 Related Work

Chudnov and Naumann [CN10] design a sound inlining approach for a monitor based on Russo and Sabelfeld's monitor. As they aim at monitoring information flows for JavaScript, they argue that VM monitors are impractical because of just-in-time compilation. Their language supports output instructions but no references. We also believe that inlining is necessary when the language may be compiled rather than interpreted.

Magazinius et al. [MRS12, MRS10] investigate sound inlining of security monitors for an imperative language supporting dynamic code evaluation but no references. Their monitor is purely dynamic since it uses a no-sensitive upgrade policy as in Austin and Flanagan [AF09]. Our program transformation approach can also be applied for such a policy in order to soundly monitor information flows for richer languages, including pointers.

4.7 Summary

This chapter proposes an inlining approach for the `PWhile` monitor, presented in Chapter 3, through a program transformation. This inlining approach is sound with respect to the `PWhile` monitor semantics. Hence, it is also sound with respect to TINI. Our program transformation enables permissive yet sound enforcement of TINI by relying on both dynamic and static analyses.

As we are aiming at supporting a large subset of the C language, we implemented our inlining approach as a plug-in of the Frama-C framework [CKK+12, KKP+15]. In addition to pointers and aliasing, our plug-in validates the scalability of our inlining approach for more complex constructions of the C language, such as arrays, function calls, as well as recursive structures. However, since our inlining approach relies on the knowledge of the underlying type of a variable, our implementation does not yet support type casts. For instance, a target program can cast and use a variable `x` of type $void*$ either as a pointer to an integer (type $ptr(\kappa)$) or as a pointer to pointer to an integer (type $ptr(ptr(\kappa))$). In both cases, the number of shadow variables the program transformation must map to variable `x` is actually different. Additionally, the program transformation must also handle the aliasing invariant proved in Lemma 2 differently in both cases. Future work will tackle this challenge by relying on static analysis to try to uncover the underlying type of a variable in order to guide the inlining approach.

The inlining approach presented in Chapter 4 relies heavily on the aliasing invariant proved in Lemma 2. While our work relies on such an invariant

to soundly inline an information flow monitor, we believe it also sets the ground for a generic approach of inlining security monitors. In future work, we will investigate sound inlining of security monitors aimed at a wide set of enforceable security policies [Sch00, LBW02].

Part II

Quantitative Information Flow

Chapter 5

Quantifying Information Leaks

This chapter introduces a new quantitative security policy aimed at relaxing termination-insensitive non-interference (TINI), while still providing the same security guarantees.

We introduce a generic deterministic operational semantics, enabling attackers to observe intermediate steps of computation, divergence as well as termination of program runs. We then quantify information leaks wrt. the probability that attackers guess the secret in one try, according to Smith's [Smi09, Smi11] definition of min-entropy. Finally, we propose relative secrecy (RS) as a quantitative security property. This security property ensures that the probability of polynomial time attackers guessing the secret in one try is negligible. We also extend RS to account for a scenario where attackers are able to observe multiple runs a program on the same secret inputs.

5.1 Introduction

Non-interference semantically characterizes the absence of information leaks. It is a baseline policy aimed at detecting and preventing information leaks. Yet, enforcing non-interference can be too much conservative.

On the one hand, information flow analyses actually enforce approximations of non-interference. Consider for instance a sound security type system such as the one proposed by Volpano et al. [VIS96] or Hunt and Sands [HS06]. If a program is typed, then it is non-interferent. However, when a program is not typed, it could be either because:

1. the program indeed leaks information, thus is interferent, or

2. the security analysis is imprecise, thus fails to type a non-interferent program.

The latter problem is characteristic of the usual trade-off between soundness and completeness. One can only strive for more precise analyses in order to type a larger set of non-interferent programs, while still guaranteeing soundness.

On the other hand, non-interference as a security policy is also conservative. Non-interference prevents $High$ inputs from leaking to Low outputs. Yet, many programs cannot comply with such a strict policy. A password checker for instance, leaks information about the secret password by accepting or rejecting a user-supplied input. A banking application also leaks information about a client's invoice, by accepting or rejecting a debit instruction, depending on available funds. However, these information leaks are deemed negligible, under the assumption that brute-force attacks are thwarted, thus impractical. While declassification policies [ZM01, SM04, MSZ06, SS09] can handle these deliberate release of sensitive information, the programmer or auditor responsible for such a release must still ensure that these information leaks are negligible. Therefore, we need tools and techniques that go beyond the detection of information leaks, by quantifying information leaks in order to decide whether they are negligible. These tools and techniques may enable the release sensitive information, while still providing strong security guarantees.

Notice that while we argue that non-interference is a baseline security policy that is often not the most suitable, some applications may require a conservative policy such as TINI or termination-sensitive non-interference (TSNI). Indeed, non-interference has been successfully applied to many cases requiring a strict control of information leaks. Such cases include preventing information leaks due to timing channels in the case of cryptographic software [BRW06, BBC+14], as well as preventing ghost code – code added to a program to aid program verification – from interfering with regular code [FGP14].

5.2 Relaxing Non-interference

Non-interference [GM82] as a policy is conservative since it prevents all information leaks. However, most information flow analyses in the literature [MRS12, MC11, BPR13, dACD+14] enforce a slightly more permissive flavour of non-interference, introduced previously in Definition 2 as TINI. Recall that TINI guarantees that two terminating execution, differing only

on *High* inputs, deliver the same *Low* outputs. Hence, this property ensures that a program does not leak sensitive information to attackers, unless the information leak is due to non-termination. For instance, let us compare both programs in Listings 5.1 and 5.2, where only variable `secret` is *High*.

```
1 if (input == secret){
2   skip;
3 }
4 else {
5   while (true){
6     skip;
7   }
8 }
9 output 1;
```

Listing 5.1: A non-interferent program wrt. TINI

```
1 if (input == secret){
2   output 1;
3 }
4 else {
5   output 0;
6 }
```

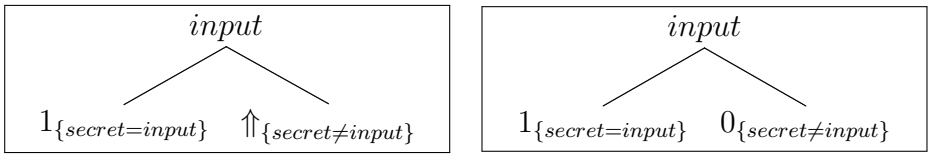
Listing 5.2: An interferent program wrt. TINI

The first program in Listing 5.1 leaks sensitive information about variable `secret`. Indeed, if attackers observe the output 1, they can deduce that the supplied input is equal to the sensitive variable `secret`. Otherwise, if attackers wait enough time without observing any outputs, they may conclude that the program diverges. Consequently, attackers can deduce that the supplied input is different from the sensitive variable `secret`.

The second program in Listing 5.2 also leaks sensitive information about variable `secret`. Indeed, attackers may observe the output 1 (resp. the output 0), thus deduce that the supplied input is equal to (resp. different from) the sensitive variable `secret`.

Figure 5.1 summarizes the observations attackers make for both programs in Listings 5.1 and 5.2, as well as the knowledge they deduce for each observation they make. Both programs are equivalent wrt. the observations

attackers make and the knowledge they deduce for each observation. However, the program in Listing 5.1 is considered secure since it satisfies TINI, whereas the program in Listing 5.2 is considered insecure since it does not satisfy TINI.



(a) Listing 5.1 (b) Listing 5.2

Figure 5.1: Attackers' observations for both programs in Listings 5.1 and 5.2

Historically, the literature in information flow control argues that non-termination channels leak at most 1 bit of security. However, when attackers are able to observe intermediate steps of computation through output instructions for instance, the leakage may be much greater [AHSS08]. Consider for instance the following program in Listing 5.3 where only variable `secret` is *High*. This program outputs the value of the counter variable `i`, until it is equal to the sensitive variable `secret`, then diverges. Although this example program leaks the value of variable `secret`, it is non-interferent wrt. TINI, hence considered secure.

```
1 while (i < max){
2   while (i == secret){
3     skip;
4   }
5   output i;
6   i := i + 1;
7 }
```

Listing 5.3: A non-interferent program wrt. TINI

In fact, Askarov et al. [AHSS08] prove that for the program in Listing 5.3 – as well as all programs satisfying TINI –, the probability of attackers guessing the secret, after observing a polynomial amount of outputs in the size of secrets, is negligible assuming the secret is uniformly distributed. Consequently, TINI is a valid characterization of security only for big secrets and against polynomial time attackers. Therefore, we propose to characterize

security of programs against this exact assumption: *a program is secure if the probability of polynomial time attackers guessing the secret is negligible.*

This latter characterization of security has the advantage of being more permissive than TINI, while providing the same security guarantees wrt. the probability of attackers guessing the secret. For instance, all three programs in Listings 5.1 to 5.3 are secure wrt. this property, assuming that the size of variable `secret` is large enough and sufficiently random. Additionally, this characterization also enables us to treat equally information leaks due to explicit information flows, implicit ones and divergence, according to the knowledge attackers gain. For instance, both programs in Listings 5.1 and 5.2 can be deemed as equivalent wrt. the advantage attackers gain, when reasoning on the probability of attackers guessing the secret.

Volpano and Smith [VS00] already propose a type system that weakens traditional type systems enforcing TINI. By allowing programs to test whether a secret is equal to a particular value, their type system accepts as secure the program in Listing 5.2 for instance. They also prove that well-typed deterministic programs can only leak the secret with a negligible probability, assuming the program runs in polynomial time in the size of the secret. Volpano and Smith coin this property as RS. Therefore, they stress out that their type system is well-suited when the secret is sufficiently large and randomly chosen.

Interestingly, recent work in quantitative information flow also proposes to quantify information leaks wrt. the probability of attackers guessing the secret. Indeed, Smith [Smi09, Smi11] coins this quantitative measure as min-entropy. Therefore, the following sections introduce the notion of min-entropy in order to provide a formal characterization of a *relative secrecy* security property.

5.3 Information Leakage

In this section, we introduce a quantitative information flow measure proposed by Smith [Smi09, Smi11]. For simplicity, we consider the simple information flow lattice introduced in Figure 3.5, consisting of two security levels: Low and $High$.

5.3.1 Operational Semantics

We consider a deterministic small-step operational semantics [Plo81, Plo04] with labelled transitions. Configurations have the form $\langle c, \varrho \rangle$, where $\mathtt{c}$

denotes an instruction and ϱ denotes a memory mapping variables to values. We also denote by $\mathtt{t}$ an output observable by attackers.

Evaluation rules have the form $\langle c, \varrho\rangle \to \langle c', \varrho'\rangle$, which corresponds to a small-step transition of a configuration $\langle c, \varrho\rangle$ to an intermediate configuration $\langle c', \varrho'\rangle$. When a small-step transition produces an observable output $\mathtt{t}$, we write $\langle c, \varrho\rangle \to_t \langle c', \varrho'\rangle$ to mean that the configuration $\langle c, \varrho\rangle$ takes one computation step to a configuration $\langle c', \varrho'\rangle$ and yields an output $\mathtt{t}$. Note that we only model outputs to *Low* channels observable by attackers. Additionally, we let ϵ denote the empty trace.

We also write $\langle c, \varrho\rangle \to_t^* \langle c_n, \varrho_n\rangle$ in the usual way for the transitive closure relation $\to^*$ of the small step transition relation $\to$, to denote the existence of a sequence of transitions $\langle c, \varrho\rangle \to_{t_1} \langle c_1, \varrho_1\rangle \to_{t_2} \ldots \to_{t_n} \langle c_n, \varrho_n\rangle$, where t is the sequence $t_1 \cdot t_2 \ldots \cdot t_n$ of observable outputs. Similarly, we write $\langle c, \varrho\rangle \to_t^*$ to denote the existence of a configuration $\langle c_n, \varrho_n\rangle$ such that $\langle c, \varrho\rangle \to_t^* \langle c_n, \varrho_n\rangle$.

Modelling termination. We explicitly model termination by considering an instruction occurring at the end of a program and denoted by `stop`. This instruction signals the end of a program's evaluation. Therefore, we write $\langle \mathit{stop}, \varrho\rangle \to_{\Downarrow} \varrho$ to mean that the intermediate configuration $\langle \mathit{stop}, \varrho\rangle$ takes a computation step to a final configuration ϱ and produces an observable output $\Downarrow$, denoting termination.

Modelling divergence. Following the work of Askarov et al. [AHSS08], we write $\langle c, \varrho\rangle\Uparrow$ to mean that the configuration $\langle c, \varrho\rangle$ does not produce any observable outputs. Note that we do not distinguish stuck configurations and divergence. Additionally, this model only considers *silent divergence*: when a program execution enters a loop that does not terminate, and eventually stops outputting new observations. Therefore, we also write $\langle c, \varrho\rangle \to_{t\cdot\Uparrow}^*$ to denote the existence of an intermediate configuration $\langle c', \varrho'\rangle$ such that $\langle c, \varrho\rangle \to_t^* \langle c', \varrho'\rangle$ and $\langle c', \varrho'\rangle\Uparrow$.

5.3.2 Attackers' Knowledge

As illustrated by Figure 5.1, attackers can make a set of observations when a program runs. Each one of these observations allows attackers to deduce information about sensitive variables. Therefore, we set up to define the observations attackers can make as well as the knowledge they gain, in order to reason about the probability of attackers guessing the secret.

We denote by $\Sigma_{L_0} \times \Sigma_{H_0}$ the partition of initial environments into *Low* (Σ_{L_0}) and *High* (Σ_{H_0}) variables. We also assume that attackers:

1. control the initial *Low* input memory $\varrho_{L_0} \in \Sigma_{L_0}$, and
2. can observe output traces t, as well as
3. known the programs' source code and have perfect deduction capabilities about their semantics, and finally
4. cannot observe the programs' timing behaviour.

Definition 9 introduces the set $Obs(\varrho_{L_0})$ of observations attackers can make for a choice of a *Low* input memory $\varrho_{L_0} \in \Sigma_{L_0}$. Moreover, since attackers are assumed to have perfect deduction capabilities about the program's semantics, whenever they observe a trace $t \in Obs(\varrho_{L_0})$, they can deduce a set of *High* input memories $\varrho_{H_0} \in \Sigma_{H_0}$ that can lead to this observation. Definition 10 introduces this set of *High* input memories as the knowledge attackers gain when observing a trace t.

Notice that the more attackers observe, the smallest their knowledge about the secret becomes, since attackers may be able to rule out *High* input memories that cannot lead to the same additional observations. Attackers' knowledge, in a sense, captures the uncertainty of attackers about the secret. The more they observe, the smaller their uncertainty becomes.

Definition 9 (Attackers' observations $Obs(\varrho_{L_0})$)**.**
For a program c, and a choice of a Low input memory $\varrho_{L_0} \in \Sigma_{L_0}$, the set of observations attackers can make is given by:

$$Obs(\varrho_{L_0}) \triangleq \{t \mid \exists \varrho_{H_0} \in \Sigma_{H_0}, \varrho_0 \triangleq (\varrho_{L_0}, \varrho_{H_0}) \text{ and } \langle c, \varrho_0 \rangle \rightarrow_t^* \}.$$

Definition 10 (Attackers' knowledge $k(\varrho_{L_0}, t)$)**.**
For a program c, and a choice of a Low input memory $\varrho_{L_0} \in \Sigma_{L_0}$, the knowledge $k(\varrho_{L_0}, t)$ attackers gain when observing a trace $t \in Obs(\varrho_{L_0})$ is defined by:

$$k(\varrho_{L_0}, t) \triangleq \{\varrho_{H_0} \in \Sigma_{H_0} \mid \varrho_0 \triangleq (\varrho_{L_0}, \varrho_{H_0}) \text{ and } \langle c, \varrho_0 \rangle \rightarrow_t^* \}.$$

Bounded attackers. So far, Definitions 9 and 10 introduce the observations and the knowledge of unbounded attackers: no matter how long a trace t can be, attackers are able to wait enough time to observe this trace t. In the following, we refine our model by bounding the observational power

of attackers. As a consequence, we also restrict the knowledge attackers gain when observing a program run by limiting the observations they can make.

Our refined attacker model includes a notion of time, that we represent by a bound on the length of output sequences attackers can observe. Let us denote by $|t|$ the length of the sequence t of outputs. Let us also denote by b, the bound on attackers' observations. Then, Definition 11 introduces the bounded observations attackers can make. The bounded version of attackers' observations $Obs_b(\varrho_{L_0})$ restricts the running time of attackers observing a program run. Notice that this hypothesis bounds the running time of attackers, rather than the running time of the program itself.

Definition 11 (Bounded attackers' observations $Obs_b(\varrho_{L_0})$)**.**
For a program c, and a choice of a Low input memory $\varrho_{L_0} \in \Sigma_{L_0}$, the set of bounded observations attackers can make is given by:

$$Obs_b(\varrho_{L_0}) \triangleq \{t \mid \exists \varrho_{H_0} \in \Sigma_{H_0}, \varrho_0 \triangleq (\varrho_{L_0}, \varrho_{H_0}), \langle c, \varrho_0\rangle \rightarrow_t^* \text{ and } |t| \leq b\}.$$

Definition 12 also introduces the bounded knowledge $k_b(\varrho_{L_0}, t)$ attackers gain after observing a bounded trace $t \in Obs_b(\varrho_{L_0})$ – such that $|t| \leq b$.

Definition 12 (Bounded attackers' knowledge $k_b(\varrho_{L_0}, t)$)**.**
For a program c, and a choice of a Low input memory $\varrho_{L_0} \in \Sigma_{L_0}$, the bounded knowledge $k_b(\varrho_{L_0}, t)$ attackers gain when observing a bounded trace $t \in Obs_b(\varrho_{L_0})$ is defined by:

$$k_b(\varrho_{L_0}, t) \triangleq \{\varrho_{H_0} \in \Sigma_{H_0} \mid \varrho_0 \triangleq (\varrho_{L_0}, \varrho_{H_0}) \text{ and } \langle c, \varrho_0\rangle \rightarrow_t^*\}.$$

Notice that the bounded knowledge $k_b(\varrho_{L_0}, t)$ is the restriction of the knowledge $k(\varrho_{L_0}, t)$ – previously introduced in Definition 10 – to bounded traces $t \in Obs_b(\varrho_{L_0})$.

Attackers' observation trees. Notice that when a set $Obs_b(\varrho_{L_0})$ of attackers' bounded observations contains a sequence $t_1 \cdot t_2$ of outputs, then the set $Obs_b(\varrho_{L_0})$ is also guaranteed to contain the output t_1. Indeed, this is guaranteed by definition of attackers observations and the transitive closure $\rightarrow_t^*$. Therefore, if the set $Obs_b(\varrho_{L_0})$ contains a trace t, then it also contains all the *prefixes* of the trace t. Said otherwise, the set $Obs_b(\varrho_{L_0})$ is a prefix-closed set.

The *prefix* relation is related to the lexicographic partial order that is usually used in a dictionary. For instance, the prefixes of the word *supertramp* are all contained in the following set:

$$\{\epsilon, s, su, sup, supe, super, supert,$$
$$supertr, supertra, supertram, supertramp\}$$

As noted by Perrin and Pin [PP04, Chapter 1], "one can associate naturally a tree with each non-empty prefix-closed set S, using the elements of S as nodes and the empty word as a root". Note that we did previously introduce such attackers' observation trees rather informally in Figure 5.1, in order to illustrate the observations attackers can make for the programs in Listings 5.1 and 5.2, as well as the knowledge they gain. However, we have used the *Low* input variable `input` as a root of the observation tree, in order to stress out that the observations attackers make are parametrized by the *Low* input memory.

Henceforth, we denote by $\mathcal{T}_b^{\varrho_{L_0}}$ the attackers' observation tree that is defined by the set $Obs_b(\varrho_{L_0})$ of attackers observations. Additionally, we also denote by $Leaves(\mathcal{T}_b^{\varrho_{L_0}})$ the set of leaves of the observation tree $\mathcal{T}_b^{\varrho_{L_0}}$. This will shortly prove useful since when reasoning on probabilities, it is often practical to consider a set of disjoint events. One way of achieving a set of disjoint events is by assuming that attackers wait for the longest observable traces $t \in \mathcal{T}_b^{\varrho_{L_0}}$, which corresponds exactly to the set $Leaves(\mathcal{T}_b^{\varrho_{L_0}})$ of disjoint observations. This assumption is safe, since the more attackers observe, the smallest the knowledge they gain. Therefore, attackers have every incentive to maximize the information they can deduce about the secret by waiting for the longest trace they can observe, which corresponds to observing only the set $Leaves(\mathcal{T}_b^{\varrho_{L_0}})$ of maximal traces.

5.3.3 Min-entropy Leakage

Traditionally, the quantitative information flow literature defines the leakage of a program as the difference between an initial uncertainty about the secret prior to any program run, and the remaining uncertainty after attackers observe the outputs of a program run [Den82]:

$$\text{leakage} = \text{initial uncertainty} - \text{remaining uncertainty}$$

Denning [Den82] proposes the first quantitative measure of a program's leakage in terms of Shannon entropy. Denning characterizes both the initial uncertainty and the remaining one in terms of Shannon entropy and conditional entropy, then defines the information leakage as the reduction of uncertainty about the secret. In a different approach, Smith [Smi09, Smi11] recently proposes to quantify information leaks wrt. the probability that attackers could guess the secret in one try. Smith then defines both the initial uncertainty and the remaining uncertainty in terms of the probability of attackers guessing the secret.

Initial uncertainty. Let us assume that attackers know the probability distribution of the $High$ input memories $\varrho_{H_0} \in \Sigma_{H_0}$. Let us also denote by H a random variable representing the $High$ input memories. Then, prior to any program run, rational attackers will assume that the $High$ input memory is the one having the maximum probability wrt. the $High$ inputs distribution. Therefore, the probability that attackers guess the secret in one try is the *maximum probability in the distribution of the high input memories*. Following Smith's work [Smi09, Smi11], Definition 13 introduces this quantity as the *vulnerability* $V(H)$ of the $High$ input memories:

Definition 13 (Vulnerability $V(H)$)**.**
The vulnerability of the high input memories is given by

$$V(H) \triangleq \max_{\varrho_{H_0} \in \Sigma_{H_0}} P(H = \varrho_{H_0}).$$

The vulnerability $V(H)$ characterizes the initial uncertainty about the secret in terms of probabilities. Usually, quantitative information flow – and more generally information theory – measure information in bits. Therefore, we can map the vulnerability $V(H)$ to the logarithm of its inverse in order to obtain a measure of the initial uncertainty in bits. By doing so, we obtain a classical entropy measure termed as Rényi's min-entropy [Rén61]. Therefore, Definition 14 introduces min-entropy $\mathcal{H}_\infty(H)$ as a measure for the initial uncertainty about the secret.

Definition 14 (Min-entropy $\mathcal{H}_\infty(H)$)**.**
The min-entropy of the High input memories is given by

$$\mathcal{H}_\infty(H) \triangleq -\log_2 V(H).$$

For instance, let us denote by H_U a random variable representing $High$ input memories that are uniformly distributed. Let us also denote by N the size in bits of the $High$ input memories. Then, all the values of the secret have equal probabilities given by $P(H_U = \varrho_{H_0}) = \frac{1}{2^N}$. Therefore, the vulnerability and min-entropy of a uniformly distributed secret are given by:

$$V(H_U) = \frac{1}{2^N}$$
$$\mathcal{H}_\infty(H_U) = N \ bits$$

This means that the initial uncertainty of attackers about a uniformly distributed secret is N *bits*, exactly the size of the secret. Notice that a uniform distribution of the secret corresponds to the case where the initial uncertainty of attackers is the highest. Indeed, any other distribution than a uniform one yields a higher vulnerability, which translates to a lower min-entropy.

Remaining uncertainty. Let us assume that attackers provide a *Low* input memory $\varrho_{L_0} \in \Sigma_{L_0}$, then observe a trace $t \in Obs_b(\varrho_{L_0})$ after a program run. Then, attackers are able to deduce the set $k_b(\varrho_{L_0}, t)$ of *High* input memories that can lead to an observation of the trace t, since we assume attackers have perfect deduction capabilities. Additionally, rational attackers will assume that the *High* input memory is the most probable input memory $\varrho_{H_0} \in \Sigma_{H_0}$. Let O denote a random variable representing the observations attackers can make. Therefore, we can define the a posteriori vulnerability of the *High* input memories, when attackers provide a *Low* input memory ϱ_{L_0} and observe a trace t, as the maximum conditional probability of the *High* input memories, given an observation $t \in Obs_b(\varrho_{L_0})$:

$$V^{\varrho_{L_0}}(H \mid O = t) \triangleq \max_{\varrho_{H_0} \in k_b(\varrho_{L_0}, t)} P(H = \varrho_{H_0} \mid O = t) \tag{5.1}$$

Given Equation (5.1), we further introduce in Definition 15 the a posteriori vulnerability of the *High* input memory, when attackers provide a *Low* input memory, as the expected probability of attackers guessing the secret.

Definition 15 (Conditional vulnerability for a *Low* input ϱ_{L_0})**.**
For a choice of a Low input memory $\varrho_{L_0} \in \Sigma_{L_0}$, the conditional vulnerability of the High input memories is given by:

$$\begin{aligned} V^{\varrho_{L_0}}(H \mid O) &\triangleq \sum_{t \in Leaves(\mathcal{T}_b^{\varrho_{L_0}})} P(O = t) V^{\varrho_{L_0}}(H \mid O = t) \\ &= \sum_{t \in Leaves(\mathcal{T}_b^{\varrho_{L_0}})} P(O = t) \max_{\varrho_{H_0} \in k_b(\varrho_{L_0}, t)} P(H = \varrho_{H_0} \mid O = t). \end{aligned}$$

Definition 15 introduces the conditional vulnerability $V^{\varrho_{L_0}}(H \mid O)$ when attackers provide a *Low* input memory and observe the outputs of a program run. Rational attackers will maximize their probability of guessing the secret in one try, by providing a *Low* input memory that maximizes the conditional vulnerability $V^{\varrho_{L_0}}(H \mid O)$. Therefore, we introduce the conditional vulnerability in Definition 16 as the maximum conditional vulnerability $V^{\varrho_{L_0}}(H \mid O)$ over all *Low* input memories $\varrho_{L_0} \in \Sigma_{L_0}$. Similarly to Definition 14 of min-entropy, Definition 16 also introduces conditional min-entropy as the logarithm of the conditional vulnerability's inverse, in order to measure the remaining uncertainty in bits.

Definition 16 (Conditional vulnerability $V(H \mid O)$ and conditional min-entropy $\mathcal{H}_\infty(H \mid O)$)**.**
The conditional vulnerability of the High input memories is given by:

$$V(H \mid O) \triangleq \max_{\varrho_{L_0} \in \Sigma_{L_0}} V^{\varrho_{L_0}}(H \mid O)$$

The conditional min-entropy is defined as:

$$\mathcal{H}_\infty(H \mid O) \triangleq -\log_2 V(H \mid O).$$

Measuring information leakage. So far, we introduced min-entropy in Definition 14 as an information measure for the initial uncertainty of attackers about the secret. Additionally, we also introduced conditional min-entropy in Definition 16 as a measure for the remaining uncertainty of attackers about the secret. Therefore, we can finally introduce a measure for information leakage as the difference between the initial uncertainty and the remaining uncertainty about the secret.

Since this measure depends on the probability distribution of the initial $High$ input memories, we let $\vec{\pi}$ denote that distribution. Then, Definition 17 introduce the information leakage $\mathcal{L}_{\vec{\pi}}$ of a program as the difference between the initial uncertainty and the remaining uncertainty about the secret.

Definition 17 (Min-entropy leakage $\mathcal{L}_{\vec{\pi}}$)**.**
The leakage is defined as follows:

$$\begin{aligned}\mathcal{L}_{\vec{\pi}} &\triangleq \mathcal{H}_\infty(H) - \mathcal{H}_\infty(H \mid O)\\ &= \log_2 \frac{V(H \mid O)}{V(H)}.\end{aligned}$$

Remaining uncertainty for deterministic programs. Equation (5.1) and Definition 15 of the conditional vulnerability $V^{\varrho_{L_0}}(H \mid O)$ for a Low input ϱ_{L_0} simplify further by for deterministic programs. Indeed, by the conditional probability law, $V^{\varrho_{L_0}}(H \mid O = t)$ as defined in Equation (5.1) is equal to:

$$\begin{aligned}V^{\varrho_{L_0}}(H \mid O = t) &= \max_{\varrho_{H_0} \in k_b(\varrho_{L_0}, t)} \frac{P(H = \varrho_{H_0} \wedge O = t)}{P(O = t)}\\ &= \max_{\varrho_{H_0} \in k_b(\varrho_{L_0}, t)} \frac{P(O = t \mid H = \varrho_{H_0}) P(H = \varrho_{H_0})}{P(O = t)}\end{aligned} \tag{5.2}$$

Moreover, for deterministic programs, the observed trace t is completely determined by the Low and $High$ input memories. Therefore, we have:

$$P(O = t \mid H = \varrho_{H_0}) = \begin{cases} 1 & \text{if } \varrho_{H_0} \in k_b(\varrho_{L_0}, t) \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

Injecting both Equations (5.2) and (5.3) into Definitions 15 and 16 yields a simpler characterization of the conditional vulnerability $V^{\varrho_{L_0}}(H \mid O)$ for

a *Low* input memory, as well as the conditional vulnerability $V(H \mid O)$, introduced as Lemma 3.

Lemma 3 (Conditional vulnerability for deterministic programs)**.**
For deterministic programs, the conditional vulnerabilities are given by:

$$V^{\varrho_{L_0}}(H \mid O) = \sum_{t \in Leaves(\mathcal{T}_b^{\varrho_{L_0}})} \max_{\varrho_{H_0} \in k_b(\varrho_{L_0}, t)} P(H = \varrho_{H_0})$$
$$V(H \mid O) = \max_{\varrho_{L_0} \in \Sigma_{L_0}} V^{\varrho_{L_0}}(H \mid O).$$

Additionally, assuming that the $High$ input memories follow a uniform distribution denoted by $\vec{\pi}_U$, Lemma 3 simplifies even further. Let us denote by N the size in bits of the set Σ_{H_0} of $High$ input memories. Then, all the $High$ input memories have equal probabilities given by $P(H_U = \varrho_{H_0}) = \frac{1}{2^N}$. Therefore, the conditional vulnerabilities as well as the information leakage have a much simpler formulation, introduced in Corollary 2.

Corollary 2 (Conditional vulnerabilities and leakage for deterministic programs, under uniformly distributed secrets)**.**
For deterministic programs, assuming the High input memories follow a uniform distribution $\vec{\pi}_U$, the conditional vulnerabilities and the leakage are given by:

$$V(H_U) = \frac{1}{2^N}$$
$$V^{\varrho_{L_0}}(H_U \mid O) = \frac{\left|Leaves(\mathcal{T}_b^{\varrho_{L_0}})\right|}{2^N}$$
$$V(H_U \mid O) = \max_{\varrho_{L_0} \in \Sigma_{L_0}} V^{\varrho_{L_0}}(H_U \mid O)$$
$$\mathcal{L}_{\vec{\pi}_U} = \max_{\varrho_{L_0} \in \Sigma_{L_0}} \log_2 \left|Leaves(\mathcal{T}_b^{\varrho_{L_0}})\right|.$$

5.4 Min-capacity leakage

The information leakage $\mathcal{L}_{\vec{\pi}}$ of a program, as introduced in Definition 17, depends on the assumed distribution of the $High$ input memories. In order to abstract away from the initial probabilities of the secret, we introduce the notion of min-capacity [Smi09, Smi11]. We also establish upper bounds on min-capacity, following Braun et al.'s [BCP09] and Smith's [Smi09, Smi11] work. This upper-bound will allow us to focus on a single quantity that is prone to static analysis.

Since the probability distribution of the $High$ input memories is unknown in the general case, we usually want to abstract away from the initial probabilities of the secret in order to measure the leakage of a program. A suitable measure can be achieved by considering the maximum leakage $\mathcal{L}_{\vec{\pi}}$, over all the $High$ input distributions $\vec{\pi}$. Definition 18 introduces this measure as min-capacity since it is defined in terms of min-entropy, as opposed to capacity that is usually defined in terms of Shannon entropy [Sha48].

Definition 18 (Min-capacity $\mathcal{ML}$)**.**
Min-capacity $\mathcal{ML}$ is defined as the maximum leakage $\mathcal{L}_{\vec{\pi}}$ over all the High input distributions $\vec{\pi}$:

$$\mathcal{ML} \triangleq \max_{\vec{\pi}} \mathcal{L}_{\vec{\pi}}.$$

When considering only deterministic programs, min-capacity $\mathcal{ML}$ has a simple characterization. Indeed, as Braun et al. [BCP09] prove, the supremum of the leakage $\mathcal{L}_{\vec{\pi}}$ is reached when $\vec{\pi}$ is a uniform distribution:

$$\begin{aligned}
2^{\mathcal{L}_{\vec{\pi}}} &= \frac{V(H \mid O)}{V(H)} \\
&= \text{(by Lemma 3 and Definition 13)} \\
&\quad \frac{\max\limits_{\varrho_{L_0} \in \Sigma_{L_0}} \left(\sum\limits_{t \in Leaves(\mathcal{T}_b^{\varrho_{L_0}})} \max\limits_{\varrho_{H_0} \in k_b(\varrho_{L_0}, t)} P(H = \varrho_{H_0}) \right)}{\max\limits_{\varrho_{H_0} \in \Sigma_{H_0}} P(H = \varrho_{H_0})} \\
&= \max_{\varrho_{L_0} \in \Sigma_{L_0}} \left(\sum_{t \in Leaves(\mathcal{T}_b^{\varrho_{L_0}})} \frac{\max\limits_{\varrho_{H_0} \in k_b(\varrho_{L_0}, t)} P(H = \varrho_{H_0})}{\max\limits_{\varrho_{H_0} \in \Sigma_{H_0}} P(H = \varrho_{H_0})} \right) \\
&\leq \max_{\varrho_{L_0} \in \Sigma_{L_0}} \sum_{t \in Leaves(\mathcal{T}_b^{\varrho_{L_0}})} 1 \\
&= \max_{\varrho_{L_0} \in \Sigma_{L_0}} \left| Leaves(\mathcal{T}_b^{\varrho_{L_0}}) \right| \\
&= \text{(By Corollary 2)} \\
&\quad 2^{\mathcal{L}_{\vec{\pi}_U}}
\end{aligned}$$

The latter proof yields Theorem 4, that characterizes the maximum leakage $\mathcal{ML}$ for deterministic programs as the leakage when the $High$ input memories are uniformly distributed.

Theorem 4 (Characterization of min-capacity $\mathcal{ML}$)**.**
For deterministic programs, the maximum leakage $\mathcal{ML}$ is reached for uniformly distributed High input memories. Moreover, $\mathcal{ML}$ is given by:

$$\mathcal{ML} = \max_{\varrho_{L_0} \in \Sigma_{L_0}} \log_2 \left| Leaves(\mathcal{T}_b^{\varrho_{L_0}}) \right|.$$

Intuitively, a deterministic program always leaks the same bits of the secret, when the *Low* inputs are fixed. If the leaked bits are initially known to attackers, then the measured leakage is actually small. On the contrary, if the leaked bits are not known to attackers initially, then the measured leakage will be greater. Therefore, in order to maximize the leakage, one has to measure it when attackers know almost nothing beforehand, meaning that the initial uncertainty is the highest. This corresponds precisely to the case where the secret is uniformly distributed.

Additionally, for a fixed *Low* input, the more output sequences a program produces, the more partitions $k_b(\varrho_{L_0}, t)$ of the $High$ input memory Σ_{H_0} there are. Plus, the more partitions $k_b(\varrho_{L_0}, t)$ there are, the smaller these partitions get, and the more vulnerable the secret gets.

Let us consider again both programs introduced earlier in Listings 5.1 and 5.2. Assuming that attackers can observe at least one output ($b \geq 1$), Figure 5.1 illustrates the attackers' observation trees of both programs. Since these observation trees have 2 leaves, we can deduce that the maximum leakage for both programs is given by :

$$\mathcal{ML} = \log_2(2) = 1 \; bit$$

Therefore, both programs are equivalent wrt. attackers guessing the secret in one try, since they both leak only 1 bit of sensitive information. Yet, should we consider both programs as secure? The maximum leakage $\mathcal{ML}$ is rather small, but this fact alone is not sufficient to decide if these programs are secure. Indeed, in the case where variable `secret` is a boolean variable, both programs do actually leak all the sensitive information although they only leak 1 bit of the secret. Therefore, in order to deem both programs as secure, we need to assume that the size of the variable `secret` is large enough, and randomly chosen so that the initial uncertainty is large. Only then, we can consider both programs to be secure wrt. attackers guessing the secret in one try.

Min-capacity for deterministic non-interferent programs. For deterministic programs, each time an attackers' observation tree $\mathcal{T}_b^{\varrho_{L_0}}$ branches, attackers are able to learn additional information about the secret and refine

their knowledge. Indeed, a new branch in the observation tree corresponds to the program outputting at least two sequences $t \cdot t_1$ and $t \cdot t_2$, where t_1 and t_2 are different. Therefore, the knowledge sets $k_b(\varrho_{L_0}, t \cdot t_1)$ and $k_b(\varrho_{L_0}, t \cdot t_2)$ – the $High$ inputs that may lead to these observations – that attackers gain when observing both traces $t \cdot t_1$ and $t \cdot t_2$ are also different, by determinism.

Since TINI forbids all information leaks, unless the information leak is due to the observation of divergence $\Uparrow$, Askarov et al. [AHSS08] proves that attackers' observation trees of programs satisfying TINI have a particular shape depicted in Figure 5.2. Intuitively, the observation tree $\mathcal{T}_b^{\varrho_{L_0}}$ of a program satisfying TINI cannot branch in order not to leak sensitive information, unless the branching is due to the observation of divergence $\Uparrow$.

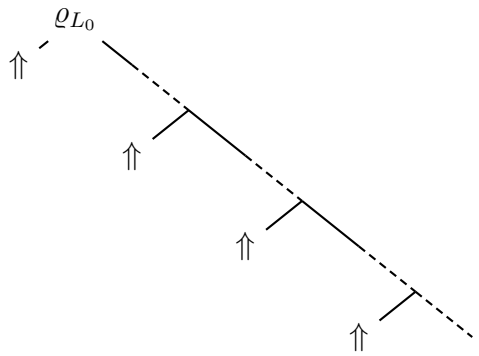


Figure 5.2: The shape of attackers' observation trees $\mathcal{T}_b^{\varrho_{L_0}}$ for deterministic non-interferent (wrt. TINI) programs

The attackers' observation tree illustrated by Figure 5.2 represents for instance the worst-case scenario of the non-interferent program in Listing 5.3, where each new observations attackers make is either an output value, or divergence $\Uparrow$.

The latter remark leads to Lemma 4 that Askarov et al. [AHSS08] prove. Intuitively, when a program satisfies TINI, then each node in its attackers' observation tree has at most two children. Moreover, one of these two children corresponds to the observation of divergence $\Uparrow$. Therefore, since the depth of an observation tree $\mathcal{T}_b^{\varrho_{L_0}}$ is bounded by b, the number of its leaves is hence bounded by $b + 1$.

Lemma 4 (Attackers' observation trees of non-interferent programs)**.** *For all deterministic programs satisfying TINI, for all Low input memories $\varrho_{L_0} \in \Sigma_{L_0}$, the attackers' observation tree $\mathcal{T}_b^{\varrho_{L_0}}$ has a cardinal of leaves bounded by:*

$$\left|Leaves(\mathcal{T}_b^{\varrho_{L_0}})\right| \leq b + 1.$$

As a consequence of Lemma 4, Askarov et al. [AHSS08] then prove that for deterministic non-interferent programs and a uniformly distributed secret, the probability that polynomial time attackers guess the secret after observing a program run is negligible in the size of the secret. This result directly translates into our quantitative framework by stating that the conditional vulnerability $V(H_U \mid O)$ – the remaining uncertainty – in the case of a uniformly distributed $High$ input memories, is negligible when attackers observe only a polynomial amount of outputs in the size of the secret:

$$\begin{aligned} V(H_U \mid O) &= \max_{\varrho_{L_0} \in \Sigma_{L_0}} \frac{\left|Leaves(\mathcal{T}_b^{\varrho_{L_0}})\right|}{2^N} \\ &\leq \frac{b+1}{2^N} \\ &= o(1) \text{ assuming } \log(b) = o(N) \end{aligned}$$

As a result, Askarov et al. deduce that TINI is a suitable security property, even when polynomial time attackers are allowed to observe intermediate steps of computation, provided that the secret is large enough and uniformly distributed.

A quantitative security property. TINI is a valid security property only when the secret is sufficiently large and sufficiently random [AHSS08, DS09, HS12]. This latter remark is the primary motivation behind our work; TINI allows an amount of information leakage that is negligible for a uniformly distributed secret since the probability of polynomial time attackers guessing the secret is negligible. Therefore, we propose to prove security of programs against this exact security guarantee, that we introduce in Definition 19 as relative secrecy (RS).

Definition 19 (Relative secrecy)**.**
Let N denote the size of the $High$ input memories in bits. Then a program is secure wrt. relative secrecy (RS) if the maximum leakage $\mathcal{ML}$ against polynomial time attackers grows strictly slower than the size of the secret:

$$\mathcal{ML} = o(N) \text{ assuming } \log(b) = o(N).$$

For deterministic programs, this condition translates into a simpler form:

$$\max_{\varrho_{L_0} \in \Sigma_{L_0}} \log_2 \left|Leaves(\mathcal{T}_b^{\varrho_{L_0}})\right| = o(N) \text{ assuming } \log(b) = o(N).$$

The quantitative security property introduced in Definition 19 is more permissive than TINI, since all deterministic programs satisfying TINI also satisfy the security property introduced in Definition 19, as a direct consequence of Lemma 4. Indeed, the cardinal number of leaves of attackers' observation trees for deterministic programs satisfying TINI is bounded by $b+1$. Thus, the maximum leakage $\mathcal{ML}$ is bounded by $\log_2(b+1)$. Therefore, if the bound b on attackers' observations is assumed to be polynomial in the size N of the secret, then the maximum leakage $\mathcal{ML}$ grows strictly slower than the size N of the secret. This result leads to Corollary 3, stating that all deterministic programs satisfying TINI are also secure wrt. RS.

Corollary 3 (Permissiveness)**.**
All deterministic programs satisfying TINI also satisfy RS.

For instance, both programs presented in Listings 5.1 and 5.3 are secure wrt. RS since they both satisfy TINI. In addition to that, the program in Listing 5.2 is also secure wrt. RS although it does not satisfy TINI.

