Computationally Secure Information Flow

Dissertation

Zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

von

Peeter Laud

Saarbrücken
April, 2002

ISBN 9985-78-703-X

Tag des Kolloquiums:	16.09.2002
Dekan:	Prof. Dr. Philipp Slusallek
Gutachter:	Prof. Dr. Reinhard Wilhelm Prof. Dr. Birgit Pfitzmann
Vorsitzender:	Prof. Dr. Harald Ganzinger

Abstract

This thesis presents a definition and a static program analysis for secure information flow. The definition of secure information flow is not based on non-interference, but on the computational independence of the program's public outputs from its secret inputs. Such definition allows cryptographic primitives to be gracefully handled, as their security is usually defined to be only computational, not information-theoretical.

The analysis works on a simple imperative programming language containing a cryptographic primitive—encryption—as a possible operation. The analysis captures the intuitive qualities of the (lack of) information flow from a plaintext to its corresponding ciphertext. We prove the analysis correct with respect to the definition of secure information flow described above. In the proof of correctness we assume that the encryption primitive hides the identity of plaintexts and keys.

This thesis also considers the case where the identities of plaintexts and keys are not hidden by encryption, i.e. given two ciphertexts it may be possible to determine whether the corresponding plaintexts are equal or not. We also give an analysis for this case, though it is not a whole program analysis. Namely, we cannot analyse loops. Nevertheless, with the help of the analysis one can check, whether two formal expressions (which are equivalent to the output of programs without loops) have indistinguishable interpretations as bit-strings.

Zusammenfassung

In dieser Dissertation wird eine Definition und eine statische Programmanalyse für sicheren Informationsfluß präsentiert. Die Definition des sicheren Informationsflusses basiert nicht auf der Unbeeinflußbarkeit, sondern auf der komplexitätstheoretischen Unabhängigkeit der öffentlichen Ausgaben des Programms von seinen geheimen Eingaben. Eine solche Definition erlaubt uns, kryptographische Primitiven elegant zu bearbeiten, weil ihre Sicherheit meistens nur komplexitätstheoretisch und nicht informationstheoretisch definiert ist.

Die Analyse arbeitet auf einer einfachen imperativen Programmiersprache, die eine kryptographische Primitive—Verschlüsselung—als eine mögliche Operation enthält. Die Analyse gibt die intuitive Eigenschaft des (nicht vorhandenen) Informationsflusses von einem Klartext zu dem entsprechenden Schlüsseltext wieder. Wir geben den Korrektheitsbeweis der Analyse in Bezug auf die obengegebene Definition des sicheren Informationflusses. Im Beweis nehmen wir an, daß die Verschlüsselungsprimitive die Identität der Klartexte und Schlüssel versteckt.

Diese Dissertation behandelt auch den Fall, daß die Verschlüsselungsprimitive die Identität der Klartexte und Schlüssel nicht versteckt, d.h. daß man aus zwei Schlüsseltexten möglicherweise herausfinden kann, ob die entsprechenden Klartexte gleich sind oder nicht. Wir geben eine Analyse auch für diesen Fall an, obwohl sie nicht auf ganze Programme anwendbar ist, da wir keine Schleifen analysieren können. Mit Hilfe dieser Analyse kann man feststellen, ob zwei formale Ausdrücke (die gleichwertig zu der Ausgabe der Programme ohne Schleifen sind) gleiche Interpretation als Bitfolgen haben.

Extended Abstract

When is a program safe to run? One aspect of safety is confidentiality, which arises when the inputs and outputs of the program are partitioned into several different security classes. Typically, some inputs of the program may be public and some confidential; some outputs of the program may become public and some remain private. In this case one wants the program to have *secure information flow* — one wants that nothing about the confidential inputs of the program can be deduced by observing the public outputs.

A standard tool for keeping information confidential is encryption. A ciphertext must not reveal the corresponding plaintext to someone that does not have the key. Theoretically, this hiding of information is not absolute (at least when the encryption key is shorter than the plaintext), though, someone with (unrealistically) high computing power may be able to uncover the plaintext from only the ciphertext.

In this thesis we present a definition for secure information flow that considers a program to be secure if no *reasonably* powerful attacker can learn something about the secret inputs of the program by only observing its public outputs. Such *computational* security contrasts with most of definitions proposed earlier, which assert *information-theoretical* security, requiring that no attacker at all (irrespective of its power) can deduce something about the secret inputs from public outputs.

Cryptology is the branch of computer science where the definitions stating the inability of *reasonably* powerful adversaries are common. Our definition of secure information flow also has cryptographic nature. The main notion used in the definition is that of *independence*. Classically, two random events are independent if the probability that the first one happens is the same no matter whether the second one has happened or not. When we are talking about the events happening, then we always assume that it is easy to figure out, whether the event has indeed happened or not. For example, we do not consider events like "formula $\mathcal{X}$ is satsifiable", where $\mathcal{X}$ is a random formula of the propositional calculus.

In computational independence, there may be a difference of the probabilities of the first event happening, depending on whether the second event happens or not, but this difference must be *negligible*. If we had not required the outcomes of events to be explicit, then the definition of computational independence would have been more complex (the negligibility of the difference of probabilities would still have been sufficient, but no longer necessary). The notion of independence also generalises to the case, where the first and the second event can have more

outcomes than just two (either happening or not). We define a program to have computationally secure information flow, if its public outputs are independent of its secret inputs.

In this thesis we also present a static program analysis for secure information flow. The concrete semantics of the program is defined to be a mapping over the probability distributions of program states — the program transforms the probability distribution of its inputs to the probability distribution of its outputs. We abstract the probability distributions over program states by pairs of sets of program variables — if (X, Y), where X and Y are sets of variables, belongs to the abstraction of a distribution over program states, then the values of variables in X are independent from the values of variables in Y in this distribution. The abstract semantics (i.e. the analysis) is a mapping over the set of pairs of sets of program variables. Given an abstraction of the probability distribution of program's inputs, the abstract semantics maps it to an abstraction of the probability distribution of the program's outputs.

Note that formally there can be several different abstractions of the same probability distribution — if the values of the variables in X are independent of the values of the variables in Y, then the pair (X, Y) can either belong or not belong to the abstraction of that distribution. On the other hand, if the values of the variables in X are not independent from the values of the variables in Y, then the pair (X, Y) certainly cannot belong to the abstraction of that distribution. We see that there exists a best abstraction of a distribution — it is the abstraction that contains all pairs (X, Y) that it can correctly contain. "Program analysis is allowed to err on the safe side" — given the best abstraction of the probability distribution of the program's inputs, the abstract semantics maps it to an abstraction of the probability distribution of the program's outputs, but not necessarily to the best abstraction. Finding the best abstraction of the distribution of program's outputs is in general uncomputable, therefore we have to be content with a possibly suboptimal abstraction. Now the aim is to get an abstraction that is still "good enough" and we believe that the analysis that we have devised gives such an abstraction.

This analysis assumes that the encryption operation satisfies some rather strong (but still totally realistic) security properties. Namely, given two ciphertexts it must be impossible to find out whether their underlying plaintext is the same or not (note that for satisfying this property, the encryption operation cannot be deterministic) and whether they are created with the same key or not. The encryption operations with such properties are usually constructed from simpler, "primitive" operations that themselves satisfy some weaker security properties. Mostly, these primitive operations are considered to be *pseudorandom permutations*, i.e. they are assumed to look like a uniformly chosen random permutation of the message space for someone with reasonable computing power.

In this thesis we also present an analysis for the case, where the encryption operation is only a pseudorandom operation and not something with stronger security properties. The analysis is not a full program analysis, we cannot analyse loops. Similarly to the previous analysis, this one gives us information about the independence of certain sets of variables from other ones. For the purposes of comparing our results with some earlier ones, we have presented the analysed structure not as a simple programming language, but as a formal language of expressions. For the same reason we have added to our analysis the capability to show that two formal expressions have the same semantics. These earlier results are still quite different from ours, mainly because they assume that the encryption operation satisfies stronger security properties (same properties that we described before). We are not aware of any other attempt to automate the analysis of systems containing pseudorandom permutations.

Ausführliche Zusammenfassung

Wann ist ein Programm sicher auszuführen? Ein Aspekt der Sicherheit ist Vertraulichkeit, die sich daraus ergibt, daß die Eingaben und Ausgaben des Programms in verschiedene Sicherheitsklassen aufgeteilt werden. Typischerweise sind einige Eingaben des Programms vertraulich und andere nicht; einige Ausgaben des Programms können öffentlich werden, andere bleiben aber geheim. In diesem Fall möchte man garantieren, daß das Programm *sicheren Informationsfluß* hat — man möchte, daß nichts über die vertraulichen Eingaben ableitbar ist, wenn man nur die öffentlichen Ausgaben des Programms kennt.

Verschlüsselung ist eine Standardoperation um Information geheim zu halten. Der Schlüsseltext darf jemandem, der den Schlüssel nicht kennt, den entsprechenden Klartext nicht verraten. Dieses Verstecken der Information ist theoretisch noch nicht absolut (wenigstens wenn der Schlüssel kürzer als der Klartext ist) — jemand der über eine (unrealistisch) hohe Rechnerleistung verfügt, kann den Klartext nur aus dem Schlüsseltext finden.

Wir geben in dieser Dissertation eine Definition für sicheren Informationsfluss. Ein Programm gilt dann als sicher, wenn kein *vernünftig* leistungsfähiger Angreifer aus den öffentlichen Ausgaben des Programms etwas über vertrauliche Eingaben ableiten kann. Solche *komplexitätstheoretische* Definition steht im Kontrast zu den meisten früher vorgeschlagenen Definitionen, die die *informationstheoretische* Sicherheit behaupten. Sie erfordern, daß überhaupt kein Angreifer (ungeachtet von seiner Leistungsfähigkeit) etwas über vertrauliche Eingaben aus den öffentlichen Ausgaben ableiten kann.

In der Kryptologie sind Aussagen über die Unfähigkeit eines *vernünftig* leistungsfähigen Angreifers weit verbreitet. Unsere Definition des sicheren Informationsflusses hat auch diesen kryptographischen Charakter. Der Hauptbegriff, der in der Definition benutzt wird, ist die *Unabhängigkeit*. Im klassischen Fall sagt man, daß zwei Ereignisse unabhängig sind, wenn die Wahrscheinlichkeit, daß das erste vorkommt, immer gleich ist, ungeachtet davon, ob das zweite Ereignis vorgekommen ist oder nicht. Wenn wir über die Vorkommen der Ereignisse reden, dann nehmen wir immer an, daß es einfach auszufinden ist, ob das jeweilige Ereignis wirklich vorgekommen ist oder nicht. Zum Beispiel, das Ereignis „Formel $\mathcal{X}$ ist erfüllbar“, wo $\mathcal{X}$ eine zufällige aussagenlogische Formel ist, betrachten wir nicht.

Bei der komplexitätstheoretischen Unabhängigkeit dürfen die Wahrscheinlichkeiten, daß das erste Ereignis vorkommt, wenn das zweite vorgekommen ist, und daß das erste Ereignis vorkommt, wenn das zweite nicht vorgekommen ist, auch unterschiedlich sein, aber dieser Unterschied darf nur *vernachlässigbar klein* sein. Hätten wir nicht erfordert, daß die Auskommen der Ereignisse explizit sind, dann wäre die Definition der komplexitätstheoretischen Unabhängigkeit komplizierter gewesen (die Vernachlässigbarkeit des Unterschieds der Wahrscheinlichkeiten wäre noch immer hinreichend gewesen, aber nicht mehr notwendig). Die Unabhängigkeit läßt sich auch für den Fall verallgemeinern, in dem die Ereignisse mehr als zwei (entweder kommt vor oder kommt nicht vor) mögliche Endergebnisse haben können. Wir haben definiert, daß ein Programm sicheren Informationsfluß hat, wenn seine öffentliche Ausgaben von seinen vertraulichen Angaben unabhängig sind.

Wir legen in dieser Dissertation auch eine statische Programmanalyse für sicheren Informationsfluß vor. Die konkrete Semantik des Programms wird als eine Abbildung über die Menge der Wahrscheinlichkeitsverteilungen über Programmzustände definiert — das Programm wandelt die Verteilung seiner Eingaben in die Verteilung seiner Ausgaben um. Wir abstrahieren die Verteilungen über Programmzustände mit Hilfe von Paaren von Mengen der Programmvariablen — falls ein Paar (X, Y) von Variablenmengen zu der Abstraktion einer Verteilung über Programmzustände gehört, dann bedeutet das, daß in dieser Verteilung die Werte der Variablen in X unabhängig von Werten der Variablen in Y sind. Die abstrakte Semantik (d.h. die Analyse) ist eine Abbildung über die Menge der Paare von Mengen der Programmvariablen. Die abstrakte Semantik wandelt eine Abstraktion der Verteilung der Programmeingaben in eine Abstraktion der Verteilung der Programmausgaben um.

Man muß beachten, daß *eine* Wahrscheinlichkeitsverteilung im allgemeinen *mehrere* Abstraktionen haben kann — wenn die Werte der Variablen in X unabhängig von den Werten der Variablen in Y sind, darf das Paar (X, Y) zu der Abstraktion gehören, aber es darf auch außerhalb der Menge, die die Abstraktion der Verteilung ist, bleiben. Andererseits darf das Paar (X, Y) keinesfalls zu der Abstraktion dieser Verteilung gehören, wenn die Werte der Variablen in X nicht unabhängig von den Werten der Variablen in Y sind. Deshalb sehen wir, daß es eine beste Abstraktion einer Verteilung gibt — diese Abstraktion enthält alle solche Paare (X, Y), die sie enthalten darf. "Die Programmanalyse darf auf der sicheren Seite irren" — angewendet auf die beste Abstraktion der Wahrscheinlichkeitsverteilung der Programmeingaben muß die abstrakte Semantik *eine* Abstraktion der Wahrscheinlichkeitsverteilung der Programmausgaben liefern, aber nicht unbedingt die beste Abstraktion. Das Problem, die beste Abstraktion der Verteilung zu finden, ist im allgemeinen unberechenbar, deshalb muß man mit einer möglicherweise suboptimalen Abstraktion zufrieden sein. Das Ziel hier ist, eine Abstraktion zu bekommen, die "gut genug" ist. Wir glauben, daß unsere Analyse eine solche Abstraktion liefert.

Bei dieser Analyse nehmen wir an, daß die Verschlüsselungsoperation einige ziemlich starke (aber immer noch völlig realistische) Sicherheitsbedingungen erfüllt. Betrachtet man zwei Schlüsseltexte, so muß es unmöglich sein festzustellen, ob die zu Grunde liegenden Klartexte gleich sind oder nicht (beachten wir, daß keine Verschlüsselungsprimitive, die diese Bedingung erfüllt, deterministisch sein kann), und ob die Schlüssel, die man beim Erzeugen dieser Schlüsseltexte benutzt hat, gleich sind oder nicht. Solche Verschlüsselungsoperationen werden meistens aus einfacheren, "primitiven" Operationen, die gewisse schwächere Sicherheitsbedingungen erfüllen, konstruiert. Meistens nimmt man an, daß diese primitiven Operationen *Pseudozufallspermutationen* sind, d.h. für einen Betrachter, der nur über eine realistische Rechnerleistung verfügt, sollen sie wie zufällig, uniform gewählte Permutationen der Textmenge aussehen.

In dieser Dissertation legen wir desweiteren eine Analyse vor, die den Fall betrachtet, wo die Verschlüsselungsoperation nur eine Pseudozufallspermutation ist und keine stärkeren Sicherheitsbedingungen erfüllt. Diese Analyse ist keine vollständige Analyse, da wir Schleifen nicht analysieren können. Diese Analyse, wie auch die vorher beschriebene, gibt uns Informationen über die Unabhängigkeit bestimmter Variablenmengen von anderen. Wir haben den analysierten Formalismus nicht als Programmiersprache, sondern als formale Sprache der Ausdrücke dargestellt, weil wir unsere Ergebnisse mit einigen früher veröffentlichten vergleichen wollen. Deshalb haben wir unserer Analyse auch die Fähigkeit gegeben, zu zeigen, daß zwei formale Ausdrücke gleiche Semantik haben. Diese früher veröffentlichten Ergebnisse sind jedoch ziemlich unterschiedlich von unseren, hauptsächlich weil sie stärkere Sicherheitsbedingungen (diejenigen die wir vorher beschrieben haben) von der Verschlüsselungsoperation erfordern. Uns sind keine weitere Versuche bekannt, die die Analyse der Pseudozufallspermutationen enthaltenden Systeme automatisieren.

Acknowledgements

I would like to thank my advisor Prof. Reinhard Wilhelm for inviting me to Saarbrücken, introducing me to the topic of language-based security and letting me to pursue it on my own. I am also thankful to him for proofreading this thesis and for numerous suggestions for better presentation of the material.

Prof. Birgit Pfitzmann has also suggested numerous improvements to this thesis. I thank her for them.

Christian Probst has proofread the parts of this thesis that are in German. Danke schön.

I am thankful to German Science Foundation for supporting the research presented in this thesis by a graduate fellowship under the graduate studies program "Quality Guarantees for Computer Systems".

In the implementation of the analysis presented in this thesis I have used some third-party software. Namely, I have used a library for binary decision diagrams provided by the model-checking group[1] at Carnegie-Mellon University. The graph visualisation package aiSee[2] by AbsInt Angewandte Informatik GmbH is used to show the results produced by the implementation. I am grateful to both of them.

Finally, I thank my co-workers for the pleasant working atmosphere and my parents, relatives and girlfriend Monica for their support and patience.

[1] `http://www.cs.cmu.edu/~modelcheck`

[2] `http://www.aisee.com`

Contents

Chapter 1

Introduction

Security is an important aspect of computer systems. In broad terms, a system is secure if it does not have any undesired functionality.

Security of a system has two basic aspects:

Confidentiality. The system does not publish something about the secret part of its local state. Typical examples of the contents of the secret part of the local state are passwords or personal data. Hence the functionality for reading secret data over the public interface of the system is absent.

Integrity. The system does not allow its state to be changed in an unauthorised manner. A typical example of the state that has to be changed with care is the balance of various accounts. Hence, the system offers only limited functionality for changing that part of its state.

Classically, *availability* is considered to be the third basic aspect of security. Availability means, that the normal operation of the system must be hard to disrupt. Obviously, any reasonable system must have availability. However, availability is functionality, not the absence of functionality. Therefore we advocate not classifying availability as an aspect of security.

In this thesis we consider a system that contains a computer that has a local storage and an interface to an outside network, and a program running on that computer. This program may access both the local storage and the network.

In this setup, we may either have written the program ourselves or have obtained it from a third party. There are a lot of such parties that offer programs. It is impossible for us to trust them all — we just do not have enough resources to convince ourselves of the trustworthiness of every single source of software that we execute on our computer. Often, we do not even notice that we have obtained a program that may be running on our computer — executability is becoming more and more ubiquitous, all kinds of document formats allow for executable content.

In the previous paragraph, trustworthiness may mean two different things — goodwill and competence. Therefore it is not unthinkable to treat even ourselves untrusted as the source of programs.

Running a program received from an untrusted source is always a security risk as the program may attempt to use the resources of its host computer in an unauthorised way.

Security is a complex matter and requires covering all of its aspects. This thesis only deals with the confidentiality aspect of system security. In a general setting, this would not be enough to convince oneself that the system is secure. However, confidentiality is sufficient for security, if the attackers of the system are passive. A passive attacker can observe the public output of the system, but it cannot influence the inputs of the system. In our setup, an attacker is some entity that is connected to the network. The public outputs of the system are the data that the program writes to the network. The passivity of the attacker means, that it does not influence what the program reads from the network. Alternatively, it may mean that the program does not read anything from the network.

1.1 Secure Information Flow

The topic of this thesis is secure information flow. A program is said to have secure information flow, if the data that it makes public (for example, by writing it to the network) do not depend on the secret input data of this program (for example, the secret data on the local storage). Secure information flow is a particularly strong form of confidentiality. If a program has secure information flow, then it is secure against passive adversaries.

If we are going to run a program obtained from an untrusted source, then we must put checks on it. The checks can be done either during or before the execution.

Dynamic checking — checks during the execution — means that the program is enclosed in some kind of a sandbox that controls the accesses of the program to the local storage and the network. Probably the most known sandbox is the Java Virtual Machine (JVM) [LY99]. In the default setting, this sandbox does not allow the program to access the local storage at all.

The JVM allows the user to authorise the program to access specific files on the local storage in a specific way (either read, write, rename,...). If we give the program extra privileges, then we must have already convinced ourselves in the trustworthiness of the supplier of the program. Mechanisms of assigning extra trust to the programs and possible meanings of such assignments are not the topic of this thesis.

Entirely preventing the program to access secret data is a rather coarse-grained approach. A useful refinement of the sandbox is its history-sensitivity. For example, the sandbox could allow the program to access secret data, but after the program has accessed it, forbid any further network activity. If we consider the *traces* of the program — possible sequences of successive states and actions of the computer executing the program — then the sandbox would only allow the program to have such traces that have no network writes after reads of secret data. The enforcement mechanisms for allowing only "good" traces have been researched by Schneider

[Sch00]. It turns out that there are certain constraints on the subset of the set of traces that can serve as the set of "good" traces.

The checks that a sandbox makes for ensuring confidentiality could be seen as instances of Bell-LaPadula's "no read up" and "no write down" rules [Gol99]. They rule out certain flows of information either by preventing the computer to access secret data or by preventing it to publish the secret data that it has already read.

The absence of certain information flows is a property of program traces. Volpano [Vol99] has argued that confidentiality is not really a *property* of program traces but a *relation* over them — all possible traces must look "the same" to an external observer. Indeed, a program that tries to write something to the network after having read secret data may still preserve confidentiality, if it always writes the same data to the network, no matter what the secret data was. Thus a method that ensures the absence of certain information flows must label some programs falsely insecure.

In some sense, thinking about confidentiality as a relation over program states makes it harder to convince ourselves about the security of the program. Namely, while one may attempt to test, whether all program traces have a certain property (of not having certain information flows), it is impossible to test, whether all program traces look "the same". Instead of testing, we must do something different, for example prove the similarity of all program traces.

Our approach has been to design static analyses for checking the observable sameness of program traces. Basically, it is an attempt to automatically derive a proof that all possible traces look the same. A nice property of this approach (designing static analyses) is, that analysing a program is a completely automatic process. The approach has been pioneered by Denning [Den76, DD77].

Another, related approach for proving the program traces similar is to use typings [VSI96, ABHR99]. The types are assigned to program variables and also to program fragments; these types are essentially the same as the security classes. The type-theoretical approach usually gives simpler checking procedures than static program analysis. They may be somewhat inadequate for our purposes.

1.2 Encryption in Programs

A standard way to ensure the confidentiality of certain data is to employ encryption. The program analysis must be able to handle the encryption operation gracefully — on the one hand, it must record that a plaintext is recoverable from the corresponding ciphertext and the key, on the other hand, it should note that having the ciphertext alone is generally not enough to recover the plaintext. An analysis without the last property cannot expect to deliver interesting results about programs that make significant use of encryption.

What is an encryption, actually? Two different approaches to describe the nature of cryptographic operations have evolved over the years. One of them is the Dolev-Yao model [DY83, BAN90, AG99], where the cryptographic operations are operators on formal expressions; the security properties of the operations are given by rules for constructing and destructing the expressions. For example, one may specify that for destructing a ciphertext (and for obtaining the underlying plaintext), one needs to know the key that was used by constructing that ciphertext. The other approach models messages as bit-strings and cryptographic operations as probabilistic functions over bit-strings. The security properties of the operations are expressed in terms of success probabilities of resource constrained adversaries. Therefore the protection provided by encryption is modelled to be not absolute, but only good enough against realistic attackers[1].

The first approach is easier to argue about and lends itself more readily to automatic tool support — reasoning about formal expressions is easier that about the maximal success probabilities of adversaries. The second approach is more natural — in real world, the messages are bit-strings, there are no formal messages in real world. One has to make a (greater) leap of faith when translating the results obtained by the first approach to the real world.

The approach that we have chosen here is the second, computational approach. It has the following advantages:

- It is closer to the real world.
- It gives us simpler semantics and security definitions.

Indeed, we have to give semantics for whole programs, these programs have richer structure than formal expressions. It is simple to define that programs work on bit-strings, it is not so simple to define that they work on formal expressions. If we had chosen the first, computational approach, then we would have had to introduce other, non-cryptographic, operations to the language of formal expressions. We also would have had to define, when two formal messages are equal (beyond the syntactic equality). And then we would have had the question, how well these definitions model the real world.

Generally, these two approaches seem rather different. There exist some results, though, hinting that they may indeed be equivalent [AR00, AJ01]. So far, their results only hold for the case where the adversary is passive and the size of messages is bounded. This actually makes their work quite comparable with ours, as we also consider only passive adversaries. We hope that our work will also advance the understanding of the equivalence of formal and computational approaches.

1.3 Our Contribution

In this thesis we give a definition of secure information flow that corresponds to the assumption that the attackers of the system, trying to find out something about the program's secret inputs from its public outputs, only have reasonable resources

[1] Indeed, one can show that if the encryption key is shorter than the plaintext, then a powerful enough attacker is able to recover something about the plaintext from the ciphertext only.

at their disposal. Such a definition is necessary for arguing about encryption as an operation preserving confidentiality. Usually, secure information flow is defined so, that no attacker at all, irregardless of its power, can deduce something about the secret inputs of the program from its public outputs. A very powerful attacker can use brute force to find the encryption key (again, we assume that the key is shorter than the plaintext that is encrypted with it), therefore the encryption is not a secure operation for such a definition of security.

The main contribution of this thesis is a program analysis for secure information flow, where the definition of secure information flow assumes resource-bound adversaries. The program analysis is designed for a simple imperative programming language. A possible operation in the language is the encryption $x := \mathcal{E}nc(k, y)$, which assigns to the variable x the ciphertext corresponding to the plaintext y under the key k. The analysis handles this operation in a special way, reflecting that one cannot find anything about the value of y from the value of x alone, the key is also necessary. We also show how to implement this analysis.

The proof of correctness of the analysis unfortunately cannot use standard results from the theory of abstract interpretation. This is caused by the incompatibility of the mathematical structures necessary for defining the semantics of programs and the structures for defining the security. We have devised an *ad-hoc* proof. The problems with the proof are likely to reappear when we are going to also consider active adversaries. We think that our proof may be instructive for the devising of future proofs of analyses for stronger security properties.

The security requirements for the encryption operation that are necessary for the analysis presented in this thesis and also in earlier papers are rather strong (but nonetheless realistic). It is demanded that the encryption operation be *repetition and which-key concealing* [AR00], i.e. it must hide the identity of both plaintexts and keys. Such operations are usually constructed from simpler operations that have weaker security properties. The "primitive" operations (i.e. operations that cannot be considered to be constructed from simpler ones) are usually only pseudorandom permutations. In this thesis we give an analysis that is correct for the case where the encryption is a pseudorandom permutation. It is not a whole program analysis, we cannot analyse loops. As far as we know, this is the first attempt to automate the analysis of systems where the encryption operation is a pseudorandom permutation.

1.4 Overview of the Thesis

In Chapter 2 we give the necessary notation and preliminaries about the complete partial orders, which are needed for defining the semantics of the programming language, and about the complexity-theoretic foundations of cryptography. Cryptography is needed for giving the security definition. Also, the proofs of correctness of the analyses are mostly cryptographic. Chapter 3 gives the syntax and semantics of our programming language and also gives the definition of secure information flow. Chapter 4 gives the program analysis and proves it correct. Chapter 5 describes our implementation of this analysis. In Chapter 6 we deal with pseudorandom permutations. As we said before, we cannot analyse loops, therefore we introduce the language of formal expressions that correspond to programs without loops. We define the computational interpretation of these expressions, this interpretation corresponds to the semantics of programs. The main result of this chapter is an analysis that allows to check, whether the interpretations of two formal expressions are indistinguishable or not. Chapter 7 reviews the related work from the areas of secure information flow and relating the two approaches to cryptography. Finally, in Chapter 8 we make some concluding remarks and suggest future research directions.

Chapter 2

Preliminaries

This chapter gives the notation used in the rest of the thesis and introduces the concepts and basic results that we use.

In Sec. 2.1.1 we start with basic definitions and results about partially ordered sets. We recall the definition of partial orders and present several instances of partially ordered sets that we are going to use in the rest of the thesis. We then proceed to define *complete partial orders* which play a big role in defining the semantics of programming languages. We define properties of functions over (complete) partial orders — monotonicity and continuity. We finish this section by defining fixed points of functions and by presenting some sufficient conditions for their existence.

In Sec. 2.1.2 we present probability distributions over sets and the necessary notations to translate between elements of sets and probability distributions over sets. These notations also include tools to "lift" functions between sets to functions between sets of probability distributions.

Sec. 2.2 gives the necessary definitions from the complexity-theoretical foundations of cryptography. In this thesis, the encryption primitives are used as black boxes, we only use the fact that they satisfy certain security definitions. The task of this section is to present these definitions and also some reductions between them.

The security definitions are asymptotic in kind. They state that the success probability of any attacker rapidly approaches zero, as the complexity of the system increases. Here the measure of system complexity is the length of the secret keys that the system uses. Because of using the asymptotics, we assume that the complexity of the system may be arbitrarily increased.

In Sec. 2.2.1 we fix the notations that we use and give some basic definitions. We clarify, what can be an argument to an algorithm. We also define the notions of polynomial time and polynomial-time computability.

In Sec. 2.2.2 we define the notion of indistinguishability of families of probability distributions. This is a very general notion, most subsequent definitions can be stated as indistinguishability of certain families of distributions. Based on this definition, we give complexity-theoretic definition of independence of random variables. It is used by the analyses in Chapters 4 and 6. We finish this section by giving another, seemingly stronger definition of indistinguishability, mostly for the purpose to show the ideas behind the proof of equivalence of two definitions. The proof of correctness for the analysis presented in Chapter 4, although much more complex, has the same ideas behind it.

Sec. 2.2.3 deals with encryption. It defines the encryption system and gives several different security definitions for it. We define, when an encryption system is a pseudorandom permutation or a pseudorandom function and show that these two notions are actually equivalent. Being a pseudorandom permutation is all that we require from the encryption system for the analysis in Chapter 6. The analysis in Chapter 4 needs more — it is necessary for the encryption to also hide the *identities* of the messages that are encrypted and the identities of the keys that are used. We give the definitions covering these notions in the end of Sec. 2.2.3.

2.1 Domains

We let $\mathbb{N}$ denote the set of nonnegative integers $\{0, 1, 2, 3, \ldots\}$, $\mathbb{Z}$ denote the set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and $\mathbb{B}$ denote the set of booleans $\{\mathsf{true}, \mathsf{false}\}$. Let $\mathbf{Pol}(\mathbb{Z})$ denote the set of polynomials with integer coefficients.

For a family of sets $\{X_n\}_{n\in\mathbb{N}}$, let $\prod_{n\in\mathbb{N}} X_n$ denote their Cartesian product — the set of countable tuples, where the n-th component has the type X_n. For $f : \prod_{n\in\mathbb{N}}(X_n \to Y_n)$ and $x \in X_i$ (we assume that the index i is clear from the context) let $f(x) \in Y_i$ denote the quantity $f_i(x)$.

For a function $f : X \to X$ we let f^0 denote the identity function on X and for each $n \in \mathbb{N}$ denote $f^{n+1} = f \circ f^n$.

For a function $f : X \to Y$ and $X' \subseteq X$ we let $f|_{X'}$ denote the *contraction* of f to X' — $f|_{X'}$ is a function from X' to Y, such that $f|_{X'}(x) = f(x)$ for all $x \in X'$.

2.1.1 Partially Ordered Sets

Let X be a set. A *partial order* on X is a relation $\leq \subseteq X \times X$ that is reflexive ($x \leq x$ for all $x \in X$), antisymmetric ($x \leq y$ & $y \leq x \implies x = y$ for all $x, y \in X$) and transitive ($x \leq y$ & $y \leq z \implies x \leq z$ for all $x, y, z \in X$). A set X together with a partial order $\leq$ is called a *partially ordered set*.

The *reverse* of a partial order $\leq \subseteq X \times X$ is the order $\geq$, where $x \geq y$ iff $y \leq x$. For a partially ordered set X let $\mathcal{R}(X)$ denote the same set with reversed order.

The smallest possible partial order on X is the diagonal relation, where $x \leq y$ iff $x = y$. We can consider each set as a partially ordered set, if we assume that the order on the set is the diagonal relation (unless specified otherwise). If the order on the set X is the diagonal relation, then we call the set X *unordered*.

$X_\perp$ denotes the set $X \uplus \{\perp\}$ (here $\uplus$ denotes the disjoint union). $X_\perp$ is partially ordered — $\perp$ is the smallest element and the other elements are ordered as in X.

The Cartesian product $X \times Y$ of partially ordered sets X and Y is again a partially ordered set. For all $x_1, x_2 \in X$ and $y_1, y_2 \in Y$ define $(x_1, y_1) \leq (x_2, y_2)$ iff $x_1 \leq x_2$ and $y_1 \leq y_2$. Similarly, if Y is a partially ordered set, then the set of functions $X \to Y$ is a partially ordered set, too. The order on $X \to Y$ is defined *pointwise*: for $f_1, f_2 : X \to Y$ define $f_1 \leq f_2$ iff $f_1(x) \leq f_2(x)$ for all $x \in X$.

For a set X, the set of all of its subsets is denoted by $\mathcal{P}(X)$. The set $\mathcal{P}(X)$ is partially ordered, the order is given by the subset inclusion. If X is partially ordered and $Y \subseteq X$, then Y is *downwards* [resp. *upwards*] *closed*, iff $x \in Y$ and $x' \leq x$ [resp. $x' \geq x$] imply $x' \in Y$ for all $x, x' \in X$. The set of all downwards [resp. upwards] closed subsets of X is denoted by $\mathcal{P}_{\mathrm{L}}(X)$ [resp. $\mathcal{P}_{\mathrm{U}}(X)$].

Let X be a set and $\Pi \subset \mathcal{P}(X)$. Then Π is a *partition* of X if for each $x \in X$ there exists exactly one $X' \in \Pi$, such that $x \in X'$. The set of all partitions of X is denoted by $\mathbf{Parts}(X)$. Note that $\mathbf{Parts}(\emptyset)$ contains one element $\Pi_\emptyset$ — the partition with no parts. I.e. $\Pi_\emptyset = \emptyset$ and $\mathbf{Parts}(\emptyset) = \{\emptyset\}$.

Let Π_1, Π_2 be partitions of the set X. We say that Π_1 is *finer* than Π_2 iff for each $X' \in \Pi_1$ there exists $X'' \in \Pi_2$, such that $X' \subseteq X''$. Equivalently, for each two elements $x, y \in X$ the following holds: if there exists a part in Π_1 that contains both x and y then there also exists a part in Π_2 that contains them both. The set $\mathbf{Parts}(X)$ together with the relation "finer" is a partially ordered set.

Let X be a partially ordered set and let $X' \subseteq X$. We say that $a \in X$ is an *upper bound* of X', if $a \geq x$ for all $x \in X'$. The *least upper bound* of X', denoted $\bigvee X'$, if it exists, is an upper bound of X', such that $\bigvee X' \leq a$ for each a that is an upper bound of X'. The *lower bounds* and *greatest lower bound* (denoted $\bigwedge X'$) of a set X' are defined *dually* — by working on $\mathcal{R}(X)$ instead of X.

If $X' = \{x, y\}$ is a two-element set then we denote $\bigvee X'$ also by $x \vee y$ and $\bigwedge X'$ also by $x \wedge y$. The partially ordered set X is a *lattice* if all its two-element subsets (and by induction, all finite subsets) have the least upper bound and the greatest lower bound.

A *chain* of a partially ordered set X is a set $X' \in X$, such that for each $x, y \in X'$, either $x \leq y$ or $y \leq x$ holds. A partially ordered set X has *finite height*, if all its chains are finite sets and there is an upper bound on their cardinality. This upper bound is called the *height* of X and denoted $\mathbf{h}(X)$. The following results clearly hold about $\mathbf{h}$.

Lemma 2.1. *Let X, Y be partially ordered sets with finite height. Let Z be a finite set. Then*

- $\mathbf{h}(X_\perp) = \mathbf{h}(X) + 1$;
- $\mathbf{h}(X \times Y) = \mathbf{h}(X) + \mathbf{h}(Y) - 1$;
- $\mathbf{h}(\mathbf{Parts}(Z)) = |Z|$, *if Z is not empty;*
- $\mathbf{h}(\mathbf{Parts}(\emptyset)) = 1$;
- $\mathbf{h}(\mathcal{P}(Z)) = |Z| + 1$.

If X is a partial order and $X' \subseteq X$ is a finite chain, then $\bigvee X'$ exists — it is equal to the greatest element of X'. If X' is an infinite chain, then it does not necessarily have the greatest element and $\bigvee X'$ does not have to exist. A partially ordered set is a *complete partial order* (*CPO*), if all its chains have least upper bounds. A lattice is a *complete lattice*, if all its subsets have least upper bounds (and then all its subsets also have greatest lower bounds [DP90, Thm. 2.16]). Note that a complete partial order X has the smallest element. Indeed, $\emptyset \subseteq X$ is a chain and each element of X is its upper bound. Therefore $\bigvee \emptyset$ is the smallest element of X.

Clearly, each partial order with finite height and each lattice with finite height are complete. If X is a complete partial order and $Y \subseteq X$ is upper closed, then Y is a complete partial order, too. If X is a complete lattice, $Y \subseteq X$ is upper closed and Y has the smallest element, then Y is a complete lattice, too.

Lemma 2.2. *If X and Y are complete partial orders, then $X \times Y$ is a complete partial order. For a chain $Z \subseteq X \times Y$,*

$$\bigvee Z = \left(\bigvee\{x \,:\, (x,y) \in Z\}, \bigvee\{y \,:\, (x,y) \in Z\}\right) .$$

Similarly, if Y is a complete partial order then $X \to Y$ is also a complete partial order. For a chain $Z \subseteq X \to Y$,

$$(\bigvee Z)(x) = \bigvee\{z(x) \,:\, z \in Z\}$$

for all $x \in X$.

Also, the Cartesian product of complete lattices is a complete lattice and the set of functions from a set to a complete lattice is a complete lattice.

A function $f : X \to Y$ between partially ordered sets X and Y is *monotone*, if $x_1 \leq x_2 \implies f(x_1) \leq f(x_2)$ for all $x_1, x_2 \in X$. A monotone function $f : X \to Y$ between complete partial orders X and Y is *continuous*, if $f(\bigvee X') = \bigvee\{f(x) : x \in X'\}$ for all chains $X' \subseteq X$. The composition of monotone [resp. continuous] functions is again a monotone [resp. continuous] function.

A *fixed point* of a function $f : X \to X$ is an element $x \in X$, such that $f(x) = x$. Let X be a partially ordered set and let $y \in X$. If the following quantities exist, then we denote

- the least fixed point of f by *lfp* f;
- the least fixed point of f that is greater or equal to y, by $\mathit{lfp}^y\, f$;
- the greatest fixed point of f by *gfp* f;
- the greatest fixed point of f that is less or equal to y, by $\mathit{gfp}^y\, f$.

We now present some sufficient conditions for the least fixed points to exist.

Proposition 2.3 (Kleene's fixed point theorem). *Let X be a complete partial order and let $f : X \to X$ be a continuous function. Then* lfp f *exists and*

$$\mathit{lfp}\; f = \bigvee\{f^n(\bot) \,:\, n \in \mathbb{N}\},$$

where $\bot$ denotes the smallest element of X.

This proposition also appears as Theorem 4.37 in [NN92]; its proof is given there, too.

Proposition 2.4. *Let X be a complete partial order, let $y \in X$ and let $f : X \to X$ be a continuous function, such that $y \leq f(y)$. Then $\mathit{lfp}^y\ f$ exists and*

$$\mathit{lfp}^y\ f = \bigvee \{f^n(y) : n \in \mathbb{N}\} \ . \tag{2.1}$$

Proof. Let $Y = \{x : x \in X, x \geq y\}$. Then Y is a complete partial order. If $x \in Y$, then $f(x) \geq f(y) \geq y$, therefore $f(x) \in Y$ and we may consider f as a (continuous) function from Y to Y. Applying proposition 2.3 to Y and f gives that $\mathit{lfp}\ f$ in Y, which equals $\mathit{lfp}^y\ f$ in X, exists and the equation (2.1) holds. ☐

Proposition 2.5 (Tarski's fixed point theorem). *Let X be a complete lattice and let $f : X \to X$ be a monotone function. Then lfp f exists. If X has finite height, then*

$$\mathit{lfp}\ f = \bigvee \{f^n(\bot) : n \in \mathbb{N}\},$$

where $\bot$ denotes the smallest element of X.

This proposition also appears as Proposition A.10 in [NNH99]; its proof is given there, too.

Proposition 2.6. *Let X be a complete lattice, let $y \in X$ and let $f : X \to X$ be a monotone function, such that $y \leq f(y)$. Then $\mathit{lfp}^y\ f$ exists. If X has finite height, then*

$$\mathit{lfp}^y\ f = \bigvee \{f^n(y) : n \in \mathbb{N}\} \ .$$

The proof of proposition 2.6 is identical to the proof of proposition 2.4.

There exists a dual result for each of the Propositions 2.3–2.6 that states a sufficient condition for the existence (and value) of *gfp* f or $\mathit{gfp}^y\ f$. We obtain them by replacing

- *lfp* by *gfp* ;
- $\leq$ by $\geq$;
- $\bigvee$ by $\bigwedge$ (also in the definition of CPO);
- $\bot$ by $\top$, where $\top$ denotes the greatest element of X

in the wordings of those propositions. For example, the dual of Prop. 2.6 is

Proposition 2.7. *Let X be a complete lattice, let $y \in X$ and let $f : X \to X$ be a monotone function, such that $y \geq f(y)$. Then $\mathit{gfp}^y\ f$ exists. If X has finite height, then*

$$\mathit{gfp}^y\ f = \bigwedge \{f^n(y) : n \in \mathbb{N}\} \ .$$

2.1.2 Probability Distributions

A *probability distribution* over a set X is a function $D : X \to [0..1]$, where $\sum_{x\in X} D(x) = 1$ (we only consider probability distributions over finite or countable sets). If $x \in X$ and $D(x) = z$ then we also say that the probability distribution D *assigns the weight* z to x. Let $\mathcal{D}(X)$ denote the set of all probability distributions over X. For a probability distribution $D \in \mathcal{D}(X)$, the expression $x \leftarrow D$ denotes that the random variable x is distributed according to D. The expression $x, x' \leftarrow D$ is equivalent to $x \leftarrow D$ and $x' \leftarrow D$, i.e. x and x' are two independent random variables distributed according to D. Multiset comprehensions, for example $\{\!| f(x) : x \leftarrow D |\!\}$, are used to define new probability distributions from existing ones. For $D \in \mathcal{D}(X)$ and $f : X \to Y$ the given example is a distribution $D' \in \mathcal{D}(Y)$ given by

$$\forall y \in Y : D'(y) = \sum_{x\in f^{-1}(y)} D(x) \ .$$

A partial order on X defines a partial order on $\mathcal{D}(X)$. $D_1 \leq D_2$, where $D_1, D_2 \in \mathcal{D}(X)$, iff

$$\forall X' \in \mathcal{P}_{\mathrm{U}}(X) : \sum_{x\in X'} D_1(x) \leq \sum_{x\in X'} D_2(x) \ .$$

As a special case, if $X = Y_\perp$ and Y is unordered, then $D_1 \leq D_2$ iff $D_1(y) \leq D_2(y)$ for all $y \in Y$.

In this case, the set of probability distributions is even a CPO.