RS also ensures the same security guarantees as TINI provides. RS guarantees that the probability of polynomial time attackers guessing the secret is negligible in the size of the secret. Indeed, assuming that the initial uncertainty about the secret is large enough, RS guarantees that the remaining uncertainty of attackers after observing a program run stays large enough, so that their probability of guessing the secret after observing a program run is still negligible. Let us assume that the initial uncertainty is large enough by supposing that min-entropy $\mathcal{H}_\infty(H)$ of the secret is equivalent to the size N of the secret ($\mathcal{H}_\infty(H) \sim N$) – for instance, this assumption is verified in the case of a uniformly distributed secret since $\mathcal{H}_\infty(H_U) = N$ *bits*. Consequently, if a program satisfies RS, then the remaining uncertainty is also large enough since it stays equivalent to N:

$$\begin{aligned}\mathcal{H}_\infty(H \mid O) &= \mathcal{H}_\infty(H) - \mathcal{L}_{\vec{\pi}} \\ &\sim N + o(N) \\ &\sim N\end{aligned}$$

Consequently, if a program satisfies RS, then the probability of polynomial time attackers guessing the secret is negligible, provided that their initial uncertainty about the secret is sufficiently large.

```
1 if (input ≤ secret){
2   output 1;
3 }
4 else {
5   output 0;
6 }
```

Listing 5.4: A program prone to linear time leakage across multiple runs

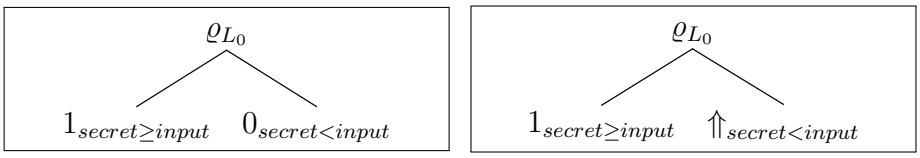
```
1 if (input ≤ secret){
2   skip;
3 }
4 else {
5   while (true){
6     skip;
7   }
8 }
9 output 1;
```

Listing 5.5: An equivalent program to the one in Listing 5.5, satisfying TINI

5.5 Min-capacity leakage against k-try attacks

Min-capacity is a crude measure of information leakage as Smith [Smi09] already notes. Indeed, min-capacity quantifies information leaks in terms of the probability that attackers could guess the secret in only one try. It does not account for scenarios where a program uses the same secret across multiple runs, allowing attackers to further refine their knowledge about the secret. As a consequence, adopting min-capacity as a measure for information leaks enables us to term both programs in Listings 5.1 and 5.2 as equivalent, since they both have a maximum leakage of 1 bit. Yet, min-capacity also deems the program in Listing 5.4 as equivalent to both the ones in Listings 5.1 and 5.2, although it is more dangerous when attackers are able to observe multiple runs of a program on the same secret.

Figure 5.3(a) illustrates the attackers' observation tree of the program in Listing 5.4. Therefore, we can deduce that the maximum leakage is $\mathcal{ML} = 1$ *bit* since the observation trees have at most 2 leaves, which means that it is indeed equivalent to the programs in Listings 5.1 and 5.2 when



(a) Listing 5.4 (b) Listing 5.5

Figure 5.3: Attackers' observation trees for the program in Listings 5.4 and 5.5

quantifying information flow for the one-try attacks scenario. Yet, if the secret remains the same across multiple executions, attackers can leverage this program to learn the secret in linear time in the size of the secret, through a dichotomous algorithm – a binary search algorithm for instance. Note that TINI also suffers from this flaw. Indeed, the program in Listing 5.5 is equivalent to the one in Listing 5.4, wrt. the observations and the knowledge attackers gain, as illustrated by Figure 5.3(b). Although this latter program in Listing 5.5 satisfies TINI, attackers can also leverage it to leak the secret across multiple executions in linear time in the size of the secret.

Note that, unlike the programs in Listings 5.4 and 5.5, both programs in Listings 5.1 and 5.2 only allow attackers to perform a brute-force attack across multiple executions. Indeed, the best strategy for attackers to learn the secret across multiple executions is exponential in the size of the secret. Therefore, we would like to improve the security property proposed in Definition 19 to account for the vulnerability of the secret in the case where attackers can observe multiple runs of a program on the same secret. Eventually, this new security property will enable us to accept the programs in Listings 5.1 and 5.2 as secure, while denying the programs in Listings 5.4 and 5.5.

Simulating k-try attacks. In the case where attackers can observe multiple runs of the program on the same secret, attackers are able to provide multiple *Low* inputs and observe the outputs of multiple runs. Therefore, they are also able to refine their knowledge about the secret. We can model such a scenario through self-composition [BDR04, BDR11]. Let us for instance illustrate this technique when attackers are able to observe 3 program runs.

```
1 P_k (input_k, secret_k)
2 {
3   if (input_k == secret_k){
4     output 1;
5   }
6   else {
7     output 0;
8   }
9 }
```

Listing 5.6: Independent copies P_k of the program in Listing 5.2

```
1 P_k (input_k, secret_k)
2 {
3   if (input_k ≤ secret_k
  ){
4     output 1;
5   }
6   else {
7     output 0;
8   }
9 }
```

Listing 5.7: Independent copies P_k of the program in Listing 5.4

In order to simulate a 3-try attack scenario for a program P, we consider 3 independent copies P_1, P_2 and P_3 of the program P, where every variable in P is renamed by, for instance, adding a subscript k in each copy P_k. Listings 5.6 and 5.7 illustrate for instance the independent copies of both programs in Listings 5.2 and 5.4. Therefore, we can simulate a scenario where attackers are able to observe 3 different runs of a program P on the same secret by considering the program $P^{(3)}$ that results from the composition of the independent copies P_1, P_2 and P_3. Listing 5.8 illustrates this latter program $P^{(3)}$. Note that $P^{(3)}$ accepts 3 *Low* input memories that represents the *Low* input memories attackers would provide if they run the program P 3 times, as well as one *High* input memory since $P^{(3)}$ simulates 3 different runs of the program P on the same secret.

```
1 secret_1 := secret;
2 secret_2 := secret;
3 secret_3 := secret;
4 P_1(input_1, secret_1)
5 P_2(input_2, secret_2)
6 P_3(input_3, secret_3)
```

Listing 5.8: The program $P^{(3)}$ simulating a 3-attack scenario for a program P through self-composition

Computing the maximum leakage of the program $P^{(3)}$ in Listing 5.8 enables us to quantify the leakage wrt. the probability of attackers guessing the secret after observing 3 different runs of a program P on the same secret.

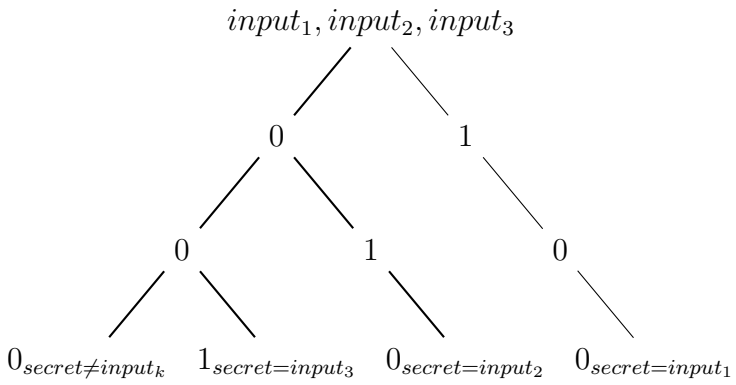


Figure 5.4: Attackers' observations of $P^{(3)}$ in the case of the program in Listing 5.2

Let us for instance compute the maximum leakage in the case of a 3-try attack scenario for both programs in Listings 5.2 and 5.4. By Theorem 4, min-capacity $\mathcal{ML}$ for the deterministic program $P^{(3)}$ that accepts 3 *Low* input memories is given by:

$$\mathcal{ML} = \max_{\varrho_{L_0} \in \Sigma_{L_0}^3} \log_2 \left| Leaves(\mathcal{T}_b^{\varrho_{L_0}}) \right|$$

Therefore, we need to compute the cardinal of leaves for attackers' observation trees of $P^{(3)}$ in order to compute the maximum leakage $\mathcal{ML}$ in the case of a 3-try attack scenario.

Let us assume that attackers provide 3 different *Low* input memories across the 3 different executions. Indeed, for deterministic programs rational attackers would not provide the second run of a program with the same *Low* input memory as the first run for instance, since they would make the same observations and would not be able to further refine their knowledge about the secret. Therefore, Figure 5.4 illustrates the attackers' observation trees for the program $P^{(3)}$ simulating a 3-try attack in the case of the program in Listing 5.2. Hence, we deduce that the maximum leakage is $\mathcal{ML} = log_2(4) = 2\ bits$. Additionally, Figure 5.5 illustrates the attackers' observation trees for the program $P^{(3)}$ in the case of the program in Listing 5.4.

Hence, we also deduce that the maximum leakage is $\mathcal{ML} = log_2(8) = 3\ bits$. As a consequence, we can deduce that the program in Listing 5.4 is indeed more dangerous than the program in Listing 5.2, in the case of a 3-try attack scenario.

In general, we can consider a program $P^{(k)}$ simulating a k-try attack scenario for a program P. In the case of the attackers' observation tree

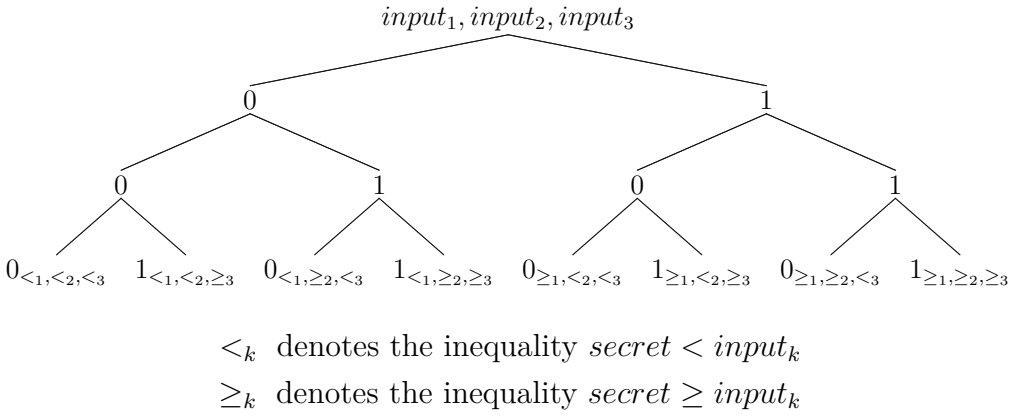


Figure 5.5: Attackers' observations of $P^{(3)}$ in the case of the program in Listing 5.4

in Figure 5.4 for instance, only the leftmost leaf will continue to branch – leading to 2 children nodes – as k grows. Therefore, we can deduce that the maximum leakage for the program in Listing 5.2 in the case of a k-try attack scenario is given by $\mathcal{ML} = log_2(k+1)$, since the number of leaves is an arithmetic sequence with a common difference of 1. In contrast, in the case of the observation tree in Figure 5.5, all the leaves continue to branch – leading to 2 children nodes – as k grows. Therefore, the maximum leakage for the program in Listing 5.4 is given by $\mathcal{ML} = log_2(2^k) = k$, since the number of leaves is a geometric sequence with a common ratio of 2. As a consequence, assuming that polynomial time attackers can only run a polynomial number of times – in the size of the secret – a program on the same secret, the program in Listing 5.2 can be considered as secure since min-capacity $\mathcal{ML}^{(k)}$ grows strictly slower than the size of the secret. On the contrary, the program in Listing 5.4 is insecure in the case of a k-try attack scenario since min-capacity $\mathcal{ML}^{(k)}$ does not grow strictly slower than the size N of the secret, assuming k is polynomial in N. Therefore, we propose in Definition 20 a security property that accounts for multiple try attack scenarios as well. This security property ensures that polynomial time attackers, if allowed to run a program on the same secret a polynomial number of times, only have a negligible probability of guessing the secret.

Definition 20 (Relative secrecy for k-try attacks)**.**
Let N denote the size of the High input memories in bits. Then a program P is secure against polynomial time attackers, in the case of a multiple try attack scenario, if the maximum leakage $\mathcal{ML}^{(k)}$ of the program $P^{(k)}$

simulating a k-try attack is negligible when k is assumed polynomial in the size of the secret:

$$\mathcal{ML}^{(k)} = o(N) \text{ assuming } \log(b) = o(N) \text{ and } \log(k) = o(N).$$

Both programs in Listings 5.1 and 5.2 are secure wrt. RS for k-try attacks, as introduced in Definition 20. Indeed, Listing 5.1 illustrates for instance the shape of the attackers' observation trees in the case of the program in Listing 5.1. As k grows, only the leftmost branch continues to branch leading to 2 new children nodes. Therefore, the cardinal of leaves for the attackers' observation trees is also equal to $k + 1$, which means that min-capacity for the program simulating a k-try attack is given by $\mathcal{ML}^{(k)} = \log_2(k + 1)$ and thus grows stricly slower than the size of the secret N – assuming k is polynomial in N.

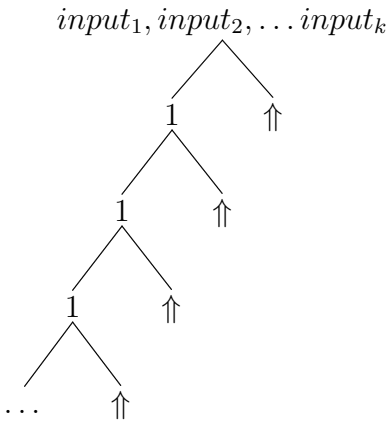


Figure 5.6: Attackers' observations of $P^{(k)}$ in the case of the programs in Listings 5.1 and 5.5

Note that Figure 5.6 also represents the attackers' observation trees for the program in Listing 5.5. Thus, this latter program is also accepted as secure since it satisfies RS for k-try attacks. This is due to the fact that the program $P^{(k)}$ simulates a k-try attack by sequential composition rather than parallel composition [BDR04, BDR11]. Indeed, the program $P^{(k)}$ implicitly allows attackers to observe the outputs of a new run of the program P only when the previous run does not diverge.

To sum up, the security property introduced in Definition 20 does indeed deem as secure both programs in Listings 5.1 and 5.2. Additionally, it also succeeds in denying the program in Listing 5.4 since this program is more dangerous when attackers are able to run it multiple times on the same secret. Yet, RS for k-try attacks fails to deny the program in Listing 5.5 since

it invalidates our intuition that both programs in Listings 5.4 and 5.5 are equivalent wrt. the observations attackers make and the knowledge they gain. Indeed, in a scenario where attackers are allowed to observe the outputs of a new run only when the previous run terminates, both programs are not equivalent: the program in Listing 5.4 remains more dangerous than the one in Listing 5.5. However, if attackers can observe multiple runs of a program on the same secret irrespective of the results of previous computations, RS for k-try attacks, as introduced in Definition 20, needs to be further tweaked to account for this particular scenario. We leave investigating RS in such a scenario for future work.

5.6 Related Work

Our formalization of RS is inspired by the work of Volpano and Smith [VS00], Smith [Smi09, Smi11] as well as Askarov et al. [AHSS08]. The common denominator of these approaches is quantifying information flow wrt. the probability of attackers guessing the secret after observing a program run.

Volpano and Smith [VS00] relaxes an existing type system [VIS96] enforcing TINI, by ignoring implicit information flows due to programs testing if the value of a $High$ variable is equal to a Low variable. They prove that their type system enforces a relative secrecy property: a well typed progam running in polynomial time can only leak the secret with negligible probability. Their type system is designed for a deterministic language with no intermediate outputs. The relative secrecy property we propose bounds the running time of attackers rather than the running time of the programs themselves. Indeed, there is no need to put the strain on the running time of a program as long as it produces no outputs observable by attackers.

Askarov et al. [AHSS08] prove that although TINI programs may leak more than one bit of information, when attackers can observe intermediate steps of computation, polynomial time attackers can only guess the secret with a negligible probability. We propose to prove security of programs by relying on the exact property they prove for programs satisfying TINI: the probability of polynomial attackers guessing the secret is negligible in the size of the secret. To this end, we build on Smith's [Smi09, Smi11] framework in order to quantify information flow wrt. the probability of attackers guessing the secret. We extend Smith's work for programs that accept both Low and $High$ inputs, and may output intermediate steps of computation. We also introduce RS as a security property ensuring that the probability of polynomial time attackers guessing the secret in one try is negligible. Finally, we extend RS for a k-try attack scenario, by relying

on sequential self-composition [BDR04, BDR11].

5.7 Summary

In this chapter, we propose RS as a new quantitative security property. This property quantifies the leakage in terms of the probability of polynomial time attackers guessing the secret in one try. RS is based on min-entropy, a quantitative measure of information proposed by Smith [Smi09, Smi11], that we adapt to a model where attackers can observe intermediate steps of computation. RS also sets the threshold beneath which programs are deemed secure such that it is more permissive than TINI, while providing the same security guarantees wrt. the probability of attackers guessing the secret.

We extend RS to quantify the leakage of programs when attackers can observe multiple runs of a program on the same secret, by relying on a self-composition technique [BDR04, BDR11]. Thus, we propose to deem a program as secure if the min-capacity of its self-composed version – simulating a k-try attack scenario – grows strictly slower than the size of the secret, assuming that attackers can only observe a polynomial number of runs.

We also characterize RS, for deterministic programs, as the maximum cardinal of leaves for attackers' observation trees over all *Low* input memories. Therefore, in the remaining chapters we develop novel static analyses techniques in order to over-approximate the cardinal of leaves for such attackers' observation trees. We stage our developments into two steps:

1. we restrict our target programs to batch-job ones – programs that only output the result of their computation at the end of their execution – and propose the *cardinal abstraction* to quantify information leaks for such programs, then

2. we build on the cardinal abstraction in order to propose the *tree abstraction*, which quantifies information leaks for programs that may also output intermediate steps of computation.

Our proposal of a new quantitative security property is driven by the need to relax the non-interference security property. Therefore, our next developments will also try to focus on proposing static analyses techniques that are at least as permissive as static enforcement techniques enforcing TINI. Ideally, both the cardinal abstraction and the tree abstraction should

deem as secure programs that are typed by state-of-the-art type systems enforcing TINI.

Chapter 6

Cardinal Abstraction

In the previous chapter, we characterize min-capacity for deterministic programs and propose relative secrecy (RS) as a quantitative security property. Our goal is therefore to propose a static analysis technique that approximates min-capacity in order to decide whether a program is secure wrt. RS.

In this chapter, we restrict ourselves to quantify information leaks for deterministic batch-job programs. Such programs do not output any intermediate steps of computation: attackers are able to observe a set of *Low* variables only at the end of the program's computation.

6.1 Min-capacity for Batch-job Programs

This section characterizes min-capacity $\mathcal{ML}$ for deterministic batch-job programs. Let us consider a command c that outputs no intermediate steps of computation. Let us also assume a set of *Low* variables $v_1, v_2, \ldots v_f$ attackers can observe at the end of command c's execution. Then, the program in Listing 6.1 illustrates the observations attackers make after running command c.

```
1 c;
2 output v1;
3 output v2;
4 ...
5 output vf;
```

Listing 6.1: Illustrating attackers' observations for batch-job programs

As a consequence, we can simply characterize min-capacity for batch-job

programs by relying on the previous framework introduced in Chapter 5. Indeed, by Theorem 4 we know that the maximum leakage $\mathcal{ML}$ is given by:

$$\mathcal{ML} = \max_{\varrho_{L_0} \in \Sigma_{L_0}} \log_2 \left| Leaves(\mathcal{T}_b^{\varrho_{L_0}}) \right|.$$

Additionally, the leaves of attackers' observation trees for the program in Listing 6.1 are solely determined by the final environments ϱ resulting from the executions of command $\mathtt{c}$. Therefore, these leaves are given by:

$$Leaves(\mathcal{T}_b^{\varrho_{L_0}}) = \\ \{\varrho(v_1) \cdot \varrho(v_2) \ldots \cdot \varrho(v_f) \mid \exists \varrho_{H_0} \in \Sigma_{H_0}, \varrho_0 \triangleq (\varrho_{L_0}, \varrho_{H_0}) \text{ and } \langle c, \varrho_0 \rangle \rightarrow^*_{\Downarrow} \varrho\} \tag{6.1}$$

Note that for batch-job programs, we ignore the bound b on attackers' observations, since we assume attackers can observe all *Low* output variables $\mathtt{v_1}, \mathtt{v_2}, \ldots \mathtt{v_f}$.

Computing min-capacity for deterministic batch-job programs requires maximizing the cardinality of the set introduced in Equation (6.1), over all *Low* input memories $\varrho_{L_0} \in \Sigma_{L_0}$. As a consequence, the problem of bounding min-capacity can be reduced to the problem of finding the set of reachable states at the end of a program for each *Low* input memory, then counting the set of observations attackers can make. This is an interesting remark even if finding the exact set of reachable program states is not computable in general. Indeed, there is a large body of the literature in program analysis that aims at approximating the set of reachable states, both in model checking [CGP99] and static analysis [NNH99]. However, we argue in the following that in the presence of *Low* inputs, reducing the problem of bounding min-capacity to a problem of approximating the reachable program states is inefficient.

Programs with no *Low* inputs. For deterministic batch-job programs that accept only *High* inputs, the problem of bounding min-capacity $\mathcal{ML}$ can be efficiently reduced to a problem of approximating the set of reachable states at the end of a program.

Consider for instance the program in Listing 6.2 where variable $\mathtt{s}$ is a *High* input and variable $\mathtt{x}$ is a *Low* observable output. Table 6.1 illustrates the reachable states for this program. Note that we let pp_k denote the program point at Line k. Since attackers can only observe the values variable $\mathtt{x}$ may take, they may therefore observe either the value 1 or 2. Thus, the maximum leakage for this program is given by $\mathcal{ML} = \log_2(2) = 1$ *bit*, which

```
1 x := s mod 2;
2 x := x + 1;
```

Listing 6.2: A program accepting only $High$ inputs

Table 6.1: The reachable states for the program in Listing 6.2

pp_1	$\{s \mapsto 0, x \mapsto 0\}, \{s \mapsto 1, x \mapsto 1\}, \{s \mapsto 2, x \mapsto 0\} \ldots \{s \mapsto 2^N - 1, x \mapsto 1\}$
pp_2	$\{s \mapsto 0, x \mapsto 1\}, \{s \mapsto 1, x \mapsto 2\}, \{s \mapsto 2, x \mapsto 1\} \ldots \{s \mapsto 2^N - 1, x \mapsto 2\}$

accounts for the intuition that this program only leaks the parity bit of the secret variable `s`.

Since finding the set of reachable states of a program is not computable in general, we can rely on existing program analysis techniques to approximate such a set. Köpf and Rybalchenko [KR10, KR13] for instance propose a generic framework to bound various quantitative measures, among which the maximum leakage $\mathcal{ML}$ – min-entropy under a uniformly distributed secret by Theorem 4. They approximate min-capacity by computing lower bounds through model checking approaches, as well as upper bounds through abstract interpretation [CC77] techniques.

Since we are interested in over-approximating min-capacity, we will focus on the framework of abstract interpretation which provides a theory of sound semantics approximations. For instance, instead of finding the concrete values variables may take, we can choose to approximate these values by an interval analysis [CC77]. Table 6.2 illustrates this analysis for the program in Listing 6.2. Thus, by computing an over-approximation of the reachable states, we can deduce an over-approximation of the observations attackers can make and at the same time an over-approximation of the maximum leakage $\mathcal{ML}$.

Table 6.2: Approximating the reachable states for the program in Listing 6.2 by an interval analysis

pp_1	$\{s \mapsto [0, 2^N - 1], x \mapsto [0, 1]\}$
pp_2	$\{s \mapsto [0, 2^N - 1], x \mapsto [1, 2]\}$

In this example, the interval analysis in Table 6.2 is precise: it computes a precise range $[1, 2]$ for the integer variable `x`, which yields a precise over-approximation of min-capacity $\mathcal{ML} = 1$ *bit*. Yet, abstract interpretation in general may incur a loss of precision, depending on the underlying

approximate representations. There is indeed a fairly large body of the literature in abstract interpretation that introduces various approximate representations, formally called abstract domains. These abstract domains cover a broad spectrum of trade-off between the precision of the analysis and its cost: polyhedra [CH78] and octagons [Min06b] lie at the most precise and costly end of the spectrum, whereas congruence relations [Gra89] and intervals [CC76, CC77] are on the lower end of the spectrum.

Köpf and Rybalchenko's approach is a generic one since one can plug in any existing abstract domain aimed at over-approximating the set of reachable states, in order to compute upper-bounds over min-capacity. Yet, for programs that also accept *Low* inputs, relying on traditional abstract domains aimed at approximating the set of reachable states is inefficient in general.

Programs with *Low* inputs. Consider for instance the program in Listing 6.3, where variable `s` is a *High* input, whereas variable `input` is a *Low* input. Similarly to the previous program in Listing 6.2, let us assume that only variable `x` is a *Low* observable output.

```
1 x := s mod 2;
2 x := x + input;
```

Listing 6.3: A program accepting both *Low* and *High* inputs

Notice first that relying on traditional abstract domains to compute an over-approximation of the set of reachable states – irrespective of the *Low* inputs – for the program in Listing 6.3 yields a valid, yet quiet imprecise, upper-bound over min-capacity $\mathcal{ML}$. Indeed, assuming that variable `input` ranges through the interval $[0, 2^N - 1]$ of integers, variable `x` will range through the interval $[0, 2^N]$. Therefore, the resulting upper-bound over min-capacity will be equal to $\log_2(2^N + 1) \approx N$ *bits*. Yet, this bound is quiet imprecise since the program in Listing 6.3 only leaks one bit of sensitive information: the parity bit of variable `s`.

In fact, according to Equation (6.1), we can compute an approximation of the set of reachable states for each *Low* input memory in order to over-approximate the sets of observations attackers can make. Then, we can deduce a more precise upper-bound over min-capacity $\mathcal{ML}$ by maximizing the cardinality of the computed sets of attackers' observations.

Intuitively, we can reduce a program accepting *Low* inputs to a set of sub-programs that accept no *Low* inputs, then rely on traditional abstract domains to over-approximate the set of reachable states for each

sub-program. For instance, Figure 6.1 illustrates the family of sub-programs induced by the program in Listing 6.3, assuming that variable `input` ranges over the interval $[0, 2^N - 1]$ of integers. Note that the maximum min-capacity over all sub-programs in Figure 6.1 is by definition equal to the min-capacity of the program in Listing 6.3. As a consequence, we can over-approximate min-capacity for all the sub-programs represented in Figure 6.1, by relying on traditional abstract domains. Finally, the maximum over all over-approximated min-capacities yields an upper bound for min-capacity of our target program in Listing 6.3.

```
1 x := s mod 2;
2 x := x + 0;
```

Listing 6.4: sub-program 0

```
x := s mod 2;
x := x + 1;
```

Listing 6.5: sub-program 1

```
x := s mod 2;
x := x + 2;
```

Listing 6.6: sub-program 2

```
1 x := s mod 2;
2 x := x + 3;
```

Listing 6.7: sub-program 3

...

```
x := s mod 2;
x := x + 2^N - 1;
```

Listing 6.8: sub-program $2^N - 1$

Figure 6.1: The sub-programs induced by the program in Listing 6.3

Table 6.3 illustrates for instance the over-approximated reachable states for all the sub-programs in Figure 6.1, by relying on an interval analysis. As a result, we deduce that min-capacity for all these sub-programs is equal to $\log_2(2) = 1$ *bit*, since attackers are only allowed to observe variable `x`. Finally, we also deduce 1 *bit* as an upper-bound for min-capacity of our target program in Listing 6.3.

In general, any target program that accepts *Low* inputs can be reduced to a set of sub-programs that accept no *Low* inputs. Thus, we can upper-bound min-capacity of the target program by relying on traditional abstract domains that over-approximate the set of reachable states for the induced sub-programs. However, this would require running as many analyses as the size of the *Low* input memories. Therefore, relying on traditional abstract domains to get a precise upper-bound over min-capacity for programs accepting *Low* inputs is computationally inefficient in general.

Towards the cardinal abstraction. We propose to abstract even more the results of the interval analyses obtained in Table 6.3. Indeed, let us

Table 6.3: Approximating the reachable states for the sub-programs in Figure 6.1 by an interval analysis

sub-program 0	
pp_1	$\{s \mapsto [0, 2^N - 1], input \mapsto [0, 0], x \mapsto [0, 1]\}$
pp_2	$\{s \mapsto [0, 2^N - 1], input \mapsto [0, 0], x \mapsto [0, 1]\}$

sub-program 1	
pp_1	$\{s \mapsto [0, 2^N - 1], input \mapsto [1, 1], x \mapsto [0, 1]\}$
pp_2	$\{s \mapsto [0, 2^N - 1], input \mapsto [1, 1], x \mapsto [1, 2]\}$

...

sub-program $2^N - 1$	
pp_1	$\{s \mapsto [0, 2^N - 1], input \mapsto [2^N - 1, 2^N - 1], x \mapsto [0, 1]\}$
pp_2	$\{s \mapsto [0, 2^N - 1], input \mapsto [2^N - 1, 2^N - 1], x \mapsto [2^N - 1, 2^N]\}$

focus only on the cardinal number of values variables may take, instead of keeping track of their range of values. Interestingly, this abstraction enables the computation of an over-approximation of min-capacity directly, without having to first approximate the set of reachable states for all the sub-programs induced by *Low* inputs. For instance, Table 6.4 abstracts the interval analyses' results presented in Table 6.3, in order to track only the cardinal number of values variables may take. Therefore, assuming that attackers only observe variable `x`, we can deduce that they make at most 2 different observations. Thus, we can also deduce that min-capacity of the program in Listing 6.3 is at most 1 *bit*.

Table 6.4: An abstraction of the interval analyses in Table 6.3

pp_1	$\{s \mapsto 2^N, input \mapsto 1, x \mapsto 2\}$
pp_2	$\{s \mapsto 2^N, input \mapsto 1, x \mapsto 2\}$

In fact, an analysis that only keeps track of the cardinal of values variables may take will lose precision whenever it encounters a conditional instruction. Consider for instance the program in Listing 6.9 that is annotated with an analysis that only tracks the cardinal of values variables may take. Let us assume that variable `secret` is a *High* input, whereas variable `input` is

a *Low* input. At both Lines 2 and 5, the analysis determines that variable `x` has only one possible value since it is assigned a constant. Then, since variable `x` has only 1 possible value in each branch of the conditional, the analysis soundly deduces that variable `x` has at most 2 different values at the merge point of the conditional at Line 6. Similarly, variable `input` also has 1 possible value in each branch of the conditional. Therefore, the analysis deduces that variable `input` has at most 2 different values at the merge point of the conditional at Line 6, although it is not modified inside the conditional branches. In order to retain some precision for variable `input`, the analysis must know that variable `input` is not modified inside the conditional branches. We achieve this by letting the cardinal abstraction track both the cardinal number of values variables may take, as well as the program points where variables may have been last assigned.

```
0 // {secret ↦ 2^N, input ↦ 1, x ↦ 1}
1 if (secret > input){
2   x := 0; // {secret ↦ 2^N, input ↦ 1, x ↦ 1}
3 }
4 else {
5   x := 1; // {secret ↦ 2^N, input ↦ 1, x ↦ 1}
6 } // {secret ↦ 2^N, input ↦ 2, x ↦ 2}
```

Listing 6.9: An analysis only keeping track of the cardinal of values

Listing 6.10 annotates the program in Listing 6.9 with the results of the cardinal abstraction. Unlike the previous analysis in Listing 6.9, the cardinal abstraction retains some precision for variables that are not modified inside conditional branches for instance. As a result, the cardinal abstraction is able to determine that, at Line 6, variable `input` does indeed have only 1 possible value.

6.2 Abstract Semantics

This section formalizes the cardinal abstraction. We consider a deterministic `While` language [Win93] introduced in Figure 6.2. Expressions include unsigned integers n of finite size κ, variables `id`, binary arithmetic operations (bop), comparison operations (cmp) as well as the modulo operation (mod). Commands are instructions identified by a unique program point $pp \in \mathbb{P}$.

```
0 // {secret ↦ ({pp0}, 2^N), input ↦ ({pp0}, 1), x ↦ ({pp0}, 1)}
1 if (secret > input){
2   x := 0;// {secret ↦ ({pp0}, 2^N), input ↦ ({pp0}, 1), x ↦ ({pp2}, 1)}
3 }
4 else {
5   x := 1;// {secret ↦ ({pp0}, 2^N), input ↦ ({pp0}, 1), x ↦ ({pp5}, 1)}
6 } // {secret ↦ ({pp0}, 2^N), input ↦ ({pp0}, 1), x ↦ ({pp2, pp5}, 2)}
```

Listing 6.10: The results of the cardinal abstraction on the program in Listing 6.9

Expressions:	a	$::= n$	(constants)
		$\mid id$	(variables)
		$\mid a_1\ bop\ a_2$	(binary operators)
		$\mid a\ mod\ n$	(modulo operator)
		$\mid a_1\ cmp\ a_2$	(comparison operators)
	bop	$::= + \mid - \mid \times \mid \div$	
	cmp	$::= \leq \mid \geq \mid < \mid > \mid ==$	
Commands:	c	$::= {}^{pp}skip$	(empty instruction)
		$\mid {}^{pp}id := a$	(assignment)
		$\mid c_1;\ c_2$	(sequence)
		$\mid {}^{pp}if\ (a)\ c_1\ else\ c_2$	(conditional)
		$\mid {}^{pp}while\ (a)\ c$	(loop)

Figure 6.2: Abstract syntax of `While`

6.2.1 Abstract Domain

The cardinal abstract domain computes an over-approximation of the cardinal number of values variables can take, when attackers provide an arbitrary *Low* input memory.

We assume that the set of variables of analysed programs are partitioned into *Low* and *High* input variables. We also assume that all input variables can range from 0 to $2^\kappa - 1$. Therefore, initially at program point pp_0, *Low* input variables have only 1 possible value, whereas *High* input variables

have 2^κ possible values as illustrated by Listing 6.10 for the program point pp_0.

Definition 21 introduces the cardinal abstract domain. Abstract values are pairs of a set s_{pp} of program points and a cardinal number $n \in [0, 2^\kappa]$. The resulting lattice is actually the cartesian product of 2 lattices:

1. the set $\mathcal{P}(\mathbb{P})$ of all subsets of $\mathbb{P}$, ordered via set inclusion $\subseteq$, with set union $\cup$ as a join operator and set intersection $\cap$ as a meet operator, as well as
2. the set of natural numbers $[0, 2^\kappa]$, ordered via the natural order $\leq$ over integers, with max as a join operator and min as a meet operator.

Definition 21 (Cardinal abstract domain)**.**
The cardinal abstract domain is defined as the lattice

$$\left(D_C^\sharp, \subseteq_\otimes, (\varnothing, 0), (\mathbb{P}, 2^\kappa), \cup_\otimes, \cap_\otimes\right)$$

where:

$$D_C^\sharp \triangleq \mathcal{P}(\mathbb{P}) \times [0, 2^\kappa] \qquad \subseteq_\otimes \triangleq \subseteq \times \leq$$
$$\cup_\otimes \triangleq \cup \times \max \qquad \cap_\otimes \triangleq \cap \times \min$$

Our analysis maps each variable id to a pair of a set s_{pp} of program points and a cardinal number n. The set s_{pp} represents the program points where variable id may have been last assigned, whereas n represents the cardinal number of values variable id may take.

6.2.2 Semantics of Expressions

We denote by $\varrho^\sharp \in Var \rightharpoonup D_C^\sharp$ an abstract environment that maps each variable id to an abstract value $(s_{pp}, n) \in D_C^\sharp$. We also denote by $proj_i()$ the projection onto the ith component of a tuple. Figure 6.3 introduces the abstract semantics of expressions $\mathbb{A}^\sharp \in Exp \times (Var \rightharpoonup D_C^\sharp) \to [0, 2^\kappa]$.

The abstract semantics of expressions $\mathbb{A}^\sharp$ evaluates an expression `a` in an abstract environment $\varrho^\sharp$, in order to yield a cardinal $n \in [0, 2^\kappa]$ representing the cardinal number of values expression `a` may evaluate to. Constant expressions always evaluate to a single value. Binary arithmetic operations $\texttt{a}_1$ `bop` $\texttt{a}_2$ yield at most the product of the cardinal of $\texttt{a}_1$ and the cardinal of $\texttt{a}_2$, or 2^κ when this product overflows. Comparison operations $\texttt{a}_1$ `cmp` $\texttt{a}_2$ evaluate to at most 2 different values (true or false), unless both $\texttt{a}_1$ and $\texttt{a}_2$ evaluates to only one possible value. The modulo operation $\texttt{a}_1$ `mod n`

$$\mathbb{A}^\sharp[\![n]\!]\varrho^\sharp = 1 \qquad \mathbb{A}^\sharp[\![id]\!]\varrho^\sharp = proj_2(\varrho^\sharp(id))$$

$$\mathbb{A}^\sharp[\![a_1\ bop\ a_2]\!]\varrho^\sharp = \min\left(\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp \times \mathbb{A}^\sharp[\![a_2]\!]\varrho^\sharp, 2^\kappa\right)$$

$$\mathbb{A}^\sharp[\![a_1\ mod\ n]\!]\varrho^\sharp = \min\left(\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp, n\right)$$

$$\mathbb{A}^\sharp[\![a_1\ cmp\ a_2]\!]\varrho^\sharp = \min\left(\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp \times \mathbb{A}^\sharp[\![a_2]\!]\varrho^\sharp, 2\right)$$

Figure 6.3: Abstract semantics of expressions

produces at most n different values or $\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp$ different values when $\texttt{a}_1$ may take less than n values.

Note that for simplicity of presentation, the second argument of the modulo operation accepts only a literal rather than an expression since the cardinal abstract domain needs to rely on numerical abstractions otherwise. For instance, if a numerical abstraction determines that an expression $\texttt{a}_2$ has a maximum value of 10 for instance, then the cardinal expression can soundly conclude that expression $\texttt{a}_1$ `mod` $\texttt{a}_2$ may evaluate to at most 10 different values or $\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp$ different values when $\texttt{a}_1$ may take less than 10 values. The process of combining different abstract domains in order to refine the precision of the analysis is called a reduced product [CC79b].

In general, relying on reduced products using traditional abstract domains can refine most of the cardinal abstract domain definitions. For instance, if an interval-based abstraction [CC76, CC77] determines that $\texttt{a}_1$ equals zero, then a simple refinement would yield only one possible value for expression $\texttt{a}_1 \times \texttt{a}_2$.

6.2.3 Semantics of Instructions

Figure 6.4 introduces the abstract semantics of instructions. This abstract semantics $[\![c]\!]^\sharp \in (Var \rightharpoonup D_C^\sharp) \to (Var \rightharpoonup D_C^\sharp)$ evaluates a command `c` in an abstract environment $\varrho^\sharp$, then yields a new abstract environment.

Commands ^{pp}`skip` do not modify input abstract environments. Assignments ^{pp}`id := a` map the abstract value $(pp, \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp)$ to variable `id` since `id` can take as many values as expression `a`, and is assigned at program point pp. A sequence of commands $\texttt{c}_1$`;` $\texttt{c}_2$ composes the abstract semantics of the second command $\texttt{c}_2$ with the abstract semantics of the first one $\texttt{c}_1$.

Recall that the cardinal abstract domain computes the cardinal number of values variables may take, when attackers provide a fixed *Low* input

$$
\begin{aligned}
[\![^{pp} skip]\!]^\sharp \varrho^\sharp &\triangleq \varrho^\sharp \\
[\![^{pp} id := a]\!]^\sharp \varrho^\sharp &\triangleq \varrho^\sharp[id \mapsto (pp, \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp)] \\
[\![c_1;\ c_2]\!]^\sharp \varrho^\sharp &\triangleq [\![c_2]\!]^\sharp([\![c_1]\!]^\sharp \varrho^\sharp) \\
[\![^{pp} if\ (a)\ c_1\ else\ c_2]\!]^\sharp \varrho^\sharp &\triangleq \text{let } n = \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp \text{ in} \\
&\quad \text{let } \varrho_1^\sharp = [\![c_1]\!]^\sharp \varrho^\sharp \text{ in} \\
&\quad \text{let } \varrho_2^\sharp = [\![c_2]\!]^\sharp \varrho^\sharp \text{ in} \\
&\quad \lambda id. \begin{cases} \varrho_1^\sharp(id) \cup_\otimes \varrho_2^\sharp(id) & \text{if } n = 1 \\ \varrho_1^\sharp(id) \cup_{add(c_1,c_2)} \varrho_2^\sharp(id) & \text{otherwise} \end{cases} \\
[\![^{pp} while\ (a)\ c]\!]^\sharp \varrho^\sharp &\triangleq \text{let } \varrho_2^\sharp = \text{lfp}^{\dot{\subseteq}_\otimes}_{\varrho^\sharp} [\![c]\!]^\sharp \text{ in} \\
&\quad \text{let } n = \mathbb{A}^\sharp[\![a]\!]\varrho_2^\sharp \text{ in} \\
&\quad \begin{cases} \varrho_2^\sharp & \text{if } n = 1 \\ \lambda id.\, \text{top}_{(c)}(\varrho_2^\sharp(id)) & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&(s_{pp}, n) \cup_{add(c_1,c_2)} (s'_{pp}, n') \triangleq \\
&\quad \begin{cases} (s_{pp}, n) \cup_\otimes (s'_{pp}, n') & \text{if } PP(c_1; c_2) \cap (s_{pp} \cup s'_{pp}) = \varnothing \\ \left(s_{pp} \cup s'_{pp}, min(n + n', 2^\kappa)\right) & \text{otherwise} \end{cases} \\
&\text{top}_{(c)}\left((s_{pp}, n)\right) \triangleq \begin{cases} (s_{pp}, n) & \text{if } PP(c) \cap s_{pp} = \varnothing \\ (s_{pp}, 2^\kappa) & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
PP(^{pp} skip) \triangleq \{pp\} \qquad PP(^{pp} id := a) \triangleq \{pp\}
$$

$$
PP(c_1;\ c_2) \triangleq PP(c_1) \cup PP(c_2) \qquad PP(^{pp} while\ (a)\ c) \triangleq \{pp\} \cup PP(c)
$$

$$
PP(^{pp} if\ (a)\ c_1\ else\ c_2) \triangleq \{pp\} \cup PP(c_1) \cup PP(c_2)
$$

Figure 6.4: Abstract semantics of instructions

memory. Recall also that we assume a partition of the set of variables into *Low* and *High* input variables, so that all variables are initialized at the initial program point pp_0. Therefore, initially at the entry program point pp_0, *Low* input variables have only 1 possible value since they are fixed by attackers. On the opposite, *High* input variables range over the interval $[0, 2^\kappa - 1]$ of integers. Therefore, *High* input variables have 2^κ possible values at the entry program point pp_0.