Lemma 2.8. *If the set Y is unordered, then $\mathcal{D}(Y_\perp)$ is a CPO.*

Proof. Let $D_i \in \mathcal{D}(Y_\perp)$, where $i \in I$ and I is a suitable index set. Let the set $\{D_i : i \in I\}$ be a chain. Define $D \in \mathcal{D}(Y_\perp)$ by

$$\begin{aligned} D(y) &= \sup\{D_i(y) : i \in I\}, \text{ for each } y \in Y \\ D(\perp) &= 1 - \sum_{y\in Y} D(y) \ . \end{aligned}$$

Some basic calculus shows that D is indeed a probability distribution — the sum $\sum_{y\in Y} D(y)$ is at most 1. It is also obvious that D is an upper bound of $\{D_i : i \in I\}$. Assume that D' is also its upper bound and $D \not\leq D'$. Then there exists $y \in Y$, such that $D'(y) < D(y)$. But then there also exists an $i \in I$, such that $D'(y) < D_i(y)$ and hence D' is not an upper bound of $\{D_i : i \in I\}$. We have shown that D is the least upper bound of $\{D_i : i \in I\}$. □

Let $\eta^{\mathcal{D}}(x) \in \mathcal{D}(X)$, where $x \in X$, denote the distribution where x occurs with the probability 1.

Let $D_1, D_2 \in \mathcal{D}(X_\perp)$, such that $D_1(\perp) + D_2(\perp) \geq 1$. Then we can define the *sum* of D_1 and D_2, denoted by $D_1 + D_2$. It is again a probability distribution over $X_\perp$. It is defined as follows:

$$(D_1 + D_2)(x) = \begin{cases} D_1(x) + D_2(x), & \text{if } x \in X \\ D_1(\perp) + D_2(\perp) - 1, & \text{if } x = \perp \ . \end{cases}$$

Note that the sum of probability distributions is associative and commutative. In general, if $D_1, \ldots, D_k \in \mathcal{D}(X_\perp)$, then their sum is defined iff $D_1(\perp) + \cdots + D_k(\perp) \geq k-1$.

A product $D \in \prod_{n\in\mathbb{N}} \mathcal{D}(X_n)$, where X_n are some sets, should be called "a family of probability distributions (over X)". However, we mostly call D just "a (probability) distribution (over X)". When we use multiset comprehensions to define new families of probability distributions, then we make explicit that the parameter n varies. Hence if $f : \prod_{n\in\mathbb{N}}(X_n \to Y_n)$ is a countable tuple of functions, then

$$\{\!|\, f_n(x_n) \,:\, x_n \leftarrow D_n \,|\!\}_{n\in\mathbb{N}}$$

is a family of probability distributions over Y, and

$$\{\!|\, f_n(x_n) \,:\, x_n \leftarrow D_n \,|\!\}$$

is a probability distribution over Y_n for some fixed $n \in \mathbb{N}$.

We introduce some shorter notation for sets of families of distributions and functions. Let $X = \{X_n\}_{n\in\mathbb{N}}$ and $Y = \{Y_n\}_{n\in\mathbb{N}}$ be families of sets. Denote

- $\prod_{n\in\mathbb{N}} \mathcal{D}(X_n)$ by $\mathcal{D}^{\mathbb{N}}(X_n)$;
- $\prod_{n\in\mathbb{N}}(X_n \to Y_n)$ by $X \stackrel{\mathbb{N}}{\to} Y$;
- $\prod_{n\in\mathbb{N}}(X_n \to \mathcal{D}(Y_n))$ by $X \stackrel{\mathbb{N}}{\rightsquigarrow} Y$.

For sets Z and W we denote the set of functions $Z \to \mathcal{D}(W)$ by $Z \rightsquigarrow W$. We call this set the set of *probabilistic functions* from Z to W.

For a function $f : X \to Y$ and $x \in X$ we denote $f(x)$ also by $f \,\$\, x$. If f and x are both complex expressions, then $f \,\$\, x$ is more readable than $f(x)$ — we have to use less parentheses. We also assume that the composition of functions $\circ$ binds stronger than \$. Thus $f \circ g \,\$\, x$ denotes $(f \circ g) \,\$\, x = f(g(x))$.

The symbol \$ is also useful for overloading. Given $f : X \to Y$ and $X' \subseteq X$, we let $f \,\$\, X'$ denote the sequence of the values $f(x)$ for all $x \in X'$. We assume here that an order of appearing in the sequence is defined for the elements of X. It is not important, how the order is defined, we only assume that this order is the same for all sequences that we may construct from the elements of X. We denote the sequence of the elements of X in this order by $\langle x \rangle_{x\in X}$.

2.2 Cryptography

The treatment of cryptography we are presenting here is based on asymptotics (this is the usual case in complexity-theoretic arguments). The specification of systems has a parameter $n \in \mathbb{N}$, called *security parameter*, that characterises the security of the system. Typically, n is the length of the keys in bits. The probability that the system reaches a certain state, or that some attacker is successful against the system, is then a function of the security parameter.

2.2.1 Basics

The next definition gives a particular meaning to "never". If the probability that something happens is negligible in the security parameter, then this is (by definition(s)) as good as that something never happens.

Definition 2.1. A function $f : \mathbb{N} \to \mathbb{R}$ is *negligible*, if

$$\forall p \in \mathbf{Pol}(\mathbb{Z})\ \exists n \in \mathbb{N}\ \forall n' > n : |f(n')| < \left|\frac{1}{p(n')}\right| \ .$$

It is obvious that the sum of negligible functions is again negligible.

Let $\mathcal{A}$ be an algorithm. All algorithms that we consider may be probabilistic. We say that $\mathcal{A}$ is a *probabilistic polynomial-time* (PPT) algorithm, if there exists a polynomial $p \in \mathbf{Pol}(\mathbb{Z})$, such that for each input $x \in \{\mathbf{0}, \mathbf{1}\}^*$ the probability that $\mathcal{A}(\mathbf{1}^n, x)$ makes more than $p(n)$ steps is negligible (the probability is a function of n). Here $\mathbf{1}^n$ denotes a bit-string of n bits $\mathbf{1}$. The usual way of defining the complexity classes of algorithms is to demand that the running time of the algorithm must be bound by some function (from a certain class of functions) of the *length* of its argument(s). Here we have defined that the running time of the algorithm must be polynomial in the length of its first argument. This explains the traditional way of representing the security parameter in unary, when giving it to an algorithm. In the unary representation, the length of n is equal to n, while for some other representations (for example, binary), the length of n might be quite different from n (for example, logarithmic).

Arguments to algorithms have to be bit-strings. In the following, when we say that some quantity x is an argument to an algorithm, then we mean that a suitable encoding of x is given to the algorithm.

Thus we may speak about tuples or sets of bit-strings as arguments of algorithms, or about functions with a finite domain and the set of bit-strings as range.

An algorithm may have access to an oracle implementing a probabilistic function $\omega : \{\mathbf{0}, \mathbf{1}\}^* \rightsquigarrow \{\mathbf{0}, \mathbf{1}\}^*$. The algorithm has an extra operation available, this operation allows the algorithm to submit a bit-string x to the oracle. Whenever the algorithm invokes this operation, a bit-string y is picked according to the distribution $\omega(x)$ and returned to the algorithm. We denote an algorithm $\mathcal{A}$ that assumes to have access to an oracle by $\mathcal{A}^{(\cdot)}$. If we want to stress that the oracle implements the function ω, then we denote the algorithm by $\mathcal{A}^{\omega(\cdot)}$. We may also say "algorithm $\mathcal{A}$ accesses ω through oracle interface".

The algorithm may also have accesses to more than one oracle. In this case, for each oracle there is an operation available for the algorithm, allowing the algorithm to submit a bit-string to this oracle.

The functions of type $\{\mathbf{0}, \mathbf{1}\}^* \rightsquigarrow \{\mathbf{0}, \mathbf{1}\}^*$ may thus also be considered to be suitable inputs to algorithms. Sometimes, when we have a set of arguments X that we want to give to an algorithm $\mathcal{A}$, we do not distinguish the arguments that are encoded as bit-strings from arguments that the algorithm can access through oracle interface. Then we denote the application of $\mathcal{A}$ on X by $\mathcal{A}(\langle x\rangle_{x\in X})$.

We also give names to the properties of distributions and functions of being "realisable" in polynomial time.

Definition 2.2. Let $X = \{X_n\}_{n\in\mathbb{N}}$ be a family of sets whose elements can be outputs of algorithms. Let $D \in \mathcal{D}^{\mathbb{N}}(X)$. We say that the distribution D is *polynomial-time constructible*, if there exists a PPT algorithm $\mathcal{A}$, such that the distribution of the outputs of $\mathcal{A}(\mathbf{1}^n)$ is equal to the distribution D_n for all $n \in \mathbb{N}$.

Here we also say that the algorithm $\mathcal{A}$ *samples* the distribution D.

Definition 2.3. Let $X = \{X_n\}_{n\in\mathbb{N}}$ and $Y = \{Y_n\}_{n\in\mathbb{N}}$ be families of sets whose elements can be inputs and outputs of algorithms. Let $f : X \overset{\mathbb{N}}{\rightsquigarrow} Y$. We say that f is *polynomial-time computable*, if there exists a PPT algorithm $\mathcal{A}$, such that the distribution $f_n(x_n)$ is equal to the distribution of the output of $\mathcal{A}(\mathbf{1}^n, x_n)$ for each $x_n \in X_n$.

2.2.2 Indistinguishability

Indistinguishability of probability distributions is a fundamental concept in cryptography. The security definitions of encryption systems and programs that we are going to present, are just statements that certain distributions are indistinguishable.

Definition 2.4. Let $X = \{X_n\}$ be a family of sets whose elements can be inputs to algorithms. Two families of probability distributions $D, D' \in \mathcal{D}^{\mathbb{N}}(X)$ are *(computationally) indistinguishable* (denoted $D \approx D'$) if for all PPT algorithms $\mathcal{A}$ the difference

$$\mathbf{Adv}_{\mathcal{A}}^{D,D'}(n) := \Pr\bigl[\mathcal{A}(\mathbf{1}^n, x) = 1 \,:\, x \leftarrow D_n\bigr] - \Pr\bigl[\mathcal{A}(\mathbf{1}^n, x) = 1 \,:\, x \leftarrow D'_n\bigr]$$

is negligible in n. $\mathbf{Adv}_{\mathcal{A}}^{D,D'}$ is called the *advantage* of $\mathcal{A}$ on D, D'.

The intuition behind this definition is the following (see also Fig. 2.1): the algorithm $\mathcal{A}$ may find itself in one of two possible worlds. In the first world the variable x is distributed according to D, in the second world according to D'. The algorithm $\mathcal{A}$ outputs 1 if it guesses that it is in the first world, it outputs a different value (for example, 2) if it guesses that it is in the second world. The definition now states that those two worlds are so similar that the guess made by $\mathcal{A}$ does not significantly depend on the world it is placed into.

This is a very universal definition, as X can be any family of sets whose elements can be inputs to algorithms. For example, it also covers the indistinguishability of probabilistic functions. According to definition 2.4, two families of probability distributions D, D' over sets of probabilistic functions are indistinguishable if for all PPT algorithms $\mathcal{A}^{(\cdot)}$ the difference

$$\Pr\bigl[\mathcal{A}^{f(\cdot)}(\mathbf{1}^n) = 1 \,:\, f \leftarrow D_n\bigr] - \Pr\bigl[\mathcal{A}^{f(\cdot)}(\mathbf{1}^n) = 1 \,:\, f \leftarrow D'_n\bigr]$$

is negligible in n.

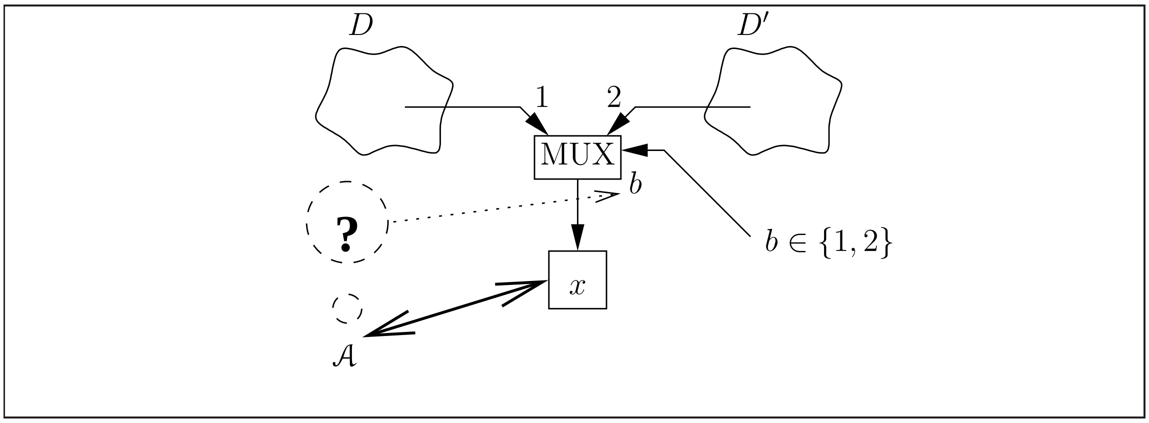


Figure 2.1: Intuition behind indistinguishability

If the advantage of some algorithm $\mathcal{A}$ on some distributions D, D' is not negligible, then we say that $\mathcal{A}$ *distinguishes* the distributions D and D'.

We now state some simple properties of indistinguishability.

Lemma 2.9. *Indistinguishability of distributions is a transitive relation.*

Proof. Let D, D', D'' be three distributions over $X = \{X_n\}_{n\in\mathbb{N}}$. Assume that D and D' are indistinguishable, and D' and D'' are indistinguishable. Let $\mathcal{A}$ be any PPT algorithm. Then

$$\begin{aligned}
&\mathbf{Adv}_{\mathcal{A}}^{D,D''}(n) = \\
&\qquad \Pr\big[\mathcal{A}(\mathbf{1}^n, x) = 1 \,:\, x \leftarrow D_n\big] - \Pr\big[\mathcal{A}(\mathbf{1}^n, x) = 1 \,:\, x \leftarrow D''_n\big] = \\
&\Pr\big[\mathcal{A}(\mathbf{1}^n, x) = 1 \,:\, x \leftarrow D_n\big] - \Pr\big[\mathcal{A}(\mathbf{1}^n, x) = 1 \,:\, x \leftarrow D'_n\big] + \\
&\qquad \Pr\big[\mathcal{A}(\mathbf{1}^n, x) = 1 \,:\, x \leftarrow D'_n\big] - \Pr\big[\mathcal{A}(\mathbf{1}^n, x) = 1 \,:\, x \leftarrow D''_n\big] = \\
&\qquad\qquad \mathbf{Adv}_{\mathcal{A}}^{D,D'}(n) + \mathbf{Adv}_{\mathcal{A}}^{D',D''}(n),
\end{aligned}$$

hence $\mathbf{Adv}_{\mathcal{A}}^{D,D''}$ is negligible. □

A very simple but powerful property of indistinguishability is the following:

Lemma 2.10. *Let $D, D' \in \mathcal{D}^{\mathbb{N}}(X)$ be indistinguishable. Let $f : X \overset{\mathbb{N}}{\rightsquigarrow} Y$ be polynomial-time computable. Then the distributions*

$$\{\!|f(x) \,:\, x \leftarrow D|\!\}_{n\in\mathbb{N}} \tag{2.2}$$

and

$$\{\!|f(x) \,:\, x \leftarrow D'|\!\}_{n\in\mathbb{N}} \tag{2.3}$$

are indistinguishable.

Proof. Suppose that the distributions (2.2) and (2.3) are distinguishable. Let $\mathcal{A}$ be a PPT algorithm that distinguishes them. Let $\mathcal{B}$ be a PPT algorithm that implements f. Then the composition of $\mathcal{A}$ and $\mathcal{B}$ is a PPT algorithm that distinguishes D and D'. □

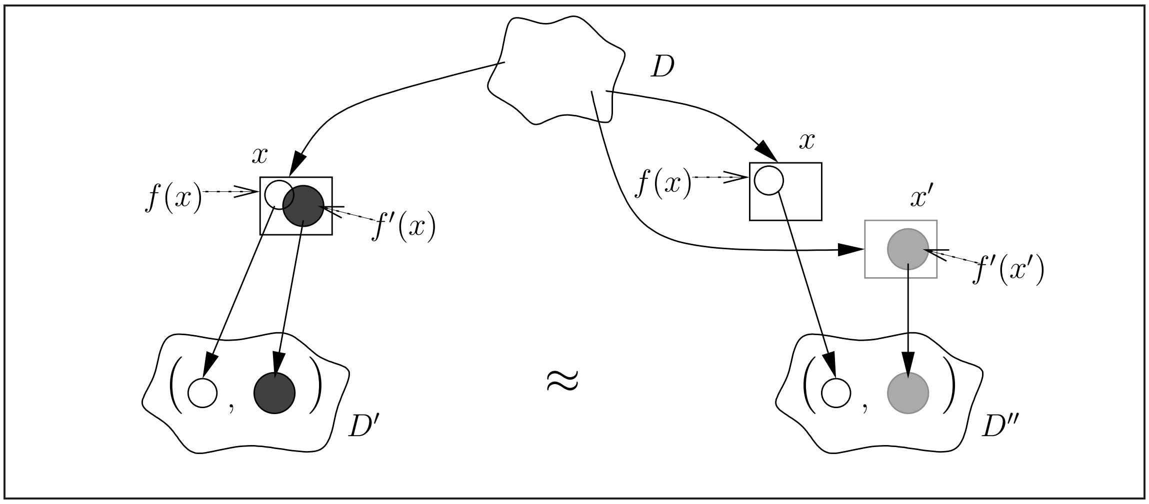


Figure 2.2: Intuition behind independence

If D is a distribution over $X = \{X_n\}_{n\in\mathbb{N}}$ and the elements of X_n have complex structure (for example, they are tuples of several components), then we may be interested, how much do the different components of the elements of X_n depend on each other. A particular example here is, when $X_n = \mathbf{State}_n$ is the set of *program states*. A program state is a function that maps each variable (the set of variables is a finite set) to its value. If Y and Z are sets of variables then we are interested whether the values of variables in Y and the values of variables in Z are independent or not.

Definition 2.5. Let $X = \{X_n\}_{n\in\mathbb{N}}$ and $Y = \{Y_n\}_{n\in\mathbb{N}}$ be families of sets and let $D \in \mathcal{D}^{\mathbb{N}}(X)$. Let $f, f' : X \stackrel{\mathbb{N}}{\rightsquigarrow} Y$. Then f and f' are *independent* [Yao82, Def. 18, Thm. 6] in the distribution D, if

$$\{\!|(y_n, y'_n) \,:\, x_n \leftarrow D_n, y_n \leftarrow f_n(x_n), y'_n \leftarrow f'_n(x_n)|\!\}_{n\in\mathbb{N}} \approx \{\!|(y_n, y'_n) \,:\, x_n, x'_n \leftarrow D_n, y_n \leftarrow f_n(x_n), y'_n \leftarrow f'_n(x'_n)|\!\}_{n\in\mathbb{N}} \ .$$

Fig. 2.2 illustrates this definition. f and f' are independent in the distribution D iff distributions D' and D'' are indistinguishable.

We mainly use this definition in a setup where the sets X_n and the families of functions f, f' have a special shape. Namely, let Y be a set, let $Z = \{Z_n\}_{n\in\mathbb{N}}$ be a family of sets and let $X_n = Y \to Z_n$. Let $D \in \mathcal{D}^{\mathbb{N}}(X)$ and let $Y', Y'' \subseteq Y$. We say that the sets Y and Y' are *independent* in the distribution D, if

$$\{\!|(g_n \,\$\, Y', g_n \,\$\, Y'') \,:\, g_n \leftarrow D_n|\!\}_{n\in\mathbb{N}} \approx \{\!|(g_n \,\$\, Y', g'_n \,\$\, Y'') \,:\, g_n, g'_n \leftarrow D_n|\!\}_{n\in\mathbb{N}} \ . \quad (2.4)$$

The previous definition of indistinguishability required that two indistinguishable distributions cannot be told apart, based on a single sample. This definition could be strengthened by allowing multiple samples.

Definition 2.6. Two families of probability distributions D, D' over $X = \{X_n\}_{n\in\mathbb{N}}$ are *indistinguishable by multiple samples*, if for all polynomials $m \in \mathbf{Pol}(\mathbb{Z})$ and PPT algorithms $\mathcal{A}$ the difference

$$\Pr\bigl[\mathcal{A}(\mathbf{1}^n, x_1, \ldots, x_{m(n)}) = 1 \,:\, x_1, \ldots, x_{m(n)} \leftarrow D_n\bigr] - \\ \Pr\bigl[\mathcal{A}(\mathbf{1}^n, x_1, \ldots, x_{m(n)}) = 1 \,:\, x_1, \ldots, x_{m(n)} \leftarrow D'_n\bigr]$$

is negligible in n.

However, at least for polynomial-time constructible distributions, these two notions are equivalent. Goldreich [Gol95, Sec. 3.2] shows that the following proposition holds.

Proposition 2.11. *Let D, D' be two polynomial-time constructible distributions. Then D and D' are indistinguishable iff they are indistinguishable by multiple samples.*

We also give the proof of this proposition here, because the used technique has an important application later.

Proof. In one direction, the claim of the proposition is trivial — if D and D' are indistinguishable by multiple samples, then by setting the number of samples to 1 we get that D and D' are indistinguishable.

Consider the other direction. Suppose that the probability distributions D and D' over $X = \{X_n\}_{n\in\mathbb{N}}$ are not indistinguishable by multiple samples. Let $\mathcal{A}$ be the algorithm that distinguishes them using $m \in \mathbf{Pol}(\mathbb{Z})$ samples. Then $\mathbf{Adv}^{\tilde{D},\tilde{D}'}_{\mathcal{A}}$ is not negligible, where

$$\tilde{D} = \{\!|(x_1, \ldots, x_{m(n)}) \,:\, x_1, \ldots, x_{m(n)} \leftarrow D_n|\!\}_{n\in\mathbb{N}}$$
$$\tilde{D}' = \{\!|(x_1, \ldots, x_{m(n)}) \,:\, x_1, \ldots, x_{m(n)} \leftarrow D'_n|\!\}_{n\in\mathbb{N}} \ .$$

For each $n \in \mathbb{N}$ and each $i \in \{0, 1, \ldots, m(n)\}$ define the following distributions $D_n^{(i)} \in \mathcal{D}(X_n)$:

$$D_n^{(i)} = \{\!|(x_1, \ldots, x_{m(n)}) \,:\, x_1, \ldots, x_i \leftarrow D'_n, x_{i+1}, \ldots, x_{m(n)} \leftarrow D_n|\!\} \ .$$

Then $D_n^{(0)} = \tilde{D}_n$ and $D_n^{(m(n))} = \tilde{D}'_n$. Consider now the algorithm $\mathcal{B}$ given in Fig. 2.3. The algorithm $\mathcal{B}$ runs in polynomial time. The following equalities between distributions hold for all $n \in \mathbb{N}$ and $i \in \{1, \ldots, m(n)\}$:

$$D_n^{(i-1)} = \{\!|\mathcal{B}(\mathbf{1}^n, i, x) \,:\, x \leftarrow D_n|\!\}$$
$$D_n^{(i)} = \{\!|\mathcal{B}(\mathbf{1}^n, i, x) \,:\, x \leftarrow D'_n|\!\} \ .$$

We can now define the algorithm $\bar{\mathcal{A}}$ that can distinguish the families of probability distributions D and D'. On input $(\mathbf{1}^n, x)$ the algorithm $\bar{\mathcal{A}}$ uniformly chooses i from the set $\{1, \ldots, m(n)\}$, then calls the algorithm $\mathcal{B}$ on $(\mathbf{1}^n, i, x)$ and finally

```
The algorithm B(1^n, i, x), where i ∈ {1, ..., m(n)} and x ∈ X_n:
  for j = 1 to i − 1 do
    generate x_j ← D'_n .................../* D' is polynomial-time constructible */
  end for
  x_i := x
  for j = i + 1 to m(n) do
    generate x_j ← D_n ................../* D is polynomial-time constructible */
  end for
  return (x_1, ..., x_m(n))
```

Figure 2.3: Making a step between distributions D and D'

invokes the algorithm $\mathcal{A}$ with the output of $\mathcal{B}$. If x is distributed according to D_n, then the probability that $\bar{\mathcal{A}}(\mathbf{1}^n, x)$ returns 1 is

$$\Pr\left[\bar{\mathcal{A}}(\mathbf{1}^n, x) = 1 \,:\, x \leftarrow D_n\right] = \frac{1}{m(n)} \sum_{i=1}^{m(n)} \Pr\left[\mathcal{A}(\mathbf{1}^n, \mathbf{x}) = 1 \,:\, \mathbf{x} \leftarrow D_n^{(i-1)}\right] \ .$$

If x is distributed according to D'_n, then the same probability is

$$\Pr\left[\bar{\mathcal{A}}(\mathbf{1}^n, x) = 1 \,:\, x \leftarrow D'_n\right] = \frac{1}{m(n)} \sum_{i=1}^{m(n)} \Pr\left[\mathcal{A}(\mathbf{1}^n, \mathbf{x}) = 1 \,:\, \mathbf{x} \leftarrow D_n^{(i)}\right] \ .$$

Their difference is

$$\begin{aligned}
&\mathbf{Adv}_{\bar{\mathcal{A}}}^{D,D'}(n) = \\
&\qquad \Pr\left[\bar{\mathcal{A}}(\mathbf{1}^n, x) = 1 : x \leftarrow D_n\right] - \Pr\left[\bar{\mathcal{A}}(\mathbf{1}^n, x) = 1 : x \leftarrow D'_n\right] = \\
&\frac{1}{m(n)} \sum_{i=1}^{m(n)} \Big(\Pr\left[\mathcal{A}(\mathbf{1}^n, \mathbf{x}) = 1 : \mathbf{x} \leftarrow D_n^{(i-1)}\right] - \Pr\left[\mathcal{A}(\mathbf{1}^n, \mathbf{x}) = 1 : \mathbf{x} \leftarrow D_n^{(i)}\right]\Big) = \\
&\quad \frac{1}{m(n)} \Big(\Pr\left[\mathcal{A}(\mathbf{1}^n, \mathbf{x}) = 1 : \mathbf{x} \leftarrow \tilde{D}_n\right] - \Pr\left[\mathcal{A}(\mathbf{1}^n, \mathbf{x}) = 1 : \mathbf{x} \leftarrow \tilde{D}'_n\right]\Big) = \\
&\qquad\qquad\qquad\qquad \frac{\mathbf{Adv}_{\mathcal{A}}^{\tilde{D},\tilde{D}'}(n)}{m(n)} \ .
\end{aligned}$$

Therefore $\mathbf{Adv}_{\bar{\mathcal{A}}}^{D,D'}$ is not negligible and the distributions D and D' are not indistinguishable. □

The underlying idea of the proof, called the *hybrid argument*, was to transform the distribution $\tilde{D}_n$ into the distribution $\tilde{D}'_n$ in *a small number of short steps*. In our case

- The hybrids were the distributions $D_n^{(i)}$. The extreme hybrids had to collide with $\tilde{D}_n$ and $\tilde{D}'_n$.

- A step related two neighbouring hybrids. The steps were done by the algorithm $\mathcal{B}$. A step was "short", the "difference" of two neighbouring hybrids is only as big as the "difference" of D_n and D'_n.

- The number of steps was $m(n)$ which is polynomial in n. Here "polynomial" means "small".

2.2.3 Encryption

Let $\mathsf{Plaintext} \subseteq \{\mathbf{0},\mathbf{1}\}^*$ be the set of permitted plaintexts to be encrypted. I.e. an encryption system must be able to encrypt all bit-strings in $\mathsf{Plaintext}$, it may fail for other bit-strings.

Definition 2.7. An *encryption system* is a triple of PPT algorithms $(\mathcal{G}, \mathcal{E}, \mathcal{D})$, where

- on input $\mathbf{1}^n$, the algorithm $\mathcal{G}$ returns a bit-string of length $\ell(n)$, where $\ell \in \mathbf{Pol}(\mathbb{Z})$ is a fixed polynomial. The type of algorithm $\mathcal{G}$ is therefore

$$\mathcal{G} : \prod_{n\in\mathbb{N}} \mathcal{D}(\{\mathbf{0},\mathbf{1}\}^{\ell(n)}) \ .$$

  The algorithm $\mathcal{G}$ is called the *key generation algorithm* and the bit-strings are called the keys.

- on input $\mathbf{1}^n$, $k \in \{\mathbf{0},\mathbf{1}\}^{\ell(n)}$ and $x \in \mathsf{Plaintext}$, the algorithm $\mathcal{E}$ returns a bit-string y. The length of y must depend only on n and $|x|$. The type of $\mathcal{E}$ is therefore

$$\mathcal{E} : \prod_{n\in\mathbb{N}} (\{\mathbf{0},\mathbf{1}\}^{\ell(n)} \times \{\mathbf{0},\mathbf{1}\}^* \rightsquigarrow \{\mathbf{0},\mathbf{1}\}^*) \ .$$

  The algorithm $\mathcal{E}$ is called the *encryption algorithm*. Its output y is called the *ciphertext* for the plaintext x with the key k.

- on input $\mathbf{1}^n$, $k \in \{\mathbf{0},\mathbf{1}\}^{\ell(n)}$ and $y \in \{\mathbf{0},\mathbf{1}\}^*$, the algorithm $\mathcal{D}$ deterministically returns a bit-string x. The type of $\mathcal{D}$ is therefore

$$\mathcal{D} : \prod_{n\in\mathbb{N}} (\{\mathbf{0},\mathbf{1}\}^{\ell(n)} \times \{\mathbf{0},\mathbf{1}\}^* \to \{\mathbf{0},\mathbf{1}\}^*) \ .$$

  The algorithm $\mathcal{D}$ is called the *decryption algorithm*.

The triple $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ must be such, that for each $x \in \mathsf{Plaintext}$ and each $k \in \{\mathbf{0},\mathbf{1}\}^{\ell(n)}$, where k has nonzero probability of being returned by $\mathcal{G}(\mathbf{1}^n)$, the decryption of encryption of x (both with the key k) must be equal to x, i.e. the equality $\mathcal{D}(\mathbf{1}^n, k, \mathcal{E}(\mathbf{1}^n, k, x)) = x$ must hold.

We say that the encryption system is *deterministic*, if the encryption algorithm $\mathcal{E}$ is deterministic. We say that a deterministic encryption system is *length-preserving*, if for all $n \in \mathbb{N}$, $k \in \{\mathbf{0},\mathbf{1}\}^{\ell(n)}$, where $k \in \mathrm{carrier}(\mathcal{G}(\mathbf{1}^n))$, and $x \in \{\mathbf{0},\mathbf{1}\}^*$, the equality $|\mathcal{E}(\mathbf{1}^n, k, x)| = |x|$ holds.

We now give definitions for the secureness of an encryption system. There are several possible definition of security and the "'right"' one depends on the intended usage of the encryption system inside a bigger system. In this thesis we consider two different kinds of secure encryption systems — *pseudorandom permutations* [LR85] and *which-key and repetition concealing* encryption systems (which-key concealing encryption is the topic of [BBDP01], repetition-concealing encryption dates back as far as [GM84]).

For a set X let $\mathcal{S}(X) \subset (X \to X)$ be the set of all permutations of X. Let $\mathcal{U}(X) \in \mathcal{D}(X)$ be the *uniform* probability distribution over X — this distribution assigns to each element of X the weight $\frac{1}{|X|}$.

Definition 2.8. A length-preserving encryption system $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is a *pseudorandom permutation (PRP)*, if for each $p \in \mathbf{Pol}(\mathbb{Z})$, where $\{\mathbf{0},\mathbf{1}\}^{p(n)} \subseteq \mathsf{Plaintext}$ for each $n \in \mathbb{N}$, the following holds:

$$\{\!| \mathcal{E}(\mathbf{1}^n, k, \cdot) \,:\, k \leftarrow \mathcal{G}(\mathbf{1}^n) |\!\}_{n\in\mathbb{N}} \approx \{\!| \pi(\cdot) \,:\, \pi \leftarrow \mathcal{U}(\mathcal{S}(\{\mathbf{0},\mathbf{1}\}^{p(n)})) |\!\}_{n\in\mathbb{N}} \ .$$

It is a *pseudorandom function (PRF)*, if the following holds:

$$\{\!| \mathcal{E}(\mathbf{1}^n, k, \cdot) \,:\, k \leftarrow \mathcal{G}(\mathbf{1}^n) |\!\}_{n\in\mathbb{N}} \approx \{\!| \varphi(\cdot) \,:\, \varphi \leftarrow \mathcal{U}(\{\mathbf{0},\mathbf{1}\}^{p(n)} \to \{\mathbf{0},\mathbf{1}\}^{p(n)}) |\!\}_{n\in\mathbb{N}} \ .$$

In the left-hand side, $\mathcal{E}(\mathbf{1}^n, k, \cdot)$ is assumed to be restricted to bit-strings of length $p(n)$.

For better understanding of this definition it may be helpful to note that no algorithm $\mathcal{A}$ (even one that does not have to run in polynomial time) can distinguish a random permutation from the stateful oracle $\mathcal{PRP}$ given in Fig. 2.4. The following equality holds for all $\mathcal{A}$ and $n \in \mathbb{N}$:

$$\{\!| \mathcal{A}^{\pi(\cdot)}(\mathbf{1}^n) \,:\, \pi \leftarrow \mathcal{U}(\mathcal{S}(\{\mathbf{0},\mathbf{1}\}^{p(n)})) |\!\} = \{\!| \mathcal{A}^{\mathcal{PRP}(\mathbf{1}^n,\cdot)}(\mathbf{1}^n) |\!\} \ . \tag{2.5}$$

The oracle $\mathcal{PRP}$ "lazily" constructs a permutation of $\{\mathbf{0},\mathbf{1}\}^{p(n)}$. Its internal state contains all pairs (x_i, y_i), where the bit-strings x_i are queries that are already made to it, and y_i-s are the answers given by it to these queries. If the oracle $\mathcal{PRP}$ is queried with a bit-string x, then it first checks, whether it has been queried with x before. If yes, then it returns the same value that it returned before. If no, then it generates a new value y, such that y has not yet been returned to any earlier query, saves the pair (x, y) for future checks, and returns y.

The only difference between left- and right-hand side is, that the permutation given to $\mathcal{A}$ is "fixed before" at the left-hand side, while it is "generated on the fly" at the right-hand side. This obviously does not change the result of $\mathcal{A}$.

State: $\sigma \subset \{\mathbf{0},\mathbf{1}\}^* \times \{\mathbf{0},\mathbf{1}\}^*$.
Initial state $\sigma_0 = \emptyset$.
On query $(\mathbf{1}^n, x)$, where $x \in \{\mathbf{0},\mathbf{1}\}^{\geq p(n)}$:

$\mathcal{PRP}$	$\mathcal{PRF}$
if $\exists y : (x,y) \in \sigma$ **then**	**if** $\exists y : (x,y) \in \sigma$ **then**
result $:= y$	*result* $:= y$
else	**else**
repeat	
generate $y \leftarrow \mathcal{U}(\{\mathbf{0},\mathbf{1}\}^{\|x\|})$	generate $y \leftarrow \mathcal{U}(\{\mathbf{0},\mathbf{1}\}^{\|x\|})$
until $\nexists x' : (x',y) \in \sigma$	
$\sigma := \sigma \cup \{(x,y)\}$	$\sigma := \sigma \cup \{(x,y)\}$
result $:= y$	*result* $:= y$
end if	**end if**
return *result*	**return** *result*

Figure 2.4: The stateful oracles $\mathcal{PRP}$ and $\mathcal{PRF}$

Similarly, for all algorithms $\mathcal{A}$ an $n \in \mathbb{N}$ holds

$$\{\!|\mathcal{A}^{\varphi(\cdot)}(\mathbf{1}^n) : \varphi \leftarrow \mathcal{U}(\{\mathbf{0},\mathbf{1}\}^{p(n)} \to \{\mathbf{0},\mathbf{1}\}^{p(n)})|\!\} = \{\!|\mathcal{A}^{\mathcal{PRF}(\mathbf{1}^n,\cdot)}(\mathbf{1}^n)|\!\} \ . \tag{2.6}$$

($\mathcal{PRF}$ is defined in Fig. 2.4) The reason is the same. From the equations (2.5) and (2.6) immediately follows:

Proposition 2.12. *An encryption system $(\mathcal{G},\mathcal{E},\mathcal{D})$ is a PRP iff it is a PRF.*

Proof. Having equations (2.5) and (2.6), it only remains to show that for all PPT algorithms $\mathcal{A}$,

$$\Pr[\mathcal{A}^{\mathcal{PRP}(\mathbf{1}^n,\cdot)}(\mathbf{1}^n) = 1] - \Pr[\mathcal{A}^{\mathcal{PRF}(\mathbf{1}^n,\cdot)}(\mathbf{1}^n) = 1]$$

is negligible (in n). The oracle $\mathcal{PRF}$ looks different from the oracle $\mathcal{PRP}$ only if the algorithm $\mathcal{A}$ manages to get the same answer from it for two different queries. Let $q(n)$, where $q \in \mathbf{Pol}(\mathbb{Z})$, be (an upper bound on) the number of different oracle queries that $\mathcal{A}^{(\cdot)}(\mathbf{1}^n)$ makes. If the oracle is $\mathcal{PRP}$ then the probability that two of the answers are equal, is 0. If the oracle is $\mathcal{PRF}$ then the probability that two of the answers are equal, is bounded by $(q(n))^2/2^{p(n)}$. The difference of these probabilities is therefore negligible. $\square$

The main results of this thesis require the encryption system to satisfy seemingly stronger security properties than being a PRP. Actually, they are incomparable, because pseudorandom permutations are deterministic, while the following definitions require the encryption algorithm to be probabilistic.

We need our encryption system to hide the identity of messages and keys. This means that if we are given two ciphertexts then we must be unable to find out whether the corresponding plaintexts were equal.

Definition 2.9. An encryption system $(\mathcal{G},\mathcal{E},\mathcal{D})$ is *repetition concealing* [AR00] if for all PPT algorithms $\mathcal{A}^{(\cdot)}$ the difference

$$\Pr\left[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\cdot)}(\mathbf{1}^n)=1\,:\,k\leftarrow\mathcal{G}(\mathbf{1}^n)\right]-\Pr\left[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|})}(\mathbf{1}^n)=1\,:\,k\leftarrow\mathcal{G}(\mathbf{1}^n)\right] \quad (2.7)$$

is negligible in n.

In the scenario depicted in (2.7), the adversary $\mathcal{A}$ can choose, which bit-strings get encrypted. The adversary becomes either the encryptions of chosen bit-strings or it becomes the encryptions of a fixed bit-string of the same length. The definition requires that the adversary cannot distinguish the encryptions of bit-strings it chose from the encryptions of $\mathbf{00}\cdots\mathbf{0}$.

We see that no encryption system where $\mathcal{E}$ is deterministic can be repetition-concealing. Indeed, if $\mathcal{E}$ were deterministic, then $\mathcal{A}$ could query its oracle twice with different arguments (of the same length). If the results of the queries are different then the oracle is $\mathcal{E}(\mathbf{1}^n,k,\cdot)$, if they are same then the oracle is $\mathcal{E}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|})$.

A security definition that looks rather different to the previous definition, but is nevertheless equivalent to it, was given by Goldwasser and Micali [GM84, MRS86][1]. In their definition of *polynomial security*, the adversary works in two stages. At the first stage it outputs two bit-strings x_0 and x_1 and possibly some information about its state that is to be inputted to the second stage. Then a bit $b\in\{0,1\}$ is uniformly randomly chosen and the second stage of the adversary is given an encryption of the bit-string x_b and the state information outputted by the first stage. The goal of the second stage is to guess the bit b. The adversary is considered successful, if it manages to guess it with probability that is significantly higher than 50%. Both stages of the adversary may access the encryption oracle. Formally, an encryption system $(\mathcal{G},\mathcal{E},\mathcal{D})$ is polynomially secure, if for all PPT algorithm pairs $(\mathcal{A}_1,\mathcal{A}_2)$ the inequality

$$\begin{aligned}\Pr\Big[b^*=b\,:\,&k\leftarrow\mathcal{G}(\mathbf{1}^n),(x_0,x_1,s)\leftarrow\mathcal{A}_1^{\mathcal{E}(\mathbf{1}^n,k,\cdot)}(\mathbf{1}^n),b\leftarrow\mathcal{U}(\{0,1\}),\\ &y\leftarrow\mathcal{E}(\mathbf{1}^n,k,x_b),b^*\leftarrow\mathcal{A}_2^{\mathcal{E}(\mathbf{1}^n,k,\cdot)}(\mathbf{1}^n,s,y)\Big]\leq\frac{1}{2}+\alpha(n)\end{aligned}$$

holds for some negligible function α. The eqivalence of this definition to Def. 2.9 is shown in [BDJR97].

Also, given two ciphertexts, we must be unable to find out whether they have been encrypted with the same key or with different keys.

Definition 2.10. An encryption system $(\mathcal{G},\mathcal{E},\mathcal{D})$ is *which-key concealing* if for all PPT algorithms $\mathcal{A}^{(\cdot),(\cdot)}$ the difference

$$\begin{aligned}&\Pr\left[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k',\cdot)}(\mathbf{1}^n)=1\,:\,k,k'\leftarrow\mathcal{G}(\mathbf{1}^n)\right]-\\ &\qquad\Pr\left[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k,\cdot)}(\mathbf{1}^n)=1\,:\,k\leftarrow\mathcal{G}(\mathbf{1}^n)\right] \quad (2.8)\end{aligned}$$

[1]Actually, their definition applies to public-key encryption systems, not secret-key systems. In public-key scenarios, the public parts of generated key(s) is/are made available to the adversary. There is a general way to transform a definition dealing with public-key encryption to a definition dealing with secret-key encryption. Instead of giving the generated public key(s) to the adversary, we give it access to oracles that encrypt with generated keys.

is negligible in n.

The definitions of repetition-concealing and which-key concealing encryption systems could also have been given in terms of indistinguishability of distributions. Namely, $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is repetition-concealing iff

$$\{\!|\mathcal{E}(\mathbf{1}^n, k, \cdot) : k \leftarrow \mathcal{G}(\mathbf{1}^n)|\!\}_{n\in\mathbb{N}} \approx \{\!|\mathcal{E}(\mathbf{1}^n, k, \mathbf{0}^{|\cdot|}) : k \leftarrow \mathcal{G}(\mathbf{1}^n)|\!\}_{n\in\mathbb{N}} \tag{2.9}$$

and which-key concealing iff

$$\{\!|(\mathcal{E}(\mathbf{1}^n, k, \cdot), \mathcal{E}(\mathbf{1}^n, k', \cdot)) : k, k' \leftarrow \mathcal{G}(\mathbf{1}^n)|\!\}_{n\in\mathbb{N}} \approx \{\!|(\mathcal{E}(\mathbf{1}^n, k, \cdot), \mathcal{E}(\mathbf{1}^n, k, \cdot)) : k \leftarrow \mathcal{G}(\mathbf{1}^n)|\!\}_{n\in\mathbb{N}} . \tag{2.10}$$

Abadi and Rogaway [AR00] also define that

Definition 2.11. An encryption system $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is *which-key and repetition concealing* if for all PPT algorithms $\mathcal{A}^{(\cdot),(\cdot)}$ the difference

$$\Pr\left[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k',\cdot)}(\mathbf{1}^n) = 1 : k, k' \leftarrow \mathcal{G}(\mathbf{1}^n)\right] - \Pr\left[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|}),\mathcal{E}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|})}(\mathbf{1}^n) = 1 : k \leftarrow \mathcal{G}(\mathbf{1}^n)\right] \tag{2.11}$$

is negligible.

It turns out that the following lemma holds:

Lemma 2.13. *An encryption system $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is which-key and repetition concealing iff it is which-key concealing and repetition concealing.*

We have not been able to find the proof of this lemma in the literature (neither have we found Def. 2.10, although we believe that they were known to Abadi and Rogaway). Therefore we present its proof here.

Proof. **Direction $(\Rightarrow)$.** Suppose that $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not repetition concealing, i.e. there exists a PPT algorithm $\mathcal{A}^{(\cdot)}$ such that (2.7) is not negligible. Modify the algorithm $\mathcal{A}^{(\cdot)}$ so, that it takes two oracles instead of one, but queries only the left one. For such algorithm, the difference (2.11) is not negligible, i.e. $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not which-key and repetition concealing.