When the cardinal abstraction determines that a variable `id` has at most 1 possible value, it also implicitly determines that variable `id` may only depend on *Low* input memories. Indeed, if variable `id` depended on *High* input memories, then there would exist two *High* input memories that yield two different values for variable `id`. Therefore, the cardinal abstraction would determine that variable `id` has a cardinal number of values greater than 2, since it over-approximates the cardinal number of values variables may take. This remark will shortly prove useful when defining the abstract semantics of conditionals and loops.

Conditional instructions. The abstract semantics of conditional instructions considers two different cases depending on the abstract value of the conditional guard.

First, if the conditional guard has only one possible value, then it may only depend on *Low* inputs. Hence, the evaluation of the conditional always executes the same conditional branch for each fixed *Low* input. Therefore, the join operator $\cup_\otimes$ lifted over environments soundly over-approximates the semantics of conditionals, by computing the set union over the set of program points where each variable may be last assigned, as well as the maximum cardinal over both branches for each variable. Indeed, if a variable `id` can take at most n_1 values (resp. n_2 values) in the then branch (resp. the else branch) assuming *Low* inputs are fixed, the cardinal abstract domain soundly concludes that variable `id` can take at most $\max(n_1, n_2)$ values after the conditional when *Low* inputs are fixed.

Second, if the conditional guard has more than one value, then it may depend on *High* inputs. Hence, the evaluation of the conditional may execute both conditional branches for each fixed *Low* inputs. Assuming that variable `id` can have at most n_1 values (resp. n_2 values) in the then branch (resp. the else branch), the cardinal abstract domain soundly concludes that variable `id` can take at most $n_1 + n_2$ values after the conditional, or 2^k values if the latter sum overflows.

However, the operator $\cup_{add(c_1,c_2)}$ still retains precision for the variables that are written neither in $\mathtt{c}_1$ nor in $\mathtt{c}_2$, by computing the join $\cup_\otimes$ over

their abstract values. Therefore, for variables that are written neither in $\texttt{c}_1$ nor $\texttt{c}_2$, the cardinal abstract domain simply computes the set union over the program points where these variables may be defined and the maximum over both their cardinals. As for variables that may be written in either $\texttt{c}_1$ or $\texttt{c}_2$, the cardinal abstract domain computes the set union over the program points where they may be defined and sums both their cardinals.

In order to determine which variables may be modified within an instruction `c`, the cardinal abstract domain relies on the set of program points where variables may be defined as well as the operator $PP(c)$, that we define as the set of program points appearing in command `c`. Hence, instruction `c` does not modify variable `id` if $PP(c) \cap s_{pp} = \varnothing$, supposing s_{pp} over-approximates the set of program points where `id` may be last assigned. Note that this condition is only a sufficient one since s_{pp} is an over-approximation of the set of program points where variable `id` may be last assigned.

In general, abstract domains retain precision in conditional branches by relying on the conditional guard to reduce the abstract environments entering both the then-branch and the else-branch. For instance, let us assume that an interval analysis determines that variable `x` ranges over the interval $[0, 9]$, whereas variable `y` ranges over the interval $[2, 5]$. Then, in the case of a conditional guard (`x == y`) testing the equality of variables `x` and `y`, the interval analysis can soundly reduce both abstract values mapped to `x` and `y` in the then-branch to the intersection $[2, 5]$ of both intervals. For simplicity of presentation, the cardinal abstract semantics of conditionals presented in Figure 6.4 does not attempt to leverage on such reductions. We leave this improvement as future work.

Loop instructions. Let us denote by $\dot{\subseteq}_\otimes$ (resp. by $\dot{\cup}_\otimes$) the pointwise lifting of the partial order relation $\subseteq_\otimes$ (resp. of the join operator $\cup_\otimes$) over environments:

$$\begin{aligned} &\text{Partial order:} && \varrho_1^\sharp \mathrel{\dot{\subseteq}_\otimes} \varrho_2^\sharp \iff \forall id, \varrho_1^\sharp(id) \subseteq_\otimes \varrho_2^\sharp(id) \\ &\text{Join operator:} && \varrho_1^\sharp \mathrel{\dot{\cup}_\otimes} \varrho_2^\sharp = \lambda id. \varrho_1^\sharp(id) \cup_\otimes \varrho_2^\sharp(id) \end{aligned}$$

The abstract semantics of loops requires computing an abstract environment $\varrho^\sharp$ that is a loop invariant $\dot{\subseteq}_\otimes$-greater than the initial abstract environment $\varrho_0^\sharp$. This loop invariant $\varrho^\sharp$ satisfies the following fixpoint equation:

$$\varrho^\sharp = \varrho_0^\sharp \mathrel{\dot{\cup}_\otimes} \left([\![{}^{pp}if\ (a)\ c\ else\ {}^{pp}skip]\!]^\sharp \varrho^\sharp \right)$$

Therefore, we can define the abstract semantics of loops as the least fixpoint of $F^\sharp$:

$$F^\sharp \triangleq \lambda\varrho^\sharp.\varrho_0^\sharp \dot{\cup}_\otimes \left([\![^{pp}if\ (a)\ c\ else\ ^{pp}skip]\!]^\sharp \varrho^\sharp\right)$$
$$[\![^{pp}while(a)\ c]\!]^\sharp \varrho_0^\sharp \triangleq \mathrm{lfp}^{\dot{\subseteq}_\otimes} F^\sharp \tag{6.2}$$

In practice, the least fixpoint of $F^\sharp$ can be computed iteratively [Tar55, CC79a] by defining a sequence $(x_n)_{n\geq 0}$ as follows:

$$x_0 = \varrho_0^\sharp$$
$$x_{n+1} = \varrho_0^\sharp \dot{\cup}_\otimes \left([\![^{pp}if\ (a)\ c\ else\ ^{pp}skip]\!]^\sharp x_n\right)$$

Tarski's theorem [Tar55] guarantees the existence of the least fixpoint of $F^\sharp$ since $F^\sharp$ is monotonic and the lattice of abstract environments is complete. Additionally, the least fixpoint of $F^\sharp$ is also equal to the limit x_∞ of the sequence $(x_n)_{n\geq 0}$.

A modified semantics for loop instructions. Our definition of the abstract semantics of loop instructions in Figure 6.4 is slightly different from the one in Equation (6.2), although both resulting abstract environments are equal. This modification will shortly prove useful to quantify information leaks for programs with intermediate outputs in Chapter 7.

The definition we retain for the abstract semantics of loop instructions is presented in Equation (6.3).

$$\begin{aligned}[\![^{pp}while\ (a)\ c]\!]^\sharp \varrho_0^\sharp \triangleq \mathrm{let}\ \varrho_2^\sharp = \mathrm{lfp}^{\dot{\subseteq}_\otimes}_{\varrho_0^\sharp} [\![c]\!]^\sharp\ \mathrm{in} \\ \mathrm{let}\ n = \mathbb{A}^\sharp[\![a]\!]\varrho_2^\sharp\ \mathrm{in} \\ \begin{cases} \varrho_2^\sharp & \text{if } n = 1 \\ \lambda id.\, \mathrm{top}_{(c)}(\varrho_2^\sharp(id)) & \text{otherwise} \end{cases}\end{aligned} \tag{6.3}$$

Note that this definition computes the least fixpoint of $[\![c]\!]^\sharp$ that is $\dot{\subseteq}_\otimes$-greater than the initial abstract environment $\varrho_0^\sharp$, instead of the least fixpoint of $F^\sharp$ as introduced in Equation (6.2). This is the main difference between both definitions:

1. the least fixpoint of $F^\sharp$ in Equation (6.2) over-approximates the cardinal number of values variables may take *at the loop entry*, whereas
2. the least fixpoint of $[\![c]\!]^\sharp$ in Equation (6.3) over-approximates the cardinal number of values variables may take *at the loop entry for each iteration*.

Note that for the particular case where the loop guard depends only on *Low* inputs, the least fixpoint of both $F^\sharp$ and $[\![c]\!]^\sharp$ coincide, since in this case we know that:

$$[\![^{pp}if\ (a)\ c\ else\ ^{pp}skip]\!]^\sharp\varrho^\sharp = \varrho^\sharp \dot{\cup}_\otimes [\![c]\!]^\sharp\varrho^\sharp$$

Intuitively, if the loop guard depends only on *Low* inputs – the loop guard has only one possible value at each iteration –, then the loop terminates after a fixed number of iterations that depends only on *Low* inputs. Therefore, the maximum cardinal number of values variables may take at each iteration of the loop is equal to the maximum cardinal number of values variables may take at the exit of the loop.This remark justifies the soundness of the first case in Equation (6.3) where the loop guard depends only on *Low* inputs ($n = 1$).

In the case where the loop guard may depend on $High$ inputs, the loop may exit after a number of iterations that may depend on $High$ inputs. Therefore, the cardinal number of values variables may take at each iteration as computed by the least fixpoint of $[\![c]\!]^\sharp$ under-approximates the cardinal number of values variables may take at the exit of the loop. Therefore, the modified abstract semantics of loops in Equation (6.3) concludes that all variables that may be modified in the loop body may have 2^κ possible values – the maximum number of values variables may take –, in order to guarantee soundness. Interestingly, the least fixpoint of $F^\sharp$ also concludes that all variables that may be modified in the loop body have at most 2^κ values in the case where the loop guard may depend on $High$ inputs.

As a consequence, the resulting abstract environment computed in Equation (6.2) is equal to the one computed in Equation (6.3). However, the intermediate abstract environments computed for instructions of the loop body are different. For now, since we are interested in over-approximating min-capacity for batch-job programs, our modification of the abstract semantics for loops is irrelevant since we are only interested in the final abstract environment that is computed at the end of the program. Yet, our modification will shorty prove useful in order to retain precision in the case of programs that may output intermediate steps of computation.

Consider for instance the program in both Listings 6.11 and 6.12 where only variable `secret` is a $High$ input. Let us also assume that attackers may observe variable `i` at the end of execution. Listing 6.11 is annotated with the results of the cardinal abstraction using the standard definition of the abstract semantics for loops in Equation (6.2), whereas Listing 6.12 illustrates the results of the cardinal abstraction using the modified definition in Equation (6.3). These annotations omit the abstract value of variable

`secret` past the initial program point pp_0 since the program does not modify it.

```
0 // [i ↦ ({pp0}, 1); secret ↦ ({pp0}, 2^κ)]
1 while (i ≤ secret) { // [i ↦ ({pp0, pp2}, 2^κ); ...]
2     i := i+1; // [i ↦ ({pp0, pp2}, 2^κ); ...]
3 }
4 // [i ↦ ({pp0, pp2}, 2^κ); ...]
```

Listing 6.11: An example analysis using the abstract semantics for loops in Equation (6.2)

```
0 // [i ↦ ({pp0}, 1); secret ↦ ({pp0}, 2^κ)]
1 while (i ≤ secret) { // [i ↦ ({pp0, pp2}, 1); ...]
2     i := i+1; [i ↦ ({pp0, pp2}, 1); ...]
3 }
4 // [i ↦ ({pp0, pp2}, 2^κ); ...]
```

Listing 6.12: An example analysis using the modified abstract semantics for loops in Equation (6.3)

In the case of the standard definition in Listing 6.11, at each iteration of the least fixpoint computation, the cardinal abstraction will always use the operator $\cup_{add}$ in order to add the current cardinal of variable `i` to the one computed at the previous iteration, since the loop guard evaluates to 2 – it depends on variable `secret`. Therefore, the least fixpoint is reached when the cardinal number of variable `i` reaches the value 2^κ.

However, in the case of the modified definition in Listing 6.12, at each iteration of the least fixpoint computation, the cardinal abstraction will always use the join operator $\cup_\otimes$ to compute the maximum of the current cardinal number of variable `i` and the one computed at the previous iteration, irrespective of the evaluation result of the loop guard. Therefore, the cardinal number of variable `i` will always be 1, which corresponds to the maximum number of values at each iteration. Yet, the cardinal abstraction still guarantees soundness, by concluding that the cardinal number of values variable `i` may take after the loop is 2^κ, through the operator *top*, since the loop guard evaluates to 2 in the computed least fixpoint.

6.2.4 Over-approximating Min-capacity

The cardinal abstract domain computes an over-approximation of the number of values variables may take, when attackers provide a *Low* input memory. Hence, we can upper-bound the cardinal number of outputs attackers may observe when providing a *Low* input memory, in order to obtain an over-approximation of capacity.

Equation (6.1) characterizes the observations attackers make for batch-job programs $\mathtt{c}$ after providing a *Low* input memory as follows :

$$\begin{aligned} &Leaves(\mathcal{T}_b^{\varrho_{L_0}}) = \\ &\quad \{\varrho(v_1) \cdot \varrho(v_2) \ldots \cdot \varrho(v_f) \mid \varrho_{H_0} \in \Sigma_{H_0}, \varrho_0 \triangleq (\varrho_{L_0}, \varrho_{H_0}) \text{ and } \langle c, \varrho_0 \rangle \rightarrow_{\Downarrow}^* \varrho\} \end{aligned}$$

Let us denote by R the set of sets of reachable states for each fixed *Low* input memory, at the end of a batch-job program $\mathtt{c}$:

$$R \triangleq \left\{ \{\varrho \mid \exists \varrho_{H_0} \in \Sigma_{H_0}, \varrho_0 \triangleq (\varrho_{L_0}, \varrho_{H_0}) \text{ and } \langle c, \varrho_0 \rangle \rightarrow_{\Downarrow}^* \varrho\} \mid \varrho_{L_0} \in \Sigma_{L_0} \right\}$$

Then, we can prove that the maximum cardinal number of outputs attackers may observe, when providing a *Low* input memory, can be over-approximated by relying on the cardinal abstraction. Indeed, assuming that $\varrho_{\Downarrow}^\sharp$ is the abstract environment computed at the end of the batch-job program $\mathtt{c}$, the cardinal number of outputs attackers may observe is given by the product of cardinals computed by $\varrho_{\Downarrow}^\sharp$, over all *Low* observable variables $\mathtt{v_1}, \mathtt{v_2}, \ldots \mathtt{v_f}$:

$$\begin{aligned} \max_{\varrho_{L_0} \in \Sigma_{L_0}} \left| Leaves(\mathcal{T}_b^{\varrho_{L_0}}) \right| &= \max_{r \in R} |\{\varrho(v_1) \cdot \varrho(v_2) \ldots \cdot \varrho(v_f) \mid \varrho \in r\}| \\ &\leq \text{(by loosing relations in } r\text{)} \\ &\quad \max_{r \in R} \prod_{1 \leq i \leq f} |\{\varrho(v_i) \mid \varrho \in r\}| \\ &\leq \text{(by loosing relations in } R\text{)} \\ &\quad \prod_{1 \leq i \leq f} \max_{r \in R} |\{\varrho(v_i) \mid \varrho \in r\}| \\ &\leq \text{(by soundness of the cardinal abstraction)} \\ &\quad \prod_{1 \leq i \leq f} proj_2(\varrho_{\Downarrow}^\sharp(v_i)) \end{aligned}$$

The latter proof yields Theorem 5 which provides an over-approximation of min-capacity for batch-job programs by relying on the cardinal abstract domain.

Theorem 5 (Over-approximation of min-capacity $\mathcal{ML}$ for batch-job programs)**.**
For deterministic batch-job programs, min-capacity $\mathcal{ML}$ is upper-bounded by:

$$\mathcal{ML} \leq \log_2 \left(\prod_{1 \leq i \leq f} proj_2(\varrho^{\sharp}_{\Downarrow}(v_i)) \right)$$

where $\varrho^{\sharp}_{\Downarrow}$ denotes the abstract environment computed at the end of the program, and $v_1, v_2, \ldots v_f$ denote the Low variables attackers are allowed to observe.

Note that the proof of Theorem 5 relies on the assumption that the cardinal abstract domain is sound. Interestingly, this same proof sets the stage for the proof of soundness of the cardinal abstraction:

1. define a semantics over a set R of sets of reachable states for each fixed *Low* input memory, then

2. define an abstraction that looses relations between variables in the sets $r \in R$

3. define an abstraction computing the cardinal number of values.

Therefore, the next section will sketch a soundness proof of the cardinal abstract domain.

6.3 Soundness

Abstract interpretation generally requires defining a standard semantics for the considered language. This standard semantics describes a set of transition rules over environments. Then, abstract interpretation focuses on defining a collecting semantics [CC77] describing the details that are relevant to the properties of interest. These properties are in general not computable. Therefore, abstract interpretation frameworks introduce approximate representations aimed at approximating the properties of interest. Finally, by linking the properties of interest to the approximate representations through a Galois connection, a sound abstract semantics can be systematically [Cou99] derived in order to tractably compute approximations of the properties of interest.

6.3.1 Standard Semantics

Let us provide a semantics for the `While` language introduced in Figure 6.2. Figure 6.5 introduces a small step operational semantics for this language.

$$\text{(SKIP)}\ \langle {}^{pp}skip, \varrho\rangle \to \varrho \qquad \text{(ASS)}\ \frac{\varrho' = \varrho[id \mapsto (pp, \mathbb{A}[\![a]\!]\varrho)]}{\langle {}^{pp}id := a, \varrho\rangle \to \varrho'}$$

$$\text{(SEQ)}\ \frac{\langle c_1, \varrho\rangle \to \langle c_1', \varrho'\rangle}{\langle c_1;\ c_2, \varrho\rangle \to \langle c_1';\ c_2, \varrho'\rangle} \qquad \text{(SEQ')}\ \frac{\langle c_1, \varrho\rangle \to \varrho'}{\langle c_1;\ c_2, \varrho\rangle \to \langle c_2, \varrho'\rangle}$$

$$\text{(IFT)}\ \frac{\mathbb{A}[\![a]\!]\varrho = v \qquad \text{istrue}(v)}{\langle {}^{pp}if\ (a)\ c_1\ else\ c_2, \varrho\rangle \to \langle c_1, \varrho\rangle}$$

$$\text{(IFF)}\ \frac{\mathbb{A}[\![a]\!]\varrho = v \qquad \text{isfalse}(v)}{\langle {}^{pp}if\ (a)\ c_1\ else\ c_2, \varrho\rangle \to \langle c_2, \varrho\rangle}$$

$$\text{(W)}\ \langle {}^{pp}while\ (a)\ c, \varrho\rangle \to \langle {}^{pp}if\ (a)\ (c;\ {}^{pp}while\ (a)\ c)\ else\ {}^{pp}skip, \varrho\rangle$$

Figure 6.5: The operational small step semantics of the language in Figure 6.2

Since the cardinal abstract domain tracks the set of program points where variables may have been last assigned, we also instrument the semantics rule for assignments (ASS) in order to track these program points as well. Therefore, environments $\varrho \in Var \rightharpoonup \mathbb{P} \times \mathbb{V}$ map variables `id` to a pair of a program point $pp \in \mathbb{P}$ and an unsigned integer $v \in \mathbb{V}$. A configuration $\langle c, \varrho\rangle$ transitions to either an intermediate configuration $\langle c', \varrho'\rangle$ or a final configuration ϱ'. Additionally, we also denote by $\mathbb{A}[\![a]\!]\varrho$ the evaluation of an expression `a` in an environment ϱ, that yields a value $v \in \mathbb{V}$.

6.3.2 Collecting Semantics

The choice of a collecting semantics depends on the problem of interest. Indeed, a collecting semantics must at least describe program behaviours that are relevant to the studied problem. Ideally, the collecting semantics should also abstract away from the details that are not relevant to the studied problem. Therefore, how should we define the collecting semantics to prove the soundness of the cardinal abstraction?

To answer the latter question intuitively, let us assume an abstract memory $m^\sharp \triangleq \{x \mapsto 2\}$ determining that a variable `x` may take at most 2 different values. What would be the concrete memories $m \in Var \rightharpoonup \mathbb{V}$ that

are represented by $m^\sharp$? Since variable x may take at most 2 different values, the possible values x may take are given by a set of sets of values $v \in \mathbb{V}$:

$$\{V \in \mathcal{P}(\mathbb{V}) : |V| \leq 2\} \in \mathcal{P}(\mathcal{P}(\mathbb{V}))$$

Therefore, the concrete memories represented by $m^\sharp$ are also given by a set of sets of memories $m \in Var \rightharpoonup \mathbb{V}$:

$$\left\{r \in \mathcal{P}(Var \rightharpoonup \mathbb{V}) : |\{m(x) : m \in r\}| \leq 2\right\} \in \mathcal{P}(\mathcal{P}(Var \rightharpoonup \mathbb{V}))$$

Intuitively, in each set r of environments, the memories $m \in r$ map at most 2 different values to variable x. Thus, we are going to define the collecting semantics over a set of sets of environments. Interestingly, if we consider a program c that accepts both *Low* and *High* inputs, such a collecting semantics also enables us to describe the reachable final states of all the sub-programs that are induced by a *Low* input, such as those illustrated by Figure 6.1.

Such structures implying an additional level of sets over environments naturally arise when dealing with security policies such as qualitative and quantitative information flow. Indeed, Clarkson and Schneider [CS08, CS10] note that important classes of security policies are best described by *hyperproperties* which they define as sets of legal sets of states – legal wrt. a security policy –, in contrast to *properties* that are defined as sets of legal states. Additionally, unlike qualitative information flow policies such as non-interference that can be reduced to a safety property through self-composition [BDR04, BDR11], Yasuoka and Terauchi [YT10a, YT10b, YT11] prove that quantitative information flow policies cannot be reduced to a safety property.

Forward collecting semantics of commands. Let us denote by $\rightarrow^*$ the transitive closure of the small step transition relation $\rightarrow$. Therefore, we write $\langle c, \varrho\rangle \rightarrow^* \varrho'$ to denote the existence of a sequence of intermediate configurations $\langle c, \varrho\rangle \rightarrow \langle c_1, \varrho_1\rangle \rightarrow \ldots \rightarrow \langle c_n, \varrho_n\rangle$ that eventually yields a final configuration $\langle c_n, \varrho_n\rangle \rightarrow \varrho'$.

Equation (6.4) defines the forward collecting semantics for commands. This collecting semantics computes the set R' of sets of final reachable environments when a command c is executed over a set R of sets of initial environments.

$$\begin{aligned} &[\![c]\!]_c \in \mathcal{P}(\mathcal{P}(Var \rightharpoonup \mathbb{P} \times \mathbb{V})) \rightarrow \mathcal{P}(\mathcal{P}(Var \rightharpoonup \mathbb{P} \times \mathbb{V})) \\ &[\![c]\!]_c R \triangleq \{\{\varrho' \mid \exists \varrho \in r, \langle c, \varrho\rangle \rightarrow^* \varrho'\} \mid r \in R\} \end{aligned} \tag{6.4}$$

Forward collecting semantics of expressions. Similarly, Equation (6.5) defines the forward collecting semantics for expressions over a set of sets of environments. This semantics then yields a set of sets of values $v \in \mathbb{V}$.

$$\begin{aligned} \mathbb{A}_c[\![a]\!] &\in \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right) \to \mathcal{P}\left(\mathcal{P}\left(\mathbb{V}\right)\right) \\ \mathbb{A}_c[\![a]\!]R &\triangleq \{\{v \in \mathbb{V} \mid \exists \varrho \in r, \mathbb{A}[\![a]\!]\varrho = v\} \mid r \in R\} \end{aligned} \tag{6.5}$$

Outline. Once we define a collecting semantics for commands, we need to construct a Galois connection that relates the lattice of concrete objects $R \in \mathcal{P}\left(\mathcal{P}\left(Var \mapsto \mathbb{P} \times \mathbb{V}\right)\right)$ to the lattice of abstract ones $\varrho^\sharp \in Var \mapsto D_C^\sharp$. With such a Galois connection, we can derive [Cou99] a sound abstract semantics $[\![c]\!]^\sharp$ for commands as well as a sound abstract semantics for expressions $\mathbb{A}^\sharp[\![a]\!]$.

Let us assume the existence of such a Galois connection $(\dot{\alpha}, \dot{\gamma})$:

$$\begin{aligned} &\left\langle \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right); \subseteq, \varnothing, \mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right), \cup, \cap \right\rangle \\ &\qquad \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} \\ &\left\langle Var \rightharpoonup \mathcal{P}\left(\mathbb{P}\right) \times [0, 2^\kappa]; \dot{\subseteq}_\otimes, \lambda x.(\varnothing, 0), \lambda x.(\mathbb{P}, 2^\kappa), \dot{\cup}_\otimes, \dot{\cap}_\otimes \right\rangle \end{aligned} \tag{6.6}$$

Then, in order to derive a sound abstract semantics for commands, we can consider the functional abstraction $\alpha^{\triangleright}_{com}$ defined in Equation (6.7), in order to transpose functions $[\![c]\!]_c$ of the concrete collecting semantics to functions $[\![c]\!]^\sharp$ of the abstract semantics. Note that Equation (6.7) denotes by Env the set $Var \rightharpoonup \mathbb{P} \times \mathbb{V}$ of concrete environments, and by $Env^\sharp$ the set $Var \rightharpoonup D_C^\sharp$ of abstract environments.

$$\begin{aligned} \alpha^{\triangleright}_{com} &\in \Big(\mathcal{P}\left(\mathcal{P}\left(Env\right)\right) \to \mathcal{P}\left(\mathcal{P}\left(Env\right)\right)\Big) \to \Big(Env^\sharp \to Env^\sharp\Big) \\ \alpha^{\triangleright}_{com}([\![c]\!]_c) &\triangleq \dot{\alpha} \circ [\![c]\!]_c \circ \dot{\gamma} \end{aligned} \tag{6.7}$$

Finally, note that $\alpha^{\triangleright}_{com}([\![c]\!]_c)$ provides the best abstraction of the collecting semantics of commands. However, this abstraction might not be computable in general. Therefore, soundness only requires that the abstract semantics $[\![c]\!]^\sharp$ of commands over-approximates the functional abstraction $\alpha^{\triangleright}_{com}([\![c]\!]_c)$ of the collecting semantics:

$$\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp \mathrel{\dot{\subseteq}_\otimes} [\![c]\!]^\sharp\varrho^\sharp$$

Similarly, we will also construct a functional abstraction $\alpha^{\triangleright}_{exp}$ in order to derive a sound abstract semantics for expressions. The functional abstraction

$\alpha^{\triangleright}_{exp}$ transposes functions $\mathbb{A}_c[\![a]\!]$ of the concrete collecting semantics of expressions to functions $\mathbb{A}^\sharp[\![a]\!]$ of the abstract semantics of expressions:

$$\alpha^{\triangleright}_{exp} \in \Big(\mathcal{P}\left(\mathcal{P}\left(Env\right)\right) \to \mathcal{P}\left(\mathcal{P}\left(\mathbb{V}\right)\right)\Big) \to \Big(Env^\sharp \to [0, 2^\kappa]\Big)$$

Therefore, the abstract semantics of expressions $\mathbb{A}^\sharp[\![a]\!]$ is sound if it verifies the following condition:

$$\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp \leq \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp$$

In the next subsections, we will therefore build the Galois connection $(\dot{\alpha}, \dot{\gamma})$ assumed in Equation (6.6). Figure 6.6 summarizes the steps towards building such an abstraction.

6.3.3 Non-relational Abstraction of Environments

The cardinal abstraction is a non-relational abstraction. Thus, we start by defining a first Galois connection that ignores the relationships among variables.

In the case of environment properties, the following abstraction function [Cou99] ignores relations between variables:

$$\begin{aligned} @ &\in \mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right) \to \left(Var \rightharpoonup \mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right) \\ @(r) &\triangleq \lambda x.\{\varrho(x) \mid \varrho \in r\} \end{aligned}$$

Since our collecting semantics is defined over hyperproperties – sets of environment properties –, we lift the previous abstraction @ over sets of properties through an element-wise abstraction [MJ08] denoted by $\alpha_@$:

$$\begin{aligned} \alpha_@(R) &\triangleq \{@(r) \mid r \in R\} \\ &= \{\lambda x.\{\varrho(x) \mid \varrho \in r\} \mid r \in R\} \\ \gamma_@(Q) &\triangleq \{r \mid @(r) \in Q\} \end{aligned}$$

Therefore, we obtain a Galois connection:

$$\begin{gathered} \left\langle \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right); \subseteq, \varnothing, \mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right), \cup, \cap \right\rangle \\ \underset{\alpha_@}{\overset{\gamma_@}{\leftrightarrows}} \\ \left\langle \mathcal{P}\left(Var \rightharpoonup \mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right); \subseteq, \varnothing, Var \rightharpoonup \mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right), \cup, \cap \right\rangle \end{gathered}$$

The abstraction $\alpha_@$ only forgets relationships among variable in each set $r \in R$ of environments. For instance, let us assume a set R_0 of sets of

$$\begin{array}{l} \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right) \\ \qquad \alpha_{@} \downarrow \uparrow \gamma_{@} \quad \text{Non-relational element-wise abstraction} \\ \mathcal{P}\left(Var \rightharpoonup \mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right) \\ \qquad \alpha^{\varpi} \downarrow \uparrow \gamma^{\varpi} \quad \text{Non-relational abstraction} \\ Var \rightharpoonup \mathcal{P}\left(\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right) \end{array}$$

$$\left\langle \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right); \subseteq \right\rangle \underset{\alpha_r \triangleq \alpha^{\varpi} \circ \alpha_{@}}{\overset{\gamma_r \triangleq \gamma_{@} \circ \gamma^{\varpi}}{\leftrightarrows}} \left\langle Var \rightharpoonup \mathcal{P}\left(\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right); \dot{\subseteq} \right\rangle$$

(a) Non-relational abstraction of environments

$$\begin{array}{l} \mathcal{P}\left(\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right) \\ \qquad \alpha_{\times} \circ \alpha_{@_{\times}} \downarrow \uparrow \gamma_{@_{\times}} \circ \gamma_{\times} \quad \text{Attribute independent abstraction} \\ \mathcal{P}\left(\mathcal{P}\left(\mathbb{P}\right)\right) \times \mathcal{P}\left(\mathcal{P}\left(\mathbb{V}\right)\right) \\ \qquad \alpha_{\otimes} \downarrow \uparrow \gamma_{\otimes} \quad \text{Component-wise abstraction} \\ \mathcal{P}\left(\mathbb{P}\right) \times [0, 2^{\kappa}] \end{array}$$

$$\left\langle \mathcal{P}\left(\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right); \subseteq \right\rangle \underset{\alpha \triangleq \alpha_{\otimes} \circ \alpha_{\times} \circ \alpha_{@_{\times}}}{\overset{\gamma \triangleq \gamma_{@_{\times}} \circ \gamma_{\times} \circ \gamma_{\otimes}}{\leftrightarrows}} \left\langle \mathcal{P}\left(\mathbb{P}\right) \times [0, 2^{\kappa}]; \subseteq_{\otimes} \right\rangle$$

(b) Abstraction of values

$$\begin{array}{l} Var \rightharpoonup \mathcal{P}\left(\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right) \\ \qquad \alpha_{c} \downarrow \uparrow \gamma_{c} \quad \text{Pointwise abstraction} \\ Var \rightharpoonup \mathcal{P}\left(\mathbb{P}\right) \times [0, 2^{\kappa}] \end{array}$$

$$(\alpha_c, \gamma_c) \triangleq \left(\lambda f.\lambda x.\alpha(f(x)), \lambda f^{\sharp}.\lambda x.\gamma(f^{\sharp}(x))\right)$$

$$\left\langle \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right); \subseteq \right\rangle \underset{\dot{\alpha} \triangleq \alpha_c \circ \alpha_r}{\overset{\dot{\gamma} \triangleq \gamma_r \circ \gamma_c}{\leftrightarrows}} \left\langle Var \rightharpoonup \mathcal{P}\left(\mathbb{P}\right) \times [0, 2^{\kappa}]; \dot{\subseteq}_{\otimes} \right\rangle$$

(c) Abstraction of environments

Figure 6.6: Building the Galois connection $(\dot{\alpha}, \dot{\gamma})$

environments defined as:

$$R_0 \triangleq \Big\{ \quad \{[x \mapsto (pp_0, 0); y \mapsto (pp_0, 2)], [x \mapsto (pp_0, 0); y \mapsto (pp_0, 3)]\}, \quad (6.8)$$
$$\{[x \mapsto (pp_0, 1); y \mapsto (pp_0, 4)], [x \mapsto (pp_0, 1); y \mapsto (pp_0, 5)]\} \quad \Big\}$$

Then, the abstraction $\alpha_{@}(R_0)$ yields:

$$\alpha_{@}(R_0) = \Big\{ \qquad [x \mapsto \{(pp_0, 0)\}; y \mapsto \{(pp_0, 2), (pp_0, 3)\}],$$
$$[x \mapsto \{(pp_0, 1)\}; y \mapsto \{(pp_0, 4), (pp_0, 5)\}] \quad \Big\}$$

In this latter set, we can further ignore relationships among variables through an additional non-relational abstraction:

$$\alpha^{\varpi}(P) \triangleq \lambda x.\{p(x) \mid p \in P\}$$
$$\gamma^{\varpi}(Q) \triangleq \{p \mid \alpha^{\varpi}(p) \in Q\}$$

Therefore, $(\alpha^{\varpi}, \gamma^{\varpi})$ yields a Galois connection [CC94]:

$$\langle \mathcal{P}\left(Var \rightharpoonup \mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right); \subseteq, \varnothing, Var \rightharpoonup \mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right), \cup, \cap \rangle$$
$$\underset{\alpha^{\varpi}}{\overset{\gamma^{\varpi}}{\leftrightarrows}}$$
$$\langle Var \rightharpoonup \mathcal{P}\left(\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right); \dot{\subseteq}, \lambda x.\varnothing, \lambda x.\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right), \dot{\cup}, \dot{\cap} \rangle$$

Since composing Galois connections yields a Galois connection, we define $(\alpha_r, \gamma_r) \triangleq (\alpha^{\varpi} \circ \alpha_{@}, \gamma^{\varpi} \circ \gamma_{@})$ to obtain a non-relational abstraction over hyperproperties:

$$\langle \mathcal{P}\left(\mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right)\right); \subseteq, \varnothing, \mathcal{P}\left(Var \rightharpoonup \mathbb{P} \times \mathbb{V}\right), \cup, \cap \rangle$$
$$\underset{\alpha_r \triangleq \alpha^{\varpi} \circ \alpha_{@}}{\overset{\gamma_r \triangleq \gamma_{@} \circ \gamma^{\varpi}}{\leftrightarrows}}$$
$$\langle Var \rightharpoonup \mathcal{P}\left(\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right)\right); \dot{\subseteq}, \lambda x.\varnothing, \lambda x.\mathcal{P}\left(\mathbb{P} \times \mathbb{V}\right), \dot{\cup}, \dot{\cap} \rangle \quad (6.9)$$

If we recall the example set R_0 of sets of environments defined in Equation (6.8), then $\alpha_r(R_0)$ ignores all relationships among variables:

$$\alpha_r(R_0) = \Big[x \mapsto \Big\{\{(pp_0, 0)\}, \{(pp_0, 1)\}\Big\};$$
$$y \mapsto \Big\{\{(pp_0, 2), (pp_0, 3)\}, \{(pp_0, 4), (pp_0, 5)\}\Big\}\Big]$$

As we have defined a non-relational abstraction (α_r, γ_r) over hyperproperties, we can now focus on defining an abstraction over sets of sets of values as shown in Figure 6.6(b). This latter abstraction can then be lifted over environments as illustrated by Figure 6.6(c).

6.3.4 Abstraction of Values

Similarly to the previous section, where we lift a non-relational abstraction to a set of sets of environments, we can also lift an *attribute independent abstraction* [Cou99], that forgets about relationships between components of pairs, to a set of sets of pairs through an element-wise abstraction.

Attribute independent abstraction. The attribute independent abstraction function $@_\times$ is given by:

$$\begin{aligned} @_\times &\in \mathcal{P}(\mathbb{P} \times \mathbb{V}) \to \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{V}) \\ @_\times(p) &\triangleq (\Pi_1(p), \Pi_2(p)) \\ \Pi_i(p) &\triangleq \{proj_i(x) \mid x \in p\} \end{aligned}$$

Therefore, we define an element-wise abstraction $\alpha_{@_\times}$ over a set of sets of pairs as follows:

$$\begin{aligned} \alpha_{@_\times}(P) &\triangleq \{@_\times(p) \mid p \in P\} \\ &= \{(\Pi_1(p), \Pi_2(p)) \mid p \in P\} \\ \gamma_{@_\times}(Q) &\triangleq \{p \mid @_\times(p) \in Q\} \end{aligned}$$

Therefore, $(\alpha_{@_\times}, \gamma_{@_\times})$ defines a Galois connection:

$$\begin{array}{c} \langle \mathcal{P}(\mathcal{P}(\mathbb{P} \times \mathbb{V})); \subseteq, \varnothing, \mathcal{P}(\mathbb{P} \times \mathbb{V}), \cup, \cap \rangle \\ \xleftarrow[\alpha_{@_\times}]{\gamma_{@_\times}} \\ \langle \mathcal{P}(\mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{V})); \subseteq, \varnothing, \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{V}), \cup, \cap \rangle \end{array} \quad (6.10)$$

Let us assume that P_0 is a set of sets of pairs defined as follows:

$$P_0 \triangleq \Big\{\{(pp_0, 2), (pp_0, 3)\}, \{(pp_0, 4), (pp_0, 5)\}\Big\} \quad (6.11)$$

Then $\alpha_{@_\times}(P_0)$ is given by:

$$\alpha_{@_\times}(P_0) = \Big\{(\{pp_0\}, \{2, 3\}), (\{pp_0\}, \{4, 5\})\Big\}$$

In this latter set, we can further ignore relationships among values through an additional attribute independent abstraction:

$$\begin{aligned} \alpha_\times(Q) &\triangleq (\Pi_1(Q), \Pi_2(Q)) \\ \gamma_\times((X, Y)) &\triangleq X \times Y \end{aligned}$$

Therefore, we obtain a Galois connection [Cou99]:

$$\begin{array}{c}\langle \mathcal{P}\left(\mathcal{P}\left(\mathbb{P}\right)\times\mathcal{P}\left(\mathbb{V}\right)\right);\subseteq,\varnothing,\mathcal{P}\left(\mathbb{P}\right)\times\mathcal{P}\left(\mathbb{V}\right),\cup,\cap\rangle \\ \xleftarrow[\alpha_\times]{\gamma_\times} \\ \langle \mathcal{P}\left(\mathcal{P}\left(\mathbb{P}\right)\right)\times\mathcal{P}\left(\mathcal{P}\left(\mathbb{V}\right)\right);\subseteq_\times,(\varnothing,\varnothing),\mathcal{P}\left(\mathbb{P}\right)\times\mathcal{P}\left(\mathbb{V}\right),\cup_\times,\cap_\times\rangle \end{array} \quad (6.12)$$

with the component-wise ordering, join and meet:

$$\begin{aligned}\subseteq_\times &\triangleq \subseteq\times\subseteq \\ \cup_\times &\triangleq \cup\times\cup \\ \cap_\times &\triangleq \cap\times\cap\end{aligned}$$

For instance, if we recall the example set P_0 defined in Equation (6.11), then $\alpha_\times\circ\alpha_{@_\times}(P_0)$ is given by:

$$\begin{aligned}\alpha_\times\circ\alpha_{@_\times}(P_0) &= \alpha_\times\left(\left\{(\{pp_0\},\{2,3\}),(\{pp_0\},\{4,5\})\right\}\right) \\ &= \left(\left\{\{pp_0\}\right\},\left\{\{2,3\},\{4,5\}\right\}\right)\end{aligned}$$

Component-wise abstraction. Finally, let us assume two Galois connections (α_v,γ_v) and $(\alpha_{pp},\gamma_{pp})$ such that:

$$\begin{aligned}\langle \mathcal{P}\left(\mathcal{P}\left(\mathbb{V}\right)\right);\subseteq,\varnothing,\mathcal{P}\left(\mathbb{V}\right),\cup,\cap\rangle &\xleftarrow[\alpha_v]{\gamma_v} \langle[0,2^\kappa];\leq,0,2^\kappa,\max,\min\rangle \\ \langle \mathcal{P}\left(\mathcal{P}\left(\mathbb{P}\right)\right);\subseteq,\varnothing,\mathcal{P}\left(\mathbb{P}\right),\cup,\cap\rangle &\xleftarrow[\alpha_{pp}]{\gamma_{pp}} \langle \mathcal{P}\left(\mathbb{P}\right);\subseteq,\varnothing,\mathbb{P},\cup,\cap\rangle\end{aligned}$$

Then, a component-wise abstraction [MJ08] $(\alpha_\otimes,\gamma_\otimes)$ defined as follows:

$$\begin{aligned}\alpha_\otimes\Big((S_{pp},S_v)\Big) &\triangleq \Big(\alpha_{pp}(S_{pp}),\alpha_v(S_v)\Big) \\ \gamma_\otimes\Big((s_{pp},n)\Big) &\triangleq \Big(\gamma_{pp}(s_{pp}),\gamma_v(n)\Big)\end{aligned}$$

yields a Galois connection:

$$\begin{array}{c}\langle \mathcal{P}\left(\mathcal{P}\left(\mathbb{P}\right)\right)\times\mathcal{P}\left(\mathcal{P}\left(\mathbb{V}\right)\right);\subseteq_\times,(\varnothing,\varnothing),\mathcal{P}\left(\mathbb{P}\right)\times\mathcal{P}\left(\mathbb{V}\right),\cup_\times,\cap_\times\rangle \\ \xleftarrow[\alpha_\otimes]{\gamma_\otimes} \\ \langle \mathcal{P}\left(\mathbb{P}\right)\times[0,2^\kappa];\subseteq_\otimes,(\varnothing\times 0),(\mathbb{P}\times 2^\kappa),\cup_\otimes,\cap_\otimes\rangle \end{array} \quad (6.13)$$

Consequently, the abstraction of values (α,γ) can be defined as the composition of the 3 Galois connections defined previously in Equations (6.10), (6.12) and (6.13):

$$\langle \mathcal{P}(\mathcal{P}(\mathbb{P}\times\mathbb{V}));\subseteq,\varnothing,\mathcal{P}(\mathbb{P}\times\mathbb{V}),\cup,\cap\rangle \xleftarrow[\alpha\triangleq\alpha_{\otimes}\circ\alpha_{\times}\circ\alpha_{@_{\times}}]{\gamma\triangleq\gamma_{@_{\times}}\circ\gamma_{\times}\circ\gamma_{\otimes}} \langle \mathcal{P}(\mathbb{P})\times[0,2^{\kappa}];\subseteq_{\otimes},(\varnothing,0),(\mathbb{P},2^{\kappa}),\cup_{\otimes},\cap_{\otimes}\rangle \quad (6.14)$$

Let us focus now on defining both Galois connections (α_v,γ_v) and $(\alpha_{pp},\gamma_{pp})$.

Abstraction of program points. The abstraction α_{pp} merges all the sets of program points:

$$\alpha_{pp}(S_{pp}) \triangleq \bigcup_{s\in S_{pp}} s$$
$$\gamma_{pp}(s_{pp}) \triangleq \mathcal{P}(s_{pp})$$

Therefore, we obtain a Galois connection:

$$(\mathcal{P}(\mathcal{P}(\mathbb{P}));\subseteq,\varnothing,\mathcal{P}(\mathbb{P}),\cup,\cap) \xleftarrow[\alpha_{pp}]{\gamma_{pp}} (\mathcal{P}(\mathbb{P});\subseteq,\varnothing,\mathbb{P},\cup,\cap)$$

Indeed, let us prove that $(\alpha_{pp},\gamma_{pp})$ is a Galois connection:

$$\begin{aligned}\alpha_{pp}(S)\subseteq s_{pp} &\iff \bigcup_{s\in S} s\subseteq s_{pp}\\ &\iff \forall s\in S, s\in\mathcal{P}(s_{pp})\\ &\iff S\subseteq\mathcal{P}(s_{pp})\\ &\iff S\subseteq\gamma_{pp}(s_{pp})\square\end{aligned}$$

Abstraction of concrete values. The abstraction α_v computes the maximum cardinal of values over the sets $s_v\in S_v$:

$$\alpha_v(S_v) \triangleq \max_{s_v\in S_v}|s_v|$$
$$\gamma_v(n) = \{V\in\mathcal{P}(\mathbb{V}) \mid |V|\leq n\}$$

Therefore, (α_v,γ_v) is a Galois connection:

$$\langle \mathcal{P}(\mathcal{P}(\mathbb{V}));\subseteq,\varnothing,\mathcal{P}(\mathbb{V}),\cup,\cap\rangle \xleftarrow[\alpha_v]{\gamma_v} \langle[0,2^{\kappa}];\leq,0,2^{\kappa},\max,\min\rangle$$

Indeed, let us prove that (α_v,γ_v) is a Galois connection:

$$\begin{aligned}\alpha_v(S_v)\leq n &\iff \max_{s_v\in S_v}|s_v|\leq n\\ &\iff \forall s_v\in S_v, s_v\in\mathcal{P}_n(\mathbb{V})\\ &\iff S_v\subseteq\gamma_v(n)\square\end{aligned}$$

6.3.5 Abstraction of Environments

As illustrated by Figure 6.6(c), the abstraction (α, γ) of values – defined in Equation (6.14) – can be lifted through a pointwise abstraction [CC94], in order to abstract a set valued environment $\varrho \in Var \rightharpoonup \mathcal{P}(\mathcal{P}(\mathbb{P} \times \mathbb{V}))$.

Indeed, let us define a pointwise abstraction (α_c, γ_c) as follows:

$$\alpha_c \triangleq \lambda f.\lambda x.\alpha\Big(f(x)\Big)$$
$$\gamma_c \triangleq \lambda f^\sharp.\lambda x.\gamma\Big(f^\sharp(x)\Big)$$

Then, (α_c, γ_c) yields a Galois connection:

$$\langle Var \rightharpoonup \mathcal{P}(\mathcal{P}(\mathbb{P} \times \mathbb{V})); \dot{\subseteq}, \lambda x.\varnothing, \lambda x.\mathcal{P}(\mathbb{P} \times \mathbb{V}), \dot{\cup}, \dot{\cap}\rangle$$
$$\underset{\alpha_c}{\overset{\gamma_c}{\leftrightarrows}}$$
$$\langle Var \rightharpoonup \mathcal{P}(\mathbb{P}) \times [0, 2^\kappa]; \dot{\subseteq}_\otimes, \lambda x.(\varnothing, 0), \lambda x.(\mathbb{P}, 2^\kappa), \dot{\cup}_\otimes, \dot{\cap}_\otimes\rangle$$

Therefore, by composing both the non-relational abstraction over environments and the pointwise abstraction of values defined in Equations (6.9) and (6.14), we can build the abstraction $(\dot{\alpha}, \dot{\gamma})$:

$$\dot{\alpha} \triangleq \alpha_c \circ \alpha_r$$
$$\dot{\gamma} \triangleq \gamma_r \circ \gamma_c$$

Thus, the pair $(\dot{\alpha}, \dot{\gamma})$ yields a Galois connection:

$$\langle \mathcal{P}(\mathcal{P}(Var \rightharpoonup \mathbb{P} \times \mathbb{V})); \subseteq, \varnothing, \mathcal{P}(Var \rightharpoonup \mathbb{P} \times \mathbb{V}), \cup, \cap\rangle$$
$$\underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}}$$
$$\langle Var \rightharpoonup \mathcal{P}(\mathbb{P}) \times [0, 2^\kappa]; \dot{\subseteq}_\otimes, \lambda x.(\varnothing, 0), \lambda x.(\mathbb{P}, 2^\kappa), \dot{\cup}_\otimes, \dot{\cap}_\otimes\rangle \quad (6.15)$$

6.3.6 Deriving an Abstract Semantics of Expressions

As mentioned in Section 6.3.2, once we construct a Galois connection relating concrete objects to abstract one, we can define functional abstractions in order to soundly approximate functions over concrete objects by functions over abstract ones.

In order to approximate the collecting semantics $\mathbb{A}_c[\![a]\!]$ of expressions, we can define the following functional abstraction [Cou99]:

$$\alpha^{\triangleright}_{exp} \in \Big(\mathcal{P}(\mathcal{P}(Env)) \to \mathcal{P}(\mathcal{P}(\mathbb{V}))\Big) \mapsto \Big(Env^\sharp \to [0, 2^\kappa]\Big)$$
$$\alpha^{\triangleright}_{exp}(\phi) \triangleq \alpha_v \circ \phi \circ \dot{\gamma}$$

Therefore, the abstract semantics $\mathbb{A}^\sharp[\![a]\!]$ of expressions is sound wrt. the collecting semantics $\mathbb{A}_c[\![a]\!]$ of expressions if:

$$\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp \leq \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp \tag{6.16}$$

Let us derive an abstract semantics for binary arithmetic operators for instance:

$$\begin{aligned}
\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a_1 \ bop \ a_2]\!])\varrho^\sharp &\triangleq \alpha_v\left(\{\{v_1 \ bop \ v_2 \mid \exists \varrho \in r, \mathbb{A}[\![a_1]\!]\varrho = v_1 \right.\\
&\qquad \left. \wedge\, \mathbb{A}[\![a_2]\!]\varrho = v_2\} \mid r \in \dot\gamma(\varrho^\sharp)\right\}\Big)\\
&\leq \text{(by loosing relationships among variables in } r)\\
&\quad \alpha_v\left(\left\{\{v_1 \ bop \ v_2 \mid \exists \varrho_1 \in r_1, \exists \varrho_2 \in r_2, \mathbb{A}[\![a_1]\!]\varrho_1 = v_1 \right.\right.\\
&\qquad \left.\left. \wedge\, \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\} \mid r_1, r_2 \in \dot\gamma(\varrho^\sharp)\right\}\right)\\
&= \text{(by definition of } \alpha_v)\\
&\quad \max_{r_1, r_2 \in \dot\gamma(\varrho^\sharp)} \left|\left\{v_1 \ bop \ v_2 \mid \exists \varrho_1 \in r_1, \mathbb{A}[\![a_1]\!]\varrho_1 = v \right.\right.\\
&\qquad \left.\left. \wedge\, \varrho_2 \in r_2, \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\right\}\right|\\
&\leq \text{(values } v \in \mathbb{V} \text{ are finite, of size } 2^\kappa)\\
&\quad \min\left(2^\kappa, \max_{r_1, r_2 \in \dot\gamma(\varrho^\sharp)} \left|\left\{v_1 \mid \exists \varrho_1 \in r_1, \mathbb{A}[\![a_1]\!]\varrho_1 = v_1\right\}\right| \right.\\
&\qquad \left. \times \left|\left\{v_2 \mid \exists \varrho_2 \in r_2, \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\right\}\right|\right)\\
&= \min\left(2^\kappa, \max_{r_1 \in \dot\gamma(\varrho^\sharp)} \left|\left\{v_1 \mid \exists \varrho_1 \in r_1, \mathbb{A}[\![a_1]\!]\varrho_1 = v_1\right\}\right| \right.\\
&\qquad \left. \times \max_{r_2 \in \dot\gamma(\varrho^\sharp)} \left|\left\{v_2 \mid \exists \varrho_2 \in r_2, \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\right\}\right|\right)\\
&\leq \text{(By induction hypothesis in Equation (6.16))}\\
&\quad \min\left(2^\kappa, \mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp \times \mathbb{A}^\sharp[\![a_2]\!]\varrho^\sharp\right)\\
&\triangleq \mathbb{A}^\sharp[\![a_1 \ bop \ a_2]\!]\varrho^\sharp
\end{aligned}$$

Theorem 6 states that the abstract semantics of expressions is sound. A proof for the remaining cases can be found in Appendix C.1.

Theorem 6 (Soundness of the abstract semantics $\mathbb{A}^\sharp[\![a]\!]$)**.**
The abstract semantics of expressions is sound:

$$\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp \leq \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp$$

6.3.7 Deriving an Abstract Semantics of Instructions

Similarly to expressions, we define the functional abstraction $\alpha^{\triangleright}_{com}$ in order to soundly approximate the collecting semantics $[\![c]\!]_c$ of commands.

$$\alpha^{\triangleright}_{com} \in \Big(\mathcal{P}\left(\mathcal{P}\left(Env\right)\right) \to \mathcal{P}\left(\mathcal{P}\left(Env\right)\right)\Big) \to \Big(Env^\sharp \to Env^\sharp\Big)$$
$$\alpha^{\triangleright}_{com}([\![c]\!]_c) \triangleq \dot{\alpha} \circ [\![c]\!]_c \circ \dot{\gamma}$$

Therefore, the abstract semantics of commands $[\![c]\!]^\sharp$ is sound wrt. the collecting semantics $[\![c]\!]_c$ of commands if:

$$\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp \mathbin{\dot{\subseteq}_\otimes} [\![c]\!]^\sharp\varrho^\sharp \tag{6.17}$$

Let us derive an abstract semantics for conditionals for instance. Let `c` denote a conditional instruction $^{\text{pp}}$`if (a) c`$_1$ `else c`$_0$. Then:

$$\begin{aligned}\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp &= \dot{\alpha}\left(\left\{\{\varrho' \mid \exists\varrho \in r, \langle c,\varrho\rangle \to^* \varrho'\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\\ &= \dot{\alpha}\left(\left\{\{\varrho' : \exists\varrho \in r, \exists v \in \{0,1\}, \mathbb{A}[\![a]\!]\varrho = v\right.\right.\\ &\qquad \left.\left.\wedge \langle c_v,\varrho\rangle \to \varrho'\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\end{aligned}$$

First, if $\mathbb{A}^\sharp[\![a]\!]\varrho^\sharp = 1$, then expression `a` evaluates to at most one value in each set $r \in \dot{\gamma}(\varrho^\sharp)$:

$$\forall r \in \dot{\gamma}(\varrho^\sharp), \exists v \in \{0,1\}, \forall \varrho \in r, \mathbb{A}[\![a]\!]\varrho = v$$

Therefore, the sets $r \in \dot{\gamma}(\varrho^\sharp)$ can be partitioned into sets r_1 (resp. r_0) where expression `a` evaluates to 1 (resp. evaluates to 0):

$$\begin{aligned}\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp &= \dot{\alpha}\left(\left\{\{\varrho' \mid \exists\varrho \in r_1, \langle c_1,\varrho\rangle \to^* \varrho'\} \mid r_1 \in \dot{\gamma}(\varrho^\sharp)\right\}\right.\\ &\qquad \left.\cup\left\{\{\varrho' \mid \exists\varrho \in r_0, \langle c_0,\varrho\rangle \to^* \varrho'\} \mid r_0 \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\\ &\mathbin{\dot{\subseteq}_\otimes} (\dot{\alpha} \text{ preserves joins})\\ &\quad \dot{\alpha}\left(\left\{\{\varrho' \mid \exists\varrho \in r_1, \langle c_1,\varrho\rangle \to^* \varrho'\} \mid r_1 \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\\ &\qquad \dot{\cup}_\otimes\, \dot{\alpha}\left(\left\{\{\varrho' \mid \exists\varrho \in r_1, \langle c_1,\varrho\rangle \to^* \varrho'\} \mid r_1 \in \dot{\gamma}(\varrho^\sharp)\right\}\right)\\ &= \left(\dot{\alpha} \circ [\![c_1]\!]_c \circ \dot{\gamma}(\varrho^\sharp)\right) \dot{\cup}_\otimes \left(\dot{\alpha} \circ [\![c_0]\!]_c \circ \dot{\gamma}(\varrho^\sharp)\right)\\ &\mathbin{\dot{\subseteq}_\otimes} (\text{by induction hypothesis Equation (6.17)})\\ &\quad [\![c_1]\!]^\sharp\varrho^\sharp \,\dot{\cup}_\otimes\, [\![c_0]\!]^\sharp\varrho^\sharp\end{aligned}$$

Second, if $\mathbb{A}^\sharp[\![a]\!]\varrho^\sharp > 1$, then for variables $\mathtt{x}$ that are modified in neither $\mathtt{c}_1$ nor $\mathtt{c}_0$, we have:

$$(\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp)(x) = \left([\![c_1]\!]^\sharp\varrho^\sharp \dot\cup_\otimes [\![c_0]\!]^\sharp\varrho^\sharp\right)(x)$$

Finally, for variables that are modified in either $\mathtt{c}_1$ or $\mathtt{c}_0$:

$$\begin{aligned}
(\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp)(x) &= \dot\alpha\left(\left\{\{\varrho' \mid \exists\varrho\in r, \exists v\in\{0,1\}, \mathbb{A}[\![a]\!]\varrho = v\right.\right.\\
&\qquad \left.\left.\wedge\langle c_v,\varrho\rangle\to\varrho'\} \mid r\in\dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&= \dot\alpha\left(\left\{\{\varrho' \mid \exists\varrho\in r, \mathbb{A}[\![a]\!]\varrho = 1\wedge\langle c_1,\varrho\rangle\to^*\varrho'\}\right.\right.\\
&\qquad \left.\left.\cup\{\varrho' \mid \exists\varrho\in r, \mathbb{A}[\![a]\!]\varrho = 0\wedge\langle c_0,\varrho\rangle\to^*\varrho'\}\right\}\right)(x)\\
&\subseteq_\otimes \dot\alpha\left(\left\{\{\varrho' \mid \exists\varrho\in r_2, \mathbb{A}[\![a]\!]\varrho = 1\wedge\langle c_1,\varrho\rangle\to^*\varrho'\}\right.\right.\\
&\qquad \cup\{\varrho' \mid \exists\varrho\in r_1, \mathbb{A}[\![a]\!]\varrho = 0\wedge\langle c_0,\varrho\rangle\to^*\varrho'\}\\
&\qquad \left.\left.\mid r_1,r_2\in\dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes \dot\alpha\left(\left\{\{\varrho' \mid \varrho'\in r_1'\}\cup\{\varrho' \mid \varrho'\in r_2'\} \mid r_1'\in[\![c_1]\!]_c\circ\dot\gamma(\varrho^\sharp),\right.\right.\\
&\qquad \left.\left.r_2'\in[\![c_0]\!]_c\circ\dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes (\dot\gamma\circ\dot\alpha \text{ is extensive, and } \dot\alpha \text{ is monotone})\\
&\qquad \dot\alpha\left(\left\{\{\varrho' \mid \varrho'\in r_1'\}\cup\{\varrho' \mid \varrho'\in r_2'\}\mid\right.\right.\\
&\qquad\quad r_1'\in\dot\gamma\circ\dot\alpha\circ[\![c_1]\!]_c\circ\dot\gamma(\varrho^\sharp),\\
&\qquad\quad \left.\left.r_2'\in\dot\gamma\circ\dot\alpha\circ[\![c_0]\!]_c\circ\dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes (\text{By hypothesis in Equation (6.17) and monotony of } \dot\alpha)\\
&\qquad \dot\alpha\left(\left\{\{\varrho' \mid \varrho'\in r_1'\}\cup\{\varrho' \mid \varrho'\in r_2'\}\mid\right.\right.\\
&\qquad\quad \left.\left.r_1'\in\dot\gamma([\![c_1]\!]^\sharp\varrho^\sharp), r_2'\in\dot\gamma([\![c_0]\!]^\sharp\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes (\text{By definition of } (\dot\alpha,\dot\gamma))\\
&\qquad \Big(proj_1([\![c_1]\!]^\sharp\varrho^\sharp(x))\cup proj_1([\![c_0]\!]^\sharp\varrho^\sharp(x)),\\
&\qquad\quad proj_2([\![c_1]\!]^\sharp\varrho^\sharp(x)) + proj_2([\![c_0]\!]^\sharp\varrho^\sharp(x))\Big)
\end{aligned}$$

Hence, the abstract semantics of conditionals is sound:

$$\begin{aligned}
[\![^{pp}if\ (a)\ c_1\ else\ c_2]\!]^\sharp\varrho^\sharp \triangleq\ &\text{let } n = \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp \text{ in}\\
&\text{let } \varrho_1^\sharp = [\![c_1]\!]^\sharp\varrho^\sharp \text{ in}\\
&\text{let } \varrho_2^\sharp = [\![c_2]\!]^\sharp\varrho^\sharp \text{ in}\\
&\lambda id.\begin{cases}\varrho_1^\sharp(id)\cup_\otimes\varrho_2^\sharp(id) & \text{if } n=1\\ \varrho_1^\sharp(id)\cup_{add(c_1,c_2)}\varrho_2^\sharp(id) & \text{otherwise}\end{cases}
\end{aligned}$$

Theorem 7 states that the abstract semantics of instructions is sound. A proof for the remaining cases can be found in Appendix C.2.

Theorem 7 (Soundness of the abstract semantics $[\![c]\!]^\sharp$)**.**
The abstract semantics of commands is sound:

$$\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp \dot{\subseteq}_\otimes [\![c]\!]^\sharp\varrho^\sharp$$

6.4 Related Work

The cardinal abstraction quantifies information flow for deterministic programs wrt. min-capacity. It does not rely on an approximation of the set of reachable states. Thus, it computes upper-bounds over min-capacity for programs that accept both *Low* and *High* inputs. Similarly to Clark et al.'s type system [CHM07] which also supports programs with *Low* and *High* inputs, but is aimed at quantifying information leaks wrt. Shannon entropy, the cardinal abstraction is also conservative for information leaks due to loops depending on *High* inputs. Indeed, the cardinal abstraction assumes that variables modified inside a loop, whose guard may depend on *High* inputs, may leak all the secret.

Most existing approaches [KR10, MS11] for quantitative information flow apply for batch-job programs accepting only *High* inputs, since they rely on approximating the set of reachable states in order to approximate min-capacity. As we argue in Section 6.1, approximating min-capacity by relying on an approximation of the set of reachable states is inefficient in the case of programs that accepts *Low* inputs as well. Indeed, to deal with such programs, such an approach requires analysing as many programs as the size of the *Low* inputs. Note that Backes, Köpf and Rybalchenko [BKR09] overcome this limitation by treating both *Low* and *High* inputs symbolically through a model checking approach.

Köpf and Rybalchenko [KR13] propose to rely on self-composition [BDR04, BDR11] in order to analyse the leakage of batch-job programs in a k-try attack scenario, where attackers can provide k different *Low* inputs. However, their approach applies for arbitrary *Low* inputs only if k is as big as the size of the *Low* input space. Our cardinal abstraction overcomes this limitation since it over-approximates the leakage when attackers provide an arbitrary *Low* inputs. We plan on building on the cardinal abstraction in order to propose a relational abstract domain, suited for quantifying information leaks for k-try attacks. Interestingly, such a relational abstract domain will also enhance the precision of the cardinal abstract domain.

Throughout our developments of the cardinal abstract domain, we consider some optimizations in order to enhance the precision of our analysis, while we leave out some others for future work. For instance, the cardinal abstract domain tracks the program points where variables may have been last assigned to, in order to improve its precision in the case of conditional instructions. However, we leave off optimizations such as reducing abstract environments by relying on conditional guards, or combining the cardinal abstraction with numerical abstract domains [Gra89, CC77]. These choices are motivated by making the precision of the cardinal abstraction reach a first checkpoint; indeed, we believe that the cardinal abstraction refines Hunt and Sands flow-sensitive type-system [HS06]. In fact, we believe that abstracting further the cardinal abstract domain, by mapping variables having a cardinal of 1 to *Low* and mapping variables having a cardinal greater than 1 to *High*, yields an analysis that is equivalent to Hunt and Sands' type system. We leave the proof of this conjecture as future work.

6.5 Summary

In this chapter, we propose a static analysis to quantify information flow for batch-job programs. Following the abstract interpretation framework, we develop the cardinal abstraction, an abstract domain aimed at over-approximating the cardinal number of values variables may take. We also prove the soundness of our abstract domain.

Unlike previous quantitative approaches aimed at quantifying information leaks, our approach does not rely on traditional safety analyses relying on approximating the set of reachable states. Therefore, the cardinal abstract domain enables an efficient over-approximation of min-capacity, for programs accepting both *Low* and *High* inputs.

Future work involves improving the cardinal abstraction, by taking advantage of the powerful framework of abstract interpretation. Indeed, the precision of our abstract domain can be improved by relying on conditional guards to reduce abstract environments in conditional branches. Additionally, the cardinal abstract domain can also be improved by combining it with different abstract domains [CCF+07] such as numerical domains [Gra89, CC77] as well as relational ones [CH78, Min06b, Min06a].

We also implemented the cardinal abstraction by modifying an existing abstract interpreter[1] for a `While` language. This implementation enhances the precision of the cardinal abstraction by relying on a reduced prod-

[1] http://www.irisa.fr/celtique/teaching/PAS/while_analyser.tgz

uct [CC79b] of both the cardinal abstract domain and an interval abstract domain [CC76].

The next chapter builds on the cardinal abstraction in order to quantify information leaks for a larger class of programs: those that may also output intermediate steps of computation.

Chapter 7

Tree Abstraction

The cardinal abstraction enables the over-approximation of min-capacity for a restricted class of deterministic programs: batch-job ones that only output the result of their computations at the end of their execution. In this chapter, we propose a novel static analysis technique in order to over-approximate min-capacity for programs that may output intermediate results of computation.

Our analysis, the tree abstraction, builds on the results of the cardinal abstraction, defined in Chapter 6, in order to construct a *regular specification* of attackers' observations. Finally, we rely on the framework of *analytic combinatorics* [FS09] in order to quantify the leakage for polynomial time attackers. We also derive sufficient conditions on the regular specification describing attackers' observations, guaranteeing that analysed programs are secure wrt. relative secrecy (RS).

7.1 Overview

Our static analysis builds a *regular specification* of attackers' observation trees by relying on the results of the cardinal abstraction, defined in Chapter 6. Using this specification, we can estimate the cardinal of leaves for attackers' observations trees, in the case of programs that may output intermediate steps of computation. Therefore, we can also estimate min-capacity by Theorem 4.

Let us for instance consider the program in Listing 7.1, where only variable `secret` is a $High$ input. We annotate this program by the results of the cardinal abstraction. Since this example program modifies neither variable `input` nor variable `secret`, the annotations omit their abstract values past the initial program point pp_0.

```
0  // ϱ♯0 = [x ↦ ({pp0}, 1); public ↦ ({pp0}, 1); secret ↦ ({pp0}, 2^κ)]
1  if (public == secret){
2    x := 1; // ϱ♯2 = [x ↦ ({pp2}, 1); ...]
3  }
4  else {
5    x := 0; // ϱ♯5 = [x ↦ ({pp5}, 1); ...]
6  }
7  output x; // ϱ♯7 = [x ↦ ({pp2, pp5}, 2); ...]
8  while (public > secret){
9    skip;
10 }
11 stop;
```

Listing 7.1: An example program with intermediate outputs

The first output instruction, at Line 5, enables attackers to observe the value of variable x. Thus, attackers may observe at most 2 different values since the cardinal abstraction determines that variable x may take at most 2 different values when attackers provide a *Low* input memory. The loop body at Line 9 outputs nothing. Therefore, if the loop instruction at Line 8 terminates, attackers only observe an empty trace denoted by ϵ. Otherwise, if the loop does not terminate, attackers may deduce that the loop silently diverges, which corresponds to the observation of divergence $\Uparrow$. Finally, at Line 11, the instruction `stop` signals the end of execution, which corresponds to the observation of termination $\Downarrow$.

To sum up, attackers may observe at most 2 different values, followed by the observation of either divergence $\Uparrow$, or an empty trace ϵ followed by the observation of termination $\Downarrow$ if the loop at Line 8 terminates. Note that in general, attackers cannot distinguish a loop that terminates without outputting any value from a loop that silently diverges, until they make the next observation. Indeed, in this case, as soon as attackers observe termination $\Downarrow$ of the program, they deduce that the loop terminates. Otherwise, if attackers wait enough time without making any new observations, they may deduce that the loop indeed diverges. In practice, attackers may also deduce that a program terminates or silently diverges by leveraging knowledge about the usage of system resources for instance.

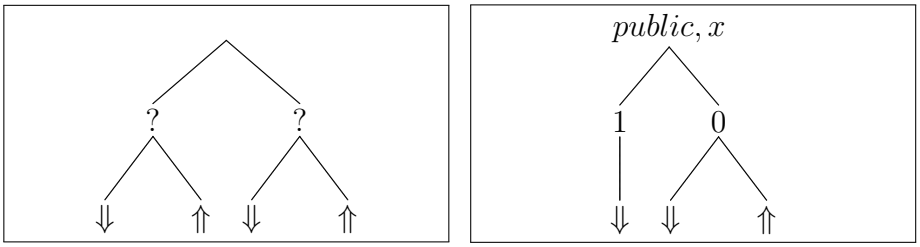
If we want to write a specification for the attackers' observations we determined earlier for the program in Listing 7.1, we may represent this specification as follows:

$$2 \cdot \Big((\epsilon \cdot \Downarrow) \oplus \Uparrow \Big)$$

Indeed, "followed by" is represented by the concatenation $\cdot$, whereas "or" is represented by the union operator $\oplus$. Since the empty trace ϵ corresponds to attackers making no observations, this specification can be further simplified to:

$$2 \cdot (\Downarrow \oplus \Uparrow) \tag{7.1}$$

This regular specification closely resembles a grammar describing a regular language. The empty trace ϵ is the neutral element wrt. concatenation. Additionally, instead of describing how to construct words of a regular language, this rational specification describes how to construct the leaves of an attackers' observation tree. Indeed, according to Equation (7.1), in order to build the leaves of attackers' observation trees, start by taking 2 undefined observable values, then concatenate them to both a divergence observation $\Uparrow$ and a termination observation $\Downarrow$. Intuitively, this specification describes an abstract observation tree illustrated by Figure 7.1(a).



(a) An abstract observation tree described by Equation (7.1)

(b) Concrete observation trees

Figure 7.1: Concrete and abstract attackers' observation trees of the program in Listing 7.1

As a consequence, we can deduce that the leaves $Leaves(\mathcal{T}_b^{\varrho_{L_0}})$ of attackers' observation trees of the program in Listing 7.1 have a cardinal number lesser than or equal to 4 – assuming that the bound b on attackers' observations is large enough. Moreover, we can also deduce that the maximum leakage $\mathcal{ML}$ is upper-bounded by: $\mathcal{ML} \leq \log_2(4) = 2\ bits$. Note that as illustrated by Figure 7.1(b), concrete attackers' observation trees of the program in Listing 7.1 have at most 3 leaves, which corresponds to a maximum leakage $\mathcal{ML} = \log_2(3) \approx 1.59\ bits$.

For the simple specification in Equation (7.1), we can easily enumerate the leaves of attackers' observation trees in order to upper-bound min-capacity $\mathcal{ML}$. However, enumerating the cardinal number of leaves described by a regular specification might not be as straightforward in general. Therefore,

we rely on the framework of analytic combinatorics [FS09] in order to solve this enumeration problem. The next section attempts to provide a gentle introduction to the analytic combinatorics framework.

7.2 Analytic Combinatorics

Analytic combinatorics [FS09] is a framework aimed at studying properties of objects built according to a finite set of rules. In a nutshell, analytic combinatorics provides a calculational approach to quantify properties of discrete objects.

Essentially, enumeration problems such as "how many objects of size n are there?" boils down to defining a specification describing how to build such objects. Indeed, analytic combinatorics then relies on mechanized methods to translate this specification into a *generating function*. Analytic combinatorics also defines mechanized methods to study *generating functions* through mathematical analysis, in order to extract asymptotic counting information and estimate the growth of, say, the sequence $(a_n)_{n\geq 0}$ where a_n represents the number of objects of size n.

7.2.1 Combinatorial Classes

Combinatorics concerns the study of finite or countable objects. Usually, one wants to count the objects of a certain class according to some characteristic parameter. Definition 22 introduces combinatorial classes as a countable set of objects, provided with a size function.

Definition 22 (Combinatorial classes $\mathcal{C}$)**.**
A combinatorial class $\mathcal{C}$ is a finite or denumerable set on which a size function is defined, satisfying the following conditions:

1. *the size of an element is a non-negative integer, and*
2. *the number of elements of any given size is finite.*

Consider for instance the following set $\mathcal{W}$ of words, provided with a size function $|\cdot| \in \mathcal{W} \mapsto \mathbb{Z}_{\geq 0}$ that represents the length of a word:

$$\mathcal{W} \triangleq \{a, b, aa, ab, bb, ba, aaa, aab, abb, aba, bba, bbb, bab, baa\} \tag{7.2}$$

The set $\mathcal{W}$ provided with a size function then defines a combinatorial class. Since we explicitly enumerated all the words of the set $\mathcal{W}$, we can also count elements of $\mathcal{W}$ wrt. their size. Indeed, let us denote by $\mathcal{W}_n$ the set of

elements in $\mathcal{W}$ that have a size n. Let us also denote by W_n the cardinal number of the set $\mathcal{W}_n$ of elements in $\mathcal{W}$ that have a size n. Then, the sequence $W_n = |\mathcal{W}_n|$ is given by:

$$W_0 = 0 \quad W_1 = 2 \quad W_2 = 4 \quad W_3 = 8 \quad W_n = 0 \ (\forall n \geq 4) \tag{7.3}$$

The sequence $(W_n)_{n\geq 0}$ is the *counting sequence* of the class $\mathcal{W}$. Definition 23 introduces the counting sequence $(C_n)_{n\geq 0}$ of a combinatorial class $\mathcal{C}$ as the cardinal number of elements in $\mathcal{C}$ that have a size n.

Definition 23 (Counting sequence $(C_n)_{n\geq 0}$)**.**
Let us denote by $\mathcal{C}_n$ the objects in a combinatorial class $\mathcal{C}$ that have a size n.

The counting sequence of a combinatorial class $\mathcal{C}$ is the sequence $(C_n)_{n\geq 0}$ where $C_n \triangleq |\mathcal{C}_n|$ is the number of objects in class $\mathcal{C}$ that have a size n.

In general, we want to avoid enumerating all the elements of a combinatorial class in order to count elements of a certain size n. To this end, we introduce ordinary generating function (OGF) in Definition 24. OGFs are a keystone to the analytic combinatorics framework.

Definition 24 (Ordinary generating functions $C(z)$)**.**
The OGF of a combinatorial class $\mathcal{C}$ is the formal power series:

$$\begin{aligned} C(z) &\triangleq \sum_{c\in\mathcal{C}} z^{|c|} \\ &\triangleq \sum_{n=0}^{\infty} C_n z^n \end{aligned}$$

where $(C_n)_{n\geq 0}$ is the counting sequence of the combinatorial class $\mathcal{C}$.

For instance, the OGF $W(z)$ of the combinatorial class $\mathcal{W}$ is given by Equation (7.4). Note that by collecting the terms in z^n, we obtain exactly the counting sequence found earlier in Equation (7.3).

$$\begin{aligned} W(z) &= z + z + z^2 + z^2 + z^2 + z^2 + z^3 + z^3 + z^3 + z^3 + z^3 + z^3 + z^3 + z^3 \\ &= 2z + 4z^2 + 8z^3 \end{aligned} \tag{7.4}$$

Intuitively, instead of enumerating all the elements of a combinatorial class $\mathcal{C}$ in order to determine its counting sequence $(C_n)_{n\geq 0}$, we can simply provide a specification for $\mathcal{C}$ in terms of simpler constructions. Then, ordinary generating functions come into play in order to systematically derive the counting sequence $(C_n)_{n\geq 0}$ of the combinatorial class $\mathcal{C}$ by leveraging its specification.

In the next subsection, we illustrate this mechanized process for the combinatorial class $\mathcal{W}$. We also introduce the specification rules the tree abstraction relies upon.

Henceforth, we adopt the same naming conventions as Flajolet and Sedgewick [FS09]: combinatorial classes, the set of elements that have size n, counting sequences and OGFs are denoted by the same groups of letters:

$$\begin{aligned} \text{Combinatorial classes} &: \mathcal{C} \\ \text{Set of objects of size } n &: \mathcal{C}_n \\ \text{Counting sequences} &: C_n = |\mathcal{C}_n| \\ \text{OGF} &: C(z) = \sum_{n \geq 0} C_n z^n \end{aligned}$$

7.2.2 Regular Specifications

Let us consider for instance the combinatorial class $\mathcal{W}$ introduced earlier in Equation (7.2). Can we define a specification describing how to construct words in $\mathcal{W}$?

We can in fact draw on an analogy from the field of formal languages in order to intuitively define a regular specification for the combinatorial class $\mathcal{W}$. Indeed, let us denote by $\mathcal{A} \triangleq \{a\}$ (resp. by $\mathcal{B} \triangleq \{b\}$) the combinatorial class consisting of one word `a` (resp. one word `b`). Then, the class $\mathcal{W}$ admits the regular specification introduced in Equation (7.5).

$$\begin{aligned} \mathcal{W} &= \underbrace{(\mathcal{A} \oplus \mathcal{B})}_{\mathcal{U}} \oplus \underbrace{(\mathcal{A} \oplus \mathcal{B})^2}_{\mathcal{V}} \oplus \underbrace{(\mathcal{A} \oplus \mathcal{B})^3}_{\mathcal{Y}} \\ &= \mathcal{U} \oplus \mathcal{V} \oplus \mathcal{Y} \end{aligned} \tag{7.5}$$

In this equation, we denote by $\mathcal{C}^2$ the concatenation $\mathcal{C} \cdot \mathcal{C}$ of words in $\mathcal{C}$. Similarly, $\mathcal{C}^3$ denotes the concatenation $\mathcal{C} \cdot \mathcal{C} \cdot \mathcal{C}$ of words in $\mathcal{C}$ 3 times. One major difference exists though, between the regular specification in Equation (7.5) and regular expressions used in the field of formal languages. Indeed, instead of denoting a set union over combinatorial classes, the operator $\oplus$ is actually a *combinatorial sum*.

Combinatorial sum. The combinatorial sum $\oplus$ captures the idea of a disjoint union of two combinatorial classes, yet without requiring both combinatorial classes to be actually disjoint. For instance, $\mathcal{A} \oplus \mathcal{B}$ can be

formalized in a set theoretic sense as the set union of two disjoint copies $\mathcal{A}^{\blacktriangle}$ and $\mathcal{B}^{\triangledown}$ of $\mathcal{A}$ and $\mathcal{B}$:

$$\mathcal{A} \oplus \mathcal{B} \triangleq (\{\blacktriangle\} \times \mathcal{A}) \cup (\{\triangledown\} \times \mathcal{B})$$

This is exactly the same definition given in Flajolet and Sedgewick [FS09]. Intuitively, objects in both combinatorial classes $\mathcal{A}$ and $\mathcal{B}$ are marked with distinct markers in order to guarantee them to be disjoint.

This is the main difference between a regular specification in combinatorics and a regular expression as used in formal languages. In fact, given a regular expression re, we can rely on OGFs to count the number of different ways we can build words of size n. Yet, this is actually different from counting the words of size n in the formal language defined by re. Indeed, using *OGFs*, we would be counting the words of size n along with their multiplicities. For instance, using OGFs to count the words generated by a regular expression $a + a$, we find that there are 2 words of size 1, since the union $+$ is interpreted as a combinatorial sum $\oplus$. Interestingly, the combinatorial sum fits quiet well our problem of counting attackers' observation trees.

As a consequence of the definition of the combinatorial sum $\oplus$, we can deduce that the counting sequence C_n of a class $\mathcal{C} \triangleq \mathcal{T} \oplus \mathcal{S}$ is equal to the sum of both counting sequences $T_n + S_n$. More importantly, this also means that the OGF of the class $\mathcal{C}$ is equal to the sum of both OGFs $T(z)$ and $S(z)$:

$$\mathcal{C} \triangleq \mathcal{T} \oplus \mathcal{S} \implies C(z) = T(z) + S(z)$$

Therefore, recalling the specification of $\mathcal{W}$ in Equation (7.5), we can deduce that the OGF of the class $\mathcal{W}$ is given by:

$$W(z) = U(z) + V(z) + Y(z)$$
$$U(z) = A(z) + B(z) = 2z$$

Concatenation. Let us consider 2 combinatorial classes $\mathcal{T}$ and $\mathcal{S}$, and denote by $\mathcal{C} \triangleq \mathcal{T} \cdot \mathcal{S}$ the combinatorial class resulting from concatenating objects in $\mathcal{T}$ with objects in $\mathcal{S}$. How can we build objects in the class $\mathcal{C}$ that have size n?

Intuitively, in order to construct objects in $\mathcal{C}$ of size n we can:

1. iterate over $k \in [0, n]$, then
2. choose any object in $\mathcal{T}$ of size k, then concatenate it with

3. any object in $\mathcal{S}$ of size $n-k$.

Therefore, the counting sequence C_n of $\mathcal{C}$ is equal to:

$$C_n = \sum_{k=0}^{n} T_k S_{n-k}$$

Interestingly, C_n is actually the coefficient resulting from the multiplication of both formal power series $T(z)$ and $S(z)$. This means that the OGF of the class $\mathcal{C}$ is equal to the product of both OGFs $T(z)$ and $S(z)$:

$$\mathcal{C} \triangleq \mathcal{T} \cdot \mathcal{S} \implies C(z) = T(z) \times S(z)$$

As a consequence, recalling the specification of $\mathcal{W}$ in Equation (7.5), we can now derive systematically the OGF of the class $\mathcal{W}$ by relying only on the specification of $\mathcal{W}$:

$$\begin{aligned} V(z) &= U^2(z) = 4z^2 \\ Y(z) &= U^3(z) = 8z^3 \\ W(z) &= U(z) + V(z) + Y(z) = 2z + 4z^2 + 8z^3 \end{aligned}$$

So far, we introduced 2 regular specification rules: the combinatorial sum $\oplus$, as well as the concatenation $\cdot$. The last regular specification we need to introduce is the sequence construction.

Sequence construction. Assuming a combinatorial class $\mathcal{A}$, then the sequence class, denoted by $\mathcal{A}^\star$, is the infinite combinatorial sum given by:

$$\mathcal{A}^\star = \{\epsilon\} \oplus \mathcal{A} \oplus \mathcal{A}^2 \oplus \mathcal{A}^3 \oplus \dots$$

where ϵ denotes the neutral element wrt. concatenation – therefore has a size of 0.

Note that $\mathcal{A}^\star$ is a well-defined combinatorial class iff. $\mathcal{A}$ contains no objects of size 0. This is necessary and sufficient to guarantee that the number of elements of any given size is finite. Intuitively, if the class $\mathcal{A}$ contains the neutral ϵ, then for all objects $a \in \mathcal{A}$, there would be an infinite number of objects in $\mathcal{A}^\star$ having the same size as object a. Indeed, this stems from the fact that all concatenations of $\mathcal{A}$ will contain at least the object a: $\forall k \geq 1, a \in \mathcal{A}^k$.

Let us denote by $\mathcal{C}$ the sequence class $\mathcal{A}^\star$. Then by definition, the OGF of $\mathcal{C}$ is equal to:

$$C(z) = 1 + A(z) + A^2(z) + A^3(z) + \dots$$

This latter equation can be further simplified by noting that $A_0 = |\mathcal{A}_0| = 0$ since $\mathcal{A}$ is assumed to not contain any objects of size 0, and finally that $C(z)$ is a geometric sequence. Therefore, the OGF of a sequence class is given by:

$$\mathcal{C} \triangleq \mathcal{A}^\star \implies C(z) = \frac{1}{1 - A(z)}$$

Translating regular specifications into OGFs. Theorem 8 summarizes the results of this subsection. The OGF of a combinatorial class described by a regular specification can be systematically computed. Indeed, regular specifications translate directly into equations over generating functions.

Theorem 8 (Regular specifications as equations over OGFs)**.**
Any regular specification of a combinatorial class translates directly into equations over generating functions.

$$\begin{aligned}
\textit{Combinatorial sum:} \quad & \mathcal{C} = \mathcal{A} \oplus \mathcal{B} \implies C(z) = A(z) + B(z) \\
\textit{Concatenation:} \quad & \mathcal{C} = \mathcal{A} \cdot \mathcal{B} \implies C(z) = A(z) \times B(z) \\
\textit{Sequence:} \quad & \mathcal{C} = \mathcal{A}^\star \implies C(z) = \frac{1}{1 - A(z)} \ (\mathcal{A}_0 = \varnothing)
\end{aligned}$$

Note that we restrict ourselves to regular specifications only, namely construction rules involving combinatorial sums, concatenations, as well as sequences. Flajolet and Sedgewick [FS09] actually prove that even more complex specification rules also translate directly into equations over OGFs.

7.2.3 Asymptotic Estimates

Once we describe a combinatorial class $\mathcal{C}$ through a regular specification, Theorem 8 enables us to systematically translate this specification into an OGF $C(z)$. Yet, how can we use this OGF in order to find the number of objects in $\mathcal{C}$ that have a size n?

Recall that Definition 24 of an OGF already embodies the counting information we seek. The counting sequence C_n is the coefficient of z^n in the formal power series $C(z)$. As a consequence, for a fixed n, a straightforward method consists in seeking the coefficient of z^n in $C(z)$. This coefficient can be extracted by expanding the formal power series until the order n, or by using Taylor's expansion formula to directly compute the coefficient of z^n – the coefficient of z^n is equal to the nth derivative of $C(z)$ evaluated at $z = 0$ and divided by $n!$.