On the other hand, suppose that $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not which-key concealing, but is which-key and repetition concealing. This means that there exists an algorithm $\mathcal{A}^{(\cdot),(\cdot)}$ for which (2.8) is not negligible, but (2.11) is negligible. The difference of (2.8) and (2.11):

$$\Pr\left[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k,\cdot)}(\mathbf{1}^n) = 1 : k \leftarrow \mathcal{G}(\mathbf{1}^n)\right] - \Pr\left[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|}),\mathcal{E}(\mathbf{1}^n,k,\mathbf{0}^{|\cdot|})}(\mathbf{1}^n) = 1 : k \leftarrow \mathcal{G}(\mathbf{1}^n)\right] \tag{2.12}$$

is not negligible as well. Let $\bar{\mathcal{A}}^{(\cdot)}$ be an algorithm that runs the algorithm $\mathcal{A}^{(\cdot),(\cdot)}$, and whenever it queries an oracle (no matter whether the left or right one), $\bar{\mathcal{A}}^{(\cdot)}$

passes the query to its own single oracle. The difference (2.7) is not negligible for the algorithm $\bar{\mathcal{A}}^{(\cdot)}$, thus $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not repetition concealing. But we just showed that a which-key and repetition concealing encryption scheme must be repetition concealing. A contradiction.

Direction ($\Leftarrow$)**.** Suppose that $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not which-key and repetition concealing, but is which-key concealing. This means that there exists an algorithm $\mathcal{A}^{(\cdot),(\cdot)}$ for which (2.8) is negligible, but (2.11) is not negligible. We again get that their difference (2.12) is not negligible and thus $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ is not repetition concealing. $\square$

It is believed that primitive block ciphers, for example DES [DES99], Idea [LM90] and AES [AES01], are pseudorandom permutations. Namely, being a PRP is their main design goal. If PRPs exist, then which-key and repetition concealing encryption systems exist, too. Abadi and Rogaway [AR00, Sec. 4.4] demostrate, how to construct a which-key and repetition concealing encryption system from a PRP.

Chapter 3

Computational Security

This chapter describes the programming language that we are handling. It gives its syntax and semantics. This chapter also gives the definition of secure information flow.

We start this chapter in Sec. 3.1.1 by introducing the syntax of the simple imperative language that we are going to use throughout this and the next two chapters. We continue in Sec. 3.1.2 by giving *the* (concrete) semantics of that language. For each program P we define a function $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$ that maps the initial state of the program to its corresponding final state. The semantics is thus denotational. As $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$ is *the* concrete semantics of P, we cannot speak about its correctness. We can only speak about its intuitiveness. It will become rather clear that this semantics indeed is basically the same semantics that is defined in [NN92, Sec. 4.1].

For the purposes of the next chapter, we give a different specification for (a part of) the semantics $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$. The difference is in defining $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$ for loops. The denotational semantics for loops is defined via a fixed-point operation. In section 3.1.3 we show that the least fixed point of some other operator is equal to the least fixed point of the operator that was used to define $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$ for loops in section 3.1.2. That other operator is more similar to the operator whose least fixed point is going to be defined to be the abstract semantics (i.e. the analysis) of loops.

Having specified the programming language and its meaning, we move towards defining the secure programs. Section 3.2.1 defines, when a program runs in expected polynomial time. The notion of confidentiality is defined only for such programs. However, we argue that this does not restrict the generality. For a program P running in expected polynomial time, if its inputs are distributed according to some fixed probability distribution D, the structure of $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$ is simpler than in the general case (the running time of the program becomes irrelevant and we do not have to deal with nontermination). We define the *output distribution* $\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)$ for the program P and the input distribution D. The distribution $\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)$ is only defined for programs P and distributions D, where P runs in expected polynomial time if its inputs are distributed according to D.

Finally, section 3.2.2 gives the definition of secure information flow. The security of information flow is defined through the independence of a program's secret inputs and public outputs. For comparison we also give another definition based on semantic security and show that it is subsumed by the first definition.

We finish by discussing some topics in Sec. 3.3. First, we show that our security definition is just a variant of defining a program secure if it is equivalent to some program that we have deemed "obviously secure". Second, we discuss how two structures on the set of distributions over program states — least upper bounds and indistinguishability — interact and what this means.

3.1 Syntax and Semantics of the Programming Language

3.1.1 Syntax

We consider a simple imperative programming language, the WHILE-language [NN92], for the treatment of our analysis. The language is nevertheless Turing-complete and can demonstrate all the interesting parts of the analyses. Adding more features to the language should be rather straightforward.

Let **Var** be the set of variables and **Op** be the set of arithmetic, relational, boolean, etc. operators. The syntax of programs is given by the following grammar:

$$\begin{array}{rcl} \mathsf{P} & ::= & x := o(x_1, \ldots, x_k) \\ & | & \mathit{skip} \\ & | & \mathsf{P}_1; \mathsf{P}_2 \\ & | & \mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2 \\ & | & \mathit{while}\ b\ \mathit{do}\ \mathsf{P}, \end{array}$$

where $b, x, x_1, \ldots, x_k$ range over **Var**, o ranges over **Op** and $\mathsf{P}_1, \mathsf{P}_2$ range over programs. Sometimes (in the definitions) we also make use of the statement *stuck* that denotes a program that never terminates.

We are handling encryption, thus we must have a binary operator $\mathcal{E}nc \in \mathbf{Op}$ that denotes the encryption operation. We also need a nullary key generation operator $\mathcal{G}en \in \mathbf{Op}$ that is used to create "good" keys. The analysis handles these operators differently from the other ones, taking into account that they are expressing secure encryption[1]. Later, when presenting the analysis, we will also need other operators with special, fixed semantics (namely, unary identity operator that we use for *simple assignments* $x := y$).

3.1.2 Denotational Semantics

We will use a denotational-style semantics as the (concrete) semantics of the programming language. This shows our disinterest in *how* a program computes, we are only interested in *what* it computes. Traditionally, a denotational semantics of an

[1]Decryption does not have to be handled separately, thus it was not mentioned here.

imperative language maps the input state of a program to its corresponding output state, i.e. it has the type $\mathbf{State} \to \mathbf{State}_\perp$, where

- $\mathbf{State} := \mathbf{Var} \to \mathbf{Val}$ is the set of program states. A program state of an imperative language maps the variables to their values; **Val** is the set of values (usually not specified any further).
- $\mathbf{State}_\perp := \mathbf{State} \uplus \{\perp\}$. The element $\perp$ denotes nontermination.

Such simple formulation is not sufficient for our purposes. We need the following into account:

The security parameter. The security of the encryption is defined as the negligibility of a certain advantage of the adversary. Negligibility is defined in terms of the security parameter $n \in \mathbb{N}$, in fact, we have a whole family of encryption functions, this family is indexed by the security parameter. We accommodate the security parameter in the semantics by parameterising the program state by it, too. Thus let $\mathbf{State}_n := \mathbf{Var} \to \mathbf{Val}_n$, where $\mathbf{Val}_n$ is the n-th set of values. The semantics $\mathbb{C}_{\text{len}}[\![\mathsf{P}]\!]$ that we are going to define as the concrete semantics of the programming language, can actually be seen as a *family* of functions mapping the input state to the output state, indexed by the security parameter. The n-th component of $\mathbb{C}_{\text{len}}[\![\mathsf{P}]\!]$ works with states of type $\mathbf{State}_n$.

The types of values. The security of the encryption is defined for encryption algorithms that work with bit-strings. To be able to use it, we fix $\mathbf{Val}_n := \{\mathbf{0}, \mathbf{1}\}^{\ell(n)}$. Here $\ell \in \mathbf{Pol}(\mathbb{Z})$ is a fixed polynomial. We call ℓ the *length polynomial.*

Obviously, we also need boolean values (as guards of conditional statements and loops). We therefore assume that the set $\mathbf{Val}_n$ is partitioned into two parts, where the values in one of the parts are meant to denote the boolean value false and the values in the other of the parts are meant to denote the boolean value true. For example, the value $\mathbf{0}^{\ell(n)}$ could denote false and all other values could denote true.

Probabilistic execution. No deterministic algorithm can satisfy the definition of security of the encryption. If our semantics of the program language did not allow probabilistic execution, then our results would hold only for a nonexistent case. To allow probabilism, the range of the semantics must be the set of probability distributions over program states.

Denote:

- $\mathbf{Distr} := \mathcal{D}^{\mathbb{N}}(\mathbf{State})$.
- $\mathbf{Distr}\perp := \prod_{n \in \mathbb{N}} \mathcal{D}(\mathbf{State}_{n\perp})$.
- $\mathbf{Distr}_{\mathsf{pol}} \subset \mathbf{Distr}$ and $\mathbf{Distr}\perp_{\mathsf{pol}} \subset \mathbf{Distr}\perp$ contain the distributions constructible in polynomial time.

Computation length. The encryption system is secure only against polynomially bounded adversaries. If the program runs for a too long time, then we no longer can make use of the secureness of encryption. We thus record the computation length in our semantics.

We let $\mathbf{T}$ be the set of possible lengths of computations. The length of a computation is defined as the number of steps that the program does, thus $\mathbf{T} = \mathbb{N}$.

Let $\mathbf{ST}_n = \mathbf{State}_n \times \mathbf{T}$. The quantities of type $\mathbf{ST}_n$ are used to record the final state of the program together with the number of steps that the program made. Let $\mathsf{state} : \mathbf{ST}_n \to \mathbf{State}_n$ and $\mathsf{time} : \mathbf{ST}_n \to \mathbf{T}$ be the projections onto the first and second component of $\mathbf{ST}_n$, respectively.

The semantics $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$ that we are going to define, maps an initial state $S_{n,s} \in \mathbf{State}_n$ to a probability distribution over $\mathbf{ST}_{n\perp}$, i.e. to the probability distribution over the pairs of possible final states and computation lengths, with $\perp$ denoting nontermination. The n-th component of $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$ has thus the type $\mathbf{Trans}_n := \mathbf{State}_n \rightsquigarrow \mathbf{ST}_{n\perp}$. The type of the entire $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$ is $\mathbf{CLenType} := \prod_{n\in\mathbb{N}} \mathbf{Trans}_n$.

The semantics $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$ is *the* (concrete) semantics of the programming language. The correctness of the abstract semantics (i.e. the analysis) given in the next chapter is defined with respect to it.

Before defining $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$, we need semantics for each $o \in \mathbf{Op}$. For an operator o of arity k, the semantics is a family of functions $[\![o]\!] : \mathbf{Val}^k \overset{\mathbb{N}}{\rightsquigarrow} \mathbf{Val}$. As the concrete semantics of the program should be computable in polynomial time (otherwise we cannot use the security definition for the encryption), we require that $[\![o]\!]$ is computable in polynomial time.

We mentioned before that we need the set $\mathbf{Op}$ to contain certain operators — namely $\mathcal{E}nc$, $\mathcal{G}en$ and the identity operator (we denote it here by id). The following must hold for the semantics of these operators:

- $([\![\mathcal{G}en]\!], [\![\mathcal{E}nc]\!])$ is a which-key and repetition concealing encryption scheme.
- $[\![id]\!]_n(x) = \eta^{\mathcal{D}}(x)$ for each $x \in \mathbf{Val}_n$.

The semantics $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$ is defined in Fig. 3.1. The definition is pretty standard [NN92, Sec. 4.1], except for the extra details about security parameter, probabilism and recording the length of the computation.

Let us describe the definition of $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$, as given on Fig. 3.1.

- The semantics of $x := o(x_1, \ldots, x_k)$ replaces the value of the variable x with a newly computed value. The length of the program is 1.
- The semantics of *skip* is basically the identity function. We have defined the length of the program *skip* to be 0 because we want to keep the equality of semantics of the programs P, $\mathsf{P}; \mathit{skip}$ and $\mathit{skip}; \mathsf{P}$. If we had not considered the computation lengths, then these three programs obviously would have

$$\boxed{\begin{array}{l}\mathbb{C}_{\text{len}}[\![x := o(x_1,\ldots,x_k)]\!]_n(S_n) = \\ \qquad \{\!|(S_n[x \mapsto x_{\text{val}}], 1) \,:\, x_{\text{val}} \leftarrow [\![o]\!]_n(S_n(x_1),\ldots,S_n(x_k))|\!\}\end{array}} \ .$$

$$\boxed{\mathbb{C}_{\text{len}}[\![\mathit{skip}]\!]_n(S_n) = \eta^{\mathcal{D}}((S_n, 0))} \ .$$

$$\boxed{\mathbb{C}_{\text{len}}[\![\mathsf{P}_1; \mathsf{P}_2]\!]_n = \text{compose}_n(\mathbb{C}_{\text{len}}[\![\mathsf{P}_1]\!]_n, \mathbb{C}_{\text{len}}[\![\mathsf{P}_2]\!]_n)}$$

where $\text{compose}_n : \mathbf{Trans}_n \times \mathbf{Trans}_n \to \mathbf{Trans}_n$ is defined as

$$\begin{array}{l}\text{compose}_n(\mathsf{T}_{1,n}, \mathsf{T}_{2,n})(S_n) = \\ \qquad \{\!|\langle\!\langle S''_n \stackrel{?}{=} \bot \ \textbf{?} \ \bot \ \textbf{:} \ (\mathsf{state}(S''_n), \mathsf{time}(S'_n) + \mathsf{time}(S''_n))\rangle\!\rangle \ : \\ \qquad\qquad S'_n \leftarrow \mathsf{T}_{1,n}(S_n), S''_n \leftarrow \langle\!\langle S'_n \stackrel{?}{=} \bot \ \textbf{?} \ \eta^{\mathcal{D}}(\bot) \ \textbf{:} \ \mathsf{T}_{2,n}(\mathsf{state}(S'_n))\rangle\!\rangle|\!\}\end{array}$$

where $\langle\!\langle b \ \textbf{?} \ x \ \textbf{:} \ y\rangle\!\rangle$ is equal to x if b is true, and is equal to y otherwise.

$$\boxed{\mathbb{C}_{\text{len}}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2]\!]_n = \text{cond}_n(b, \mathbb{C}_{\text{len}}[\![\mathsf{P}_1]\!]_n, \mathbb{C}_{\text{len}}[\![\mathsf{P}_2]\!]_n)}$$

where $\text{cond}_n : \mathbf{Var} \times \mathbf{Trans}_n \times \mathbf{Trans}_n \to \mathbf{Trans}_n$ is defined as

$$\text{cond}_n(b, \mathsf{T}_{1,n}, \mathsf{T}_{2,n})(S_n) = \langle\!\langle S_n(b) \ \textbf{?} \ \mathsf{T}_{1,n}(S_n) \ \textbf{:} \ \mathsf{T}_{2,n}(S_n)\rangle\!\rangle \ .$$

$$\boxed{\mathbb{C}_{\text{len}}[\![\mathit{while}\ b\ \mathit{do}\ \mathsf{P}]\!] = \mathit{lfp}\ F_{\mathsf{P}}}$$

where $F_{\mathsf{P}} : \mathbf{CLenType} \to \mathbf{CLenType}$ is defined by

$$[F_{\mathsf{P}}(\mathsf{T})]_n = \text{cond}_n(b, \text{compose}_n(\mathbb{C}_{\text{len}}[\![\mathsf{P}]\!]_n, \mathsf{T}_n), \mathbb{C}_{\text{len}}[\![\mathit{skip}]\!]_n) \ .$$

$$\boxed{\mathbb{C}_{\text{len}}[\![\mathit{stuck}]\!]_n(S_n) = \eta^{\mathcal{D}}(\bot)} \ .$$

Figure 3.1: The semantics $\mathbb{C}_{\text{len}}[\![\mathsf{P}]\!]$

had the same semantics. It is more convenient to keep this equality. The security definition will remain the same, no matter whether the length of the computation of the program *skip* is 0 or 1.

- The semantics of $\mathsf{P}_1; \mathsf{P}_2$ is basically the functional composition of the semantices of P_1 and P_2. The auxiliary function compose is used to deal with lifting the intermediate values to the right domains. From the definition of compose we also see that the running time of the program $\mathsf{P}_1; \mathsf{P}_2$ is the sum of the running times of programs P_1 and P_2.

- The semantics of *if b then* P_1 *else* P_2 chooses either the semantics of P_1 or the semantics of P_2, depending on the value of the variable b in the initial state. An interesting detail here is that checking the value of the variable b takes no time. Again, this does not change the security definition. The reason for this definition is to make the proof of Proposition 3.7 simpler.

- The semantics of *while b do* P is defined so that the equation

$$\mathbb{C}_{\mathrm{len}}[\![\textit{while } b \textit{ do } \mathsf{P}]\!] = \mathbb{C}_{\mathrm{len}}[\![\textit{if } b \textit{ then } (\mathsf{P}; \textit{while } b \textit{ do } \mathsf{P}) \textit{ else skip}]\!] \qquad (3.1)$$

  is satisfied. Generally, this equation may have several solutions. When comparing equation (3.1) with the definition of the function F_{P} in Fig. 3.1 we see, that the solutions of (3.1) are fixed points of F_{P} and vice versa. The definition in Fig. 3.1 is such, that we choose the *smallest* solution of equation (3.1) as the semantics of *while b do* P. This is the standard way of defining the semantics of loops.

- The semantics of *stuck* maps each program state to nontermination.

The semantics of *while b do* P is defined through a fixed point computation. We must make sure that this fixed point exists. It is sufficient to show that **CLenType** and F_{P} satisfy the assumptions of Prop. 2.3. Propositions 3.1 and 3.2 show it.

Proposition 3.1. **CLenType** *is a complete partial order.*

Proof. $\mathbf{CLenType} = \prod_{n\in\mathbb{N}} \mathbf{Trans}_n$. If $\mathbf{Trans}_n$ is a CPO for each $n \in \mathbb{N}$, then **CLenType** is also a CPO and the least upper bounds on **CLenType** are defined componentwise. It suffices to show that $\mathbf{Trans}_n$ is a CPO.

$\mathbf{Trans}_n = \mathbf{State}_n \to \mathcal{D}(\mathbf{ST}_{n\perp})$. If $\mathcal{D}(\mathbf{ST}_{n\perp})$ is a CPO then $\mathbf{Trans}_n$ is also a CPO and the least upper bounds on $\mathbf{Trans}_n$ are defined pointwise. But Lemma 2.8 gives that $\mathcal{D}(\mathbf{ST}_{n\perp})$ is a CPO. □

Proposition 3.2. *The function* F_{P} *in Fig. 3.1 is continuous.*

The arguments in the proof of this proposition follow the proof of a similar result in [NN92, Sec. 4.3] We first show that the functions $\mathrm{compose}_n$ and cond_n are continuous in one of their arguments.

Lemma 3.3. *Let* $\mathsf{U} \in \mathbf{CLenType}$. *Let* $F^{(1)} : \mathbf{CLenType} \to \mathbf{CLenType}$ *be defined by*

$$[F^{(1)}(\mathsf{T})]_n = \mathrm{compose}_n(\mathsf{U}_n, \mathsf{T}_n) \enspace .$$

Then the function $F^{(1)}$ *is continuous.*

Proof. The function $F^{(1)}$ is obviously monotone. Indeed, if $S_n \in \mathbf{State}_n$ and $\mathsf{T}' \leq \mathsf{T}''$, then

$$\begin{aligned}
&[F^{(1)}(\mathsf{T}')]_n(S_n) = \mathrm{compose}_n(\mathsf{U}_n, \mathsf{T}'_n)(S_n) = \\
&\qquad \{\!|\langle\!\langle S''_n \stackrel{?}{=} \bot \ \mathbf{?}\ \bot \mathbin{\mathbf{:}} (\mathsf{state}(S''_n), \mathsf{time}(S'_n) + \mathsf{time}(S''_n))\rangle\!\rangle \ : \\
&\qquad S'_n \leftarrow \mathsf{U}_n(S_n), S''_n \leftarrow \langle\!\langle S'_n \stackrel{?}{=} \bot \ \mathbf{?}\ \eta^{\mathcal{D}}(\bot) \mathbin{\mathbf{:}} \mathsf{T}'_n(\mathsf{state}(S'_n))\rangle\!\rangle |\!\} \leq \\
&\qquad \{\!|\langle\!\langle S''_n \stackrel{?}{=} \bot \ \mathbf{?}\ \bot \mathbin{\mathbf{:}} (\mathsf{state}(S''_n), \mathsf{time}(S'_n) + \mathsf{time}(S''_n))\rangle\!\rangle \ : \\
&\qquad S'_n \leftarrow \mathsf{U}_n(S_n), S''_n \leftarrow \langle\!\langle S'_n \stackrel{?}{=} \bot \ \mathbf{?}\ \eta^{\mathcal{D}}(\bot) \mathbin{\mathbf{:}} \mathsf{T}''_n(\mathsf{state}(S'_n))\rangle\!\rangle |\!\} = \\
&\qquad\qquad \mathrm{compose}_n(\mathsf{U}_n, \mathsf{T}''_n)(S_n) = [F^{(1)}(\mathsf{T}'')]_n(S_n) \enspace .
\end{aligned}$$

Now let I be an index set and let $\mathsf{T}^{(i)} \in \mathbf{CLenType}$ for all $i \in I$. Moreover, let $\mathsf{T}^{(i)}$ and $\mathsf{T}^{(j)}$ be comparable for each $i, j \in I$. Let $S_n \in \mathbf{State}_n$. We have

$$\begin{aligned}
&[F^{(1)}(\bigvee_{i\in I} \mathsf{T}^{(i)})]_n(S_n) = \mathrm{compose}_n(\mathsf{U}_n, \bigvee_{i\in I} \mathsf{T}^{(i)}_n)(S_n) = \\
&\qquad \{\!|\langle\!\langle S''_n \stackrel{?}{=} \bot \ \mathbf{?}\ \bot \mathbin{\mathbf{:}} (\mathsf{state}(S''_n), \mathsf{time}(S'_n) + \mathsf{time}(S''_n))\rangle\!\rangle \ : \\
&\qquad S'_n \leftarrow \mathsf{U}_n(S_n), S''_n \leftarrow \langle\!\langle S'_n \stackrel{?}{=} \bot \ \mathbf{?}\ \eta^{\mathcal{D}}(\bot) \mathbin{\mathbf{:}} \bigvee_{i\in I} \mathsf{T}^{(i)}_n(\mathsf{state}(S'_n))\rangle\!\rangle |\!\} = \\
&\qquad \bigvee_{i\in I} \{\!|\langle\!\langle S''_n \stackrel{?}{=} \bot \ \mathbf{?}\ \bot \mathbin{\mathbf{:}} (\mathsf{state}(S''_n), \mathsf{time}(S'_n) + \mathsf{time}(S''_n))\rangle\!\rangle \ : \\
&\qquad S'_n \leftarrow \mathsf{U}_n(S_n), S''_n \leftarrow \langle\!\langle S'_n \stackrel{?}{=} \bot \ \mathbf{?}\ \eta^{\mathcal{D}}(\bot) \mathbin{\mathbf{:}} \mathsf{T}^{(i)}_n(\mathsf{state}(S'_n))\rangle\!\rangle |\!\} = \\
&\qquad \bigvee_{i\in I} \mathrm{compose}_n(\mathsf{U}_n, \mathsf{T}_n)(S_n) = \bigvee_{i\in I} [F^{(1)}(\mathsf{T}^{(i)})]_n(S_n) = [\bigvee_{i\in I} F^{(1)}(\mathsf{T}^{(i)})]_n(S_n) \enspace .
\end{aligned}$$

Here, at the first equals-sign, we have used Lemma 2.2 for Cartesian products. At the second equals-sign (between first and second row), we have used the same lemma for functions. □

Lemma 3.4. *Let* $b \in \mathbf{Var}$ *and let* $F^{(2)} : \mathbf{CLenType} \to \mathbf{CLenType}$ *be defined by*

$$[F^{(2)}(\mathsf{T})]_n = \mathrm{cond}_n(b, \mathsf{T}_n, \mathbb{C}_{\mathrm{len}}[\![\mathsf{skip}]\!]_n) \enspace .$$

Then the function $F^{(2)}$ *is continuous.*

Proof. Similar to Lemma 3.3. □

Proof of proposition 3.2. The function F_{P} in Fig. 3.1 is equal to the composition $F^{(2)} \circ F^{(1)}$, where

$$[F^{(1)}(\mathsf{T})]_n = \mathrm{compose}_n(\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]_n, \mathsf{T}_n)$$
$$[F^{(2)}(\mathsf{T})]_n = \mathrm{cond}_n(b, \mathsf{T}_n, \mathbb{C}_{\mathrm{len}}[\![\mathsf{skip}]\!]_n) \ .$$

Both $F^{(1)}$ and $F^{(2)}$ are continuous, therefore their composition is continuous, too [NN92, Lem. 4.35]. □

3.1.3 An Alternative Formulation for Loops

We give another description for the semantics of while-loops. It is easier to use in the proof of correctness of the analysis.