In fact, sometimes we can compute an explicit formula for the counting sequence C_n. For instance, when the nth derivative of $C(z)$ can be determined symbolically, we can rely on Taylor's formula [Wil05] to explicitly determine C_n, as a function of n. An alternative method consists in seeking recurrences for C_n and solving them. Nevertheless, the most general method relies on *complex analysis* – studying $C(z)$ as a function of the complex space $\mathbb{C}$ – in order to approximate the growth of the sequence C_n, by means of asymptotic estimates. We will expose in the following the most important results [FS09] in complex analysis of OGFs obtained by translating a regular specification.

Rational functions. The first result, introduced in Corollary 4, is a direct consequence of Theorem 8. A combinatorial class described by a regular specification has an OGF that is a rational function. Intuitively, regular specifications involve combinatorial sums, concatenations as well as sequences. These specifications translate into sums, products and inverses, which yield an OGF that is a rational function.

Corollary 4 (Regular specifications as rational OGFs)**.**
Any combinatorial class $\mathcal{C}$ described by a regular specification has an OGF that is a rational function:

$$C(z) = \frac{N(z)}{D(z)} = Q(z) + \sum_{\alpha, r_\alpha} \frac{c_{\alpha,r_\alpha}}{(z-\alpha)^{r_\alpha}}$$

where both $N(z)$ and $D(z)$ are polynomials, $Q(z)$ is a polynomial of degree $n_0 = deg(Q) = \max(0, deg(N) - deg(D))$, the complex α ranges over the poles of $C(z)$, c_{α,r_α} is a constant, and r_α is upper-bounded by the multiplicity of α as a pole of $C(z)$.

Corollary 4 also proposes a partial fraction decomposition for OGFs obtained through a regular specification. Interestingly, this decomposition sets up the stage for the next important result we introduce, aimed at asymptotically approximating the counting sequence $(C_n)_{n\geq 0}$ of a regular combinatorial class $\mathcal{C}$.

Asymptotic approximation. Flajolet and Sedgewick [FS09] prove that the singularities of an OGF $C(z)$ determine the asymptotic behaviour of its counting sequence $(C_n)_{n\geq 0}$. This is an important result since it provides a systematic method to translate an OGF into an asymptotic approximation of its counting sequence. While this result applies to any OGF, we are going

to restrict ourselves to the simpler case of rational OGFs, namely those we obtain by translating a regular specification.

Using the partial fraction decomposition of a rational OGF, we can determine an explicit formula for its counting sequence. Let us denote by $[z^n]f(z)$ the coefficient of z^n in a generating function $f(z)$. Let us also assume a rational OGF $C(z)$. Then $C(z)$ admits a partial fraction decomposition:

$$C(z) = \frac{N(z)}{D(z)} = Q(z) + \sum_{\alpha, r_\alpha} \frac{c_{\alpha,r_\alpha}}{(z-\alpha)^{r_\alpha}}$$

where $Q(z)$ is a polynomial of degree $n_0 = \max(0, deg(N) - deg(D))$, α ranges over the poles of $C(z)$, r_α is upper-bounded by the multiplicity of α as a pole and c_{α,r_α} is a constant. Therefore, for n larger than n_0, the counting sequence C_n depends only on the poles of the rational OGF $C(z)$:

$$C_n = [z^n]C(z) = \sum_{\alpha, r_\alpha} [z^n] \frac{c_{\alpha,r_\alpha}}{(z-\alpha)^{r_\alpha}} \qquad (n \geq n_0) \tag{7.6}$$

Moreover, using Newton's generalised binomial theorem, we have:

$$\begin{aligned}
[z^n]\frac{1}{(z-\alpha)^{r_\alpha}} &= \frac{(-1)^{r_\alpha}}{\alpha^{r_\alpha}}[z^n]\frac{1}{(1-\frac{z}{\alpha})^{r_\alpha}} \\
&= \frac{(-1)^{r_\alpha}}{\alpha^{r_\alpha}}[z^n]\sum_{k=0}^{\infty}\binom{k+r_\alpha-1}{r_\alpha-1}(\frac{z}{\alpha})^k \\
&= \frac{(-1)^{r_\alpha}}{\alpha^{r_\alpha}}[z^n]\sum_{k=0}^{\infty}\binom{k+r_\alpha-1}{r_\alpha-1}\alpha^{-k}z^k \\
&= \frac{(-1)^{r_\alpha}}{\alpha^{r_\alpha}}\binom{n+r_\alpha-1}{r_\alpha-1}\alpha^{-n}
\end{aligned} \tag{7.7}$$

Finally, we can simplify Equation (7.6) by using Equation (7.7) in order to obtain an explicit formula for the counting sequence C_n, assuming n is larger than the degree n_0 of $Q(z)$:

$$C_n = \sum_{\alpha, r_\alpha} c_{\alpha,r_\alpha}\frac{(-1)^{r_\alpha}}{\alpha^{r_\alpha}}\binom{n+r_\alpha-1}{r_\alpha-1}\alpha^{-n} \tag{7.8}$$

This last equation yields Theorem 9 by collecting the terms associated to each pole α. This theorem states that the counting sequence C_n of a rational OGF $C(z)$ is a sum of exponential polynomials determined by the poles of $C(z)$. Note that the binomial coefficient $\binom{n+r_\alpha-1}{r_\alpha-1}$ yields a polynomial in n of degree $r_\alpha - 1$, and r_α is upper-bounded by the multiplicity of the pole r_α.

Theorem 9 (Expansion of rational OGFs)**.**
If a rational OGF has its poles at points $\alpha_1, \alpha_2, \ldots, \alpha_m$, then its coefficients C_n are a sum of exponential-polynomials: there exists m polynomials $\{F_j\}_{j=1}^m$ such that, for n larger than some fixed n_0:

$$C_n = [z^n]C(z) = \sum_{j=1}^{m} F_j(n)\alpha_j^{-n}$$

Furthermore, the order of the polynomial $F_j(n)$ is equal to the multiplicity order of the pole α_j minus one.

Theorem 9 actually provides a direct method for asymptotic estimation of the counting sequence C_n. Indeed, if we order the exponential-polynomial terms according to the α's of increasing modulus, each group is exponentially smaller than the other. Let us assume for instance that a rational OGF have a unique dominant pole $|\alpha_1| < |\alpha_2| \leq |\alpha_3| \leq \ldots \leq |\alpha_m|$. Then, the counting sequence C_n's asymptotic behaviour is given by :

$$C_n \sim F_1(n)\alpha_1^{-n} \sim \frac{(-1)^r c}{(r-1)!} n^{r-1}\alpha_1^{-n-r} \tag{7.9}$$

where r is the multiplicity of the pole α_1, and c is a constant equal to $c = \lim_{z\to\alpha_1} C(z)(z-\alpha)^r$.

To sum up, extracting asymptotic counting information from a rational OGF boils down to finding its poles and comparing their modulus. According to Theorem 9, the asymptotic behaviour of the counting sequence is defined by the poles having the smallest modulus. Recall that we only presented the simpler case of rational OGFs. In general, Flajolet and Sedgewick [FS09] also prove that the asymptotic behaviour of the counting sequence of an OGF is determined by its singularities.

7.3 Regular Specifications of Attackers' Observations

By relying on Theorem 8, a regular specification of attackers' observations translates directly into a generating function. The obtained generating function embodies counting information about the leaves of attackers' observation trees. Therefore, we can leverage the tools from the field of analytic combinatorics in order to estimate min-capacity by Theorem 4. In this section, we introduce atomic combinatorial classes for attackers' observations, as well as a normal form in order to accommodate the framework of regular specifications.

Atomic abstract observations. For programs that output intermediate results of computation, attackers may either observe the value of an expression `a` through a command $^{\text{pp}}$`output a`, silent divergence $\Uparrow$ or termination $\Downarrow$. Therefore, we introduce atomic combinatorial classes representing these observations.

The tree abstraction is parametrized by the cardinal abstraction. For each command $^{\text{pp}}$`output a`, the tree abstraction concludes that attackers may observe at most k different values, assuming the cardinal abstraction evaluates expression `a` to k. Since command $^{\text{pp}}$`output a` produces an output that have length 1, we can therefore represent the observations attackers make in this case by a combinatorial class $\mathcal{O}^k$, containing k undefined values of size 1:

$$\mathcal{O}^k \triangleq \oplus_{i=1}^{k} ?$$

As a consequence, the OGF of the combinatorial class $\mathcal{O}^k$ is given by:

$$\mathcal{O}^k(z) = kz$$

Similarly, we can introduce an atomic combinatorial class $\mathcal{O}^{\Uparrow}$ containing only one observation of size 1, representing divergence $\Uparrow$ as well as an atomic class $\mathcal{O}^{\Downarrow}$ containing one observation of size 1, representing termination $\Downarrow$. However, both combinatorial classes are isomorphic since they have the same generating function z. Therefore, we only define $\mathcal{O}^{\Uparrow}$ as a combinatorial class containing only one object of size 1, representing either divergence or termination. Definition 25 summarizes these definitions of the atomic abstract observations attackers can make.

Definition 25 (Atomic abstract observations)**.**
For all $k \in [1, 2^\kappa]$, the atomic combinatorial class $\mathcal{O}^k$ contains k undefined values of size 1, representing the values attackers may observe through output instructions.

The atomic combinatorial class $\mathcal{O}^\epsilon$ contains one object of size 0, representing an empty observation trace ϵ.

Additionally, the atomic combinatorial class $\mathcal{O}^{\Uparrow}$ contains one observation of size 1, representing either divergence $\Uparrow$ or termination $\Downarrow$.

Their generating functions are given by:

$$O^k(z) \triangleq kz \qquad O^\epsilon(z) \triangleq 1 \qquad O^{\Uparrow}(z) \triangleq z.$$

In fact, the combinatorial class $\mathcal{O}^1$ is also isomorphic to $\mathcal{O}^{\Uparrow}$. Yet, the class $\mathcal{O}^{\Uparrow}$ enjoys a property that $\mathcal{O}^1$ does not. Indeed, since $\mathcal{O}^{\Uparrow}$ represents either the observation of divergence or termination, it represents a terminal observation: attackers cannot make further observations whenever a program

terminates or diverges silently. Therefore, we need to separatly define both combinatorial classes $\mathcal{O}^1$ and $\mathcal{O}^{\Uparrow}$. Additionally, despite the expressive power of regular specifications, there is no straightforward way to ensure that the combinatorial class $\mathcal{O}^{\Uparrow}$ is terminal. To accommodate abstract attackers' observations to regular specifications, we opt to simplify them during the analysis by relying on a normal form.

Normal form. We opt to simplify the regular specifications of attackers' observations as the analysis proceeds, in order to make sure that the observation of either divergence or termination is terminal. Therefore, we rely on a normal form and simplification operators for regular specifications rules.

The normal form we consider is defined as the combinatorial sum of 3 regular specifications r_1, r_2 and $r_3 \cdot \mathcal{O}^{\Uparrow}$, such that:

1. r_1 is either the combinatorial class $\mathcal{O}^{\epsilon}$ representing an empty observation ϵ, or the empty regular specification $\varnothing$,

2. r_2 is either a regular specification involving only the combinatorial classes $\mathcal{O}^k$ for $k \in [1, 2^{\kappa}]$ and not containing any empty observation of size 0, or the empty regular specification $\varnothing$, and

3. r_3 is either a regular specification involving only the combinatorial classes $\mathcal{O}^k$ and $\mathcal{O}^{\epsilon}$, or the empty regular specification $\varnothing$.

The normal form for regular specifications of attackers' observations are written as follows:

$$r_1 \oplus r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow}$$

The main idea behind this normal form is to make sure that neither the regular specification r_1, nor r_2, nor r_3 contains a terminal combinatorial class $\mathcal{O}^{\Uparrow}$. Moreover, the normal form also ascertains that neither r_2, nor $r_3 \cdot \mathcal{O}^{\Uparrow}$ contains the atomic combinatorial class $\mathcal{O}^{\epsilon}$, representing an empty observation ϵ of size 0. This latter requirement proves useful in order for the sequence construction to be well-defined, wrt. Theorem 8.

We also define 3 operators $\mathrm{s}_{\mathrm{sum}}$, $\mathrm{s}_{\mathrm{con}}$ and $\mathrm{s}_{\mathrm{seq}}$ for the 3 regular specification rules – combinatorial sum, concatenation and sequence construction –, in order to simplify the regular specifications of attackers' observations. Figure 7.2 summarizes the definitions of these 3 operators. These definitions rely on the associativity of both the combinatorial sum and the concatenation rules, in addition to the distributivity of the concatenation over the combinatorial sum. Note that these operators also keep the simplified regular specifications into a normal form $r_1 \oplus r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow}$.

$$\mathrm{s}_{\mathrm{sum}}\left[r_1 \oplus r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow}, s_1 \oplus s_2 \oplus s_3 \cdot \mathcal{O}^{\Uparrow}\right] \triangleq \begin{cases} \varnothing \oplus (r_2 \oplus s_2) \oplus (r_3 \oplus s_3) \cdot \mathcal{O}^{\Uparrow} & \text{if } r_1 = s_1 = \varnothing \\ \mathcal{O}^{\epsilon} \oplus (r_2 \oplus s_2) \oplus (r_3 \oplus s_3) \cdot \mathcal{O}^{\Uparrow} & \text{otherwise} \end{cases}$$

$$\mathrm{s}_{\mathrm{con}}\left[r_1 \oplus r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow}, s_1 \oplus s_2 \oplus s_3 \cdot \mathcal{O}^{\Uparrow}\right] \triangleq (r_1 \cdot s_1) \oplus (r_1 \cdot s_2 \oplus r_2 \cdot s_1 \oplus r_2 \cdot s_2) \oplus (r_1 \cdot s_3 \oplus r_2 \cdot s_3 \oplus r_3) \cdot \mathcal{O}^{\Uparrow}$$

$$\mathrm{s}_{\mathrm{seq}}\left[r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow}\right] \triangleq \mathcal{O}^{\epsilon} \oplus r_2 \cdot r_2^{\star} \oplus r_2^{\star} \cdot r_3 \cdot \mathcal{O}^{\Uparrow}$$

Figure 7.2: Simplification operators $\mathrm{s}_{\mathrm{sum}}$, $\mathrm{s}_{\mathrm{con}}$ and $\mathrm{s}_{\mathrm{seq}}$

The operator $\mathrm{s}_{\mathrm{sum}}$ simplifies the combinatorial sum $\mathcal{O}^{\epsilon} \oplus \mathcal{O}^{\epsilon}$ to only $\mathcal{O}^{\epsilon}$, since from an attackers' perspective, observing an empty trace ϵ or another empty trace ϵ amounts to observing nothing. Note that such a simplification cannot not be performed for the combinatorial classes $\mathcal{O}^k \oplus \mathcal{O}^k$. For instance, the regular specification $\mathcal{O}^1 \oplus \mathcal{O}^1$ states that attackers may observe 1 output value or 1 output value. Since both values are undefined, they may as well be different, meaning that attackers may observe at most 2 different values. The combinatorial sum is therefore well suited to describe such observations since it suitably captures the fact that attackers may observe 2 different output values.

The operator $\mathrm{s}_{\mathrm{con}}$ distributes the concatenation over combinatorial sums, then simplifies the term $r_3 \cdot \mathcal{O}^{\Uparrow} \cdot (s_1 \oplus s_2 \oplus s_3 \cdot \mathcal{O}^{\Uparrow})$ to $r_3 \cdot \mathcal{O}^{\Uparrow}$. Indeed, whenever attackers observe a trace $r_3 \cdot \mathcal{O}^{\Uparrow}$ that silently diverges or terminates, they cannot make any additional observations.

The operator $\mathrm{s}_{\mathrm{seq}}$ is only defined for regular specifications having a normal form $r_2 \oplus r_3 \cdot \Uparrow$ in order to ensure the combinatorial classes they describe do not contain the empty observation ϵ. Additionally, it draws on an intuition from the theory of formal languages in order to simplify the regular specification $(r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow})^{\star}$. Indeed, assuming that $\mathcal{S}$ and $\mathcal{T}$ are two combinatorial classes containing no objects of size 0, then the *denesting rule* [KP00] applies:

$$(\mathcal{S} \oplus \mathcal{T})^{\star} = (\mathcal{S}^{\star}\mathcal{T})^{\star}\mathcal{S}^{\star}$$

We can easily prove a slightly weaker version of this equality over regular specifications, by proving that both the right-hand side and the left-hand

side combinatorial classes are isomorphic since their OGFs are equal:

$$\frac{1}{1-S(z)-T(z)} = \frac{1}{1-\frac{T(z)}{1-S(z)}}\frac{1}{1-S(z)}$$

Therefore, we have:

$$\begin{aligned}
(r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow})^\star &= \text{(By the denesting rule)} \\
&\quad (r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow})^\star \cdot r_2^\star \\
&= \text{(Since } \mathcal{C}^\star = \mathcal{O}^\epsilon \oplus \mathcal{C} \cdot \mathcal{C}^\star \text{, with } \mathcal{C} = r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow}\text{)} \\
&\quad \left(\mathcal{O}^\epsilon \oplus (r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow}) \cdot (r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow})^\star\right) \cdot r_2^\star \\
&= \text{(By distributivity)} \\
&\quad r_2^\star \oplus r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow} \cdot (r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow})^\star \cdot r_2^\star \\
&= \text{(Since } \mathcal{C}^\star = \mathcal{O}^\epsilon \oplus \mathcal{C} \cdot \mathcal{C}^\star \text{, with } \mathcal{C} = r_2^\star\text{)} \\
&\quad \mathcal{O}^\epsilon \oplus r_2 \cdot r_2^\star \oplus r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow} \cdot (r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow})^\star \cdot r_2^\star
\end{aligned}$$

Finally, the operator $\mathrm{s}_{\mathrm{seq}}$ simplifies the term $r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow} \cdot (r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow})^\star \cdot r_2^\star$ to $r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow}$ to ascertain that $\mathcal{O}^{\Uparrow}$ is terminal in the resulting regular specification of attackers' observations.

7.4 Abstract Semantics

Our static analysis computing the regular specification for attackers' observations is parametrized by the cardinal abstraction. Figure 7.3 introduces the abstract semantics $[\![c]\!]^\sharp$ of instructions. This abstract semantics is defined over a pair $(e^\sharp, \varrho^\sharp)$ of a regular specification $e^\sharp$ in normal form and an abstract environment $\varrho^\sharp$ representing the results of the cardinal abstraction. The abstract semantics also yields a new pair $(e_2^\sharp, \varrho_2^\sharp)$, where $e_2^\sharp$ is a regular specification describing attackers' observations after the execution of command `c`. Note that all the regular specifications are constructed using the simplification operators $\mathrm{s}_{\mathrm{sum}}$, $\mathrm{s}_{\mathrm{con}}$ and $\mathrm{s}_{\mathrm{seq}}$, so that the computed regular specification remains under a normal form $r_1 \oplus r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow}$.

Command $^{\mathrm{pp}}$`skip` and assignments do not modify the input regular specification. Command $^{\mathrm{pp}}$`stop` enables attackers to observe termination. Thus, the output regular specification is the concatenation of the combinatorial class $\mathcal{O}^{\Uparrow}$, representing both the observation of either termination $\Downarrow$ or divergence $\Uparrow$, to the input regular specification. The abstract semantics for the sequence command $\mathtt{c}_1\mathtt{;}\ \mathtt{c}_2$ is the composition of both the abstract semantics $[\![c_1]\!]^\sharp$ and $[\![c_2]\!]^\sharp$.

$$
\begin{aligned}
[\![^{pp} skip]\!]^\sharp \left(e^\sharp, \varrho^\sharp\right) &\triangleq \left(e^\sharp, \varrho^\sharp\right) \\
[\![^{pp} id := a]\!]^\sharp \left(e^\sharp, \varrho^\sharp\right) &\triangleq \left(e^\sharp, \varrho^\sharp \left[id \mapsto (pp, \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp)\right]\right) \\
[\![c_1;\ c_2]\!]^\sharp \left(e^\sharp, \varrho^\sharp\right) &\triangleq [\![c_2]\!]^\sharp\Big([\![c_1]\!]^\sharp \left(e^\sharp, \varrho^\sharp\right)\Big) \\
[\![^{pp} stop]\!]^\sharp \left(e^\sharp, \varrho^\sharp\right) &\triangleq \left(\mathrm{s}_{\mathrm{con}} \left[e^\sharp, \mathcal{O}^{\Uparrow}\right], \varrho^\sharp\right) \\
[\![^{pp} output\ a]\!]^\sharp \left(e^\sharp, \varrho^\sharp\right) &\triangleq \text{let } k = \mathbb{A}[\![a]\!]\varrho^\sharp \text{ in } \quad \left(\mathrm{s}_{\mathrm{con}} \left[e^\sharp, \mathcal{O}^k\right], \varrho^\sharp\right) \\
[\![^{pp} if\ (a)\ c_1\ else\ c_2]\!]^\sharp \left(e^\sharp, \varrho^\sharp\right) &\triangleq \text{let } \left(e_1^\sharp, \varrho_1^\sharp\right) = [\![c_1]\!]^\sharp \left(\mathcal{O}^\epsilon, \varrho^\sharp\right) \text{ in} \\
&\quad \text{let } \left(e_2^\sharp, \varrho_2^\sharp\right) = [\![c_2]\!]^\sharp \left(\mathcal{O}^\epsilon, \varrho^\sharp\right) \text{ in} \\
&\quad \text{let } \varrho_3^\sharp = \\
&\qquad \lambda id. \begin{cases} \varrho_1^\sharp(id) \cup_\otimes \varrho_2^\sharp(id) & \text{if } \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp = 1 \\ \varrho_1^\sharp(id) \cup_{add(c_1,c_2)} \varrho_2^\sharp(id) & \text{otherwise} \end{cases} \\
&\quad \text{in} \\
&\quad \text{let } e_3^\sharp = \mathrm{s}_{\mathrm{sum}} \left[e_1^\sharp, e_2^\sharp\right] \text{ in} \\
&\quad \left(\mathrm{s}_{\mathrm{con}} \left[e^\sharp, e_3^\sharp\right], \varrho_3^\sharp\right) \\
[\![^{pp} while\ (a)\ c]\!]^\sharp \left(e^\sharp, \varrho^\sharp\right) &\triangleq \text{let } \varrho_2^\sharp = \mathrm{lfp}^{\dot{\subseteq}_\otimes}_{\varrho^\sharp} [\![c]\!]^\sharp \text{ in} \\
&\quad \text{let } \varrho_3^\sharp = \begin{cases} \varrho_2^\sharp & \text{if } \mathbb{A}^\sharp[\![a]\!]\varrho_2^\sharp = 1 \\ \lambda id. \top_{(c)}(\varrho_2^\sharp(id)) & \text{otherwise} \end{cases} \\
&\quad \text{in} \\
&\quad \text{let } \left(e_2^\sharp, \_\right) = [\![c]\!]^\sharp \left(\mathcal{O}^\epsilon, \varrho_2^\sharp\right) \text{ in} \\
&\quad \text{let } e_3^\sharp = \mathrm{s}_{\mathrm{while}} \left[e_2^\sharp\right] \text{ in} \\
&\quad \text{if } (e_3^\sharp \neq \top) \text{ then} \\
&\quad \left(\mathrm{s}_{\mathrm{con}} \left[e^\sharp, e_3^\sharp\right], \varrho_3^\sharp\right)
\end{aligned}
$$

$$\mathrm{s}_{\mathrm{while}} \left[\mathcal{O}^\epsilon\right] \triangleq \mathcal{O}^\epsilon \oplus \mathcal{O}^{\Uparrow} \qquad \mathrm{s}_{\mathrm{while}} \left[\mathcal{O}^\epsilon \oplus \mathcal{O}^{\Uparrow}\right] \triangleq \mathcal{O}^\epsilon \oplus \mathcal{O}^{\Uparrow}$$

$$\mathrm{s}_{\mathrm{while}} \left[r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow}\right] \triangleq \mathrm{s}_{\mathrm{seq}} \left[r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow}\right] = \mathcal{O}^\epsilon \oplus r_2 \cdot r_2^\star \oplus r_2^\star \cdot r_3 \cdot \mathcal{O}^{\Uparrow}$$

$$\mathrm{s}_{\mathrm{while}} \left[\mathcal{O}^\epsilon \oplus r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow}\right] \triangleq \top \text{ if } (r_2 \neq \varnothing \text{ or } (r_3 \neq \varnothing \text{ and } r_3 \neq \mathcal{O}^\epsilon))$$

Figure 7.3: The tree abstract semantics of instructions

Output commands $^{\mathrm{pp}}\texttt{output a}$ enables attackers to observe k different values, where k is the evaluation result of the cardinal abstract semantics for expression `a`. Therefore, the output regular specification is the concatenation of the combinatorial class $\mathcal{O}^k$ to the input regular specification.

The abstract semantics of conditionals computes both the output regular specification $e_1^\sharp$ of the then-branch and the output regular specification $e_2^\sharp$ of the else-branch. Therefore, the regular specification of attackers' observation due to both conditional branches is the combinatorial sum $e_3^\sharp$ of both $e_1^\sharp$ and $e_2^\sharp$. Finally, the output regular specification is the concatenation of $e_3^\sharp$ to the input regular specification $e^\sharp$. Note that unlike the cardinal abstraction, the tree abstraction does not attempt to gain precision when the conditional guard depends only on *Low* inputs.

In fact, whenever the conditional guard depends only on *Low* inputs, the tree abstraction can keep both specifications that result from both conditional branches separate. This can be achieved by letting the tree abstraction compute a set of regular specifications describing attackers' observations rather than a single regular specification, which amounts to a form of selective trace partitioning [MR05, RM07]. Since the precision of the cardinal abstraction can also benefit from such an improvement, we leave off building a selective trace partitioning abstract domain on top of both the cardinal abstraction and the tree abstraction as future work.

Loop instructions. The abstract semantics of loops computes a regular specification $e_2^\sharp$ of attackers' observation for the loop body `c`. Note that the regular specification $e_2^\sharp$ of attackers' observations is computed by relying on the abstract environment $\varrho_2^\sharp = \mathrm{lfp}^{\dot{\subseteq}\otimes}_{\varrho^\sharp} [\![c]\!]^\sharp$ which determines the maximum cardinal number of values variables may take, for each iteration of the loop body. Therefore, $e_2^\sharp$ represents the observations attackers make for each iteration of the loop body `c`. Therefore, after the execution of the loop, attackers may either observe $\mathcal{O}^\epsilon$ if the loop body never executes, or $e_2^\sharp$ if the loop body executes once, or $e_2^\sharp \cdot e_2^\sharp$ if the loop body executes twice... Consequently, attackers observations after the execution of the loop can be described by the regular specification $(e_2^\sharp)^\star$. However, this sequence construction is not defined when $e_2^\sharp$ contains an observation of size 0. The normal form for regular specifications enables us to decide when such a construction is well-defined, and also bypasses the use of the sequence rule when it is possible.

The most general case where the regular specification $(e_2^\sharp)^\star$ is well defined corresponds to the case where $e_2^\sharp = r_2 \oplus r_3 \cdot \mathcal{O}^{\Uparrow}$, meaning that r_1 is the empty regular specification $\varnothing$ rather than the combinatorial class $\mathcal{O}^\epsilon$ representing

an empty observation. Indeed, our normal form guarantees that neither r_2 nor $r_3 \cdot \mathcal{O}^{\Uparrow}$ contains an observation of size 0. Therefore, in this case the output regular specification of the loop concatenates the input regular specification to $\mathrm{s}_{\mathrm{seq}}\left[e_2^\sharp\right]$, which is a simplification of the sequence construction $(e_2^\sharp)^\star$ in order to keep the regular specification under normal form.

In all other cases, the sequence construction is not defined. However, we can consider 2 more cases where we can actually define a regular specification for attackers' observations. Indeed, if the loop body outputs nothing ($e_2^\sharp = \mathcal{O}^\epsilon$), then the execution of the loop might terminate, leading attackers to observe nothing, or never terminate which leads attackers to observe a silent divergence. Therefore, in this case we can conclude that the loop enables attackers to observe either an empty trace ($\mathcal{O}^\epsilon$) or divergence ($\mathcal{O}^{\Uparrow}$).

Similarly, if the loop body outputs either nothing or silently diverges ($e_2^\sharp = \mathcal{O}^\epsilon \oplus \mathcal{O}^{\Uparrow}$), then the loop may also exit without any outputs ($\mathcal{O}^\epsilon$), or diverge silently ($\mathcal{O}^{\Uparrow}$).

In the remaining case, when the loop body might allow attackers to observe an empty trace, in addition to other observations different from a silent divergence, the analysis halts since it cannot conclusively describe attackers' observations at this level of abstraction.

```
1  i := 0;
2  while (i ≤ 2^κ − 1){
3    if (secret == i){
4      skip;
5    }
6    else {
7      output 2;
8    }
9    i := i+1;
10 }
11 stop;
```

Listing 7.2: An example program causing the analysis to halt

For instance, let us consider the example program in Listing 7.2, where the only input is variable `secret`, which is *High*. This program leaks completely the value of variable secret. Indeed, the leaves of attackers' observation tree are given by:

$$\{2 \cdot \Downarrow, 2 \cdot 2 \cdot \Downarrow, \ldots, \underbrace{2 \cdot 2 \cdot \ldots \cdot 2}_{2^\kappa \text{ times}} \cdot \Downarrow\}$$

Therefore, min-capacity is given by $\mathcal{ML} = \log_2(2^\kappa) = \kappa$, meaning that this program leaks all bits of variable `secret`.

The regular specification computed for the loop body is given by $\mathcal{O}^\epsilon \oplus \mathcal{O}^1$, which describes the fact that the loop body allows attackers to observe either an empty trace, or 1 undefined value. However, without additional knowledge about the loop semantics, the tree abstraction can only conclude that attackers may make infinitely many observations of size n, for each $n \geq 1$. Indeed, in the case where attackers may only make an observation of size 1 for instance, then the loop might execute i times without outputting anything, then output 1 undefined value at iteration $i + 1$, then output nothing during all subsequent iterations, for all $i \in \mathbb{N}$.

Analysing the leakage for looping constructs is complex and tedious, as illustrated by Malacaria's risk assessment [Mal07, Mal10] for looping constructs in batch-job programs. We also share Malacaria's enthusiasm for future static analyses that would require combining different techniques from theorem proving, model checking as well as quantitative static analyses in order to precisely estimate the leakage of loops.

Example programs. Let us illustrate the results of the tree abstraction on the example programs introduced so far, in both Chapters 5 and 7.

Considering the first example program in Listing 7.1, presented in Section 7.1, Listing 7.3 illustrates the results of the tree abstraction. Therefore, the computed regular specification for attackers' observations is given by:

$$(\mathcal{O}^2 \oplus \mathcal{O}^2) \cdot \mathcal{O}^{\Uparrow}$$

Additionally, recalling both example programs introduced in Section 5.2, Listings 7.4 and 7.5 illustrate the results of the tree abstraction on both programs introduced in Listings 5.1 and 5.2.

Both example programs in Listings 5.1 and 5.2 introduce the motivations behind our investigations, in Chapter 5, of a quantitative security property that relaxes termination-insensitive non-interference (TINI), while providing the same security guarantees wrt. attackers guessing the secret in one try. These investigations led us to introduce Definition 19 of RS, in order to quantify information leaks wrt. attackers guessing the secret. Unlike TINI, the quantitative security property RS does indeed deem both programs as equivalent wrt. a one-try attack scenario, since they both have the same min-capacity $\mathcal{ML}$. The tree abstraction also enables us to computes the same upper-bound on min-capacity for both programs. Indeed, the computed regular specifications enable us to represent for instance an abstract attackers' observation tree for both programs, as illustrated by Figure 7.4. Thus,

```
0  // (O^ϵ, [x ↦ ({pp0}, 1); public ↦ ({pp0}, 1); secret ↦ ({pp0}, 2^κ)])
1  if (public == secret){
2    x := 1; // (O^ϵ, [x ↦ ({pp2}, 1); ...])
3  }
4  else {
5    x := 0; // (O^ϵ, [x ↦ ({pp5}, 1); ...])
6  }
7  output x; // (O^2, [x ↦ ({pp2, pp5}, 2); ...])
8  while (public > secret){
9    skip;
10 } // (O^2 ⊕ O^2 · O^⇑, [x ↦ ({pp2, pp5}, 2); ...])
11 stop; // ((O^2 ⊕ O^2) · O^⇑, [x ↦ ({pp2, pp5}, 2); ...])
```

Listing 7.3: The program in Listing 7.1, annotated by the results of the tree abstraction

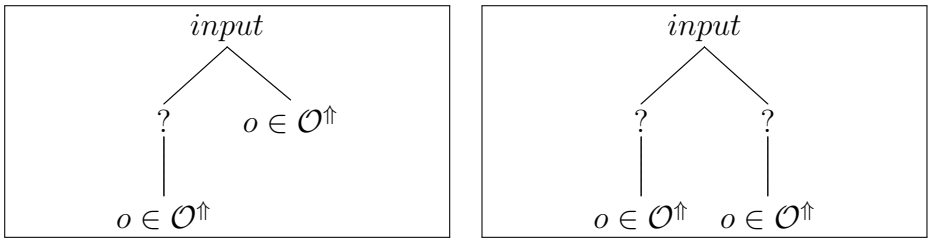
```
0  // (O^ϵ, [input ↦ ({pp0}, 1); secret ↦ ({pp0}, 2^κ)])
1  if (input == secret){
2    skip; // (O^ϵ, ...)
3  }
4  else {
5    while (true){
6      skip;
7    } // (O^ϵ ⊕ O^⇑, ...)
8  } // (O^ϵ ⊕ O^⇑, ...)
9  output 1; // (O^1 ⊕ O^⇑, ...)
10 stop; // ((O^1 ⊕ ϵ) · O^⇑, ...)
```

Listing 7.4: The program in Listing 5.1, annotated with the results of the tree abstraction

```
0 // (O^ϵ, [input ↦ ({pp0}, 1); secret ↦ ({pp0}, 2^κ)])
1 if (input == secret){
2   output 1; // (O^1, ...)
3 }
4 else {
5   output 0; (O^1, ...)
6 } (O^1 ⊕ O^1, ...)
7 stop; ((O^1 ⊕ O^1) · O^⇑, ...)
```

Listing 7.5: The program in Listing 5.2, annotated with the results of the tree abstraction

we can deduce that min-capacity for both programs is upper-bounded by $\mathcal{ML} \leq 1$ *bit*, which corresponds in this case to a tight bound.



(a) Listing 5.1 (b) Listing 5.2

Figure 7.4: Abstract attackers' observation trees for both programs in Listings 7.4 and 7.5

Additionally, Listing 7.6 illustrates the results of the tree abstraction on the example program in Listing 5.3.

The tree abstraction computes the following regular specification of attackers' observation trees:

$$\left(\mathcal{O}^{\epsilon} \oplus \mathcal{O}^{1} \cdot (\mathcal{O}^{1})^{\star} \oplus (\mathcal{O}^{1})^{\star}\right) \cdot \mathcal{O}^{\Uparrow}$$

Intuitively, the regular specification $\mathcal{O}^{\epsilon} \cdot \mathcal{O}^{\Uparrow}$ accounts for observations attackers make if execution of the program never enters the loop body. Moreover, the regular specification $\mathcal{O}^{1} \cdot (\mathcal{O}^{1})^{\star} \cdot \mathcal{O}^{\Uparrow}$ accounts for observations attackers make when the loop iterates many times, outputting during each iteration

```
0 // (O^ϵ, [i ↦ ({pp0}, 1); max ↦ ({pp0}, 1); secret ↦ ({pp0}, 2^κ)])
1 while (i < max){ // (O^ϵ, [i ↦ ({pp0, pp6}, 1); ...])
2   while (i == secret){
3     skip; // (O^ϵ, [i ↦ ({pp0, pp6}, 1); ...])
4   } // (O^ϵ ⊕ O^⇑, [i ↦ ({pp0, pp6}, 1); ...])
5   output i; // (O^1 ⊕ O^⇑, [i ↦ ({pp0, pp6}, 1); ...])
6   i := i + 1; // (O^1 ⊕ O^⇑, [i ↦ ({pp0, pp6}, 1); ...])
7 } // (O^ϵ ⊕ O^1 · (O^1)^⋆ ⊕ (O^1)^⋆ · O^⇑, [i ↦ ({pp0, pp6}, 1); ...])
8 stop; // ((O^ϵ ⊕ O^1 · (O^1)^⋆ ⊕ (O^1)^⋆) · O^⇑, [i ↦ ({pp0, pp6}, 1); ...])
```

Listing 7.6: The program in Listing 5.3, annotated with the results of the tree abstraction

only 1 undefined value, then terminates. Finally, the regular specification $(\mathcal{O}^1)^\star \cdot \mathcal{O}^{\Uparrow}$ accounts for observations attackers make when the loop iterates many times, outputting during each iteration only 1 undefined value, then silently diverges.

In the next section, we leverage the regular specification of attackers' observations in order to over-approximate min-capacity for programs with intermediate outputs, as well as derive sufficient conditions to prove that programs satisfy the RS security property.

7.5 Over-approximating Min-capacity

The regular specification computed by the tree abstraction describes the observations attackers can make. Therefore, it also enables us to upper-bound min-capacity.

By relying on both Theorem 8 and Definition 25, the computed regular specification $\mathcal{C}$ systematically translates into a generating function $C(z)$, whose coefficients C_n are equal to the cardinal number of attackers' observations that have size n. Finally, by Theorem 4 that characterizes min-capacity for deterministic programs, assuming a sufficiently large bound b on the length of output sequences attackers observe, we can upper-bound min-capacity by the logarithm of the cardinal number of attackers' observations whose size is less or equal to the bound b on attackers' observations:

$$
\begin{aligned}
\mathcal{ML} &= \max_{\varrho_{L_0} \in \Sigma_{L_0}} \log_2 \left| Leaves(\mathcal{T}_b^{\varrho_{L_0}}) \right| \\
&= \log_2 \left(\max_{\varrho_{L_0} \in \Sigma_{L_0}} \sum_{n=0}^{b} \left| \{ t \in Leaves(\mathcal{T}_b^{\varrho_{L_0}}) : |t| = n \} \right| \right) \\
&\leq \log_2 \left(\sum_{n=0}^{b} \max_{\varrho_{L_0} \in \Sigma_{L_0}} \left| \{ t \in Leaves(\mathcal{T}_b^{\varrho_{L_0}}) : |t| = n \} \right| \right) \\
&\leq \text{(Assuming the specification of attackers' observations is sound)} \\
&\quad \log_2 \left(\sum_{n=0}^{b} C_n \right) \qquad (7.10)
\end{aligned}
$$

Example programs. Let us consider for instance the regular specification $(\mathcal{O}^2 \oplus \mathcal{O}^2) \cdot \mathcal{O}^{\Uparrow}$, computed in Listing 7.3. This regular specification translates into the following OGF:

$$C = (\mathcal{O}^2 \oplus \mathcal{O}^2) \cdot \mathcal{O}^{\Uparrow} \implies C(z) = 4z^2$$

Therefore, assuming the bound b is greater than 2, we can upper-bound min-capacity by: $\mathcal{ML} \leq \log_2(4) = 2\ bits$.

In general, recall that the OGF of a regular specification is a rational function. Therefore, there exists two relatively prime polynomials $N(z)$ and $D(z)$ such that:

$$C(z) = \frac{N(z)}{D(z)}$$

Therefore, we are going to assume that the bound b on attackers' observations is greater than the degree $n_0 = \max(0, deg(N) - deg(Q))$. Otherwise, we would have to truncate the computed regular specification before computing its generating function. This assumption is without loss of generality since, eventually, we want to quantify the leakage with respect to polynomial time attackers: the bound b is assumed to be large enough, yet polynomial in the size of the secret.

Table 7.1 summarizes the results of the tree abstraction, as well as the resulting upper-bounds over min-capacity for the example programs in Listings 7.3 to 7.5. In all these programs, no output instruction occurs inside a loop. As a consequence, the regular specification of attackers' observations does not involve the sequence construction *, and the attackers' observations each regular specification describes are finite. Therefore, the growth of min-capacity does not depend on the bound b of attackers' observations, and these programs can be deemed secure wrt. RS as long as the computed

upper-bound over min-capacity grows strictly slower than the size of the secret.

Table 7.1: Computing upper-bounds over min-capacity $\mathcal{ML}$ for the programs in Listings 7.3 to 7.5

Listing	Regular specification	OGF	Upper-bound $\mathcal{ML} \leq \log_2(\sum_{n=0}^{b} C_n)$
7.3	$(\mathcal{O}^2 \oplus \mathcal{O}^2) \cdot \mathcal{O}^{\Uparrow}$	$4z^2$	$\log_2(4) = 2\ bits$
7.4	$(\mathcal{O}^1 \oplus \epsilon) \cdot \mathcal{O}^{\Uparrow}$	$z + z^2$	$\log_2(2) = 1\ bit$
7.5	$(\mathcal{O}^1 \oplus \mathcal{O}^1) \cdot \mathcal{O}^{\Uparrow}$	$2z^2$	$\log_2(2) = 1\ bit$

Quantifying the leakage for polynomial time attackers. For programs where output instructions occur inside loops, the attackers' observations that the computed regular specification describes may be infinite. Therefore, in this case, we want to asymptotically estimate min-capacity in order to prove security wrt. the quantitative security property RS.

Let us for instance consider the regular specification that is computed in Listing 7.6 for the program in Listing 5.3:

$$\left(\mathcal{O}^{\epsilon} \oplus \mathcal{O}^1 \cdot (\mathcal{O}^1)^{\star} \oplus (\mathcal{O}^1)^{\star}\right) \cdot \mathcal{O}^{\Uparrow}$$

This regular specification translates into the following generating function:

$$\begin{aligned} C(z) &= (1 + \frac{z}{1-z} + \frac{1}{1-z})z \\ &= (\text{By a partial fraction expansion}) \\ &\quad -2 + \frac{2}{1-z} \\ &= \sum_{k=1}^{\infty} 2z^k \end{aligned}$$

Therefore, assuming that the bound b is large enough, by Equation (7.10), an upper-bound over min-capacity is given by:

$$\begin{aligned} \mathcal{ML} &\leq \log_2\left(\sum_{k=1}^{b} 2\right) \\ &= \log_2(2b) \end{aligned}$$

As a consequence, assuming that the bound b is polynomial in the size κ of the variable `secret`, we can deduce that min-capacity for polynomial time attackers grows strictly slower than the size of the secret since $log_2(2b) = o(\kappa)$. Thus, we can also deduce that the program in Listing 7.6 is secure wrt. the quantitative property RS.

Note that the upper-bound $log_2(2b)$ over min-capacity that we compute through the tree abstraction is in the same order of magnitude as the actual min-capacity. Indeed, as we note in Chapter 5, the program in Listing 5.3 is secure wrt. TINI, and its min-capacity is bounded by $log_2(b+1)$.