Instead of the equation (3.1) there are other equations that intuitively may serve as the specification of the semantics for *while*-statements. Here we intend to define this semantics as a solution of the equation

$$\mathbb{C}_{\mathrm{len}}[\![\textit{while } b \textit{ do } \mathsf{P}]\!] = \mathbb{C}_{\mathrm{len}}[\![\textit{if } b \textit{ then } \mathsf{P} \textit{ else skip}; \textit{while } b \textit{ do } \mathsf{P}]\!] \ . \tag{3.2}$$

The solutions of this equations are exactly the fixed points of the function $\bar{F}_{\mathsf{P}}$, where

$$[\bar{F}_{\mathsf{P}}(\mathsf{S})]_n = \mathrm{compose}_n(\mathbb{C}_{\mathrm{len}}[\![\textit{if } b \textit{ then } \mathsf{P} \textit{ else skip}]\!]_n, \mathsf{S}) \ .$$

However, we cannot define $\mathbb{C}_{\mathrm{len}}[\![\textit{while } b \textit{ do } \mathsf{P}]\!]$ as the least solution of the equation (3.2), because the least solution is $\mathbb{C}_{\mathrm{len}}[\![\textit{stuck}]\!]$.

Let $\mathsf{U} = \mathbb{C}_{\mathrm{len}}[\![\textit{if } b \textit{ then stuck else skip}]\!]$. For a program P we will show that the semantics of *while* b *do* P is the least solution of (3.2) that is greater than or equal to U. In other words, $\mathbb{C}_{\mathrm{len}}[\![\textit{while } b \textit{ do } \mathsf{P}]\!]$ could also be defined as $\mathit{lfp}^{\mathsf{U}}\ \bar{F}_{\mathsf{P}}$. First, we must show that this least fixed point exists.

Lemma 3.5. *The function $\bar{F}_{\mathsf{P}}$ is continuous for each program* P.

Proof. This follows directly from lemma 3.3. □

Lemma 3.6. $\mathsf{U} \leq \bar{F}_{\mathsf{P}}(\mathsf{U})$.

Proof. Let $S_n \in \mathbf{State}_n$, then

$$\mathsf{U}_n(S_n) = \begin{cases} \eta^{\mathcal{D}}(\perp), & \text{if } S_n(b) = \mathsf{true} \\ \eta^{\mathcal{D}}((S_n, 0)), & \text{if } S_n(b) = \mathsf{false} \end{cases}$$

and

$$[\bar{F}_{\mathsf{P}}(\mathsf{U})]_n(S_n) = \begin{cases} \mathbb{C}_{\mathrm{len}}[\![\textit{if } b \textit{ then } \mathsf{P} \textit{ else skip}]\!]_n(S_n), & \text{if } S_n(b) = \mathsf{true} \\ \eta^{\mathcal{D}}((S_n, 0)), & \text{if } S_n(b) = \mathsf{false} \end{cases}$$

Obviously $\eta^{\mathcal{D}}(\perp) \leq \mathbb{C}_{\mathrm{len}}[\![\textit{if } b \textit{ then } \mathsf{P} \textit{ else skip}]\!]_n(S_n)$, therefore $\mathsf{U} \leq \bar{F}_{\mathsf{P}}(\mathsf{U})$. □

From the preceding two lemmas follows that $\mathit{lfp}^{\mathsf{U}}\ \bar{F}_{\mathsf{P}}$ exists.

Proposition 3.7. *For all programs* P, $\mathbb{C}_{\mathrm{len}}[\![\mathit{while}\ b\ \mathit{do}\ \mathsf{P}]\!] = \mathit{lfp}^{\mathsf{U}}\ \bar{F}_{\mathsf{P}}$.

Proof. We show that $\bar{F}^i_{\mathsf{P}}(\mathsf{U}) = F^{i+1}_{\mathsf{P}}(\bot)$. The claim of the proposition follows then from Propositions 2.3 and 2.4 (precisely, from the formulae for computing the least fixed points).

$\bar{F}^i_{\mathsf{P}}(\mathsf{U}) = F^{i+1}_{\mathsf{P}}(\bot)$ is shown by induction over i. If $i = 0$ then

$$\bar{F}^0_{\mathsf{P}}(\mathsf{U}) = \mathbb{C}_{\mathrm{len}}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}\ \mathit{else}\ \mathit{skip}]\!] = F^1_{\mathsf{P}}(\bot)\ .$$

Suppose that $\bar{F}^i_{\mathsf{P}}(\mathsf{U}) = F^{i+1}_{\mathsf{P}}(\bot)$ for some $i \in \mathbb{N}$. Let $S_n \in \mathbf{State}_n$. Then either $S_n(b) = \mathsf{true}$ or $S_n(b) = \mathsf{false}$. If $S_n(b) = \mathsf{true}$, then

$$\begin{aligned}
&[\bar{F}^{i+1}_{\mathsf{P}}(\mathsf{U})]_n(S_n) = [\bar{F}_{\mathsf{P}}(\bar{F}^i_{\mathsf{P}}(\mathsf{U}))]_n(S_n) = \\
&\quad \mathrm{compose}_n(\mathbb{C}_{\mathrm{len}}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}\ \mathit{else}\ \mathit{skip}]\!]_n, [\bar{F}^i_{\mathsf{P}}(\mathsf{U})]_n)(S_n) = \\
&\quad \{\!|\langle\!\langle S''_n \stackrel{?}{=} \bot\ \mathbf{?}\ \bot : (\mathsf{state}(S''_n), \mathsf{time}(S'_n) + \mathsf{time}(S''_n))\rangle\!\rangle\ : \\
&\qquad S'_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}\ \mathit{else}\ \mathit{skip}]\!]_n(S_n), \\
&\qquad S''_n \leftarrow \langle\!\langle S'_n \stackrel{?}{=} \bot\ \mathbf{?}\ \eta^{\mathcal{D}}(\bot) : [\bar{F}^i_{\mathsf{P}}(\mathsf{U})]_n(\mathsf{state}(S'_n))\rangle\!\rangle |\!\} = \\
&\quad \{\!|\langle\!\langle S''_n \stackrel{?}{=} \bot\ \mathbf{?}\ \bot : (\mathsf{state}(S''_n), \mathsf{time}(S'_n) + \mathsf{time}(S''_n))\rangle\!\rangle\ : \\
&\qquad S'_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]_n(S_n), S''_n \leftarrow \langle\!\langle S'_n \stackrel{?}{=} \bot\ \mathbf{?}\ \eta^{\mathcal{D}}(\bot) : [\bar{F}^i_{\mathsf{P}}(\mathsf{U})]_n(\mathsf{state}(S'_n))\rangle\!\rangle |\!\} = \\
&\quad \mathrm{compose}_n(\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]_n, [\bar{F}^i_{\mathsf{P}}(\mathsf{U})]_n)(S_n) = \mathrm{compose}_n(\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]_n, [F^{i+1}_{\mathsf{P}}(\bot)]_n)(S_n) = \\
&\quad \langle\!\langle S_n(b)\ \mathbf{?}\ \mathrm{compose}_n(\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]_n, [F^{i+1}_{\mathsf{P}}(\bot)]_n)(S_n) : \mathbb{C}_{\mathrm{len}}[\![\mathit{skip}]\!]_n(S_n)\rangle\!\rangle = \\
&\quad \mathrm{cond}_n(b, \mathrm{compose}_n(\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]_n, [F^{i+1}_{\mathsf{P}}(\bot)]_n), \mathbb{C}_{\mathrm{len}}[\![\mathit{skip}]\!]_n))(S_n) = \\
&\qquad [F_{\mathsf{P}}(F^{i+1}_{\mathsf{P}}(\bot))]_n(S_n) = [F^{i+2}_{\mathsf{P}}(\bot)]_n(S_n)\ .
\end{aligned}$$

If $S_n(b) = \mathsf{false}$, then

$$\begin{aligned}
&[\bar{F}^{i+1}_{\mathsf{P}}(\mathsf{U})]_n(S_n) = [\bar{F}_{\mathsf{P}}(\bar{F}^i_{\mathsf{P}}(\mathsf{U}))]_n(S_n) = \\
&\quad \mathrm{compose}_n(\mathbb{C}_{\mathrm{len}}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}\ \mathit{else}\ \mathit{skip}]\!]_n, [\bar{F}^i_{\mathsf{P}}(\mathsf{U})]_n)(S_n) = \\
&\quad \{\!|\langle\!\langle S''_n \stackrel{?}{=} \bot\ \mathbf{?}\ \bot : (\mathsf{state}(S''_n), \mathsf{time}(S'_n) + \mathsf{time}(S''_n))\rangle\!\rangle\ : \\
&\qquad S'_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}\ \mathit{else}\ \mathit{skip}]\!]_n(S_n), \\
&\qquad S''_n \leftarrow \langle\!\langle S'_n \stackrel{?}{=} \bot\ \mathbf{?}\ \eta^{\mathcal{D}}(\bot) : [\bar{F}^i_{\mathsf{P}}(\mathsf{U})]_n(\mathsf{state}(S'_n))\rangle\!\rangle |\!\} = \\
&\quad \{\!|\langle\!\langle S''_n \stackrel{?}{=} \bot\ \mathbf{?}\ \bot : (\mathsf{state}(S''_n), \mathsf{time}(S'_n) + \mathsf{time}(S''_n))\rangle\!\rangle\ : \\
&\qquad S'_n = (S_n, 0), S''_n \leftarrow \langle\!\langle S'_n \stackrel{?}{=} \bot\ \mathbf{?}\ \eta^{\mathcal{D}}(\bot) : [\bar{F}^i_{\mathsf{P}}(\mathsf{U})]_n(\mathsf{state}(S'_n))\rangle\!\rangle |\!\} = \\
&\quad \{\!|\langle\!\langle S''_n \stackrel{?}{=} \bot\ \mathbf{?}\ \bot : (\mathsf{state}(S''_n), \mathsf{time}(S''_n))\rangle\!\rangle\ :\ S''_n \leftarrow [\bar{F}^i_{\mathsf{P}}(\mathsf{U})]_n(S_n) |\!\} = \\
&\quad [\bar{F}^i_{\mathsf{P}}(\mathsf{U})]_n(S_n) = [\bar{F}^{i-1}_{\mathsf{P}}(\mathsf{U})]_n(S_n) = \cdots = [\bar{F}^0_{\mathsf{P}}(\mathsf{U})]_n(S_n) = \eta^{\mathcal{D}}((S_n, 0)) = \\
&\quad \langle\!\langle S_n(b)\ \mathbf{?}\ \mathrm{compose}_n(\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]_n, [F^{i+1}_{\mathsf{P}}(\bot)]_n)(S_n) : \mathbb{C}_{\mathrm{len}}[\![\mathit{skip}]\!]_n(S_n)\rangle\!\rangle = \\
&\quad \mathrm{cond}_n(b, \mathrm{compose}_n(\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]_n, [F^{i+1}_{\mathsf{P}}(\bot)]_n), \mathbb{C}_{\mathrm{len}}[\![\mathit{skip}]\!]_n))(S_n) = \\
&\qquad [F_{\mathsf{P}}(F^{i+1}_{\mathsf{P}}(\bot))]_n(S_n) = [F^{i+2}_{\mathsf{P}}(\bot)]_n(S_n)\ .
\end{aligned}$$

Therefore $\bar{F}_{\mathsf{P}}^{i+1}(\mathsf{U}) = F_{\mathsf{P}}^{i+2}(\perp)$. □

3.2 Security Definition

The model of security that we have in mind here is the following: There is a certain set of *private* variables $\mathbf{Var}_{\mathrm{S}} \subseteq \mathbf{Var}$ whose initial values we want to keep secret. After the program P has run, the values of the variables in a certain set $\mathbf{Var}_{\mathsf{P}} \subseteq \mathbf{Var}$ become *public*. The attacker tries to find out something about the initial values of secret variables. It can read the final values of public variables.

We assume that $\mathbf{Var}_{\mathrm{S}} \cap \mathbf{Var}_{\mathrm{P}} = \emptyset$. This assumption obviously does not restrict the generality, it may be overcome by introducing extra output variables. This assumption makes the subsequent arguments a bit simpler (especially Fig. 3.3 and Sec. 3.3), as we have less cases to handle.

3.2.1 "Terminating" Programs

We define security only for programs that run in polynomial time. We claim that this decision causes us no loss of generality. Namely, before the attacker obtains the final values of public variables, it is expected to wait for the program to finish its execution. If the program runs for too long time and the attacker keeps waiting, then it cannot find out anything about the initial values of secret variables. Alternatively, at a certain moment the attacker may decide that the program is taking too long time to run and should be considered to be effectively nonterminating; the final state should be considered to be $\perp$. We "compose" the original program and the attacker's decision-making process about the running time of the program. The result is a program that runs in polynomial time. We could define the original program to be secure iff the composed program is.

The next definition formalises the notion of running in polynomial time. Fig. 3.2 gives a transformation for turning an arbitrary program to one that terminates in polynomial time. This transformation corresponds to the attacker's decision-making process about the running time of the program.

Definition 3.1. A program P runs in *expected polynomial time* for a distribution of inputs $D \in \mathbf{Distr}_{\mathsf{pol}}$ if there is a polynomial $p \in \mathbf{Pol}(\mathbb{Z})$, such that the probability

$$\Pr\left[\mathsf{T}_n = \perp \vee \mathsf{time}(\mathsf{T}_n) > p(n) \;:\; S_n \leftarrow D_n, \mathsf{T}_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!](S_n)\right]$$

is negligible.

Let $\mathbf{TerD}[\mathsf{P}] \subseteq \mathbf{Distr}_{\mathsf{pol}}$ denote the set of all distributions D of inputs, for which P runs in expected polynomial time.

For a distribution $D \in \mathbf{TerD}[\mathsf{P}]$, the distribution $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!](D)$ puts only negligible weight on $\perp$ or on final states where the running time is greater than polynomial. We can say here, that the program P transforms the distribution D into another

Let l be a new variable. Let $len \in \mathbf{Op}$ be such, that $[\![len]\!]_n()$ is the binary representation of some number that is polynomial in n. Let "0", "+1", "$\leq$" and "$\wedge$" have their usual meanings.
Construct a program Q as follows:

P	Q
skip	$l := l + 1$
$x := o(x_1, \ldots, x_k)$	$x := o(x_1, \ldots, x_k);\ l := l + 1$
$\mathsf{P}_1; \mathsf{P}_2$	$\mathsf{Q}_1; \mathsf{Q}_2$
$\mathit{if}\ b\ \mathit{then}\ \mathsf{P}_1\ \mathit{else}\ \mathsf{P}_2$	$\mathit{if}\ b\ \mathit{then}\ \mathsf{Q}_1\ \mathit{else}\ \mathsf{Q}_2$
$\mathit{while}\ b\ \mathit{do}\ \mathsf{P}'$	$\mathit{while}\ b\ \mathit{do}\ \big(\mathsf{Q}'; b := b \wedge (l \leq len())\big)$

The program P is transformed to

$$\begin{array}{l} l := 0; \\ \mathsf{Q}; \\ \mathit{if}\ l > len()\ \mathit{then} \\ \quad \left.\begin{array}{l} x_1 := 0; \\ \ldots\ldots. \\ x_m := 0 \end{array}\right\} \{x_1, \ldots, x_m\} = \mathbf{Var} \\ \mathit{else}\ \mathit{skip} \end{array}$$

Figure 3.2: Making the program P terminating

distribution over the program states. We denote this distribution by $\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)$ and define it by

$$\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D) = \{\!|\mathsf{state}(\mathsf{T}_n) \ :\ S_n \leftarrow D_n, \mathsf{T}_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!](S_n)|\!\}_{n\in\mathbb{N}} \ .$$

This distribution (or actually one that is indistinguishable from it) is polynomial-time constructible. Indeed, an algorithm that samples the distribution $\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)$ could work as follows: it first obtains a sample of D (the distribution D is polynomial-time constructible) and then runs the program P for at most $p(n)$ steps. Thus $\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!]$ is a function from $\mathbf{TerD}[\mathsf{P}]$ to $\mathbf{Distr}_{\mathsf{pol}}$.

3.2.2 Security Definitions

The "canonical" form of the definition of secure information flow is the following:

Definition 3.2. Let P be a program and let $D \in \mathbf{TerD}[\mathsf{P}]$. Then P with inputs distributed according to D has *secure information flow* if the distributions

$$\{\!|(S_n \mathbin{\$} \mathbf{Var}_{\mathrm{S}}, \mathsf{state}(\mathsf{T}_n) \mathbin{\$} \mathbf{Var}_{\mathrm{P}}) \ :\ S_n \leftarrow D_n, \mathsf{T}_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!](S_n)|\!\}_{n\in\mathbb{N}} \qquad (3.3)$$

and

$$\{\!|(S_n \mathbin{\$} \mathbf{Var}_{\mathrm{S}}, \mathsf{state}(\mathsf{T}'_n) \mathbin{\$} \mathbf{Var}_{\mathrm{P}}) \ :\ S_n, S'_n \leftarrow D_n, \mathsf{T}'_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!](S'_n)|\!\}_{n\in\mathbb{N}} \qquad (3.4)$$

Let P be a program and $\mathbf{Var}_{\mathrm{S}} \subseteq \mathbf{Var}$.
Define

- $\mathbf{Var}'_{\mathrm{S}} = \{x' : x \in \mathbf{Var}_{\mathrm{S}}\}$
- $\mathbf{Var}' = \mathbf{Var} \uplus \mathbf{Var}'_{\mathrm{S}}$
- P' is the program P, where each occurrence of a variable $x \in \mathbf{Var}_{\mathrm{S}}$ is replaced with x'.

Replace P with

$$\left.\begin{array}{l} x'_1 := x_1; \\ \dots\dots\dots \\ x'_m := x_m; \end{array}\right\} \{x_1, \dots, x_m\} = \mathbf{Var}_{\mathrm{S}}$$
$$\mathsf{P}'$$

Figure 3.3: Making the variables in $\mathbf{Var}_{\mathrm{S}}$ read-only

are indistinguishable.

Because of the program P running in expected polynomial time, there is only negligible probability that T_n or T'_n are equal to $\perp$. When we compare this definition with the definition of independence then we see that a program has secure information flow for a certain distribution of inputs, if the initial values of the variables in $\mathbf{Var}_{\mathrm{S}}$ and the final values of the variables in $\mathbf{Var}_{\mathrm{P}}$ are computationally independent. Thus the final values of the variables in $\mathbf{Var}_{\mathrm{P}}$ are useless for the attacker.

This definition becomes more compact if we assume that the program P does not change the values of the variables in $\mathbf{Var}_{\mathrm{S}}$. This may seem a significant constraint but in reality, it is not. We may introduce new variables and use them instead of the variables in $\mathbf{Var}_{\mathrm{S}}$. Fig. 3.3 shows, how to transform any program to a program that does not assign to the variables in $\mathbf{Var}_{\mathrm{S}}$.

Proposition 3.8. *Suppose that the program* P *does not assign to the variables in* $\mathbf{Var}_{\mathrm{S}}$. *Let* $D \in \mathbf{TerD}[\mathsf{P}]$. *Then* P *with inputs distributed according to* D *has secure information flow iff* $\mathbf{Var}_{\mathrm{S}}$ *and* $\mathbf{Var}_{\mathrm{P}}$ *are independent in the distribution* $\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)$.

Proof. Follows directly from the definitions of $\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!]$, independence, and secure information flow. ☐

Finding out whether $\mathbf{Var}_{\mathrm{S}}$ and $\mathbf{Var}_{\mathrm{P}}$ are independent or not in the distribution $\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)$ is the task of the analysis presented in the next chapter.

Secure information flow could also be characterised differently, through semantic security. Semantic security is a more direct description of the uselessness of the final values of variables in $\mathbf{Var}_{\mathrm{P}}$ for the attacker.