Necessary conditions to prove security wrt. Relative Secrecy. Let us assume a regular specification $\mathcal{C}$ of attackers' observations that is computed through the tree abstraction. Then, this regular specification translates into a rational function:

$$C(z) = \frac{N(z)}{D(z)} = Q(z) + \sum_{\alpha, r_\alpha} \frac{c_{\alpha, r_\alpha}}{(z-\alpha)^{r_\alpha}}$$

By Theorem 9, we know that the poles of a generating function define the asymptotic behaviour of its counting sequence. However, instead of the asymptotic behaviour of the counting sequence $(C_n)_{n\geq 0}$, we are actually interested in finding the asymptotic behaviour of the sum $\sum_{n=0}^{b} C_n$, in order to upper-bound min-capacity by Equation (7.10).

Interestingly, the sum $\sum_{k=0}^{b} C_k$ is actually the counting sequence of the intermediate generating function $A(z)$ defined as follows:

$$A(z) \triangleq C(z) \times \frac{1}{1-z}$$

Indeed, with such a definition of the generating function $A(z)$, the counting sequence A_b is equal to the sum $\sum_{k=0}^{b} C_k$:

$$\begin{aligned} A(z) &= C(z) \times \frac{1}{1-z} \\ &= \left(\sum_{k=0}^{\infty} C_k z^k\right) \times \left(\sum_{n=0}^{\infty} z^n\right) \\ &= \sum_{n=0}^{\infty} \left(\sum_{k=0}^{n} C_k\right) z^n \end{aligned}$$

Therefore, studying the singularities of the intermediate generating function $A(z)$ enables us to estimate the counting sequence A_b, thus the growth of min-capacity.

Since the poles of the generating function $A(z)$ are the poles of $C(z)$ and the pole at $z = 1$, we can derive a first condition that is necessary to prove that min-capacity grows strictly slower than the size of the secret using the computed regular specification $\mathcal{C}$.

We introduce this necessary condition on the computed regular specification of attackers' observations in Corollary 5. Essentially, if the poles of $C(z)$ with the smallest modulus – the ones that contribute the most to the growth of the counting sequence A_b – have a modulus μ in the open interval $]0, 1[$ of real numbers, then the upper-bound over min-capacity cannot enable us to prove that the program is secure wrt. RS. Indeed, in such a case, assuming that α_j ranges through the poles of $A(z)$ and assuming a sufficiently large bound b, there exist polynomials $\{F_j\}_j$ by Theorem 9 such that the counting sequence A_b is given by:

$$A_b = \sum_{j=1}^{m} F_j(b)\alpha_j^{-b}$$

Therefore, if the poles with the smallest modulus have a modulus in the open interval $]0, 1[$, then the upper-bound over min-capacity $\log_2(A_b)$ does not grow strictly slower than the size of the secret when the bound b on attackers' observations is polynomial in the size of the secret. In such a case, the program either does not comply with RS, or the tree abstraction is not precise enough to enable the computation of a tight bound that proves compliance with RS.

Corollary 5 (Necessary condition to prove RS)**.**
Let $\mathcal{C}$ be a regular specification describing attackers' observations for a program P. If the computed bound over min-capacity proves that the program P complies with RS, then neither of the generating function $C(z)$'s poles has a modulus in the open interval $]0, 1[$ of real numbers.

Sufficient conditions to prove security wrt. Relative Secrecy. The necessary condition stated in Corollary 5 is not a sufficient condition to prove security wrt. RS. Indeed, consider for instance the program in Listing 7.7, that is a variant of the program in Listing 7.6.

Listing 7.7 is annotated with the results of the tree abstraction. For this example program, the tree abstraction computes the following regular specification:

$$\mathcal{C} = (\mathcal{O}^{\epsilon} \oplus \mathcal{O}^{2^{\kappa}} \oplus \mathcal{O}^{1} \cdot (\mathcal{O}^{1})^{\star} \oplus (\mathcal{O}^{1})^{\star}) \cdot \mathcal{O}^{\Uparrow}$$

```
0  // (O^ϵ, [i ↦ ({pp0}, 1); max ↦ ({pp0}, 1); secret ↦ ({pp0}, 2^κ)])
1  if (max > secret) {
2    output secret;   // (O^{2^κ}, ...)
3  }
4  else {
5    while (i < max){
6      while (i == secret){
7        skip;
8      }
9      output i;
10     i := i + 1;
11   } // (O^ϵ ⊕ O^1 · (O^1)^⋆ ⊕ (O^1)^⋆ · O^⇑, ...)
12 } // (O^ϵ ⊕ (O^{2^κ} ⊕ O^1 · (O^1)^⋆) ⊕ (O^1)^⋆ · O^⇑, ...)
13 stop; // ((O^ϵ ⊕ O^{2^κ} ⊕ O^1 · (O^1)^⋆ ⊕ (O^1)^⋆) · O^⇑, ...)
```

Listing 7.7: An example program whose OGF's asymptotic behaviour does not account for the leakage of all the secret

Therefore, the corresponding OGF $C(z)$ is given by:

$$\begin{aligned} C(z) &= \left(1 + 2^\kappa z + \frac{z}{1-z} + \frac{1}{1-z}\right) z \\ &= -2 + 2^\kappa z^2 + \frac{2}{1-z} \end{aligned}$$

Moreover, the intermediate generating function $A(z) \triangleq C(z)/(1-z)$ is given by:

$$\begin{aligned} A(z) &= \left(-2 + 2^\kappa z^2 + \frac{2}{1-z}\right) \times \frac{1}{1-z} \\ &= (\text{By a partial fraction expansion}) \\ &\quad - 2^\kappa - 2^\kappa z + \frac{2}{(z-1)^2} + \frac{2-2^\kappa}{z-1} \end{aligned}$$

Since the intermediate generating function $A(z)$ has only one pole at $z = 1$ with multiplicity $r = 2$, then by Theorem 9 and Equation (7.9), the counting sequence $(A_n)_{n\geq 0}$'s asymptotic behaviour is given by:

$$A_b \sim 2b \tag{7.11}$$

This last asymptotic expansion might lead us to deduce that the growth of min-capacity is at most logarithmic in the size of the secret, assuming that the bound b on attackers' observations is polynomial in the size of the secret. Yet, let us find an explicit formula for the counting sequence A_n in order to illustrate why this deduction is false. Indeed, the counting sequence A_n is given by:

$$\begin{aligned}
A_n &= [z^n]A(z) \\
&= [z^n](-2^\kappa - 2^\kappa z) + [z^n]\frac{2}{(1-z)^2} + [z^n]\frac{2-2^\kappa}{z-1} \\
&= (\text{Assuming } n \geq 2) \\
&\quad [z^n]\frac{2}{(1-z)^2} + [z^n]\frac{2-2^\kappa}{z-1} \\
&= (\text{By Equation (7.7)}) \\
&\quad 2\binom{n+1}{1} - (2-2^\kappa)\binom{n}{0} \\
&= 2n + 2^\kappa
\end{aligned}$$

Therefore, an explicit formula for an upper-bound over min-capacity $\mathcal{ML}$ is given by:

$$A_b = 2b + 2^\kappa \tag{7.12}$$

Note that the main difference between both Equations (7.11) and (7.12) is the term 2^κ. In fact, the asymptotic expansion in Equation (7.11) fails to capture the implicit relation between the bound b on attackers' observations and the size of the secret κ. Indeed, while Equation (7.11) simplifies the asymptotic expansion by considering the term 2^κ as a constant, the bound b actually depends on κ, since we assume that b is polynomial in the size of the secret κ. This is exactly why we cannot rely on the necessary condition stated in Corollary 5, in order to prove that a program is secure wrt. RS. Instead, we must find an explicit formula for the counting sequence A_n, as illustrated by Equation (7.12), then try to prove that min-capacity grows strictly slower than the size of the secret. For this example program in Listing 7.7, the explicit bound that we find for min-capacity is given by:

$$\begin{aligned}
\mathcal{ML} &\leq \log_2(A_b) \\
&= \log_2(2b + 2^\kappa)
\end{aligned}$$

Thus, we can deduce that the computed bound over min-capacity does not prove that min-capacity grows strictly slower than the size of the secret κ; a fitting result since the program in Listing 7.7 does indeed leak all the secret.

Therefore, while the necessary condition in Corollary 5 might help us to quickly disprove the compliance of the computed regular specification for attackers' observations wrt. RS, we need to rely on an explicit formula of the upper-bound over min-capacity in order to prove that a program complies with RS. Interestingly, along the way of proving Theorem 9, we also exhibited an explicit formula for the counting sequence C_n in Equation (7.8).

Consequently, assuming that the regular specification $\mathcal{C}$ of attackers' observations translates into a rational OGF decomposed as follows:

$$C(z) = Q(z) + \sum_{\alpha} \sum_{r=1}^{r_\alpha} \frac{c_{\alpha,r_\alpha}}{(z-\alpha)^{r_\alpha}}$$

where α ranges over the poles of $C(z)$, and r_α denotes the multiplicity of α, then an upper-bound over min-capacity is given by:

$$\begin{aligned}
A_b &= \sum_{n=0}^{b} C_n \\
&= \text{(By using the rational fraction decomposition in Section 7.5)} \\
&\quad \sum_{n=0}^{b} [z^n]Q(z) + \sum_{n=0}^{b} [z^n] \left(\sum_{\alpha} \sum_{r=1}^{r_\alpha} \frac{c_{\alpha,r_\alpha}}{(z-\alpha)^{r_\alpha}} \right) \\
&= \sum_{n=0}^{b} [z^n]Q(z) + \sum_{n=0}^{b} \sum_{\alpha} \sum_{r=1}^{r_\alpha} [z^n] \frac{c_{\alpha,r_\alpha}}{(z-\alpha)^{r_\alpha}} \\
&= \text{(By Equation (7.8))} \\
&\quad \sum_{n=0}^{b} [z^n]Q(z) + \sum_{n=0}^{b} \sum_{\alpha} \sum_{r=1}^{r_\alpha} c_{\alpha,r} \frac{(-1)^r}{\alpha^r} \binom{n+r-1}{r-1} \alpha^{-n} \\
&= \sum_{n=0}^{b} [z^n]Q(z) + \sum_{\alpha} \sum_{r=1}^{r_\alpha} (-1)^r c_{\alpha,r} \sum_{n=0}^{b} \binom{n+r-1}{r-1} \alpha^{-n-r} \\
&= \text{(By assuming } b \geq deg(Q)\text{)} \\
&\quad Q(1) + \sum_{\alpha} \sum_{r=1}^{r_\alpha} (-1)^r c_{\alpha,r} \sum_{n=0}^{b} \binom{n+r-1}{r-1} \alpha^{-n-r} \qquad (7.13)
\end{aligned}$$

Therefore, the explicit formula in Equation (7.13) enables us to derive sufficient conditions to prove that min-capacity grows strictly slower than the size of the secret, by proving that the computed bound $\log_2(A_b)$ grows strictly slower than the size of the secret. Indeed, the counting sequence A_b is guaranteed to be polynomial in the size of the secret, if no pole α has a modulus in the open interval $]0, 1[$, and $Q(1)$ as well as the coefficients

$c_{\alpha,r}$ are all negligible in the size of the secret. In this case, the upper-bound $log_2(A_b)$ over min-capacity is guaranteed to be logarithmic in the size of the secret, and therefore grows strictly slower than the size of the secret. Corollary 6 introduces these sufficient conditions in order to prove that a program satisfies RS, by relying on the computed regular specification of attackers' observations.

Corollary 6 (Sufficient conditions to prove RS)**.**
Let $\mathcal{C}$ be a regular specification describing attackers' observations for a program P, and $C(z)$ be its rational generating function decomposed as follows:

$$C(z) = Q(z) + \sum_{\alpha} \sum_{r=1}^{r_\alpha} \frac{c_{\alpha,r_\alpha}}{(z-\alpha)^{r_\alpha}}$$

The program P satisfies RS if:

1. *no pole α has a modulus in the open interval $]0,1[$ of real numbers, and*
2. *$Q(1)$ as well as the coefficients $c_{\alpha,r}$ are all negligible in the size of the secret.*

7.6 Summary

In this chapter, we build on the results of the cardinal abstraction, introduced in Chapter 6, in order to quantify the leakage for programs with intermediate outputs. We propose the tree abstraction, a static analysis approach aimed at computing a regular specification of attackers' observations.

Once the tree abstraction computes a regular specification of attackers' observations, we rely on the framework of analytic combinatorics [FS09] in order to translate it systematically into a generating function. This generating function embodies counting information about the cardinal number of observations attackers may make. Thus, it enables us to derive an upper-bound over min-capacity when attackers are allowed to observe intermediate steps of computation, by Theorem 4.

Finally, we derive a necessary condition on the computed generating function in order for min-capacity to grow strictly slower than the size of the secret. This necessary condition may quickly disprove the compliance of the regular specification of attackers' observations with the quantitative security property RS, introduced in Chapter 5. Additionally, we also derive sufficient conditions on the computed generating function in order to guarantee that min-capacity grows strictly slower than the size of the secret, assuming

polynomial time attackers. These sufficient conditions enable us to prove that the analysed programs comply with RS, by relying on a rational expansion of the generating function and a study of its singularities.

Future work involves improving the precision of both the cardinal abstraction and the tree abstraction through the combination of multiple static analysis techniques. Indeed, as the tree abstraction is parametrized by the cardinal abstraction, improving the latter also enhances the precision of the first. Additionally, the tree abstraction can also be improved by taking advantage of additional semantics information, provided through loop bounds worst-case analysis as well as deductive verification techniques for instance.

Chapter 8

Conclusions and Perspectives

This chapter summarizes the contributions of this thesis and discusses future work.

8.1 Summary

This thesis develops program analysis techniques for qualitative and quantitative information flow.

The first part on qualitative information flow formalizes the `PWhile` monitor, a hybrid information flow monitor. The `PWhile` monitor extends existing flow-sensitive monitors [LGBJS06, RS10] in order to support a language with pointers and aliasing. It soundly enforces a termination-insensitive non-interference (TINI) property, preventing the leakage of sensitive information.

We also inline the `PWhile` monitor through a program transformation. This program transformation soundly preserves the initial behaviour of programs and reproduces the exact semantics of the `PWhile` monitor. The transformed program allows the enforcement of TINI by relying on either static or dynamic analysis. Indeed, running the transformed program prevents illegal information flows since the program self-monitors its information flows. Additionally, off-the-shelf abstract interpretation tools can also enable the verification of TINI, without implementing new abstract domains. We implemented the inlining approach as a plug-in of the Frama-C framework [CKK+12, KKP+15]. This implementation supports a large subset of the C language including arrays, pointer arithmetic, function calls, as well as complex structures. Support for type casts is missing, however.

Non-interference is a baseline security policy that is often not the desired security policy in practice. Some programs, that inherently leak a small

amount of sensitive information, cannot comply with non-interference. Therefore, we propose relative secrecy (RS) as a quantitative security property. RS is more permissive than TINI, while still providing the same guarantees wrt. polynomial time attackers guessing the secret. RS builds on min-capacity, a quantitative information metric proposed by Smith [Smi09, Smi11]. RS is also inspired by the work for Volpano and Smith [VS00], as well as Askarov et al. [AHSS08].

The cardinal abstraction is the first step toward the verification of RS. This abstract domain computes the cardinal number of values variables may take. The cardinal abstract domain enables quantifying information flow wrt. min-capacity for batch-job programs. Since it does not rely on traditional safety abstract domains approximating the set of reachable states, it efficiently computes an over-approximation of min-capacity for programs that accept both *Low* and *High* inputs.

The tree abstraction builds on the cardinal abstraction in order to quantify information flow for programs that may also output intermediate steps of computation. It draws on analytic combinatorics [FS09] in order to construct a regular specification describing attackers' observations. This regular specification directly translates into a generating function. We also derive sufficient conditions on the generating function in order to prove that a program satisfies the quantitative security property RS.

8.2 Future Work

The inlining approach of the `PWhile` monitor we propose relies on the type of variables in order to assign them shadow variables. While this approach scales to complex constructions of the C language, type casts are challenging. Future work will address this problem by designing static analysis techniques in order to guide the inlining approach in presence of type casts, and support as much as possible such constructs.

While our inlining approach targets an information flow monitor, we also believe it sets the ground for a more generic approach of inlining security monitors. Such a framework, provided with a user defined micro-policy [dAGH+15], may automatically inline the corresponding monitor in order to support a wider set of enforceable security policies [Sch00, LBW02].

Beyond proving the soundness of the cardinal abstract domain, one advantage of formalizing this static analysis as an abstract domain resides in the potential to improve its precision, by relying on a large body of research dedicated to abstract interpretation. Future work involves enhancing both the cardinal abstract domain and the tree abstraction by relying on

traditional abstract domains. Additionally, some abstract domains initially designed to reason on safety properties, may also need tweaking in order to fit the larger class of hyperproperties [CS08, CS10].

Bibliography

[ACPS12] Mário S Alvim, Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Geoffrey Smith. Measuring Information Leakage Using Generalized Gain Functions. In *2012 IEEE 25th Computer Security Foundations Symposium (CSF)*, pages 265–279. IEEE, 2012.

[AF09] Thomas H Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. *SIGPLAN Notices*, 44(8):20–31, August 2009.

[AF10] Thomas H. Austin and Cormac Flanagan. Permissive Dynamic Information Flow Analysis. In *PLAS '10: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–12. ACM, 2010.

[AHSS08] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Computer Security - ESORICS 2008*, volume 5283 of *Lecture Notes in Computer Science*, 2008.

[ASTT13a] Mounir Assaf, Julien Signoles, Frédéric Tronel, and Eric Totel. Moniteur hybride de flux d'information pour un langage supportant des pointeurs. In *SARSSI - 8ème Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information*, Mont de Marsan, France, September 2013.

[ASTT13b] Mounir Assaf, Julien Signoles, Frédéric Tronel, and Éric Totel. Program Transformation for Non-interference Verification on Programs with Pointers. In *Security and Privacy Protection in Information Processing Systems*, pages 231–244. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2013.

[BBC+14] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Luna, and David Pichardie. System-level non-interference for

constant-time cryptography. *IACR Cryptology ePrint Archive*, page 422, 2014.

[BBJ13] Frederic Besson, Nataliia Bielova, and Thomas Jensen. Hybrid information flow monitoring against web tracking. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 240–254. IEEE, 2013.

[BCP09] Christelle Braun, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Quantitative notions of leakage for one-try attacks. *Electronic Notes in Theoretical Computer Science*, 249:75–91, 2009.

[BDR04] Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 100–114. IEEE, 2004.

[BDR11] Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[BKR09] Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 141–153. IEEE, 2009.

[BL09] Sandrine Blazy and Xavier Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[BPR13] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference Java bytecode verifier. *Mathematical Structures in Computer Science*, 23(05):1032–1081, October 2013.

[BRW06] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing Timing Leaks Through Transactional Branching Instructions. *Electronic Notes in Theoretical Computer Science*, 153(2):33–55, May 2006.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the second International Symposium on Programming*, pages 106–130. Paris, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, New York, USA, January 1977. ACM Request Permissions.

[CC79a] Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific journal of Mathematics*, 82(1):43–57, 1979.

[CC79b] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979.

[CC94] Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Computer Languages, 1994., Proceedings of the 1994 International Conference on*, pages 95–112. IEEE Comput. Soc. Press, 1994.

[CCF+07] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Programming Languages and Systems*, pages 272–300. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. *POPL*, pages 84–96, 1978.

[CHM07] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Program Analysis Perspective. *Software Engineering and Formal Methods*, pages 233–247, 2012.

[Cla05] David Clark. Quantitative Information Flow, Relations and Polymorphic Types. *Journal of Logic and Computation*, 15(2):181–199, 2005.

[CLVS08] Roberto Capizzi, Antonio Longo, V N Venkatakrishnan, and A Prasad Sistla. Preventing Information Leaks through Shadow Executions. In *Annual Computer Security Applications Conference (ACSAC)*, pages 322–331. IEEE, 2008.

[CMS09] Michael R Clarkson, Andrew C Myers, and Fred B Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 17:655–701, 2009.

[CN10] Andrey Chudnov and David A Naumann. Information Flow Monitor Inlining. In *2010 IEEE 23rd Computer Security Foundations Symposium (CSF)*, pages 200–214. IEEE, 2010.

[Coh77] Ellis Cohen. Information transmission in computational systems. In *SOSP '77: Proceedings of the sixth ACM symposium on Operating systems principles*, pages 133–139, New York, New York, USA, November 1977. ACM Request Permissions.

[Cou99] Patrick Cousot. The calculational design of a generic abstract interpreter. *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, 173:421–506, 1999.

[CP10] David Cachera and David Pichardie. A Certified Denotational Abstract Interpreter. In *Interactive Theorem Proving*, pages 9–24. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2010.

[CS08] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Computer Security Foundations Symposium, 2008. CSF '08. IEEE 21st*, pages 51–65, 2008.

[CS10] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[CT06] Thomas M Cover and Joy A Thomas. *Elements of Information Theory 2nd Edition*. Wiley-Interscience, 2006.

[dACD+14] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Catalin Hritcu, David Pichardie, Benjamin C Pierce, Randy Pollack, and Andrew Tolmach. A

verified information-flow architecture. *POPL*, 49(1):165–178, 2014.

[dAGH+15] Arthur Azevedo de Amorim, Maxime Dénes1 Nick Giannarakis, Catalin Hritcu, Benjamin C Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: A framework for verified, tag-based security monitors. In *IEEE Symposium on Security and Privacy*, 2015. To appear.

[DD77] Dorothy Elizabeth Robling Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[DDG76] Dorothy E Denning, Peter J Denning, and G Scott Graham. On the derivation of lattice structured information flow policies. Technical report, Purdue University, march 1976. CSD TR 180.

[Den76] Dorothy E Denning. A Lattice Model of Secure Information Flow. *Commun. ACM ()*, 19(5):236–243, 1976.

[Den82] Dorothy Elizabeth Robling Denning. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[DP10] Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. *IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[DS09] Delphine Demange and David Sands. All Secrets Great and Small. In *Programming Languages and Systems*, pages 207–221. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2009.

[Eck10] Peter Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.

[ES13] Barbara Espinoza and Geoffrey Smith. Min-entropy as a resource. *Information and Computation*, 226:57–75, May 2013.

[FGP14] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The Spirit of Ghost Code. *CAV*, 8559(Chapter 1):1–16, 2014.

[FS09] Philippe Flajolet and Robert Sedgewick. *Analytic combinatorics*. cambridge University press, 2009.

[GM82] Joseph A Goguen and José Meseguer. Security Policies and Security Models. *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 1989.

[HS06] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–90, New York, New York, USA, January 2006.

[HS12] Daniel Hedin and Andrei Sabelfeld. A Perspective on Information-Flow Control. *Software Safety and Security*, pages 319–347, 2012.

[Kah87] Gilles Kahn. *Natural semantics*. Springer, 1987.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing*, pages 1–37, January 2015.

[KP00] Dexter Kozen and Maria-Christina Patron. Certification of Compiler Optimizations Using Kleene Algebra with Tests. *Computational Logic*, 1861(Chapter 38):568–582, 2000.

[KR10] Boris Köpf and Andrey Rybalchenko. Approximation and Randomization for Quantitative Information-Flow Analysis. In *CSF '10: Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, pages 3–14. IEEE Computer Society, July 2010.

[KR13] Boris Köpf and Andrey Rybalchenko. Automation of Quantitative Information-Flow Analysis. *SFM*, 7938(Chapter 1):1–28, 2013.

[Lam95] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, pages 600–608, 1995.

[LBW02] Jay Ligatti, Lujo Bauer, and David Walker. *More enforceable security policies*. Foundations of Computer Security Workshop, 2002.

[LBW05] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, February 2005.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[LG07] Gurvan Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. *CSF*, pages 218–232, 2007.

[LG08] Gurvan Le Guernic. Precise dynamic verification of confidentiality. In *Proceedings of the 5th International Verification Workshop in connection with IJCAR 2008, Sydney, Australia, August 10-11, 2008*, 2008.

[LGBJS06] Gurvan Le Guernic, Anindya Banerjee, Thomas P Jensen, and David A Schmidt. Automata-based confidentiality monitoring. In *ASIAN'06: Proceedings of the 11th Asian computing science conference on Advances in computer science: secure software and related issues*. Springer-Verlag, December 2006.

[Mal07] Pasquale Malacaria. Assessing security threats of looping constructs. In *ACM SIGPLAN Notices*, volume 42, pages 225–235. ACM, 2007.

[Mal10] Pasquale Malacaria. Risk assessment of security threats for looping constructs. *Journal of Computer Security*, 18(2):191–228, 2010.

[MC11] Scott Moore and Stephen Chong. Static Analysis for Efficient Hybrid Information-Flow Control. In *2011 IEEE 24th Computer Security Foundations Symposium (CSF)*, pages 146–160. IEEE, 2011.

[McL94] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. *IEEE Symposium on Security and Privacy*, pages 79–93, 1994.

[Min06a] Antoine Miné. Symbolic Methods to Enhance the Precision of Numerical Abstract Domains. In *Verification, Model Checking, and Abstract Interpretation*, pages 348–363. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2006.

[Min06b] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, March 2006.

[MJ08] Jan Midtgaard and Thomas P Jensen. A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. *SAS*, 5079(Chapter 23):347–362, 2008.

[MMHS11] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. Dynamic enforcement of knowledge-based security policies. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 114–128. IEEE, 2011.

[MNZZ01] Andrew C Myers, Nate Nystrom, Lantian Zheng, and Steve Zdancewic. *Jif: Java Information Flow*, May 2001. Software release. http://www.cs.cornell.edu/jif.

[MR05] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP'05: Proceedings of the 14th European conference on Programming Languages and Systems*, pages 5–20, Berlin, Heidelberg, April 2005. Springer-Verlag.

[MRS10] Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly Inlining of Dynamic Security Monitors. In *Security and Privacy – Silver Linings in the Cloud*, pages 173–186. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[MRS12] Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 31(7):827–843, 2012.

[MS11] Ziyuan Meng and Geoffrey Smith. Calculating bounds on information leakage using two-bit patterns. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, page 1. ACM, 2011.

[MSZ06] Andrew C Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 2006.

[Mye99] Andrew C Myers. JFlow: Practical Mostly-Static Information Flow Control. *POPL*, pages 228–241, 1999.

[NNH99] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Science & Business Media, January 1999.

[Plo81] Gordon D Plotkin. A structural approach to operational semantics, 1981.

[Plo04] Gordon D Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program. () 60-*, pages 17–139, 2004.

[PP04] Dominique Perrin and Jean-Éric Pin. *Infinite words : automata, semigroups, logic and games*. Pure and applied mathematics. Academic, London, San Diego (Calif.), 2004.

[PS02] François Pottier and Vincent Simonet. Information flow inference for ML. *POPL*, pages 319–330, 2002.

[Rén61] Alfréd Rényi. On measures of entropy and information. In *the Fourth Berkeley Symposium on Mathematical Statistics and Probability*, 1961.

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007.

[RS10] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. *Computer Security Foundations Symposium, IEEE*, 0:186–199, 2010.

[Sch00] Fred B Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[Sha48] Claude E Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[Sim03] Vincent Simonet. *The Flow Caml System: documentation and user's manual*, July 2003. Software release. http://www.normalesup.org/~simonet/soft/flowcaml/.

[SM03] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[SM04] Andrei Sabelfeld and Andrew C Myers. A Model for Delimited Information Release. In *Software Security - Theories and Systems*, pages 174–191. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2004.

[Smi09] Geoffrey Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FOSSACS '09, pages 288–302, Berlin, Heidelberg, 2009. Springer-Verlag.

[Smi11] Geoffrey Smith. Quantifying information flow using min-entropy. In *Quantitative Evaluation of Systems (QEST), 2011 Eighth International Conference on*, pages 159–167. IEEE, 2011.

[SS09] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5), October 2009.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *Journal in Computer Security*, 4(2-3):167–187, 1996.

[VS97] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 156–168, 1997.

[VS00] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 268–276, New York, New York, USA, January 2000. ACM Request Permissions.

[Wil05] Herbert S Wilf. *generatingfunctionology*. Third Edition. A K Peters/CRC Press, December 2005.

[Win93] Glynn Winskel. The formal semantics of programming languages: an introduction. *The formal semantics of programming languages: an introduction*, February 1993.

[YT10a] Hirotoshi Yasuoka and Tachio Terauchi. On Bounding Problems of Quantitative Information Flow. In *Computer Security – ESORICS 2010*, pages 357–372. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2010.

[YT10b] Hirotoshi Yasuoka and Tachio Terauchi. Quantitative Information Flow - Verification Hardness and Possibilities. *2010 IEEE 23rd Computer Security Foundations Symposium (CSF)*, pages 15–27, 2010.

[YT11] Hirotoshi Yasuoka and Tachio Terauchi. On bounding problems of quantitative information flow. *Journal of Computer Security*, 19(6):1029–1082, 2011.

[Zda02] Stephan Arthur Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.

[ZM01] Steve Zdancewic and Andrew C Myers. Robust declassification. In *Computer Security Foundations Workshop, 2001. Proceedings. 14th IEEE*, pages 15–23, 2001.

Acronyms

OGF ordinary generating function. xx, 123, 125, 126, 127, 128, 129, 130, 131, 133, 142, 145, 148

RS relative secrecy. 3, 57, 73, 74, 80, 81, 82, 85, 119, 138, 141, 142, 143, 145, 147, 148, 149, 151, 152

TINI termination-insensitive non-interference. xi, xvii, xix, 3, 7, 8, 9, 21, 22, 25, 35, 36, 50, 51, 52, 57, 58, 59, 60, 61, 72, 73, 74, 75, 81, 82, 138, 144, 151, 173

TSNI termination-sensitive non-interference. 7, 9, 58

Appendices

Appendix A

Monitor Semantics

We prove that our monitor is sound with respect to TINI. The proofs in Appendices A and B are structured using Lamport's style [Lam95].

We first take advantage of the symmetry of both runs by introducing a partial order relation 3 on security memories Γ. Definition 26 is equivalent to Definition 2 when both input security memories and both execution contexts are equal in both runs.

Definition 3 (Less restrictive up to label s ($\sqsubseteq_s$))**.**
For all $s \in \mathbb{S}$*, for all* Γ_1, Γ_2*,*

Γ_2 *is less restrictive than* Γ_1 *up to security label* s *(*$\Gamma_2 \sqsubseteq_s \Gamma_1$*) iff :*

$$\textit{for all } l \in \mathit{Loc}, \Gamma_1(l) \sqsubseteq s \implies \Gamma_2(l) \sqsubseteq \Gamma_1(l).$$

Definition 26 (Termination-insensitive non-interference)**.**
For all $c, E, \Gamma_1, \Gamma_2, M_1, M_1', M_2, M_2'$*, for all* $s, \underline{pc}_1, \underline{pc}_2 \in \mathbb{S}$*, such that* $E \vdash c, M_1, \Gamma_1, \underline{pc}_1 \Rightarrow M_1', \Gamma_1'$ *and* $E \vdash c, M_2, \Gamma_2, \underline{pc}_2 \Rightarrow M_2', \Gamma_2'$*, then*

$\underline{pc}_2 \sqsubseteq \underline{pc}_1$ *and* $\Gamma_2 \sqsubseteq_s \Gamma_1$ *and* $M_1 \sim^s_{\Gamma_1} M_2$ *implies* $\Gamma_2' \sqsubseteq_s \Gamma_1'$ *and* $M_1' \sim^s_{\Gamma_1'} M_2'$

A.1 Semantics of Expressions

We introduce Lemma 1, which proves that for two s-equivalent memories, if the l-value evaluation of an expression yields a label below s, then it is evaluated to the same location in the second run, and to a label that is less restrictive than the label of the first run.

Lemma 1 (L-value evaluation in s-equivalent memories)**.**
For all environments E*, for all memories* M_1*,* M_2*, for all security memories*

Γ_1, Γ_2, *for all security levels* $s \in \mathbb{S}$, *such that* $\Gamma_2 \sqsubseteq_s \Gamma_1$ *and* $M_1 \sim^s_{\Gamma_1} M_2$, *for all* $a \in Exp$ *such that* $E \vdash a, M_1, \Gamma_1 \Leftarrow l_1, s_1$ *and* $E \vdash a, M_2, \Gamma_2 \Leftarrow l_2, s_2$ *then*

$s_1 \sqsubseteq s \implies l_1 = l_2$ *and* $s_2 \sqsubseteq s_1$.

PROOF:
LET: $E, M_1, M_2, \Gamma_1, \Gamma_2$ and $s \in \mathbb{S}$.
LET: $a \in Exp$ such that $E \vdash a, M_1, \Gamma_1 \Leftarrow l_1, s_1$ and $E \vdash a, M_2, \Gamma_2 \Leftarrow l_2, s2$.
ASSUME: 1. $\Gamma_2 \sqsubseteq_s \Gamma_1$
2. $M_1 \sim^s_{\Gamma_1} M_2$
3. $s_1 \sqsubseteq s$
PROVE: 1. $l_1 = l_2$
2. $s_2 \sqsubseteq s_1$
PROOF SKETCH: By induction on l-value evaluations of expressions.
$\langle 1 \rangle$1. CASE: LV_{id}
$\langle 2 \rangle$1. $s_2 = s_1 = public$
$\langle 3 \rangle$1. Q.E.D.
PROOF: By rule LV_{id}, labels associated to l-values of variables are defined as the bottom of $\mathbb{S}$
$\langle 2 \rangle$2. $E(a) = l_1 = l_2$
$\langle 3 \rangle$1. Q.E.D.
PROOF: By rule LV_{id}. Environment E is the same for both runs.
$\langle 2 \rangle$3. Q.E.D.
PROOF: By $\langle 2 \rangle$1 and $\langle 2 \rangle$2.
$\langle 1 \rangle$2. CASE: LV_{MEM}
LET: l_a, s_l, l, s_r and l'_a, s'_l, l', s'_r such that

$$LV_{MEM}\ \frac{RV\ \dfrac{(\mathbf{1})E \vdash a, M_1, \Gamma_1 \Leftarrow l_a, s_l \qquad M_1(l_a) = ptr(l_1) \qquad \Gamma_1(l_a) = s_r \qquad s_1 = s_l \sqcup s_r}{E \vdash a, M_1, \Gamma_1 \Rightarrow ptr(l_1), s_1}}{E \vdash *a, M_1, \Gamma_1 \Leftarrow l_1, s_1}$$

$$LV_{MEM}\ \frac{RV\ \dfrac{E \vdash a, M_2, \Gamma_2 \Leftarrow l'_a, s'_l \qquad M_2(l'_a) = ptr(l_2) \qquad \Gamma_2(l'_a) = s'_r \qquad s_2 = s'_l \sqcup s'_r}{E \vdash a, M_2, \Gamma_2 \Rightarrow ptr(l_2), s_2}}{E \vdash *a, M_2, \Gamma_2 \Leftarrow l_2, s_2}$$

SUFFICES ASSUME: $s_1 \sqsubseteq s$
PROVE: 1. $s_2 \sqsubseteq s_1$
2. $l_2 = l_1$
$\langle 2 \rangle$1. $s_l \sqsubseteq s$ and $s_r \sqsubseteq s$
$\langle 3 \rangle$1. Q.E.D.

PROOF: By assumption of $\langle 1\rangle 2$ $s_1 = s_l \sqcup s_r \sqsubseteq s$
$\langle 2\rangle 2$. $l'_a = l_a$ and $s'_l \sqsubseteq s_l$
$\langle 3\rangle 1$. Q.E.D.
PROOF: By induction on derivation depth of $*a$, using $\langle 2\rangle 1$ and assumptions 1 and 2
$\langle 2\rangle 3$. $s'_r \sqsubseteq s_r$
$\langle 3\rangle 1$. Q.E.D.
PROOF: By $\langle 2\rangle 1$ and assumption 1: $\Gamma_2(l_a) \sqsubseteq \Gamma_1(l_a) \sqsubseteq s$
$\langle 2\rangle 4$. Q.E.D.
PROOF: By $\langle 2\rangle 2$ and $\langle 2\rangle 3$, $s_2 = s'_l \sqcup s'_r \sqsubseteq s_l \sqcup s_r = s_1$, and $l_1 = l_2$ by assumption 2 and $\langle 2\rangle 1$.
$\langle 1\rangle 3$. Q.E.D.
PROOF: By induction on the evaluation of l-values and $\langle 1\rangle 1$ and $\langle 1\rangle 2$.

Corollary 1 generalizes the result of Lemma 1 to include all expressions.

Corollary 1 (R-value evaluation in s-equivalent memories)**.**
For all environments E, for all memories M_1, M_2, for all security memories Γ_1, Γ_2, for all security levels $s \in \mathbb{S}$, such that $\Gamma_2 \sqsubseteq_s \Gamma_1$ and $M_1 \sim^s_{\Gamma_1} M_2$, for all $a \in Exp$ such that $E \vdash a, M_1, \Gamma_1 \Rightarrow v_1, s_1$ and $E \vdash a, M_2, \Gamma_2 \Rightarrow v_2, s_2$ then

$$s_1 \sqsubseteq s \implies v_1 = v_2 \text{ and } s_2 \sqsubseteq s_1.$$

PROOF:
By induction on r-value evaluations, using lemma 1. □

A.2 Semantics of Instructions

Theorem 1 proves the soundness of our semantics with respect to TINI.

Theorem 1 (Soundness)**.**
The `PWhile` monitor semantics is sound with respect to TINI as introduced in Definition 2.

PROOF:
By induction on the evaluation of instructions.
LET: $E, \Gamma_1, \Gamma_2, M_1, M'_1, M_2, M'_2$ and $\underline{pc}_1, \underline{pc}_2 \in \mathbb{S}$ such that:
$E \vdash c, M_1, \Gamma_1, \underline{pc}_1 \Rightarrow M'_1, \Gamma'_1$ and $E \vdash c, M_2, \Gamma_2, \underline{pc}_2 \Rightarrow M'_2, \Gamma'_2$.
LET: Let $s \in \mathbb{S}$.

ASSUME: 1. $\underline{pc}_2 \sqsubseteq \underline{pc}_1$
2. $\Gamma_2 \sqsubseteq_s \Gamma_1$
3. $M_1 \sim^s_{\Gamma_1} M_2$

PROVE: 1. $\Gamma'_2 \sqsubseteq_s \Gamma'_1$
2. $M'_1 \sim^s_{\Gamma'_1} M'_2$

⟨1⟩1. CASE: Skip
PROOF: By assumptions 2 and 3.

⟨1⟩2. CASE: Assign
LET: $l_1, s_1, v_2, s_2, l'_1, s'_1, v'_2, s'_2$ such that the evaluation of instruction $a_1 := a_2$ in both M_1, Γ_1 and M_2, Γ_2 yield:

$$(Assign)\ \frac{\begin{array}{c} E \vdash a_1, M_1, \Gamma_1 \Leftarrow l_1, s_1 \quad E \vdash a_2, M_1, \Gamma_1 \Rightarrow v_2, s_2 \\ s_{l_1} = s_1 \sqcup s_2 \sqcup \underline{pc}_1 \quad M'_1 = M_1[l_1 \mapsto v_2] \\ \Gamma''_1 = \Gamma_1[l_1 \mapsto s_{l_1}] \quad \Gamma'_1 = update(a_1 := a_2, s_1, \Gamma''_1) \end{array}}{E \vdash a_1 := a_2, M_1, \Gamma_1, \underline{pc}_1 \Rightarrow M'_1, \Gamma'_1}$$

$$(Assign)\ \frac{\begin{array}{c} E \vdash a_1, M_2, \Gamma_2 \Leftarrow l'_1, s'_1 \quad E \vdash a_2, M_2, \Gamma_2 \Rightarrow v'_2, s'_2 \\ s'_{l_1} = s'_1 \sqcup s'_2 \sqcup \underline{pc}_2 \quad M'_2 = M_2[l'_1 \mapsto v'_2] \\ \Gamma''_2 = \Gamma_2[l'_1 \mapsto s'_{l_1}] \quad \Gamma'_2 = update(a_1 := a_2, s'_1, \Gamma''_2) \end{array}}{E \vdash a_1 := a_2, M_2, \Gamma_2, \underline{pc}_2 \Rightarrow M'_2, \Gamma'_2}$$

SUFFICES ASSUME: $l \in \Gamma'^{-1}_1(s)$
PROVE: 1. $l \in \Gamma'^{-1}_2(s)$
2. $M'_1(l) = M'_2(l)$

⟨2⟩1. CASE: $l \in S_p(a_1 := a_2)$
PROOF SKETCH: In this case, the same location $l_1 = l'_1$ is modified by the assignment $a_1 := a_2$ for both runs. It suffices to consider two cases:

- $l = l_1$: the initial values mapped to l are modified in both runs. Yet, the same values are mapped to l after the execution of the assignment since its output security label is below s
- $l \neq l_1$: only the security label of l is modified by the update operator. Yet, it is still below s for both runs.