Definition 3.3. Program P, whose inputs are distributed accordingly to $D \in \mathbf{TerD}[\mathsf{P}]$, is *semantically secure*, if for all polynomial-time computable functions $f, h : \prod_{n\in\mathbb{N}}((\mathbf{Var}_\mathrm{S} \to \mathbf{Val}_n) \to \{0,1\}^*)$ and all PPT algorithms $\mathcal{A}$ there exists a PPT algorithm $\mathcal{B}$, such that the difference of

$$\Pr\big[\mathcal{A}\big(\mathbf{1}^n, \mathsf{state}(\mathsf{T}_n) \mathbin{\$} \mathbf{Var}_\mathrm{P}, h(S_n \mathbin{\$} \mathbf{Var}_\mathrm{S})\big) = f(S_n \mathbin{\$} \mathbf{Var}_\mathrm{S}) \; : \; S_n \leftarrow D_n, \mathsf{T}_n \leftarrow \mathbb{C}_\mathrm{len}[\![\mathsf{P}]\!](S_n)\big] \quad (3.5)$$

and

$$\Pr\big[\mathcal{B}\big(\mathbf{1}^n, h(S_n \mathbin{\$} \mathbf{Var}_\mathrm{S})\big) = f(S_n \mathbin{\$} \mathbf{Var}_\mathrm{S}) \; : \; S_n \leftarrow D_n\big] \quad (3.6)$$

is negligible in n.

This definition corresponds to the following scenario. The adversary already has some knowledge (the value of h) about the secret inputs of P and it wants to find out something more about them (the value of f). In its quest, the public outputs of the program must be of no use to it — if the adversary has a certain success when using the public outputs, then there must also exist a possibility to not use the public outputs and nevertheless have the same success rate.

Secure information flow and semantic security are related in the following way:

Proposition 3.9. *Let* P *be a program and let* $D \in \mathbf{TerD}[\mathsf{P}]$*. If* P *with inputs distributed according to* D *has secure information flow then it is semantically secure.*

Proof. Suppose that P with inputs distributed according to D is not semantically secure, i.e. there exist polynomial-time computable functions $f, h : \prod_{n\in\mathbb{N}}((\mathbf{Var}_\mathrm{S} \to \mathbf{Val}_n) \to \{0,1\}^*)$ and a PPT algorithm $\mathcal{A}$, such that there is no such algorithm $\mathcal{B}$, such that the distributions (3.5) and (3.6) are indistinguishable.

Also suppose that P with inputs distributed according to D has secure information flow, i.e. the distributions (3.3) and (3.4) are indistinguishable.

We are going to construct an algorithm $\mathcal{B}'$. The inputs of $\mathcal{B}'$ are

- the security parameter $\mathbf{1}^n$;
- a bit-string $x \in \{0,1\}^*$.

The algorithm $\mathcal{B}'(\mathbf{1}^n, x)$ does the following:

1. Generates a program state S'_n according to the distribution D_n.
2. Generates $\mathsf{T}'_n \in \mathbf{ST}_n$ according to $\mathbb{C}_\mathrm{len}[\![\mathsf{P}]\!](S'_n)$;
3. Calls the algorithm $\mathcal{A}$ with arguments $\mathbf{1}^n$, $\mathsf{state}(\mathsf{T}'_n) \mathbin{\$} \mathbf{Var}_\mathrm{P}$ and x;
4. Outputs the value returned by $\mathcal{A}$.

We also construct an algorithm $\mathcal{C}$. Its inputs are

- the security parameter $\mathbf{1}^n$;

- a sequence V_{S} of the elements of $\mathbf{Val}_n$, indexed by $\mathbf{Var}_{\mathrm{S}}$;
- a sequence V_{P} of the elements of $\mathbf{Val}_n$, indexed by $\mathbf{Var}_{\mathrm{P}}$.

The algorithm $\mathcal{C}(\mathbf{1}^n, V_{\mathrm{S}}, V_{\mathrm{P}})$ does the following:

1. Calls the algorithm $\mathcal{A}$ with arguments $\mathbf{1}^n$, V_{P} and $h(V_{\mathrm{S}})$.
2. If the value returned by $\mathcal{A}$ is equal to $f(V_{\mathrm{S}})$, then $\mathcal{C}$ outputs 1. Otherwise it outputs 2.

We now have

$$\begin{aligned}
&\Pr\big[\mathcal{B}'\big(\mathbf{1}^n, h(S_n \mathbin{\$} \mathbf{Var}_{\mathrm{S}})\big) = f(S_n \mathbin{\$} \mathbf{Var}_S) \,:\, S_n \leftarrow D_n\big] = \\
&\quad \Pr\big[\mathcal{A}\big(\mathbf{1}^n, \mathsf{state}(\mathsf{T}'_n) \mathbin{\$} \mathbf{Var}_{\mathrm{P}}, h(S_n \mathbin{\$} \mathbf{Var}_{\mathrm{S}})\big) = f(S_n \mathbin{\$} \mathbf{Var}_{\mathrm{S}}) \,: \\
&\qquad\qquad S_n, S'_n \leftarrow D_n, \mathsf{T}'_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!](S'_n)\big] = \\
&\quad \Pr\big[\mathcal{C}(\mathbf{1}^n, S_n \mathbin{\$} \mathbf{Var}_{\mathrm{S}}, \mathsf{state}(\mathsf{T}'_n) \mathbin{\$} \mathbf{Var}_{\mathrm{P}}) = 1 \,: \\
&\qquad\qquad S_n, S'_n \leftarrow D_n, \mathsf{T}'_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!](S'_n)\big] = \\
&\quad \Pr\big[\mathcal{C}(\mathbf{1}^n, S_n \mathbin{\$} \mathbf{Var}_{\mathrm{S}}, \mathsf{state}(\mathsf{T}_n) \mathbin{\$} \mathbf{Var}_{\mathrm{P}}) = 1 \,: \\
&\qquad\qquad S_n \leftarrow D_n, \mathsf{T}_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!](S_n)\big] + \alpha(n) = \\
&\quad \Pr\big[\mathcal{A}\big(\mathbf{1}^n, \mathsf{state}(\mathsf{T}_n) \mathbin{\$} \mathbf{Var}_{\mathrm{P}}, h(S_n \mathbin{\$} \mathbf{Var}_{\mathrm{S}})\big) = f(S_n \mathbin{\$} \mathbf{Var}_{\mathrm{S}}) \,: \\
&\qquad\qquad S_n \leftarrow D_n, \mathsf{T}_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!](S_n)\big] + \alpha(n),
\end{aligned}$$

where α is some negligible function. Therefore the algorithm $\mathcal{B}'$ is a suitable candidate for the algorithm $\mathcal{B}$ that supposedly did not exist. A contradiction. □

3.3 Discussion

Security definition. The security definition that we have given here is quite similar to one given for programs specified in some process algebra, for example the spi-calculus [AG99]. One defines a program to be secure if it is equivalent to some *obviously secure* program. There can be different ways to define equivalence and obvious security, these definitions depend on the problem domain.

In our case with a passive adversary, we define two programs $\mathsf{P}^{(1)}$ and $\mathsf{P}^{(2)}$ (having the same set $\mathbf{Var}$ of variables) with the distributions of inputs $D^{(1)}$ and $D^{(2)}$, respectively, to be equivalent iff the distributions

$$\{\!|(S_n \mathbin{\$} \mathbf{Var}_{\mathrm{S}}, \mathsf{state}(\mathsf{T}_n) \mathbin{\$} \mathbf{Var}_{\mathrm{P}}) \,:\, S_n \leftarrow D^{(1)}_n, \mathsf{T}_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathsf{P}^{(1)}]\!](S_n)|\!\}_{n\in\mathbb{N}}$$

and

$$\{\!|(S_n \mathbin{\$} \mathbf{Var}_{\mathrm{S}}, \mathsf{state}(\mathsf{T}_n) \mathbin{\$} \mathbf{Var}_{\mathrm{P}}) \,:\, S_n \leftarrow D^{(2)}_n, \mathsf{T}_n \leftarrow \mathbb{C}_{\mathrm{len}}[\![\mathsf{P}^{(2)}]\!](S_n)|\!\}_{n\in\mathbb{N}}$$

are indistinguishable. Obvious security is not really defined at all, it is given by listing the programs that we consider being obviously secure. In our case, we consider a program P obviously secure, if it does not read the initial values of its variables. Clearly, the output of such program does not depend on its input.

If a program is secure according to Def. 3.2, then there exists an obviously secure program that is equivalent to it. Namely, given a program P whose inputs are distributed according to $D \in \mathbf{Distr}_{\mathsf{pol}}$, consider the program P', that first generates new values for all of its variables (and thereby discarding their input values) by sampling the distribution D, and then works like the program P. From the definitions follows that P and P' are equivalent iff the two distributions in Def. 3.2 are indistinguishable.

Structures on Distr⊥. In this chapter we have considered two very different structures on the set $\mathbf{Distr}\bot$ (or on its subsets). Namely, we have used completeness of $\mathbf{Distr}\bot$ and least upper bounds on it to define the semantics $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$, and we have used indistinguishability of distributions over program states to define secure information flow. A natural question about the relationship of these two structures arises. As we show below, there is no meaningful relationship.

Let $D^{(1)}, D^{(2)} \in \mathbf{Distr}\bot$ be two families of distributions over program states. We now construct two sets of families of distributions $D^{(1,m)}, D^{(2,m)} \in \mathbf{Distr}\bot$, where $m \in \mathbb{N}$, such that

- if $m_1 \leq m_2$ then $D^{(i,m_1)} \leq D^{(i,m_2)}$ for each $i \in \{1,2\}$ and $m_1, m_2 \in \mathbb{N}$. Therefore the sets $\{D^{(1,m)} : m \in \mathbb{N}\}$ and $\{D^{(2,m)} : m \in \mathbb{N}\}$ are chains in $\mathbf{Distr}\bot$.
- $\bigvee\{D^{(i,m)} : m \in \mathbb{N}\} = D^{(i)}$ for each $i \in \mathbb{N}$.
- $D^{(1,m)} \approx D^{(2,m)}$ for all $m \in \mathbb{N}$. Even more, $D^{(i_1,m_1)} \approx D^{(i_2,m_2)}$ for all $i_1, i_2 \in \{1,2\}$ and $m_1, m_2 \in \mathbb{N}$.

We let $D_n^{(i,m)} \in \mathcal{D}(\mathbf{State}_{n\bot})$, where $i \in \{1,2\}$ and $m, n \in \mathbb{N}$ be the following distribution:

$$D_n^{(i,m)} = \begin{cases} D_n^{(i)}, & \text{if } n \leq m \\ \eta^{\mathcal{D}}(\bot), & \text{if } n > m \end{cases} .$$

It is obvious that the families of distributions $D^{(i,m)}$ are ordered as described and that their least upper bound is the distribution $D^{(i)}$, because order and least upper bounds are defined componentwise. The indistinguishability of families of distributions $D^{(i_1,m_1)}$ and $D^{(i_2,m_2)}$ follows from the usage of asymptotics in defining indistinguishability and from the equation $D_n^{(i_1,m_1)} = D_n^{(i_2,m_2)}$ for all large enough $n \in \mathbb{N}$ (namely, for all $n > \max(m_1, m_2)$).

Thus the (families of) distributions $D^{(i,m)}$ are all indistinguishable from each other and $D^{(1)}$, $D^{(2)}$ are their least upper bounds. But no requirements at all were put on $D^{(1)}$ and $D^{(2)}$. They could have been any two distributions.

Let A be a CPO. If f is a function from $\mathbf{Distr}\bot$ to A with the property that $f(D^{(1)}) = f(D^{(2)})$ for all families of distributions $D^{(1)}, D^{(2)} \in \mathbf{Distr}\bot$, where

$D^{(1)} \approx D^{(2)}$ (clearly, an abstraction function associated with an analysis for secure information flow, should have this property), then the function f cannot be continuous. Because of that, in the next chapter we cannot use existing fixed point approximation theorems (for example, [Cou00, Thm. 1, Thm. 2]) for proving our analysis correct. We have to give a rather *ad hoc* proof.

Chapter 4

Analysis

In this chapter we present the static program analysis for secure information flow. The analysis of a program P has a structure similar to the semantics of P, it is a function mapping *the abstraction of* the distribution of the inputs of P to *the abstraction of* the distribution of the outputs. The analysis is therefore also called an/the *abstract semantics* of the program. The key difference between concrete and abstract semantices is, that the abstract semantics is computable (the concrete semantics was only "samplable"). Given the program and the abstract input, there exists an algorithm that computes the abstract output.

In Sec. 4.1 we present the abstract domain $\mathcal{PF}(\mathbf{Var})$ (the abstract domain depends on the set of variables $\mathbf{Var}$) — the set of abstractions of distributions over program states. We define the *representation function* $\beta^{\mathbb{KI}}_{\mathbf{Var}}$ from $\mathbf{Distr}_{\mathsf{pol}}$ to $\mathcal{PF}(\mathbf{Var})$ that maps each distribution to its (best possible) abstraction.

Sec. 4.2 gives the abstract semantics itself. As it maps the abstraction of an input distribution to the abstraction of the output distribution, it does not explicitly relate the (secret) inputs and (public) outputs of the program. We use the analysis together with Proposition 3.8 to derive conclusions about their relationship. The usage of the analysis is spelled out precisely in Corollary 4.4.

The rest of this chapter is devoted to proving the abstract semantics correct. In Sec. 4.3 we state the correctness theorem and also sketch the proof idea in Sec. 4.3.1. In Sec. 4.3.2 we give the roadmap for the rest of this chapter (Secs. 4.4–4.7), containing the correctness proof itself.

4.1 Abstraction of Distributions

The abstraction maps each $D \in \mathbf{Distr}_{\mathsf{pol}}$ to a pair, where the first, "main" component describes, which (sets of) variables are independent of each other, and the second component records, the values of which variables are distributed (and hence may be used) as encryption keys.

4.1.1 Independence

The abstraction of $D \in \mathbf{Distr}_{\mathsf{pol}}$ records the independence of (sets of) variables. Recalling the general definition from Sec. 2.2.2, two sets of variables $X, Y \subseteq \mathbf{Var}$ are independent in the distribution D, if the distributions

$$\{\!|(S_n \$ X, S_n \$ Y) \,:\, S_n \leftarrow D_n|\!\}_{n\in\mathbb{N}} \tag{4.1}$$

and

$$\{\!|(S_n \$ X, S'_n \$ Y) \,:\, S_n, S'_n \leftarrow D_n|\!\}_{n\in\mathbb{N}} \tag{4.2}$$

are indistinguishable.

Only recording the independence of variables leaves us unable to discriminate between different "kinds of independence". For example: let k, l and x be variables and let D be a distribution, such that

- the value of k is distributed as a key;
- the value of x is a ciphertext, encrypted with the key k;
- the value of l has been obtained by the operation $l = k + 1$.

Then neither l nor x are independent of k in the distribution D. But the information that l conveys about k is much stronger than the information that x conveys. Someone knowing l may be able to decrypt ciphertexts created by k, but someone knowing only x certainly cannot. This is a difference that our abstraction should certainly reflect.

Consider the object called $[k]_{\mathcal{E}}$ that we use in the following way: given a state $S_n \in \mathbf{State}_n$, we define the *value* of $[k]_{\mathcal{E}}$ in the state S_n (similarly to the values of variables). The value of $[k]_{\mathcal{E}}$ in S_n (denoted $S_n([k]_{\mathcal{E}})$) is a *black box* that encrypts its inputs with the value of k in the state S_n. The notion of independence is also defined for such objects (see discussion after Def. 2.4).

Returning to our example, being independent from $[k]_{\mathcal{E}}$ distinguishes l and x — x is independent from $[k]_{\mathcal{E}}$ (at least when the encryption is which-key and repetition concealing) while l is not.

Let $\widetilde{\mathbf{Var}}$ denote the set $\mathbf{Var} \cup \{[k]_{\mathcal{E}} \,:\, k \in \mathbf{Var}\}$. If $S_n \in \mathbf{State}_n$, i.e. S_n is a function from $\mathbf{Var}$ to $\mathbf{Val}_n$, then we consider the domain of S_n to be $\widetilde{\mathbf{Var}}$ (and its range to be $\widetilde{\mathbf{Val}}_n := \mathbf{Val}_n \cup (\mathbf{Val}_n \rightsquigarrow \mathbf{Val}_n)$ — $S_n([k]_{\mathcal{E}})$ is a probabilistic function from $\mathbf{Val}_n$ to $\mathbf{Val}_n$).

The abstraction $\beta^{\mathbb{I}}_{\mathbf{Var}}(D)$ is equal to the set of all pairs (X, Y), where $X, Y \subseteq \widetilde{\mathbf{Var}}$, such that X is independent from Y in the distribution D. Let $\mathcal{F}(\mathbf{Var})$ denote the range of $\beta^{\mathbb{I}}_{\mathbf{Var}}$, then $\mathcal{F}(\mathbf{Var})$ is a subset of $\mathcal{P}(\mathcal{P}(\widetilde{\mathbf{Var}}) \times \mathcal{P}(\widetilde{\mathbf{Var}}))$. The following trivially holds for each element $\mathcal{X}$ of $\mathcal{F}(\mathbf{Var})$:

- It is symmetric: if $(X, Y) \in \mathcal{X}$ then also $(Y, X) \in \mathcal{X}$, because of the symmetry in the definition of independence.

- It is monotone: if $(X, Y) \in \mathcal{X}$ and $X' \subseteq X$ and $Y' \subseteq Y$ then $(X', Y') \in \mathcal{X}$. Indeed, if a PPT algorithm can show the dependence between X' and Y' in a distribution D, then the same algorithm also shows the dependence between X and Y.
- $k \in \mathbf{Var}$ contains all the information that $[k]_{\mathcal{E}}$ contains: if $(\{k\} \cup X, Y) \in \mathcal{X}$ then also $(\{k, [k]_{\mathcal{E}}\} \cup X, Y) \in \mathcal{X}$, because the value of $[k]_{\mathcal{E}}$ can be computed from the value of k.

We have made the set of variables **Var** explicit in the notation (we have defined $\widetilde{\mathbf{Var}}$, $\beta^{\mathbb{I}}_{\mathbf{Var}}$ and $\mathcal{F}(\mathbf{Var})$). The reason for this is, that in defining the abstract semantics for a program we may have to consider the abstract semantices of its subprograms with respect to different sets of variables (see the def. of abstr. sem. for branches) and it is not always implicitly clear, what the assumed underlying set of variables is.

4.1.2 Keys

Our analysis treats encryption in a special way. The abstract semantics for a statement $x := \mathcal{E}nc(k, y)$ is different, more optimistic than the abstract semantics of a generic statement $x := o(x_1, \ldots, x_m)$. The correctness of this special treatment stems from the security definition for the encryption system. But this definition (and therefore also the special treatment) is applicable only if the key used for encrypting is generated by the corresponding key generation algorithm. Our abstraction thus has to record, which variables are distributed as keys.

The abstraction $\beta^{\mathbb{K}}_{\mathbf{Var}}(D)$, where $D \in \mathbf{Distr}_{\mathsf{pol}}$, is a subset of **Var**, consisting of all variables k, such that

$$\{\!|S_n([k]_{\mathcal{E}}) \,:\, S_n \leftarrow D_n|\!\}_{n\in\mathbb{N}} \approx \{\!|[\![\mathcal{E}nc]\!]_n(k', \cdot) \,:\, k' \leftarrow [\![\mathcal{G}en]\!]_n|\!\}_{n\in\mathbb{N}} \qquad (4.3)$$

holds. Here in the left hand side we have the distribution of the encrypting black box $[k]_{\mathcal{E}}$, whereas on the right hand side we have an encrypting black box that encrypts with a real key.

The whole abstraction $\beta^{\mathbb{K}\mathbb{I}}_{\mathbf{Var}}$ is the pair of abstractions presented so far, $\beta^{\mathbb{K}\mathbb{I}}_{\mathbf{Var}}(D) = (\beta^{\mathbb{K}}_{\mathbf{Var}}(D), \beta^{\mathbb{I}}_{\mathbf{Var}}(D))$. The range of this abstraction is $\mathcal{P}(\mathbf{Var}) \times \mathcal{F}(\mathbf{Var})$, we denote it by $\mathcal{PF}(\mathbf{Var})$. Let $\mathsf{keys} : \mathcal{PF}(\mathbf{Var}) \to \mathcal{P}(\mathbf{Var})$ and $\mathsf{indeps} : \mathcal{PF}(\mathbf{Var}) \to \mathcal{F}(\mathbf{Var})$ be the projections onto the first and second component of $\mathcal{PF}(\mathbf{Var})$, respectively.

4.1.3 Discussion

Here we want give an intuitive explanation, what the (in)dependence of a black box $[k]_{\mathcal{E}}$ from a set $X \subseteq \widetilde{\mathbf{Var}}$ means, versus the (in)dependence of the key k from a set $X \subseteq \widetilde{\mathbf{Var}}$.

Given a key k and a set $X \subseteq \widetilde{\mathbf{Var}}$, there are three different possibilities for their independence:

- X is independent of $\{k\}$. In this case X contains no information about the key k. This is the usual independence of two sets of variables.
- X is not independent of $\{k\}$, but is independent of $\{[k]_{\mathcal{E}}\}$. In this case the values of variables and black boxes in X may contain ciphertexts that have been created with the key k. However, X still contains so little information about k, that it is of no use in *decrypting* ciphertexts created with the key k. Here "decrypting" means not only completely, but also partially revealing the corresponding plaintext (similarly to semantic security).
- X is not independent of $\{[k]_{\mathcal{E}}\}$. Then the values of variables and black boxes in X may help in decrypting ciphertexts created with key k.

Another thing worth mentioning is that $(\{x\},\{x\}) \in \beta^{\mathbb{I}}_{\mathbf{Var}}(D)$ is possible also for variables $x \in \mathbf{Var}$ (for black boxes, this is the definition of which-key concealedness). This situation occurs if x has a constant value in D — the probability

$$\Pr\big[S_n(x) \neq S'_n(x) \,:\, S_n, S'_n \leftarrow D_n\big] \tag{4.4}$$

is negligible in n.

We have shown that $\beta^{\mathbb{I}}_{\mathbf{Var}}(D)$ has certain closedness properties — namely symmetry and downwards closedness. There is also another interesting closedness property that may help to understand the abstract semantics.