⟨3⟩1. $l_1 = l'_1$ and $s'_1 \sqsubseteq s$
⟨4⟩1. $s_1 \sqsubseteq s$
⟨5⟩1. $\Gamma'_1(l) = \Gamma''_1(l) \sqcup s_1$
⟨5⟩2. Q.E.D.
PROOF: By assumption of ⟨1⟩2 ($\Gamma'_1(l) \sqsubseteq s$)
⟨4⟩2. Q.E.D.
PROOF: by ⟨4⟩1, and Lemma 1 on the l-value evaluation of a_1 in both M_1, Γ_1 and M_2, Γ_2, and assumptions 2 and 3.
⟨3⟩2. CASE: $l = l_1$
⟨4⟩1. $\Gamma'_1(l_1) = s_1 \sqcup s_2 \sqcup \underline{pc}_1 \sqsubseteq s$
⟨4⟩2. $v_2 = v'_2$ and $s'_2 \sqsubseteq s$
⟨5⟩1. $s_2 \sqsubseteq s$

⟨6⟩1. Q.E.D.
PROOF: By ⟨4⟩1
⟨5⟩2. Q.E.D.
PROOF: By ⟨5⟩1, and Corollary 1 on the r-value evaluation of a_2 in both M_1, Γ_1 and M_2, Γ_2, and assumptions 2 and 3.
⟨4⟩3. $\Gamma'_2(l_1) = s'_1 \sqcup s'_2 \sqcup \underline{pc}_2 \sqsubseteq s$
⟨5⟩1. $\underline{pc}_2 \sqsubseteq s$
⟨6⟩1. Q.E.D.
PROOF: By ⟨4⟩1 and assumption 1
⟨5⟩2. Q.E.D.
PROOF: By ⟨3⟩1 and ⟨4⟩2 and ⟨4⟩3
⟨4⟩4. Q.E.D.
PROOF: By ⟨4⟩3 and ⟨4⟩2
⟨3⟩3. CASE: $l \neq l_1$
⟨4⟩1. $\Gamma_1(l) \sqsubseteq s$
⟨5⟩1. $\Gamma'_1(l) = \Gamma''_1(l) \sqcup s_1$
⟨6⟩1. Q.E.D.
PROOF: By the semantics rule of the execution on M_1, Γ_1 and ⟨2⟩1
⟨5⟩2. $\Gamma''_1(l) = \Gamma_1(l)$
⟨6⟩1. Q.E.D.
PROOF: By ⟨3⟩3 and the semantic rule on M_1, Γ_1
⟨5⟩3. Q.E.D.
PROOF: By assumption of ⟨1⟩2 and ⟨5⟩1 and ⟨5⟩2
⟨4⟩2. $M'_1(l) = M'_2(l)$
⟨5⟩1. $M'_1(l) = M_1(l)$
⟨6⟩1. Q.E.D.
PROOF: By ⟨3⟩3
⟨5⟩2. $M'_2(l) = M_2(l)$
⟨6⟩1. Q.E.D.
PROOF: By ⟨3⟩3 and ⟨3⟩1
⟨5⟩3. $M_1(l) = M_2(l)$
⟨6⟩1. Q.E.D.
PROOF: By ⟨4⟩1 and assumptions 3 and 2
⟨5⟩4. Q.E.D.
PROOF: By ⟨5⟩1 and ⟨5⟩2 and ⟨5⟩3
⟨4⟩3. $\Gamma'_2(l) \sqsubseteq s$
⟨5⟩1. $\Gamma_2(l) \sqsubseteq s$
⟨6⟩1. Q.E.D.
PROOF: By ⟨4⟩1 and assumption 2
⟨5⟩2. $\Gamma_2(l) = \Gamma''_2(l)$

⟨6⟩1. Q.E.D.
PROOF: By ⟨3⟩3 and ⟨3⟩1
⟨5⟩3. $\Gamma'_2(l) = \Gamma''_2(l) \sqcup s'_1$
⟨6⟩1. Q.E.D.
PROOF: By the update operator of the semantic rule on M_2, Γ_2 and ⟨2⟩1
⟨5⟩4. Q.E.D.
PROOF: By ⟨5⟩1, ⟨5⟩2, ⟨5⟩3 and ⟨3⟩1
⟨4⟩4. Q.E.D.
PROOF: By ⟨4⟩3 and ⟨4⟩2
⟨2⟩2. CASE: $l \notin S_p(a_1 := a_2)$
PROOF SKETCH: In this case, neither the value nor the security label associated to the location l changes.
⟨3⟩1. $\Gamma_1(l) = \Gamma'_1(l) \sqsubseteq s$
⟨4⟩1. Q.E.D.
PROOF: By ⟨2⟩2 and assumption of ⟨1⟩2
⟨3⟩2. $M'_1(l) = M'_2(l)$
⟨4⟩1. $M'_1(l) = M_1(l)$
⟨5⟩1. Q.E.D.
PROOF: By ⟨2⟩2 since $l_1 \neq l$
⟨4⟩2. $M'_2(l) = M_2(l)$
⟨5⟩1. Q.E.D.
PROOF: By ⟨2⟩2 since $l'_1 \neq l$
⟨4⟩3. $M_1(l) = M_2(l)$
⟨5⟩1. Q.E.D.
PROOF: By ⟨3⟩1 and assumptions 2 and 3
⟨4⟩4. Q.E.D.
PROOF: By ⟨4⟩3, ⟨4⟩2 and ⟨4⟩1
⟨3⟩3. $\Gamma'_2(l) \sqsubseteq s$
⟨4⟩1. $\Gamma'_2(l) = \Gamma_2(l)$
⟨5⟩1. Q.E.D.
PROOF: By ⟨2⟩2
⟨4⟩2. $\Gamma_2(l) \sqsubseteq s$
⟨5⟩1. Q.E.D.
PROOF: By ⟨3⟩1 and assumption 2
⟨4⟩3. Q.E.D.
PROOF: By ⟨4⟩2 and ⟨4⟩1
⟨3⟩4. Q.E.D.
PROOF: By ⟨3⟩3 and ⟨3⟩2
⟨2⟩3. Q.E.D.
PROOF:By ⟨2⟩1 and ⟨2⟩2

⟨1⟩3. CASE: If_{tt}

LET: v_1, s_1, v_2, s_2 such that the evaluation of instruction $if\ (a)\ c_1\ else\ c_2$ in M_1, Γ_1 yield:

$$(If_{tt})\ \frac{\begin{array}{c} E \vdash a, M_1, \Gamma_1 \Rightarrow v_1, s_1 \quad istrue(v_1) \\ \underline{pc}'_1 = s_1 \sqcup \underline{pc}_1 \quad E \vdash c_1, M_1, \Gamma_1, \underline{pc}'_1 \Rightarrow M'_1, \Gamma''_1 \\ \Gamma'_1 = update(c_2, \underline{pc}'_1, \Gamma''_1) \end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2, M_1, \Gamma_1, \underline{pc}_1 \Rightarrow M'_1, \Gamma'_1}$$

$$RV\ E \vdash a, M_2, \Gamma_2 \Rightarrow v_2, s_2$$

LET: $l \in Loc$

SUFFICES ASSUME: $\Gamma'_1(l) \sqsubseteq s$

PROVE: 1. $\Gamma'_2(l) \sqsubseteq \Gamma'_1(l)$

2. $M'_1(l) = M'_2(l)$

⟨2⟩1. CASE: $s_1 \sqsubseteq s$

PROOF SKETCH: In this case, both executions in M_1, Γ_1 and M_2, Γ_2 execute instruction c_1. Therefore, an induction on c_1 is sufficient.

⟨3⟩1. $v_2 = v_1$ and $s_2 \sqsubseteq s_1$

⟨3⟩2. Q.E.D.

PROOF: By assumption of ⟨2⟩1 and Corollary 1.

Execution is M_2, Γ_2 yield:

$$If_{tt}\ \frac{\begin{array}{c} E \vdash a, M_2, \Gamma_2 \Rightarrow v_2, s_2 \quad istrue(v_2) \\ pc'_2 = s_2 \sqcup \underline{pc}_2 \quad E \vdash c_1, M_2, \Gamma_2 \underline{pc}'_2 \Rightarrow M'_2, \Gamma''_2 \\ \Gamma'_1 = update(c_2, \underline{pc}'_2, \Gamma''_2) \end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2, M_2, \Gamma_2, \underline{pc}_2 \Rightarrow M'_2, \Gamma'_2}$$

⟨3⟩3. $pc'_2 \sqsubseteq \underline{pc}'_1$

⟨4⟩1. Q.E.D.

PROOF: By ⟨3⟩1 and assumption 1 and rules If_{tt} of both executions.

⟨3⟩4. $M'_1 \sim^s_{\Gamma_1} M'_2$ and $\Gamma''_2 \sqsubseteq_s \Gamma''_1$

⟨4⟩1. Q.E.D.

PROOF: By ⟨3⟩3, assumptions 3 and 2 and induction hypothesis on the evaluation of instruction c_1.

⟨3⟩5. $\Gamma'_2 \sqsubseteq_s \Gamma'_1$

LET: $l \in loc$

SUFFICES ASSUME: $\Gamma'_1(l) \sqsubseteq s$

PROVE: $\Gamma'_2(l) \sqsubseteq \Gamma'_1(l)$

⟨4⟩1. CASE: $l \notin S_p(c_2)$

⟨5⟩1. Q.E.D.

PROOF: By ⟨3⟩4, assumption of ⟨4⟩1 and definition of operator update, $\Gamma'_1(l) = \Gamma''_2(l) \sqsubseteq \Gamma''_1(l) = \Gamma'_1(l)$

⟨4⟩2. Case: $l \in S_p(c_2)$
⟨5⟩1. Q.E.D.
Proof: By $\Gamma'_2(l) = \underline{pc}'_2 \sqsubseteq \Gamma''_2(l)$ and $\Gamma'_1(l) = \underline{pc}'_1 \sqsubseteq \Gamma''_1(l)$ and ⟨3⟩4 and ⟨3⟩3.
⟨4⟩3. Q.E.D.
Proof: By cases ⟨4⟩1 and ⟨4⟩2
⟨3⟩6. Q.E.D.
Proof: By ⟨3⟩5 and ⟨3⟩4.
⟨2⟩2. Case: $s_1 \not\sqsubseteq s$
Proof sketch: In this case, if the label of location l is below s, that means that location l could not have been assigned neither by instructions c_1, nor by instruction c_2. Otherwise, $\underline{pc}$ would have been propagated to the label of l which would be greater than s_1.
⟨3⟩1. $l \notin S_p(c_2)$
⟨4⟩1. Q.E.D.
Proof: By assumption of ⟨1⟩3 ($\Gamma'_1(l) \sqsubseteq s$). In fact, $l \in S_p(c_2)$ implies that $s_1 \sqsubseteq \Gamma'_1(l)$ which means $\Gamma'_1(l) \not\sqsubseteq s$.
⟨3⟩2. $l \notin S_P(c_1)$
⟨4⟩1. Q.E.D.
Proof: If there exists an assignment in c_1 that may write to location l, then the update operator would propagate $\underline{pc}'_1 \not\sqsubseteq s$ to $\Gamma'_1(l)$.
⟨3⟩3. $M_1(l) = M'_1(l)$ and $\Gamma_1(l) = \Gamma'_1(l)$
⟨4⟩1. Q.E.D.
Proof: By ⟨3⟩1 and ⟨3⟩2 since l is neither written by c_1 nor operator update.
⟨3⟩4. $M_2(l) = M'_2(l)$ and $\Gamma_2(l) = \Gamma'_2(l)$
⟨4⟩1. Q.E.D.
Proof: By ⟨3⟩1 and ⟨3⟩2 since l is neither written by c_1, nor c_2.
⟨3⟩5. Q.E.D.
Proof: By ⟨3⟩3 and ⟨3⟩4 and assumptions 2 and 3.
⟨2⟩3. Q.E.D.
Proof: By ⟨2⟩1 and ⟨2⟩2.
⟨1⟩4. Case: If_{ff}
Proof: by symmetry of ⟨1⟩3.
⟨1⟩5. Case: W_{tt} and W_{ff}
⟨2⟩1. Q.E.D.
Proof: *While* rule is semantically equivalent to $if\ (a)\ c; while\ (a)\ c;\ else\ skip$
⟨1⟩6. Case: Composition
⟨2⟩1. Q.E.D.
Proof: By induction on c_1, then on c_2.
⟨1⟩7. Q.E.D.

PROOF: By $\langle 1\rangle 1$, $\langle 1\rangle 2$, $\langle 1\rangle 3$, $\langle 1\rangle 4$, $\langle 1\rangle 5$, $\langle 1\rangle 6$ and induction on instruction evaluation $\Rightarrow$.

Appendix B

Inlining Approach

B.1 Semantics Preservation

Theorem 2 proves that our program transformation preserves the behavior of the initial program.

Theorem 2 (Initial semantics preservation)**.**
For all instructions c, for all environments E, for all memories M, for all security memories Γ, for all security contexts $\underline{pc}$ and variables pc such that:

$$E|_{Var(P)} \vdash c, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc} \Rightarrow M_1, \Gamma_1 \text{ and}$$
$$E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2$$

$$\text{Then, } M_2|_{Loc(P)} = M_1 \text{ and } \Gamma_2|_{Loc(P)} = \Gamma_1.$$

Proof sketch: By induction on instructions evaluation $\Rightarrow$, knowing that the instructions added by transformation T handle only shadow variables.
Let: $E, M, \Gamma, \underline{pc}, M_1, \Gamma_1, M_2, \Gamma_2$.
Assume: 1. $E \vdash c, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc} \Rightarrow M'_1, \Gamma'_1$
2. $E \vdash T[c, \underline{pc}], M, \Gamma, \underline{pc} \Rightarrow M'_2, \Gamma'_2$
Prove: 1. $M'_2|_{Loc(P)} = M'_1$
2. $\Gamma'_2|_{Loc(P)} = \Gamma'_1$
$\langle 1 \rangle 1$. Locations pointed by shadow variables are disjoint from initial locations $Loc(P)$
Proof:
Let:
$Loc_M(T[P]) \triangleq \{l : \forall x \in Var(P) \wedge \forall k \in [0, \mathscr{D}(x)] \wedge \forall r \in [0, \mathscr{D}(k)], E \vdash *^r\Lambda(x, k), M \Leftarrow l)\}$. Then $Loc_M(T[P]) \cap Loc(P) = \emptyset$ since for any

$x, y \in Var(P)$, $*^n y$ has a type $ptr^{(*)}(\kappa)$ whereas $*^r \Lambda(x,k)$ has a type $ptr^{(*)}(\tau_s)$. Hence $*^n y$ and $*^r \Lambda(x,k)$ cannot point to the same location l. That stems for the fact that the transformed program is typable if the initial program is typable.

⟨1⟩2. Case: Skip

⟨2⟩1. Q.E.D.

Proof: Holds since output memories are equal to input memories.

⟨1⟩3. Case: Assign

Evaluation of instruction $a_1 := a_2$ in $M|_{Loc(P)}, \Gamma|_{Loc(P)}$ yield :

$$(Assign)\ \frac{\begin{array}{c} E \vdash a_1, M|_{Loc(P)}, \Gamma|_{Loc(P)} \Leftarrow l_1, s_1 \\ E \vdash a_2, M|_{Loc(P)}, \Gamma|_{Loc(P)} \Rightarrow v_2, s_2 \\ s = s_1 \sqcup s_2 \sqcup \underline{pc} \qquad s' = s_1 \sqcup \underline{pc} \\ M_1' = M|_{Loc(P)}[l_1 \mapsto v_2] \qquad \Gamma_1'' = \Gamma|_{Loc(P)}[l_1 \mapsto s] \\ \Gamma_1' = update(a_1 := a_2, s', \Gamma_1'') \end{array}}{E \vdash a_1 := a_2, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc} \Rightarrow M_1', \Gamma_1'}$$

Transformation T maps to $a_1 = a_2$ instructions :

$$T[a_1 = a_2, \underline{pc}] \mapsto$$

$$\begin{cases} \Lambda(a_1, 0) = \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup \underline{pc}; \\ \Lambda(a_1, k) = \Lambda(a_2, k); \forall k \in [1, \mathscr{D}(a_1)] \\ \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \mathcal{L}_L(a_1) \sqcup \underline{pc}; \forall l \in S_P(a_1 = a_2) \\ a_1 = a_2; \end{cases}$$

Let: c_T be the instructions added by the transformation T, and M_T, Γ_T such that: $E \vdash c_T, M, \Gamma, \underline{pc} \Rightarrow M_T, \Gamma_T$ and $E \vdash a_1 = a_2, M_T, \Gamma_T, \underline{pc} \Rightarrow M_2', \Gamma_2'$.

⟨2⟩1. $M_T|_{Loc(P)} = M|_{Loc(P)}$ and $\Gamma_T|_{Loc(P)} = M_T|_{Loc(P)}$

⟨3⟩1. Q.E.D.

Proof: By ⟨1⟩1, c_T does not modify neither values nor security labels mapped by M, Γ for locations $l \in Loc(P)$. Hence, output memories M_T, Γ_T are equal to initial memories M, Γ when both pairs are restricted to the set of initial locations $Loc(P)$.

⟨2⟩2. Q.E.D.

Proof: By ⟨2⟩1, and since instruction $a_1 := a_2$ handles only locations in $Loc(P)$, output memories are equal when restricted to $Loc(P)$.

⟨1⟩4. Case: If_{tt}

Supposing that the conditional guard is evaluated to true, evaluation of instruction $if\ (a)\ c_1\ else\ c_2$ in $M|_{Loc(P)}, \Gamma|_{Loc(P)}$ yield:

$$(If_{tt})\ \frac{\begin{array}{c} E \vdash a, M|_{Loc(P)}, \Gamma|_{Loc(P)} \Rightarrow v, s \qquad istrue(v) \\ \underline{pc'} = s \sqcup \underline{pc} \qquad E \vdash c_1, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc'} \Rightarrow M_1, \Gamma_1 \\ \Gamma'_1 = update(c_2, \underline{pc'}, \Gamma_1) \end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc} \Rightarrow M'_1, \Gamma'_1}$$

Transformation T maps the initial conditional to the following instructions :

$$T[if\ (a)\ c_1\ else\ c_2, \underline{pc}] \mapsto$$

$$\begin{cases} \underline{pc'} := \mathcal{L}_R(a) \sqcup \underline{pc}; \\ if\ (a)\ \{ \\ \quad T[c_1, \underline{pc'}] \\ \quad \Lambda(E^{-1}(l), 0) := \Lambda(E^{-1}(l), 0) \sqcup \underline{pc'}; \forall l \in S_P(c_2) \\ \}\ else\ \{ \\ \quad T[c_2, \underline{pc'}]; \\ \quad \Lambda(E^{-1}(l), 0) := \Lambda(E^{-1}(l), 0) \sqcup \underline{pc'}; \forall l \in S_P(c_1) \\ \} \end{cases}$$

The same branch is executed for both runs since a is evaluated to the same value (assumed to be $true$).

LET: M_T, Γ_T such that $E \vdash T[c_1, \underline{pc'}], M, \Gamma, \underline{pc'} \Rightarrow M_T, \Gamma_T$.

⟨2⟩1. $M'_1|_{Loc(P)} = M_T$ and $\Gamma'_1|_{Loc(P)} = \Gamma_T$

⟨3⟩1. Q.E.D.

PROOF: By induction on the evaluation of both $T[c_1, \underline{pc'}]$ in M, Γ and also c_1 in $M|_{Loc(P)}, \Gamma|_{Loc(P)}$.

⟨2⟩2. $M'_2|_{Loc(P)} = M_T|_{Loc(P)}$ and $\Gamma'_2|_{Loc(P)} = \Gamma_T|_{Loc(P)}$

⟨3⟩1. Q.E.D.

PROOF: By ⟨1⟩1, instructions $\Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \underline{pc'}; \forall l \in S_P(c_2)$ and operator $update$ only handle shadow variables. Hence output memories M'_2, Γ'_2 are equal to input memories M_T, Γ_T when restricted to $Loc(P)$.

⟨2⟩3. Q.E.D.

PROOF:By transitivity and ⟨2⟩2 and ⟨2⟩1.

⟨1⟩5. CASE: if_{ff}

⟨2⟩1. Q.E.D.

PROOF: This case is symmetrical to if_{tt}

⟨1⟩6. CASE: W_{tt}

Evaluation of *while* (a) c yields:

$$(W_{tt})\ \frac{\begin{array}{c}E \vdash a, M|_{Loc(P)}, \Gamma|_{Loc(P)} \Rightarrow v, s \qquad istrue(v) \qquad \underline{pc'} = s \sqcup \underline{pc} \\ E \vdash c, M, \Gamma, \underline{pc'} \Rightarrow M_1'', \Gamma_1'' \\ E \vdash while\ (a)\ c, M_1'', \Gamma_1'', \underline{pc} \Rightarrow M_1', \Gamma_1 s'\end{array}}{E \vdash while\ (a)\ c, M|_{Loc(P)}, \Gamma, \underline{pc} \Rightarrow M_1', \Gamma_1'}$$

Transformation T yields the following instructions :

$$T[while\ (a)\ c, \underline{pc}] \mapsto \begin{cases} while\ (a)\ \{ \\ \quad \underline{pc'} := \mathcal{L}_R(a) \sqcup \underline{pc}; \\ \quad T[c, \underline{pc'}]; \} \\ \underline{pc'} := \mathcal{L}_R(a) \sqcup \underline{pc}; \\ \Lambda(E^{-1}(l), 0) := \Lambda(E^{-1}(l), 0) \sqcup \underline{pc'}; \forall l \in S_P(c) \end{cases}$$

LET: $M_T, \Gamma_T, M_T', \Gamma_T'$ such that :
$E \vdash \underline{pc'} = \mathcal{L}_R(a) \sqcup \underline{pc};, M, \Gamma, \underline{pc} \Rightarrow M_T, \Gamma_T$, and
$E \vdash T[c, \underline{pc'}]; M_T, \Gamma_T, \underline{pc'} \Rightarrow M_T', \Gamma_T'$, and
$E \vdash T[while\ (a)\ c, \underline{pc}], M_T', \Gamma_T', \underline{pc} \Rightarrow M_2', \Gamma_2'$. Then,

$$W_{tt}\ \frac{\begin{array}{c}E \vdash a, M, \Gamma \Rightarrow v, s \qquad istrue(v) \qquad \underline{pc'} = s \sqcup \underline{pc} \\ E \vdash \underline{pc'} = \mathcal{L}_R(a) \sqcup \underline{pc};\ T[c, \underline{pc'}];, M, \Gamma, \underline{pc} \Rightarrow M_T', \Gamma_T' \\ E \vdash T[while\ (a)\ c, \underline{pc}], M_T', \Gamma_T', \underline{pc} \Rightarrow M_2', \Gamma_2'\end{array}}{E \vdash T[while\ (a)\ c, \underline{pc}], M, \Gamma, \underline{pc} \Rightarrow M_2', \Gamma_2'}$$

⟨2⟩1. $M_T|_{Loc(P)} = M|_{Loc(P)}$ and $\Gamma_T|_{Loc(P)} = \Gamma|_{Loc(P)}$
⟨3⟩1. Q.E.D.
PROOF: By ⟨1⟩1, knowing that $\underline{pc'} := \mathcal{L}_R(a) \sqcup \underline{pc}$ handles only shadow variables.
⟨2⟩2. $M_T'|_{Loc(P)} = M_1''$ and $\Gamma_T'|_{Loc(P)} = \Gamma_1''$
⟨3⟩1. Q.E.D.
PROOF: By ⟨2⟩1 and induction on evaluation of both c and $T[c, \underline{pc'}]$.
⟨2⟩3. Q.E.D.
PROOF: By ⟨2⟩2 and induction on both evaluations of $T[while\ (a)\ c, \underline{pc}]$ in M_T', Γ_T' and *while* (a) c in M_1'', Γ_1''.
⟨1⟩7. CASE: W_{ff}
Evaluation of *while* (a) c yields :

$$(W_{ff})\ \frac{\begin{array}{c}E \vdash a, M|_{Loc(P)}, \Gamma|_{Loc(P)} \Rightarrow v, s \\ isfalse(v) \qquad \underline{pc'} = s \sqcup \underline{pc} \qquad \Gamma_1' = update(c, \underline{pc'}, \Gamma|_{Loc(P)})\end{array}}{E \vdash while\ (a)\ c, M, \Gamma, \underline{pc} \Rightarrow M|_{Loc(P)}, \Gamma_1'}$$

Evaluation of *while* (a) $\{E \vdash \underline{pc'} := \mathcal{L}_R(a) \sqcup \underline{pc};\ T[c, \underline{pc'}];\}$ yields :

$$\frac{E \vdash a, M, \Gamma \Rightarrow v, s \qquad isfalse(v) \qquad \underline{pc'} = s \sqcup \underline{pc} \qquad \Gamma''_2 = update(c, \underline{pc'}, \Gamma)}{E \vdash while\ (a)\ \{E \vdash \underline{pc'} := \mathcal{L}_R(a) \sqcup \underline{pc};\ T[c, \underline{pc'}];\}, M, \Gamma, \underline{pc} \Rightarrow M, \Gamma''_2}$$

Then, evaluation of the remaining instructions yields :
$E \vdash \underline{pc'} = \mathcal{L}_R(a) \sqcup \underline{pc}; \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \underline{pc'}; \forall l \in S_P(c), M, \Gamma''_2, \underline{pc} \Rightarrow M'_2, \Gamma'_2$
$\langle 2 \rangle 1$. $\Gamma'_1 = \Gamma''_2|_{Loc(P)}$
$\langle 3 \rangle 1$. Q.E.D.
PROOF: Operator *update* modifies only the values mapped to locations of $S_P(c)$ in Γ and $\Gamma|_{Loc(P)}$ by assigning to them the same values $\Gamma(l) \sqcup \underline{pc'}, \forall l \in S_P(c)$.
$\langle 2 \rangle 2$. Q.E.D.
PROOF: By $\langle 1 \rangle 1$ and $\langle 2 \rangle 1$, $M'_2|_{Loc(P)} = M|_{Loc(P)}$ and $\Gamma'_2|_{Loc(P)} = \Gamma'_1$.
$\langle 1 \rangle 8$. CASE: Composition
$\langle 2 \rangle 1$. Q.E.D.
PROOF: By Induction on both c_1 and c_2
$\langle 1 \rangle 9$. Q.E.D.
PROOF: By $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, $\langle 1 \rangle 4$, $\langle 1 \rangle 5$, $\langle 1 \rangle 6$, $\langle 1 \rangle 7$, $\langle 1 \rangle 8$ and induction on the evaluation rule $\Rightarrow$.

B.2 Aliasing Invariant

Lemma 2 proves that our program transformation maintains the aliasing invariant introduced in Definition 7.

Lemma 2 (Maintaining the aliasing invariant)**.**
For all environments E, for all instructions c, for all memories M, for all security memories Γ, for all security context $\underline{pc}$, and variable pc such that:

$$E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$$

then, the program transformation maintains the aliasing invariant of Definition 8:

$$\Omega(M) \implies \Omega(M').$$

PROOF SKETCH: By induction on instructions evaluation $\Rightarrow$.
LET: $c, E, M, \Gamma, \underline{pc}, M', \Gamma', pc$ such that $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$.
LET: $\Omega(M) \triangleq \forall x, y \in Var(P)$, for all $r \in [0, \mathscr{D}(y)]$,

$$x \sim^M_{lval} *^r y \quad (\mathbf{1})$$
$$\iff \forall k \in [0, \mathscr{D}(x)], \Lambda(x, k) \sim^M_{lval} \Lambda(*^r y, k) \quad (\mathbf{2})$$
$$\iff \exists k \geq 0, \Lambda(x, k) \sim^M_{lval} \Lambda(*^r y, k) \quad (\mathbf{3})$$

ASSUME: $\Omega(M)$.
PROVE: $\Omega(M')$.
$\langle 1 \rangle 1$. CASE: Skip
$\langle 2 \rangle 1$. Q.E.D.
PROOF: $M = M'$.
$\langle 1 \rangle 2$. CASE: Composition
$\langle 2 \rangle 1$. Q.E.D.
PROOF: By induction, M' holds after evaluation of $T[c_1, pc]$. Hence, by induction on $T[c_2, pc]$ $\Omega(M')$ holds after execution of $T[c_1; c_2, pc]$.
$\langle 1 \rangle 3$. CASE: If_{tt}
Transformation T maps the following instructions to conditionals :

$$T[if\ (a)\ c_1\ else\ c_2, pc] \mapsto$$

$$\begin{cases} pc' := \mathcal{L}_R(a) \sqcup pc; \\ if\ (a)\ \{ \\ \quad T[c_1, pc'] \\ \quad \Lambda(E^{-1}(l), 0) := \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_2) \\ \} \ else\ \{ \\ \quad T[c_2, pc']; \\ \quad \Lambda(E^{-1}(l), 0) := \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_1) \\ \} \end{cases}$$

LET: M_T, Γ_T such that : $E \vdash pc' := \mathcal{L}_R(a) \sqcup pc, M, \Gamma, \underline{pc} \Rightarrow M_T, \Gamma_T$.
LET: M_1, Γ_1 such that $E \vdash T[c_1, pc'], M_T, \Gamma_T, \underline{pc'} \Rightarrow M_1, \Gamma_1$.
LET: M_1', Γ_1' such that $E \vdash \Lambda(E^{-1}(l), 0) := \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_2), M_1, \Gamma_1, \underline{pc'} \Rightarrow M_1', \Gamma_1'$.
LET: M', Γ' such that $E \vdash if(a)\{T[c_1, pc']; \ldots; \}\ else\ \{\ldots\}, M_T, \Gamma_T, \underline{pc'} \Rightarrow M', \Gamma'$.
$\langle 2 \rangle 1$. $\Omega(M_T)$ holds
$\langle 3 \rangle 1$. Q.E.D.
PROOF: $pc' := \mathcal{L}_R(a) \sqcup pc$; only modifies pc', which is of type τ_s. No l-values are modified, hence $\Omega(M_T)$ holds since $\Omega(M)$ holds.
$\langle 2 \rangle 2$. $\Omega(M_1)$ holds
$\langle 3 \rangle 1$. Q.E.D.
PROOF: By induction on $T[c_1, pc']$ knowing $\Omega(M_T)$.
$\langle 2 \rangle 3$. $\Omega(M_1')$ holds
$\langle 3 \rangle 1$. Q.E.D.
PROOF: Modified variables are of type τ_s, therefore no l-values are modified. Hence $\Omega(M_1')$ holds from $\Omega(M_1)$.
$\langle 2 \rangle 4$. Q.E.D.

PROOF: M' is equal to M'_1 by the evaluation rule if_{tt}. Hence $\Omega(M')$ holds by ⟨2⟩3.

⟨1⟩4. CASE: if_{ff}

⟨2⟩1. Q.E.D.

PROOF: This case is symmetrical to the if_{tt} rule.

⟨1⟩5. CASE: W_{tt} and W_{ff}

⟨2⟩1. Q.E.D.

PROOF: Derives from the cases if_{tt} and if_{ff}.

⟨1⟩6. CASE: Assign

PROOF SKETCH: This is the most interesting case. We are going to prove that new aliasing relations for initial variables are reproduced for shadow variables, without breaking old aliasing relations. Transformation T maps the following instructions to assignment $a_1 := a_2$:

$$T[a_1 := a_2, pc] \mapsto$$

$$\begin{cases} \Lambda(a_1, 0) := \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup pc; & \text{(instruction } c_T^0\text{)} \\ \Lambda(a_1, k) := \Lambda(a_2, k); \forall k \in [1, \mathscr{D}(a_1)] & \text{(instruction } c_T^1\text{)} \\ \Lambda(E^{-1}(l), 0) := \Lambda(E^{-1}(l), 0) \sqcup \mathcal{L}_L(a_1) \sqcup pc; \forall l \in S_P(a_1 := a_2) & \text{(instruction } c_T^2\text{)} \\ a_1 := a_2; \end{cases}$$

LET: instructions c_T^0, c_T^1 and c_T^2 be as shown above, and $c_T = c_T^0; c_T^1; c_T^2;$.

LET: M_1, Γ_1, M_2, Γ_2 and M_3, Γ_3 such that: $E \vdash c_T^0, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma_1$, and
$E \vdash c_T^1, M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2$, and $E \vdash c_T^2, M, \Gamma, \underline{pc} \Rightarrow M_3, \Gamma_3$.

Then (only relevant semantics operations are shown below) :

$$(Assign)\ \frac{\begin{array}{c} E \vdash \Lambda(a_1, 0), M, \Gamma \Leftarrow l_{\Lambda(a_1,0)}, s_{\Lambda(a_1,0)} \\ E \vdash \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup pc, M, \Gamma \Rightarrow v_{\Lambda(a_1,0)}, s'_{\Lambda(a_1,0)} \\ M_1 = M[l_1 \mapsto v_2] \qquad \ldots \end{array}}{E \vdash c_T^0, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma_1}$$

$$E \vdash c_T^1, M_1, \Gamma_1, \underline{pc} \Rightarrow M_2, \Gamma_2 \qquad E \vdash c_T^2, M_2, \Gamma_2, \underline{pc} \Rightarrow M_3, \Gamma_3$$

$$(Assign)\ \frac{\begin{array}{c} E \vdash a_1, M_3, \Gamma_3 \Leftarrow l_{a_1}, s_{a_1} \\ E \vdash a_2, M_3, \Gamma_3 \Rightarrow s_{a_2}, v_{a_2} \qquad M' = M_3[l_{a_1} \mapsto v_{a_2}] \qquad \ldots \end{array}}{E \vdash a_1 := a_2, M_3, \Gamma_3, \underline{pc} \Rightarrow M', \Gamma'}$$

$$E \vdash T[a_1 := a_2, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$$

⟨2⟩1. $M_3|_{Loc(P)} = M_2|_{Loc(P)} = M_1|_{Loc(P)} = M|_{Loc(P)}$

⟨3⟩1. Q.E.D.

PROOF: Since locations pointed by initial variables are disjoint from

locations pointed by shadow variables, and c_T only handles shadow variables (this have been proven in previous lemma).

$\langle 2 \rangle 2$. CASE: $\Omega(M').(1) \implies \Omega(M').(2)$

ASSUME: $\Omega(M)$ and $\forall x, y \in Var(P)$, for all $r \in [0, \mathscr{D}(y)]$,

$$x \sim_{lval}^{M'} *^r y \quad (1)$$

$$\implies \forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim_{lval}^{M'} \Lambda(*^r y, k) \quad (2)$$

PROOF SKETCH: By induction on $r \in \mathbb{N}$. The special case when $r = 1$ is also proven in order to be used later during the proof.

$\langle 3 \rangle 1$. CASE: $r = 0$

$\langle 4 \rangle 1$. Q.E.D.

PROOF: $x \sim_{lval}^{M'} y$ implies x and y are the same variable since environment E is a bijection.

$\langle 3 \rangle 2$. CASE: $r = 1$

ASSUME: $x \sim_{lval}^{M'} *y$

PROVE: $\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim_{lval}^{M'} \Lambda(*y, k)$

LET: $l_y \in Loc(P)$ such that $E \vdash y, M' \Leftarrow l_y$ and $M'[l_y] = l_x$ (We will explicit neither Γ nor security labels as they are useless for this). Notice that $E \vdash y, M \Leftarrow l_y$ since $E(y) = l_y$.

PROOF SKETCH: Either a_1 is an alias for y (in both M or M'), then a_1 modifies the r-value of y, and T create new aliasing relations for shadow variables of x and $*y$. Or a_1 is not an alias for y, then the aliasing relations already exist in M and are not modified in M'.

$\langle 4 \rangle 1$. CASE: $a_1 \sim_{lval}^{M'} y$

$\langle 5 \rangle 1$. $a_1 \sim_{lval}^{M} y$

$\langle 6 \rangle 1$. Q.E.D.

PROOF: $M_3|_{Loc(P)} = M|_{Loc(P)}$ ($\langle 2 \rangle 1$) and $a_1 = a_2$ modifies only the r-value of a_1, not its l-value.

$\langle 5 \rangle 2$. $*a_2 \sim_{lval}^{M} x$

$\langle 6 \rangle 1$. Q.E.D.

PROOF: $a_1 := a_2$ maps v_2 to l_{a_1} in M'. As $*a_1$ end up being aliased to $\&x$ in M' (assumption of $\langle 3 \rangle 2$), v_2 must be equal to $ptr(l_x)$ in M_3. Hence $*a_2$ is an alias for x in M_3, and also in M by $\langle 2 \rangle 1$.

$\langle 5 \rangle 3$. $\forall k \in [0, \mathscr{D}(x)], \Lambda(*a_2, k) \sim_{lval}^{M} \Lambda(x,k)$

$\langle 6 \rangle 1$. Q.E.D.

PROOF: By $\langle 5 \rangle 2$ and $\Omega(M)$.

$\langle 5 \rangle 4$. $\forall k \in [0, \mathscr{D}(x)], \Lambda(*a_2, k) \sim_{lval}^{M'} \Lambda(x,k)$

$\langle 6 \rangle 1$. Q.E.D.

PROOF: The l-values of $\Lambda(*a_2, k) = *\Lambda(a_2, k+1)$ in M are equal to those in M'. If the r-value of $\Lambda(a_2, k+1)$ were to change, it would definitely be because of c_T^1 (the only instructions

manipulations pointer shadow variables). That would mean that there exists k_1 such that $\Lambda(a_1, k_1) \sim^M_{lval} \Lambda(a_2, k+1)$. That would imply that $k_1 = k+1$ by typing. Hence, $\Omega(M)$ implies that a_1 and a_2 are aliased, and so $\Lambda(a_2, k) \sim^M_{lval} \Lambda(a_1, k)$ for all $k \in [0, \mathscr{D}(a_1)]$, meaning that c_T overrides the r-values of $\Lambda(a_2, k+1)$ in M by writing the same r-value in M'.

⟨5⟩5. $\forall k \in [0, \mathscr{D}(y)], \Lambda(a_1, k) \sim^M_{lval} \Lambda(y, k)$

⟨6⟩1. Q.E.D.

PROOF: By ⟨5⟩1 and $\Omega(M)$.

⟨5⟩6. $\forall k \in [0, \mathscr{D}(y)], \Lambda(a_1, k) \sim^{M'}_{lval} \Lambda(y, k)$

⟨6⟩1. Q.E.D.

PROOF: Only r-values of $\Lambda(a_1, k)$ are modified by c_T, and the l-value of $\Lambda(y, k) = E(\Lambda(y, k))$ is constant.

⟨5⟩7. $\forall k \in [0, \mathscr{D}(x)], \Lambda(*a_1, k) \sim^{M'}_{lval} \Lambda(*a_2, k)$

⟨6⟩1. Q.E.D.

PROOF: After the assignments c_T, for all $k \in [1, \mathscr{D}(a_2)]$, the r-values of $\Lambda(a_1, k)$ are equal to $\Lambda(a_2, k)$. Dereferencing both means that $\forall k \in [1, \mathscr{D}(a_2)]$, we have $*\Lambda(a_1, k) \sim^{M'}_{lval} *\Lambda(a_2, k)$. Hence $\forall k \in [1, \mathscr{D}(a_2)]$, $\Lambda(*a_1, k-1) \sim^{M'}_{lval} \Lambda(*a_2, k-1)$. A change of variable $k \mapsto k-1$ gives the desired result, knowing that $\mathscr{D}(a_2) = \mathscr{D}(x) + 1$ by ⟨5⟩2 and typing.

⟨5⟩8. $\forall k \in [0, \mathscr{D}(*y)], \Lambda(*a_1, k) \sim^{M'}_{lval} \Lambda(*y, k)$

⟨6⟩1. Q.E.D.

PROOF: By dereferencing ⟨5⟩6 for $k \in [1, \mathscr{D}(y)]$ and a variable change $k \mapsto k-1$.

⟨5⟩9. Q.E.D.

PROOF: By transitivity and ⟨5⟩8, ⟨5⟩7 and ⟨5⟩4, $\forall k \in [0, \mathscr{D}(x)], \Lambda(x, k) \sim^{M'}_{lval} \Lambda(*y, k)$.

⟨4⟩2. CASE: $\neg(a_1 \sim^{M'}_{lval} y)$

⟨5⟩1. $x \sim^M_{lval} y$

⟨6⟩1. Q.E.D.

PROOF: The r-value of y is not changed by assignment $a_1 := a_2$ since ⟨4⟩2. It is neither changed by c_T since ⟨2⟩1.

⟨5⟩2. $\forall k \in [0, \mathscr{D}(x)], \Lambda(x, k) \sim^M_{lval} \Lambda(*y, k)$

⟨6⟩1. Q.E.D.

PROOF: By $\Omega(M)$.

⟨5⟩3. $\forall k \in [0, \mathscr{D}(x)], \Lambda(x, k) \sim^{M'}_{lval} \Lambda(*y, k)$

⟨6⟩1. Q.E.D.

PROOF: If exists k such that the l-value of $\Lambda(*y, k) = *\Lambda(y, k+1)$ is changed by c_T, then the r-value of $\Lambda(y, k+1)$ is overridden by c_T. That means that there exists k_1 such that $\Lambda(a_1, k_1) \sim^M_{lval}$

$\Lambda(y, k+1)$. Hence, by typing $k_1 = k+1$ and $\Lambda(a_1, k_1) \sim^M_{lval} \Lambda(y, k_1)$. Then $\Omega(M)$ implies that $a_1 \sim^M_{lval} y$ which was supposed to not hold in this case $\langle 4\rangle 2$.

$\langle 5\rangle 4$. Q.E.D.

PROOF: By $\langle 5\rangle 3$.

$\langle 4\rangle 3$. Q.E.D.

PROOF: By $\langle 4\rangle 1$ and $\langle 4\rangle 2$.

$\langle 3\rangle 3$. CASE: $r \geq 1$

ASSUME: $x \sim^{M'}_{lval} *^{r+1}y$

PROVE: $\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M'}_{lval} \Lambda(*^{r+1}y, k)$

$\langle 4\rangle 1$. $\exists z \in Var(P)$, such that $*^r y \sim^{M'}_{lval} z$ and $\forall k \in [0, \mathscr{D}(z)], \Lambda(z,k) \sim^{M'}_{lval} \Lambda(*^r y, k)$

$\langle 5\rangle 1$. Q.E.D.

PROOF: Since the only way to create locations is through variable declaration, there exists a variable z which is pointed by $*^r y$. By induction on r, we have $\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M'}_{lval} \Lambda(*^r y, k)$.

$\langle 4\rangle 2$. $\forall k \in [0, \mathscr{D}(*z)], \Lambda(*z,k) \sim^{M'}_{lval} \Lambda(*^{r+1}y, k)$

$\langle 5\rangle 1$. Q.E.D.

PROOF: By dereferencing one time both sides of the aliasing relation in $\langle 4\rangle 1$ for $k \in [1, \mathscr{D}(z)]$ and a variable change $k \mapsto k-1$.

$\langle 4\rangle 3$. $x \sim^{M'}_{lval} *z$ and $\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M'}_{lval} \Lambda(*z, k)$

$\langle 5\rangle 1$. Q.E.D.

PROOF: By $\langle 4\rangle 1$, we have $x \sim^{M'}_{lval} *z$. Then by the case $r = 1$ ($\langle 3\rangle 2$), we have the aliasing relation for shadow variables.

$\langle 4\rangle 4$. Q.E.D.

PROOF: By $\langle 4\rangle 3$ and $\langle 4\rangle 2$ and transitivity, $\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M'}_{lval} \Lambda(*^{r+1}y, k)$.

$\langle 3\rangle 4$. Q.E.D.

PROOF: By induction on r, $\langle 3\rangle 3$ and $\langle 3\rangle 1$.

$\langle 2\rangle 3$. CASE: $\Omega(M').(3) \iff \Omega(M').(2) \implies \Omega(M').(1)$

ASSUME: $\Omega(M)$ and $\forall x, y \in Var(P)$, for all $r \in [0, \mathscr{D}(y)]$,

$$\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^M_{lval} \Lambda(*^r y, k) \quad (2)$$
$$\iff \exists k \geq 0, \Lambda(x,k) \sim^M_{lval} \Lambda(*^r y, k) \quad (3)$$
$$\implies x \sim^M_{lval} *^r y \quad (1)$$

PROOF SKETCH: (2) $\implies$ (3) holds. We prove (3) $\implies$ (2) by induction on r as in $\langle 2\rangle 2$. We focus on the most interesting case, $r = 1$.

$\langle 3\rangle 1$. CASE: $r = 1$

ASSUME: $x, y \in Var(P)$, such that $\exists k_1, \Lambda(x, k_1) \sim^{M'}_{lval} \Lambda(*y, k_1)$.

$\langle 4\rangle 1$. CASE: $\exists k'$ such that the l-value of $\Lambda(*y, k')$ changed between M and M'

$\langle 5\rangle 1$. $a_1 \sim^M_{lval} y$

⟨6⟩1. Q.E.D.

PROOF: The l-value of $\Lambda(*y, k') = *\Lambda(y, k'+1)$ changed, meaning that the r-value of $\Lambda(y, k'+1)$ has been overridden by c_T. That means that there exists k_1 such that $\Lambda(a_1, k_1) \sim^M_{lval} \Lambda(y, k'+1)$ and $k'+1 = k_1$. Then by $\Omega(M)$, $a_1 \sim^M_{lval} y$.

⟨5⟩2. $x \sim^M_{lval} *a_2$

⟨6⟩1. Q.E.D.

PROOF: Since a_2 has been assigned to a_1, and $a_1 \sim^M_{lval} y$ (⟨5⟩1), the r-value of $\Lambda(a_2, k_1+1)$ in M has been assigned to $\Lambda(y, k_1)+1)$ in M'. Knowing $\Lambda(y, k_1+1)$ in M' points to $\Lambda(x, k_1)$, we conclude that $\Lambda(a_2, k_1+1)$ in M' points to $\Lambda(x, k_1)$. Hence $*\Lambda(a_2, k_1+1) \sim^M_{lval} \Lambda(x, k_1)$. Then by $\Omega(M)$ and $\Lambda(*a_2, k_1) \sim^M_{lval} \Lambda(x, k_1)$, we conclude that $x \sim^M_{lval} *a_2$.

⟨5⟩3. Q.E.D.

PROOF: For all $0 \leq k \leq \mathscr{D}(a_1)-1$, the r-value of $\Lambda(a_1, k+1)$ in M' is equal to that of $\Lambda(a_2, k+1)$ in M after evaluation of assignments c_T. By ⟨5⟩2, $\Lambda(a_2, k+1)$ points to $\Lambda(x, k)$. Therefore, ⟨5⟩1, $\Lambda(y, k+1)$ also points to $\Lambda(x, k)$, meaning that $\Lambda(*y, k) \sim^{M'}_{lval} \Lambda(x, k)$, for all $k \in [0, \mathscr{D}(x)]$. Also, by ⟨5⟩1 and ⟨5⟩2, $x \sim^{M'}_{lval} *y$ holds after evaluation of $a_1 = a_2$.

⟨4⟩2. CASE: $\neg$(∃k' such that the l-value of $\Lambda(*y, k')$ changed between M and M')

⟨5⟩1. Q.E.D.

PROOF: $\forall k \in [0, \mathscr{D}(*y)]$, the l-value of $\Lambda(*y, k')$ in M is equal to that in M'. Particularly, $\Lambda(*y, k_1) \sim^M_{lval} \Lambda(x, k_1)$. By $\Omega(M)$, $\forall k \in [0, \mathscr{D}(x)], \Lambda(*y, k_1) \sim^M_{lval} \Lambda(x, k)$. Which means that $\forall k \in [0, \mathscr{D}(x)], \Lambda(*y, k_1) \sim^{M'}_{lval} \Lambda(x, k)$ since neither l-value of $\Lambda(*y, k_1)$ changed. Also, By $\Omega(M)$, we have $x \sim^M_{lval} *y$. Hence, $x \sim^{M'}_{lval} *y$ since if the r-value of y changes, that would mean that a_1 is an alias for y, causing the r-value of $\Lambda(*y, k)$ to be overridden which is not the case since ⟨4⟩2.

⟨4⟩3. Q.E.D.

PROOF: By ⟨4⟩2 and ⟨4⟩1.

⟨3⟩2. Q.E.D.

PROOF: Using the case $r = 1$ (⟨3⟩1), we conclude by introducing a variable z such that $*z \sim^{M'}_{lval} x$ and then by induction on r as done in ⟨2⟩2.

B.3 Monitoring Information Flows

Theorem 3 proves that our program transformation is sound with respect to our monitor semantics.

Theorem 3 (Sound monitoring of information flows)**.**
For all environments E, for all memories M, for all security memories Γ, the invariant $\Upsilon(E, M, \Gamma)$ is defined as the following predicate:

$$\Upsilon(E, M, \Gamma) \triangleq \forall x \in Var(P), \forall k \in [0, \mathscr{D}(x)]\,, \text{ then}$$

$$E \vdash *^k x, M \Leftarrow l_{xk} \text{ and } \Gamma(l_{xk}) = s_{xk}$$
$$\implies E \vdash *^k \Lambda(x, k), M \Rightarrow s_{xk}.$$

Hence, for all instructions c, for all memories M, for all security memories Γ, for all security context $\underline{pc}$ and variable pc such that $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$ and $E \vdash pc, M \Rightarrow \underline{pc}$, the following result holds :

$$\Upsilon(E, M, \Gamma) \implies \Upsilon(E, M', \Gamma').$$

First, we prove that operators $\mathcal{L}_L$ and $\mathcal{L}_R$ captures resp. the labels of l-value evaluation and r-value evaluation. We introduce hence Lemma 5 and Corollary 7.

Lemma 5 (Sound operator $\mathcal{L}_L$)**.** *Let E, M, Γ, such that $\Upsilon(E, M, \Gamma)$ holds. for all $a \in Exp$,*

$$E \vdash a, M, \Gamma \Leftarrow l, s \implies E \vdash \mathcal{L}_L(a), M \Rightarrow s$$

Proof sketch:By induction on l-value evaluations.
Let: E, M, Γ, such that $\Upsilon(E, M, \Gamma)$ holds.
Let: $a \in Exp, l \in Loc, s \in \mathbb{S}$, such that $E \vdash a, M, \Gamma \Leftarrow l, s$.
Prove: $E \vdash \mathcal{L}_L(a), M \Rightarrow s$
⟨1⟩1. Case: LV_{ID}
⟨2⟩1. Q.E.D.
Proof: By definition of $\mathcal{L}_L(id)$ and rule LV_{ID}, both are equal to *public*
⟨1⟩2. Case: LV_{MEM}
Evaluation of $*^k x$ in l-value position yields:

$$LV_{MEM}\ \dfrac{RV\ \dfrac{E \vdash a, M, \Gamma \Leftarrow l_a, s_l \qquad M(l_a) = ptr(l) \qquad \Gamma(l_a) = s_r \qquad s = s_l \sqcup s_r}{E \vdash a, M, \Gamma \Rightarrow ptr(l), s}}{E \vdash *a, M, \Gamma \Leftarrow l, s}$$

Also, by definition we have: $\mathcal{L}_L(*a) = \mathcal{L}_R(a) = \mathcal{L}_L(a) \sqcup \mathcal{L}(a)$.

⟨1⟩3. Q.E.D.
PROOF: By induction, we have $E \vdash \mathcal{L}_L(a), M \Rightarrow s_l$. Also, $\Upsilon(E, M, \Gamma)$ implies that $E \vdash \mathcal{L}(a), M \Rightarrow s_r$. Hence, $E \vdash \mathcal{L}_L(*a), M \Rightarrow s$.

Corollary 7 (Sound operator $\mathcal{L}_R$)**.** *Let E, M, Γ, such that $\Upsilon(E, M, \Gamma)$ holds. for all $a \in Exp$,*

$$E \vdash a, M, \Gamma \Rightarrow v, s \implies E \vdash \mathcal{L}_R(a), M \Rightarrow s$$

⟨1⟩1. Q.E.D.
PROOF: By induction on r-value evaluations of expressions, using Lemma 5.

Theorem 3 proves that our program transformation is sound with respect to our monitor semantics.

Theorem 3 (Sound monitoring of information flows)**.**
For all environments E, for all memories M, for all security memories Γ, the invariant $\Upsilon(E, M, \Gamma)$ is defined as the following predicate:

$$\Upsilon(E, M, \Gamma) \triangleq \forall x \in Var(P), \forall k \in [0, \mathscr{D}(x)]\,, \text{ then}$$

$$E \vdash *^k x, M \Leftarrow l_{xk} \text{ and } \Gamma(l_{xk}) = s_{xk}$$

$$\implies E \vdash *^k \Lambda(x, k), M \Rightarrow s_{xk}.$$

Hence, for all instructions c, for all memories M, for all security memories Γ, for all security context $\underline{pc}$ and variable pc such that $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$ and $E \vdash pc, M \Rightarrow \underline{pc}$, the following result holds :

$$\Upsilon(E, M, \Gamma) \implies \Upsilon(E, M', \Gamma').$$

PROOF SKETCH: By induction on instructions evaluation $\Rightarrow$. The most interesting case is the assignment.
LET: $c, E, M, \Gamma, \underline{pc}, M', \Gamma', \underline{pc}$ such that $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$.
LET: $\Upsilon(E, M, \Gamma) \triangleq$ for all $x \in Var(P)$, for all $k \in [0, \mathscr{D}(x)]$,
$E \vdash *^k x, M \Leftarrow l_{xk}$ and $\Gamma(l_{xk}) = s_{xk} \implies E \vdash *^k \Lambda(x, k), M \Rightarrow s_{xk}$.
ASSUME: $\Upsilon(E, M, \Gamma)$ and $E \vdash pc, M \Rightarrow \underline{pc}$.
PROVE: $\Upsilon(E, M', \Gamma')$.
⟨1⟩1. CASE: Skip
⟨2⟩1. Q.E.D.
PROOF: By assumption.
⟨1⟩2. CASE: Assign

PROOF SKETCH: We prove $\Upsilon(E, M', \Gamma')$ for any $x \in Var(P)$. Then we generalize this result by Lemma 2 to every l-value $*^k x$. Transformation T maps the following instructions to assignment $a_1 := a_2$:

$$T[a_1 := a_2, pc] \mapsto$$

$$\begin{cases} \Lambda(a_1, 0) := \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup pc; & \text{(instruction } c_T^0\text{)} \\ \Lambda(a_1, k) := \Lambda(a_2, k); \forall k \in [1, \mathscr{D}(a_1)] & \text{(instruction } c_T^1\text{)} \\ \Lambda(E^{-1}(l), 0) := \Lambda(E^{-1}(l), 0) \sqcup \mathcal{L}_L(a_1) \sqcup pc; \forall l \in S_P(a_1 := a_2) & \text{(instruction } c_T^2\text{)} \\ a_1 := a_2; \end{cases}$$

LET: instructions c_T^0, c_T^1 and c_T^2 be as shown above, and $c_T = c_T^0; c_T^1; c_T^2;$.
LET: M_1, Γ_1, M_2, Γ_2 and M_3, Γ_3 such that: $E \vdash c_T^0, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma_1$, and
$E \vdash c_T^1, M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2$, and $E \vdash c_T^2, M, \Gamma, \underline{pc} \Rightarrow M_3, \Gamma_3$.
Then (only relevant semantics operations are shown below) :

$$(Assign)\ \frac{\begin{array}{c} E \vdash \Lambda(a_1, 0), M, \Gamma \Leftarrow l_{\Lambda(a_1,0)}, s_{\Lambda(a_1,0)} \\ E \vdash \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup pc, M, \Gamma \Rightarrow v_{\Lambda(a_1,0)}, s'_{\Lambda(a_1,0)} \\ M_1 = M[l_{\Lambda(a_1,0)} \mapsto v_{\Lambda(a_1,0)}] \qquad \ldots \end{array}}{E \vdash c_T^0, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma_1}$$

$$E \vdash c_T^1, M_1, \Gamma_1, \underline{pc} \Rightarrow M_2, \Gamma_2 \qquad E \vdash c_T^2, M_2, \Gamma_2, \underline{pc} \Rightarrow M_3, \Gamma_3$$

$$(Assign)\ \frac{\begin{array}{c} E \vdash a_1, M_3, \Gamma_3 \Leftarrow l_{a_1}, s_{a_1} \\ E \vdash a_2, M_3, \Gamma_3 \Rightarrow s_{a_2}, v_{a_2} \qquad M' = M_3[l_{a_1} \mapsto v_{a_2}] \\ s = s_{a_1} \sqcup s_{a_2} \sqcup \underline{pc} \qquad \Gamma'' = \Gamma_3[l_{a_1} \mapsto s] \\ \Gamma' = update(a_1 := a_2, s_{a_1} \sqcup \underline{pc}, \Gamma'') \qquad \ldots \end{array}}{E \vdash a_1 = a_2, M_3, \Gamma_3, \underline{pc} \Rightarrow M', \Gamma'}$$

$$E \vdash T[a_1 := a_2, \underline{pc}], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$$

⟨2⟩1. $\forall x \in Var(P), \Gamma'(E(x)) = s \implies E \vdash \Lambda(x, 0), M' \Rightarrow s$
LET: $x \in Var(P)$ such that $\Gamma'(E(x)) = s$.
⟨3⟩1. CASE: $a_1 \sim_{lval}^{M'} x$
PROOF SKETCH: In this case, $T[c, pc]$ updates both x and $\Lambda(x, 0)$ such that Υ still holds.
⟨4⟩1. $s = \Gamma'(E(x)) = s_{a_1} \sqcup s_2 \sqcup \underline{pc}$
⟨5⟩1. Q.E.D.
PROOF: By assignment rule of $a_1 := a_2$ since only instruction $a_1 := a_2$ modifies values mapped to $l_{a_1} = E(x) \in Loc(P)$.
⟨4⟩2. $s = v_{\Lambda(a_1,0)}$
⟨5⟩1. Q.E.D.
PROOF: By Lemma 2, we have $\Lambda(a_1, 0) \sim_{lval}^{M'} \Lambda(x, 0)$. Then we

conclude by knowing that $\Lambda(a_1, 0)$ is assigned by instruction c_T^0. That means that $E \vdash \Lambda(a_1, 0), M' \Rightarrow v_{\Lambda(a_1,0)}$. (notice that c_T^2 also assigns $\Lambda(a_1, 0)$. But since it just propagates r-values of $\mathcal{L}_L(a_1) \sqcup pc$ which are already propagated by c_T^0, the value of $\Lambda(a_1, 0)$ keeps being equal to $v_{\Lambda(a_1,0)}$)

$\langle 4 \rangle 3$. Q.E.D.

PROOF: By $\langle 4 \rangle 1$, $\langle 4 \rangle 2$ and assumption $\Upsilon(E, M, \Gamma)$, $v_{\Lambda(a_1,0)} = s_{a_1} \sqcup s_2 \sqcup \underline{pc}$ (Lemma 5 and Corollary 7).

$\langle 3 \rangle 2$. CASE: $\neg(a_1 \sim_{lval}^{M'} x)$ and $E(x) \notin S_P(a_1 = a_2)$

LET: $x \in Var(P)$ such that $\Gamma'(E(x)) = s$.

LET: $v_{\Lambda(x,0)}$ such that $E \vdash \Lambda(x, 0), M' \Rightarrow v_{\Lambda(x,0)}$

PROOF SKETCH: In this case, neither x nor $\Lambda(x, 0)$ are updated. Hence the invariant holds from $\Upsilon(E, M, \Gamma)$.

$\langle 4 \rangle 1$. $E \vdash \Lambda(x, 0), M \Rightarrow v_{\Lambda(x,0)}$

$\langle 5 \rangle 1$. Q.E.D.

PROOF: Since $\Lambda(x, 0)$ in not modified by $T[a_1 = a_2, pc]$. $a_1 = a_2$ modifies only locations in $Loc(P)$. c_T^2 do not modify $\Lambda(x, 0)$ since $E(x) \notin S_P(a_1 = a_2)$. c_T^1 modifies only pointers and $\Lambda(x, 0)$ is not. c_T^0 does not assign $\Lambda(x, 0)$ since $\neg(a_1 \sim_{lval}^{M'} x)$.

$\langle 4 \rangle 2$. $\Gamma(E(x)) = s$

$\langle 5 \rangle 1$. Q.E.D.

PROOF: By the semantics of assignments, knowing that $E(x) \notin S_P(a_1 := a_2)$.

$\langle 4 \rangle 3$. Q.E.D.

PROOF:By $\langle 3 \rangle 2$ and $\langle 4 \rangle 2$ and $\Upsilon(E, M, \Gamma)$, $s = v_{\Lambda(x,0)}$

$\langle 3 \rangle 3$. CASE: $\neg(a_1 \sim_{lval}^{M'} x)$ and $E(x) in S_P(a_1 = a_2)$

PROOF SKETCH: In this case, $s_{a_1} \sqcup \underline{pc}$ and $\mathcal{L}_L(a_1)$ are respectively propagated to respectively $\Gamma(E(x))$ and $\Lambda(x, 0)$. We show that r-value evaluation of $\Lambda(x, 0)$ in M' is equal to the value of $\Gamma(l_{a_1})$.

LET: $x \in Var(P)$ such that $\Gamma'(E(x)) = s$.

LET: $v'_{\Lambda(x,0)}$ such that $E \vdash \Lambda(x, 0), M' \Rightarrow v_{\Lambda(x,0)}$.

LET: $v_{\Lambda(x,0)}$ such that $E \vdash \Lambda(x, 0), M \Rightarrow v_{\Lambda(x,0)}$.

$\langle 4 \rangle 1$. Q.E.D.

PROOF: By $\Gamma'(E(x)) = s_{a_1} \sqcup \underline{pc} \sqcup \Gamma(E(x))$ and $v'_{\Lambda(x,0)} = \mathcal{L}_L(a_1) \sqcup pc \sqcup v_{\Lambda(x,0)}$. Also, by $\Upsilon(E, M, \Gamma)$ we have $s_{a_1} = \mathcal{L}_L(a_1)$ and $v_{\Lambda(x,0)} = \Gamma(E(x))$.

$\langle 3 \rangle 4$. Q.E.D.

PROOF: By $\langle 3 \rangle 3$ and $\langle 3 \rangle 2$ and $\langle 3 \rangle 1$.

$\langle 2 \rangle 2$. Q.E.D.

PROOF:By $\langle 2 \rangle 1$ we can conclude. For all $y \in Var(P)$, $k \in [0, \mathscr{D}(y)]$, there

exists $x \in Var(P)$ such that $x \sim^{M'}_{lval} *^k y$. Then $E(x)$ is the l-value of $*^k y$ in memory M'.

Let $s = \Gamma'(E(x))$. That implies that $E \vdash \Lambda(x,0), M' \Rightarrow s$ ($\langle 2 \rangle 1$), and also $E \vdash *^k \Lambda(y,k), M' \Rightarrow s$ since $\Lambda(*^k y, 0) \sim^{M'}_{lval} \Lambda(x,0)$ (Lemma 2).

$\langle 1 \rangle 3$. Case: Composition

$\langle 2 \rangle 1$. Q.E.D.

By induction on c_1 and then on c_2.

$\langle 1 \rangle 4$. Case: If_{tt}

Transformation T maps the following instructions to conditionals :

$$T[if\ (a)\ c_1\ else\ c_2, pc] \mapsto$$

$$\begin{cases} pc' = \mathcal{L}_R(a) \sqcup pc; \\ if\ (a)\ \{ \\ \quad T[c_1, pc'] \\ \quad \Lambda(E^{-1}(l),0) = \Lambda(E^{-1}(l),0) \sqcup pc'; \forall l \in S_P(c_2) \\ \}\ else\ \{ \\ \quad T[c_2, pc']; \\ \quad \Lambda(E^{-1}(l),0) = \Lambda(E^{-1}(l),0) \sqcup pc'; \forall l \in S_P(c_1) \\ \} \end{cases}$$

Let: M_T, Γ_T such that : $E \vdash pc' = \mathcal{L}_R(a) \sqcup pc, M, \Gamma, \underline{pc} \Rightarrow M_T, \Gamma_T$.

Let: M_1, Γ_1 such that $E \vdash T[c_1, pc'], M_T, \Gamma_T, \underline{pc'} \Rightarrow M_1, \Gamma_1$.

Let: M'_1, Γ'_1 such that $E \vdash \Lambda(E^{-1}(l),0) = \Lambda(E^{-1}(l),0) \sqcup pc'; \forall l \in S_P(c_2), M_1, \Gamma_1, \underline{pc'} \Rightarrow M'_1, \Gamma'_1$.

Let: M', Γ' such that $E \vdash if(a)\{T[c_1, \underline{pc'}]; \ldots;\}\ else\ \{\ldots\}, M_T, \Gamma_T, \underline{pc'} \Rightarrow M', \Gamma'$.

$\langle 2 \rangle 1$. $\Upsilon(E, M_T, \Gamma_T)$ holds and $E \vdash pc', M_T \Rightarrow \underline{pc'}$

$\langle 3 \rangle 1$. Q.E.D.

Proof: Since assignment pc' does not modify neither locations in $Loc(P)$ nor locations associated to shadow variables defined by Λ, $\Upsilon(E, M_T, \Gamma_T)$ holds. Additionally, $\underline{pc'} = s_a \sqcup \underline{pc}$ where s_a is the result of r-value evaluation of a in M and $E \vdash \mathcal{L}_R(a), M \Rightarrow s_a$. Hence $E \vdash pc', M \Rightarrow \underline{pc'}$.

$\langle 2 \rangle 2$. $\Upsilon(E, M_1, \Gamma_1)$ holds

$\langle 3 \rangle 1$. Q.E.D.

By induction on the execution of $T[c_1, pc']$, knowing $\langle 2 \rangle 1$.

$\langle 2 \rangle 3$. $\Upsilon(E, M', \Gamma')$

$\langle 3 \rangle 1$. Q.E.D.

Proof: $\underline{pc'}$ is propagated to all shadow variables corresponding to locations in $S_P(c_2)$. So does the update operator on Γ'_1 before yielding

Γ'. The result for other locations (not in $S_P(c_2)$) shadow variables stems from $\langle 2 \rangle 2$.

$\langle 2 \rangle 4$. Q.E.D.

PROOF: By $\langle 2 \rangle 3$.

$\langle 1 \rangle 5$. CASE: If_{ff}

$\langle 2 \rangle 1$. Q.E.D.

Symmetrical case to If_{tt}.

$\langle 1 \rangle 6$. CASE: W_{tt} and W_{ff}

$\langle 2 \rangle 1$. Q.E.D.

Same as conditionals.

$\langle 1 \rangle 7$. Q.E.D.

PROOF: By induction on instructions evaluation and $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, $\langle 1 \rangle 4$, $\langle 1 \rangle 5$, $\langle 1 \rangle 6$.

Appendix C

Cardinal Abstraction

C.1 Abstract Semantics of Expressions

Theorem 6 (Soundness of the abstract semantics $\mathbb{A}^\sharp[\![a]\!]$)**.**
The abstract semantics of expressions is sound:

$$\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp \leq \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp$$

Let us derive an abstract semantics $\mathbb{A}^\sharp[\![a]\!]R$ for expressions:
— Case `a = n`:

$$\begin{aligned}
\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![n]\!])\varrho^\sharp &\triangleq \alpha_v \circ \mathbb{A}_c[\![n]\!] \circ \dot{\gamma}(\varrho^\sharp) \\
&= \alpha_v\left(\left\{\{v \in \mathbb{V} \mid \exists \varrho \in r, \mathbb{A}[\![n]\!]\varrho = v\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right) \\
&= \alpha_v(\{\{n\}\}) \\
&= 1 \\
&\triangleq \mathbb{A}^\sharp[\![n]\!]\varrho^\sharp
\end{aligned}$$

— Case `a = id`:

$$\begin{aligned}
\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![id]\!])\varrho^\sharp &\triangleq \alpha_v \circ [\![id]\!]^\sharp \circ \dot{\gamma}(\varrho^\sharp) \\
&= \alpha_v\left(\left\{\{v \mid \exists \varrho \in r, \mathbb{A}[\![id]\!]\varrho = v\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right) \\
&= \alpha_v\left(\left\{\{proj_2(\varrho(id)) \mid \exists \varrho \in r\} \mid r \in \dot{\gamma}(\varrho^\sharp)\right\}\right) \\
&= (\text{By definition of } \dot{\gamma}) \\
&\quad \alpha_v \circ \gamma_v(\varrho^\sharp(id)) \\
&\leq (\alpha_v \circ \gamma_v \text{ is reductive}) \\
&\quad \varrho^\sharp(id) \\
&\triangleq \mathbb{A}^\sharp[\![id]\!]\varrho^\sharp
\end{aligned}$$

— Case `a = a₁ mod n` :

$$\begin{aligned}
\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp &\triangleq \alpha_v\left(\left\{\{v_1 \ mod \ n \mid \exists \varrho \in r, \mathbb{A}[\![a_1]\!]\varrho = v_1\} \mid \right.\right.\\
&\qquad \left.\left. r \in \dot\gamma(\varrho^\sharp)\right\}\right)\\
&= \min\left(\alpha_v\left(\left\{\{v_1 \mid \exists \varrho \in r, \mathbb{A}[\![a_1]\!]\varrho = v_1\} \mid \right.\right.\right.\\
&\qquad \left.\left.\left. r \in \dot\gamma(\varrho^\sharp)\right\}\right), n\right)\\
&= \min(\alpha_v \circ \mathbb{A}_c[\![a_1]\!] \circ \dot\gamma(\varrho^\sharp), n)\\
&\leq \text{(By induction hypothesis)}\\
&\qquad \min(\mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp, n)\\
&\triangleq \mathbb{A}^\sharp[\![a_1 \ mod \ n]\!]\varrho^\sharp
\end{aligned}$$

— Case `a = a₁ bop a₂`:

$$\begin{aligned}
\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a_1 \ bop \ a_2]\!])\varrho^\sharp &\triangleq \alpha_v\left(\{\{v_1 \ bop \ v_2 \mid \exists \varrho \in r, \mathbb{A}[\![a_1]\!]\varrho = v_1\right.\\
&\qquad \left.\wedge \mathbb{A}[\![a_2]\!]\varrho = v_2\} \mid r \in \dot\gamma(\varrho^\sharp)\right\}\Big)\\
&\leq \text{(by loosing relationships among variables in } r)\\
&\qquad \alpha_v\left(\left\{\{v_1 \ bop \ v_2 \mid \exists \varrho_1 \in r_1, \exists \varrho_2 \in r_2, \mathbb{A}[\![a_1]\!]\varrho_1 = v_1\right.\right.\\
&\qquad\qquad \left.\left.\wedge \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\} \mid r_1, r_2 \in \dot\gamma(\varrho^\sharp)\right\}\right)\\
&= \text{(by definition of } \alpha_v)\\
&\qquad \max_{r_1, r_2 \in \dot\gamma(\varrho^\sharp)} \left|\left\{v_1 \ bop \ v_2 \mid \exists \varrho_1 \in r_1, \mathbb{A}[\![a_1]\!]\varrho_1 = v\right.\right.\\
&\qquad\qquad \left.\left.\wedge \varrho_2 \in r_2, \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\right\}\right|\\
&\leq \text{(values } v \in \mathbb{V} \text{ are finite, of size } 2^\kappa)\\
&\qquad \min\left(2^\kappa, \max_{r_1, r_2 \in \dot\gamma(\varrho^\sharp)} \left|\left\{v_1 \mid \exists \varrho_1 \in r_1, \mathbb{A}[\![a_1]\!]\varrho_1 = v_1\right\}\right|\right.\\
&\qquad\qquad \left.\times \left|\left\{v_2 \mid \exists \varrho_2 \in r_2, \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\right\}\right|\right)\\
&= \min\left(2^\kappa, \max_{r_1 \in \dot\gamma(\varrho^\sharp)} \left|\left\{v_1 \mid \exists \varrho_1 \in r_1, \mathbb{A}[\![a_1]\!]\varrho_1 = v_1\right\}\right|\right.\\
&\qquad\qquad \left.\times \max_{r_2 \in \dot\gamma(\varrho^\sharp)} \left|\left\{v_2 \mid \exists \varrho_2 \in r_2, \mathbb{A}[\![a_2]\!]\varrho_2 = v_2\right\}\right|\right)\\
&\leq \text{(By induction hypothesis in Equation (6.16))}\\
&\qquad \min\left(2^\kappa, \mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp \times \mathbb{A}^\sharp[\![a_2]\!]\varrho^\sharp\right)\\
&\triangleq \mathbb{A}^\sharp[\![a_1 \ bop \ a_2]\!]\varrho^\sharp
\end{aligned}$$

— Case `a = a1 cmp a2`: this case is similar to the previous case, apart that expression `a1 cmp a2` may evaluate to only 2 different boolean values.

$$\begin{aligned}\alpha^{\triangleright}_{exp}(\mathbb{A}_c[\![a]\!])\varrho^\sharp &\dot{\leq} \min\left(2, \mathbb{A}^\sharp[\![a_1]\!]\varrho^\sharp \times \mathbb{A}^\sharp[\![a_2]\!]\varrho^\sharp\right)\\ &\triangleq \mathbb{A}^\sharp[\![a_1\ cmp\ a_2]\!]\end{aligned}$$

C.2 Abstract Semantics of Instructions

Theorem 7 (Soundness of the abstract semantics $[\![c]\!]^\sharp$)**.**
The abstract semantics of commands is sound:

$$\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp \dot{\subseteq}_\otimes [\![c]\!]^\sharp\varrho^\sharp$$

Let us derive an abstract semantics for instructions:
— Case `c = pp skip`:

$$\begin{aligned}\alpha^{\triangleright}_{com}([\![^{pp}skip]\!]_c)\varrho^\sharp &= \dot\alpha \circ [\![^{pp}skip]\!]_c \circ \dot\gamma(\varrho^\sharp)\\ &= \dot\alpha \circ \dot\gamma(\varrho^\sharp)\\ &\subseteq_\otimes (\dot\alpha \circ \dot\gamma \text{ is reductive})\\ &\quad\ \varrho^\sharp\\ &\triangleq [\![^{pp}skip]\!]^\sharp\varrho^\sharp\end{aligned}$$

— For instruction `c = pp id := a`:

$$\begin{aligned}\alpha^{\triangleright}_{com}(c)\varrho^\sharp &= \dot\alpha \circ [\![^{pp}id := a]\!]_c \circ \dot\gamma(\varrho^\sharp)\\ &= \dot\alpha\left(\left\{\{\varrho' \mid \exists\varrho \in r, \langle id := a, \varrho\rangle \to \varrho'\} \mid r \in \dot\gamma(\varrho^\sharp)\right\}\right)\\ &= \dot\alpha\left(\left\{\{\varrho[id \mapsto (pp, v)] \mid \exists\varrho \in r, \mathbb{A}[\![a]\!]\varrho = v\} \mid r \in \dot\gamma(\varrho^\sharp)\right\}\right)\end{aligned}$$

Hence, for all $x \in Var$ such that $x \neq id$:

$$(\alpha^{\triangleright}_{com}([\![^{pp}id := a]\!]_c)\varrho^\sharp)(x) \subseteq_\otimes \varrho^\sharp(x)$$

Additionnally, for $x = id$:

$$\begin{aligned}(\alpha^{\triangleright}_{com}(c)\varrho^\sharp)(id) &= \Big(\bigcup_{r' \in [\![c]\!]_c \circ \dot\gamma(\varrho^\sharp)} \Pi_1(@(r')(id)), \max_{r' \in [\![c]\!]_c \circ \dot\gamma(\varrho^\sharp)} |\Pi_2(@(r')(id))|\Big)\\ &= \Big(\{pp\}, \max_{r' \in [\![c]\!]_c \circ \dot\gamma(\varrho^\sharp)} |\Pi_2(\{\varrho(id) : \varrho \in r'\})|\Big)\\ &= \Big(\{pp, \max_{r \in \dot\gamma(\varrho^\sharp)} |\{\mathbb{A}[\![a]\!]\varrho : \varrho \in r\}|\}\Big)\\ &= \Big(\{pp\}, \alpha_v \circ \mathbb{A}_c[\![a]\!]\varrho^\sharp \circ \dot\gamma(\varrho^\sharp)\Big)\\ &\subseteq_\otimes \Big(\{pp\}, \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp\Big)\end{aligned}$$

Hence,

$$\begin{aligned}\alpha^{\triangleright}_{com}({}^{pp}id := a)\varrho^\sharp &\subseteq_\otimes \varrho^\sharp[id \mapsto (pp, \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp)]\\ &\triangleq [\![{}^{pp}id := a]\!]^\sharp\varrho^\sharp\end{aligned}$$

– For conditional instructions $c = {}^{pp}if\ (a)\ c_1\ else\ c_2$:

$$\begin{aligned}\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp &= \dot\alpha\left(\left\{\{\varrho' \mid \exists\varrho\in r, \langle c,\varrho\rangle \to^* \varrho'\} \mid r\in\dot\gamma(\varrho^\sharp)\right\}\right)\\ &= \dot\alpha\left(\left\{\{\varrho' : \exists\varrho\in r, \exists v\in\{0,1\}, \mathbb{A}[\![a]\!]\varrho = v\right.\right.\\ &\qquad\left.\left.\wedge\langle c_v,\varrho\rangle\to\varrho'\} \mid r\in\dot\gamma(\varrho^\sharp)\right\}\right)\end{aligned}$$

First, if $\mathbb{A}^\sharp[\![a]\!]\varrho^\sharp = 1$, then expression a evaluates to at most one value in each set $r\in\dot\gamma(\varrho^\sharp)$:

$$\forall r\in\dot\gamma(\varrho^\sharp), \exists v\in\{0,1\}, \forall\varrho\in r, \mathbb{A}[\![a]\!]\varrho = v$$

Therefore, the sets $r\in\dot\gamma(\varrho^\sharp)$ can be partitioned into sets r_1 (resp. r_0) where expression a evaluates to 1 (resp. evaluates to 0):

$$\begin{aligned}\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp &= \dot\alpha\left(\left\{\{\varrho' \mid \exists\varrho\in r_1, \langle c_1,\varrho\rangle\to^*\varrho'\} \mid r_1\in\dot\gamma(\varrho^\sharp)\right\}\right.\\ &\qquad\left.\cup\left\{\{\varrho' \mid \exists\varrho\in r_0, \langle c_0,\varrho\rangle\to^*\varrho'\} \mid r_0\in\dot\gamma(\varrho^\sharp)\right\}\right)\\ &\dot\subseteq_\otimes (\dot\alpha \text{ preserves joins})\\ &\quad\dot\alpha\left(\left\{\{\varrho' \mid \exists\varrho\in r_1, \langle c_1,\varrho\rangle\to^*\varrho'\} \mid r_1\in\dot\gamma(\varrho^\sharp)\right\}\right)\\ &\qquad\dot\cup_\otimes\,\dot\alpha\left(\left\{\{\varrho' \mid \exists\varrho\in r_1, \langle c_1,\varrho\rangle\to^*\varrho'\} \mid r_1\in\dot\gamma(\varrho^\sharp)\right\}\right)\\ &= \left(\dot\alpha\circ[\![c_1]\!]_c\circ\dot\gamma(\varrho^\sharp)\right)\dot\cup_\otimes\left(\dot\alpha\circ[\![c_0]\!]_c\circ\dot\gamma(\varrho^\sharp)\right)\\ &\dot\subseteq_\otimes (\text{by induction hypothesis Equation (6.17)})\\ &\quad[\![c_1]\!]^\sharp\varrho^\sharp\,\dot\cup_\otimes\,[\![c_0]\!]^\sharp\varrho^\sharp\end{aligned}$$

Second, if $\mathbb{A}^\sharp[\![a]\!]\varrho^\sharp > 1$, then for variables x that are modified in neither c_1 nor c_0, we have:

$$(\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp)(x) = \left([\![c_1]\!]^\sharp\varrho^\sharp\,\dot\cup_\otimes\,[\![c_0]\!]^\sharp\varrho^\sharp\right)(x)$$

Finally, for variables that are modified in either $\mathtt{c}_1$ or $\mathtt{c}_0$:

$$
\begin{aligned}
(\alpha^{\triangleright}_{com}([\![c]\!]_c)\varrho^\sharp)(x) &= \dot\alpha\left(\left\{\{\varrho' \mid \exists \varrho \in r, \exists v \in \{0,1\}, \mathbb{A}[\![a]\!]\varrho = v \right.\right.\\
&\qquad \left.\left. \wedge \langle c_v, \varrho\rangle \to \varrho'\} \mid r \in \dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&= \dot\alpha\left(\left\{\{\varrho' \mid \exists \varrho \in r, \mathbb{A}[\![a]\!]\varrho = 1 \wedge \langle c_1, \varrho\rangle \to^* \varrho'\}\right.\right.\\
&\qquad \left.\left. \cup \{\varrho' \mid \exists \varrho \in r, \mathbb{A}[\![a]\!]\varrho = 0 \wedge \langle c_0, \varrho\rangle \to^* \varrho'\}\right\}\right)(x)\\
&\subseteq_\otimes \dot\alpha\left(\left\{\{\varrho' \mid \exists \varrho \in r_2, \mathbb{A}[\![a]\!]\varrho = 1 \wedge \langle c_1, \varrho\rangle \to^* \varrho'\}\right.\right.\\
&\qquad \cup \{\varrho' \mid \exists \varrho \in r_1, \mathbb{A}[\![a]\!]\varrho = 0 \wedge \langle c_0, \varrho\rangle \to^* \varrho'\}\\
&\qquad \left.\left. \mid r_1, r_2 \in \dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes \dot\alpha\left(\left\{\{\varrho' \mid \varrho' \in r_1'\} \cup \{\varrho' \mid \varrho' \in r_2'\} \mid r_1' \in [\![c_1]\!]_c \circ \dot\gamma(\varrho^\sharp),\right.\right.\\
&\qquad \left.\left. r_2' \in [\![c_0]\!]_c \circ \dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes (\dot\gamma \circ \dot\alpha \text{ is extensive, and } \dot\alpha \text{ is monotone})\\
&\qquad \dot\alpha\left(\left\{\{\varrho' \mid \varrho' \in r_1'\} \cup \{\varrho' \mid \varrho' \in r_2'\} \mid \right.\right.\\
&\qquad\quad r_1' \in \dot\gamma \circ \dot\alpha \circ [\![c_1]\!]_c \circ \dot\gamma(\varrho^\sharp),\\
&\qquad\quad \left.\left. r_2' \in \dot\gamma \circ \dot\alpha \circ [\![c_0]\!]_c \circ \dot\gamma(\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes (\text{By hypothesis in Equation (6.17) and monotony of } \dot\alpha)\\
&\qquad \dot\alpha\left(\left\{\{\varrho' \mid \varrho' \in r_1'\} \cup \{\varrho' \mid \varrho' \in r_2'\} \mid \right.\right.\\
&\qquad\quad \left.\left. r_1' \in \dot\gamma([\![c_1]\!]^\sharp\varrho^\sharp), r_2' \in \dot\gamma([\![c_0]\!]^\sharp\varrho^\sharp)\right\}\right)(x)\\
&\subseteq_\otimes (\text{By definition of } (\dot\alpha, \dot\gamma))\\
&\qquad \Big(proj_1([\![c_1]\!]^\sharp\varrho^\sharp(x)) \cup proj_1([\![c_0]\!]^\sharp\varrho^\sharp(x)),\\
&\qquad\quad proj_2([\![c_1]\!]^\sharp\varrho^\sharp(x)) + proj_2([\![c_0]\!]^\sharp\varrho^\sharp(x))\Big)
\end{aligned}
$$

Hence, the abstract semantics of conditionals is sound:

$$
\begin{aligned}
[\![^{pp}if\ (a)\ c_1\ else\ c_2]\!]^\sharp\varrho^\sharp \triangleq\ & \text{let } n = \mathbb{A}^\sharp[\![a]\!]\varrho^\sharp \text{ in}\\
& \text{let } \varrho_1^\sharp = [\![c_1]\!]^\sharp\varrho^\sharp \text{ in}\\
& \text{let } \varrho_2^\sharp = [\![c_2]\!]^\sharp\varrho^\sharp \text{ in}\\
& \lambda id. \begin{cases} \varrho_1^\sharp(id) \cup_\otimes \varrho_2^\sharp(id) & \text{if } n = 1\\ \varrho_1^\sharp(id) \cup_{add(c_1,c_2)} \varrho_2^\sharp(id) & \text{otherwise} \end{cases}
\end{aligned}
$$

— For sequences $c_1;\ c_2$:

$$\alpha^{\triangleright}_{com}(c_1;\ c_2)\varrho^\sharp = \dot\alpha\Big(\{\{\varrho_2 : \exists \varrho \in r, \varrho \to_{c_1;\ c_2} \varrho_2\} : r \in \dot\gamma(\varrho^\sharp)\}\Big)$$

$$
\begin{aligned}
&= \dot{\alpha}\Big(\{\{\varrho_2 : \exists \varrho_1 \in r_1, \varrho_1 \to_{c_2} \varrho_2\} : r_1 \in [\![c_1]\!]_c \dot{\gamma}(\varrho^\sharp)\}\Big) \\
&\dot{\subseteq}_\otimes \dot{\alpha}\Big(\{\{\varrho_2 : \exists \varrho_1 \in r_1, \varrho_1 \to_{c_2} \varrho_2\} : r_1 \in \dot{\gamma} \circ \dot{\alpha}\left([\![c_1]\!]_c \dot{\gamma}(\varrho^\sharp)\right)\}\Big) \\
&\dot{\subseteq}_\otimes \dot{\alpha}\Big(\{\{\varrho_2 : \exists \varrho_1 \in r_1, \varrho_1 \to_{c_2} \varrho_2\} : r_1 \in \dot{\gamma}\left([\![c_1]\!]^\sharp \varrho^\sharp\right)\}\Big) \\
&\dot{\subseteq}_\otimes [\![c_2]\!]^\sharp \left([\![c_1]\!]^\sharp \varrho^\sharp\right)
\end{aligned}
$$

— For while loops ${}^{pp}while\ (a)\ c$:

$$
\begin{aligned}
\alpha^{\triangleright}_{com}({}^{pp}while\ (a)\ c)\varrho^\sharp &= \dot{\alpha} \circ [\![{}^{pp}while\ (a)\ c]\!]_c \circ \dot{\gamma}(\varrho^\sharp) \\
&= (\text{By characterizing the collecting semantics for loops} \\
&\quad \text{as a least fixpoint, as in [CP10, Section 4],} \\
&\quad \text{with } F_{X_0} = \lambda X.X_0 \cup [\![{}^{pp}if\ (a)\ c\ else\ {}^{pp}skip]\!]_c\ ) \\
&\quad \dot{\alpha}\left(\mathrm{lfp}^{\subseteq}_{\dot{\gamma}(\varrho^\sharp)} F\right) \\
&\dot{\subseteq}_\otimes (\text{By the fixpoint transfer theorem}) \\
&\quad \mathrm{lfp}^{\dot{\subseteq}_\otimes}_{\varrho^\sharp} [\![{}^{pp}if\ (a)\ c\ else\ {}^{pp}skip]\!]^\sharp
\end{aligned}
$$