Definition 4.1. Let $\mathcal{X} \in \mathcal{F}(\mathbf{Var})$. We say that $\mathcal{X}$ is *complete*, if for each $X, Y, Z \subseteq \widetilde{\mathbf{Var}}$ the following implication holds: if

$$(X, Y) \in \mathcal{X} \tag{4.5}$$

and

$$(X \cup Y, Z) \in \mathcal{X}, \tag{4.6}$$

then also

$$(X, Y \cup Z) \in \mathcal{X} \ . \tag{4.7}$$

Lemma 4.1. *Let $D \in \mathbf{Distr}_{\mathsf{pol}}$. Then $\beta^{\mathbb{KI}}_{\mathbf{Var}}(D)$ is complete.*

Proof. Let $X, Y, Z \in \widetilde{\mathbf{Var}}$. Then

$$\begin{aligned}
&\{\!|(S_n \$ X, S_n \$ Y, S_n \$ Z) \,:\, S_n \leftarrow D_n|\!\}_{n\in\mathbb{N}} \approx^{(4.6)}\\
&\qquad \{\!|S_n \$ X, S_n \$ Y, S'_n \$ Z) \,:\, S_n, S'_n \leftarrow D_n|\!\}_{n\in\mathbb{N}} \approx^{(4.5)}\\
&\qquad \{\!|S_n \$ X, S''_n \$ Y, S'_n \$ Z) \,:\, S_n, S'_n, S''_n \leftarrow D_n|\!\}_{n\in\mathbb{N}} \approx^{(4.6)}\\
&\qquad\qquad \{\!|S_n \$ X, S'_n \$ Y, S'_n \$ Z) \,:\, S_n, S'_n \leftarrow D_n|\!\}_{n\in\mathbb{N}},
\end{aligned}$$

which shows that (4.7) holds for $\beta^{\mathbb{I}}_{\mathbf{Var}}(D)$. $\square$

From the proof of this lemma we also see that (4.5) and (4.6) imply

$$\begin{aligned}
&\{\!|(S_n \$ X, S_n \$ Y, S_n \$ Z) \,:\, S_n \leftarrow D_n|\!\}_{n\in\mathbb{N}} \approx\\
&\qquad\qquad \{\!|S_n \$ X, S'_n \$ Y, S''_n \$ Z) \,:\, S_n, S'_n, S''_n \leftarrow D_n|\!\}_{n\in\mathbb{N}},
\end{aligned}$$

i.e. the values of variables in X, the values of variables in Y and the values of variables in Z are in some sense "three-way" independent in distribution D.

$$\begin{tikzcd}
\mathbf{TerD}[\mathsf{P}] \arrow[rr, "\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!]"] \arrow[d, "\beta^{\mathbb{KI}}_{\mathbf{Var}}"] & & \mathbf{Distr}_{\mathsf{pol}} \arrow[d, "\beta^{\mathbb{KI}}_{\mathbf{Var}}"] \\
\mathcal{PF}(\mathbf{Var}) \arrow[r, "A"] & \mathcal{PF}(\mathbf{Var}) \arrow[r, dash, dotted, "\le" description] & \mathcal{PF}(\mathbf{Var})
\end{tikzcd}$$

Figure 4.1: Abstracting the concrete semantics

4.2 Abstract Semantics

The abstract semantics $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ for a program P is a function whose domain and range are both $\mathcal{PF}(\mathbf{Var})$, where the set of variables $\mathbf{Var}$ must contain at least all the variables that occur in P.

Let us give a precise description, what the *correctness* of the abstract semantics means.

Definition 4.2. Let P be a program with variables from the set $\mathbf{Var}$. Let A be a function with domain and range both equal to $\mathcal{PF}(\mathbf{Var})$. We say that A *abstracts* the concrete semantics of P, if for each $D \in \mathbf{TerD}[\mathsf{P}]$, the inequality

$$\beta^{\mathbb{KI}}_{\mathbf{Var}} \circ \mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!] \,\$\, D \geq A \circ \beta^{\mathbb{KI}}_{\mathbf{Var}} \,\$\, D \tag{4.8}$$

holds.

Fig. 4.1 illustrates this definition.

The definition of $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ is given inductively over the syntax of P.

4.2.1 Assignments

Let $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$ and let program P be the statement $x := o(x_1, \ldots, x_k)$. We define $\mathcal{Y} = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\mathcal{X})$ as the smallest element of $\mathcal{PF}(\mathbf{Var})$, such that

- $\mathcal{Y}$ satisfies the rules in Fig. 4.2 and Fig. 4.3;
- $\mathsf{indeps}(\mathcal{Y})$ is symmetric and downwards closed.

We do not demand $\mathcal{Y}$ to be complete. Demanding the completeness would complicate the correctness proof. However, it turns out (although we are not going to prove it here) that it is unnecessary to demand it at all — if $\mathcal{X}$ is complete then $\mathcal{Y}$ is complete, too.

Let us give an explanation for the rules in Figures 4.2 and 4.3.

Rule (4.9): The program P changes only the value of the variable x and therefore also the black box $[x]_{\mathcal{E}}$. It does not change the values of any other variables or black boxes. In rule (4.9), the sets $X, Y \subseteq \widetilde{\mathbf{Var}}$ only contain variables and black boxes whose values are not changed by the program P; the values of variables and

$$\frac{\begin{array}{c}\mathsf{P} \text{ is } x := o(x_1,\ldots,x_k)\\ (X,Y) \in \mathsf{indeps}(\mathcal{X})\\ x,[x]_{\mathcal{E}} \notin X \cup Y\end{array}}{(X,Y) \in \mathsf{indeps}(\mathcal{Y})} \tag{4.9}$$

$$\frac{\begin{array}{c}\mathsf{P} \text{ is } x := o(x_1,\ldots,x_k)\\ (X,Y) \in \mathsf{indeps}(\mathcal{X})\\ x,[x]_{\mathcal{E}} \notin X \cup Y\\ x_1,\ldots,x_k \in Y\end{array}}{(X, Y \cup \{x,[x]_{\mathcal{E}}\}) \in \mathsf{indeps}(\mathcal{Y})} \tag{4.10}$$

$$\frac{\begin{array}{c}\mathsf{P} \text{ is } x := \mathcal{E}nc(k,y)\\ (X,Y) \in \mathsf{indeps}(\mathcal{X})\\ x,[x]_{\mathcal{E}} \notin X \cup Y\\ y,[k]_{\mathcal{E}} \in Y\end{array}}{(X, Y \cup \{x,[x]_{\mathcal{E}}\}) \in \mathsf{indeps}(\mathcal{Y})} \tag{4.10^1}$$

$$\frac{\begin{array}{c}\mathsf{P} \text{ is } x := y\\ (X,Y) \in \mathsf{indeps}(\mathcal{X})\\ x,[x]_{\mathcal{E}} \notin X \cup Y\\ X' := X \cup \langle\!\langle y \in X \ \mathbf{?}\ \{x\} \ \mathbf{:}\ \emptyset\rangle\!\rangle \cup \langle\!\langle [y]_{\mathcal{E}} \in X \ \mathbf{?}\ \{[x]_{\mathcal{E}}\} \ \mathbf{:}\ \emptyset\rangle\!\rangle\\ Y' := Y \cup \langle\!\langle y \in Y \ \mathbf{?}\ \{x\} \ \mathbf{:}\ \emptyset\rangle\!\rangle \cup \langle\!\langle [y]_{\mathcal{E}} \in Y \ \mathbf{?}\ \{[x]_{\mathcal{E}}\} \ \mathbf{:}\ \emptyset\rangle\!\rangle\end{array}}{(X',Y') \in \mathsf{indeps}(\mathcal{Y})} \tag{4.10^2}$$

$$\frac{\begin{array}{c}\mathsf{P} \text{ is } x := \mathcal{E}nc(k,y)\\ (X,Y) \in \mathsf{indeps}(\mathcal{X})\\ x,[x]_{\mathcal{E}} \notin X \cup Y\\ k \in \mathsf{keys}(\mathcal{X})\\ (\{[k]_{\mathcal{E}}\}, X \cup (Y \backslash \{[k]_{\mathcal{E}}\}) \cup \{y\}) \in \mathsf{indeps}(\mathcal{X})\end{array}}{(X, Y \cup \{x,[x]_{\mathcal{E}}\}) \in \mathsf{indeps}(\mathcal{Y})} \tag{4.11}$$

$$\frac{\begin{array}{c}\mathsf{P} \text{ is } x := \mathcal{G}en()\\ (X,Y) \in \mathsf{indeps}(\mathcal{X})\\ x \notin X \cup Y\end{array}}{(X \cup \{[x]_{\mathcal{E}}\}, Y \cup \{[x]_{\mathcal{E}}\}) \in \mathsf{indeps}(\mathcal{Y})} \tag{4.12}$$

$$\frac{\begin{array}{c}\mathsf{P} \text{ is } x := o()\\ (X,Y) \in \mathsf{indeps}(\mathcal{X})\\ x,[x]_{\mathcal{E}} \notin X \cup Y\\ [\![o]\!] \text{ is deterministic}\end{array}}{(X \cup \{x,[x]_{\mathcal{E}}\}, Y \cup \{x,[x]_{\mathcal{E}}\}) \in \mathsf{indeps}(\mathcal{Y})} \tag{4.13}$$

Figure 4.2: The semantics $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ for assignments; indeps-part

$$\frac{\begin{array}{c}\mathsf{P} \text{ is } x := o(\ldots)\\ k \in \mathsf{keys}(\mathcal{X})\backslash\{x\}\end{array}}{k \in \mathsf{keys}(\mathcal{Y})} \tag{4.14}$$

$$\frac{\begin{array}{c}\mathsf{P} \text{ is } x := y\\ y \in \mathsf{keys}(\mathcal{X})\end{array}}{x \in \mathsf{keys}(\mathcal{Y})} \tag{4.14^1}$$

$$\frac{\mathsf{P} \text{ is } x := \mathcal{G}en()}{x \in \mathsf{keys}(\mathcal{Y})} \tag{4.15}$$

Figure 4.3: The semantics $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ for assignments; keys-part

black boxes in those sets are equal before and after the program. Therefore, if the variables and black boxes in the set X are independent from the variables and black boxes in the set Y before the program P, then the variables and black boxes in the set X are also independent from the variables and black boxes in the set Y after the program P.

We see that all rules in Fig. 4.2 have $x, [x]_{\mathcal{E}} \notin X \cup Y$ as one of their antecedents. This is not strictly necessary for all the rules (although it is necessary for rule (4.9)). It is included in the rules to make them more orthogonal — as x and $[x]_{\mathcal{E}}$ are overwritten by the assignment, their independence from other variables does not influence the independence of variables and black boxes after the program P.

Note that the independence of X and Y before the program is a necessary condition for the independence of X and Y after the program, if neither X nor Y contains x or $[x]_{\mathcal{E}}$. This follows again from the fact that the program P does not change the values of variables and black boxes in X and Y.

Rule (4.10): Compared to rule (4.9), the additional antecedent of rule (4.10) states that someone who knows the values of variables and black boxes in Y has all the necessary information to additionally compute the value of the variable x (and further, of the black box $[x]_{\mathcal{E}}$) after executing the program P. Moreover, this computation can be done in polynomial time. The independence of X and $Y \cup \{x, [x]_{\mathcal{E}}\}$ now follows from Lemma 2.10. In this lemma we instantiate the distributions D and D' with (4.1) and (4.2). The probabilistic function f in Lemma 2.10 takes the values of the variables $x_1, \ldots, x_k$ and returns these values and (a sample of) the value of x after executing the program P.

Rule (4.10^1): This is a special case of rule (4.10). According to the last antecedent of this rule, someone who knows the values of variables and black boxes in Y still has all the necessary information to additionally compute the value of the variable x. A black box encrypting with the value of k is enough for doing the encryption.

Rule (4.10^2): This is again a special case of rule (4.10). It says that if the program P is $x := y$, then the value of x after the program can be used instead of the value of y before the program. Also, the values of x and y together after the program are no more useful than the value of y before the program.

Rule (4.11): Here we make use of the repetition-concealedness of the encryption. So, what do we need for the independence of X and $Y \cup \{x\}$ after the program P?

Certainly, a necessary condition is the independence of X and Y before the program P. Indeed, from the downwards closedness of $\mathsf{indeps}(\mathcal{Y})$ follows that X and Y also must be independent after the program P. As we explained before, this is a necessary condition for the independence of X and Y before the program P.

If we want to make use of the cryptographic properties of encryption, then the variable k must be a key. Also, it must be rather independent of the variables in X and Y, as well as the variable y. Namely, it must be impossible to decrypt the value of x, as created by P. As explained in Sec. 4.1.3, the last antecedent of rule (4.11) is enough to ensure this.

Rule (4.12): This rule is a rather direct corollary of the which-key concealedness of the encryption. When we compare the definition of which-key concealedness, stated through indistinguishability of certain pairs of black boxes (2.10) with the definition of independence (2.4), then we see that an encryption system is which-key concealing, if an encrypting black box is independent of itself. Rule (4.12) states exactly this.

Rule (4.13): This rule is similar to rule (4.12). Namely, the result of the operation o is always the same value, no matter how many times we do that operation. Therefore the result of the operation o is independent of itself.

Rule (4.14): If an assignment does not change a variable, then the distribution of the value of this variable (and also the distribution of the black box encrypting with the value of this variable) after the assignment is the same as this distribution before the assignment. Therefore, if the value of the variable k was distributed as a key before the assignment, and k was not assigned to, then the value of k is distributed as a key also after the assignment.

Rule (4.14^1): This rule is similar to the previous rule. The value of x after the assignment $x := y$ is distributed identically to the value of y before the assignment.

Rule (4.15): The key generation operation creates keys.

4.2.2 Control Flow

Here we define $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ for all other constructs.

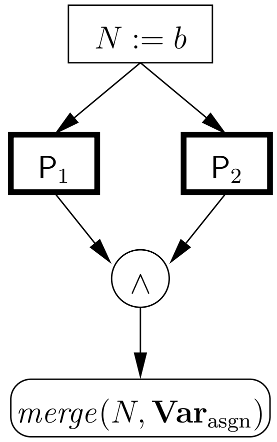


Figure 4.4: Scheme of defining $\mathbb{A}^{(\mathbf{Var})}[\![\textit{if } b \textit{ then } \mathsf{P}_1 \textit{ else } \mathsf{P}_2]\!]$

Monoid operations

It is easy to see how $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ should look like, if P is constructed by one of the "monoid operations", i.e. if P is either *skip* or $\mathsf{P}_1;\mathsf{P}_2$ for some programs P_1, P_2. We define

$$\begin{aligned}\mathbb{A}^{(\mathbf{Var})}[\![\textit{skip}]\!] &= \mathbf{id}_{\mathcal{PF}(\mathbf{Var})}\\ \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}_1;\mathsf{P}_2]\!] &= \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}_2]\!] \circ \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}_1]\!],\end{aligned}$$

where $\mathbf{id}_{\mathcal{PF}(\mathbf{Var})}$ denotes the identity function on $\mathcal{PF}(\mathbf{Var})$.

if–then–else

Next we consider the case where P is *if* b *then* P_1 *else* P_2. Let $\mathbf{Var}_{\mathrm{asgn}} \subseteq \mathbf{Var}$ be the set of variables that are at the left hand side of some assignment in at least on of the programs P_1 and P_2, let N be a variable that is not an element of $\mathbf{Var}$ and let $\mathbf{Var}' = \mathbf{Var} \uplus \{N\}$. The shape of the definition of $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ is shown on Fig. 4.4.

To compute $\mathcal{Y} = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\mathcal{X})$, where $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$, we first compute

$$\begin{aligned}\mathcal{Z}_1 &= \mathbb{A}^{(\mathbf{Var}')}[\![N := b;\mathsf{P}_1]\!](\mathcal{X})\\ \mathcal{Z}_2 &= \mathbb{A}^{(\mathbf{Var}')}[\![N := b;\mathsf{P}_2]\!](\mathcal{X})\ .\end{aligned} \tag{4.16}$$

It should be clear that prior to computing the abstract semantices of P_1 and P_2 we have to "save" the value of b because the operation of merging $\mathcal{Z}_1$ and $\mathcal{Z}_2$ back again must take into account, what the variable b was independent of at the start of the program P. Without saving b this information may get lost as b may get overwritten.

We then compute $\mathcal{Z} = \mathcal{Z}_1 \wedge \mathcal{Z}_2$ (the operation $\wedge$ on $\mathcal{PF}(\mathbf{Var}) = \mathcal{P}(\mathbf{Var}) \times \mathcal{F}(\mathbf{Var})$ is defined componentwise, on both components, the operation $\wedge$ is set intersection). Last, $\mathcal{Y}$ is defined as the smallest element of $\mathcal{PF}(\mathbf{Var})$, such that

- $\mathcal{Y}$ satisfies the rules in Fig. 4.5; square brackets are used for grouping similar antecedents on this figure;
- $\mathcal{Y}$ is symmetric and downwards closed.

In Fig. 4.5 (and in the rest of this thesis), $\widetilde{\mathbf{Var}}_{\mathrm{asgn}}$ denotes the set $\mathbf{Var}_{\mathrm{asgn}} \cup \{[k]_{\mathcal{E}} : k \in \mathbf{Var}_{\mathrm{asgn}}\}$.

Let us explain the rules in Fig. 4.5.

Rule (4.17): In this rule, the only entities that may have been changed in one of the branches are the encrypting black boxes $[x_{i_1}]_{\mathcal{E}}, \ldots, [x_{i_m}]_{\mathcal{E}}$. At the end of the branches P_1 and P_2 (i.e. before merging them together; while giving the intuitive explanation of the rules in Fig. 4.5 we assume that both branches have been run), they are "genuine" encrypting black boxes — the variables $x_{i_1}, \ldots, x_{i_m}$ are all distributed as keys.

These black boxes are independent of everything else (as given by the antecedent right above the first group in square brackets) and also of each other (this is given by the first group of antecedents in square brackets). Indeed, from the first group of antecedents in square brackets follows through Lemma 4.1, that if K_1 and K_2 are two subsets of $\{[x_{i_1}]_{\mathcal{E}}, \ldots, [x_{i_m}]_{\mathcal{E}}\}$, such that $K_1 \cap K_2 = \emptyset$, then K_1 and K_2 are independent of each other.

Therefore it is impossible to find out, whether these black boxes came from the branch P_1 or the branch P_2 — this follows from the which-key concealedness of the encryption primitive.

If a black box is independent of itself in both branches (these are the boxes $[x_{i_{l+1}}]_{\mathcal{E}}, \ldots, [x_{i_m}]_{\mathcal{E}}$, then this black box is also independent of itself after merging.

Rule (4.18): This is a more powerful variant of the previous rule. With respect to the variables x (with subscripts), it is very similar to the previous rule.

If $s = 0$, then the two antecedents above the group in square brackets are a slightly more conservative version of (this follows from Lemma 4.1)

- The antecedent of rule (4.17) above the two groups in square brackets;
- The antecedents in the second group of square brackets in rule (4.17).

Also, the group of antecedents in square brackets in rule (4.18) is exactly the first group of antecedents in square brackets in rule (4.17). This covers the whole rule (4.17).

Suppose now, that $m = 0$. Then the only remaining antecedent containing $\mathcal{Z}$ is the one broken over three lines. Someone that knows the values of variables and black boxes in $Y \cup \{\ldots\}$ (the second element of the pair in the antecedent broken over three lines) at the end of one of the branches (before merging) *may be* able to compute the values of variables and black boxes in $Y \cup \{\ldots\}$ (the second element of the pair in the consequent) after the merge. Indeed, it may be the case that exactly

$$\frac{\begin{array}{c}(X,Y)\in\mathsf{indeps}(\mathcal{Z})\\ x_{i_1},\dots,x_{i_l},x_{i_{l+1}},\dots,x_{i_m}\in\mathbf{Var}_{\text{asgn}}\\ (\{[x_{i_1}]\varepsilon,\dots,[x_{i_m}]\varepsilon\},X\cup Y\cup\{N\})\in\mathsf{indeps}(\mathcal{Z})\\ \left[\begin{array}{c}(\{[x_{i_1}]\varepsilon\},\{[x_{i_2}]\varepsilon,\dots,[x_{i_m}]\varepsilon\})\in\mathsf{indeps}(\mathcal{Z})\\ (\{[x_{i_2}]\varepsilon\},\{[x_{i_3}]\varepsilon,\dots,[x_{i_m}]\varepsilon\})\in\mathsf{indeps}(\mathcal{Z})\\ \cdots\cdots\cdots\cdots\cdots\cdots\\ (\{[x_{i_{m-1}}]\varepsilon\},\{[x_{i_m}]\varepsilon\})\in\mathsf{indeps}(\mathcal{Z})\end{array}\right]\\ \left[\begin{array}{c}(\{[x_{i_{l+1}}]\varepsilon\},\{[x_{i_{l+1}}]\varepsilon\})\in\mathsf{indeps}(\mathcal{Z})\\ \cdots\cdots\cdots\cdots\cdots\\ (\{[x_{i_m}]\varepsilon\},\{[x_{i_m}]\varepsilon\})\in\mathsf{indeps}(\mathcal{Z})\end{array}\right]\\ x_{i_1},\dots,\widetilde{x_{i_m}}\in\mathsf{keys}(\mathcal{Z})\\ (X\cup Y)\cap(\mathbf{Var}_{\text{asgn}}\cup\{N\})=\emptyset\end{array}}{(X\cup\{[x_{i_1}]\varepsilon,\dots,[x_{i_m}]\varepsilon\},Y\cup\{[x_{i_{l+1}}]\varepsilon,\dots,[x_{i_m}]\varepsilon\})\in\mathsf{indeps}(\mathcal{Y})} \tag{4.17}$$

$$\frac{\begin{array}{c}x_{i_1},\dots,x_{i_l},x_{i_{l+1}},\dots,x_{i_m},y_{j_1},\dots,y_{j_r},y_{j_{r+1}},\dots,y_{j_s}\in\mathbf{Var}_{\text{asgn}}\\ \big(X\cup\{[x_{i_1}]\varepsilon,\dots,[x_{i_m}]\varepsilon\},\\ Y\cup\{N,[x_{i_{l+1}}]\varepsilon,\dots,[x_{i_m}]\varepsilon,y_{j_1},[y_{j_1}]\varepsilon,\dots,y_{j_r},[y_{j_r}]\varepsilon,[y_{j_{r+1}}]\varepsilon,\dots,[y_{j_s}]\varepsilon\}\big)\\ \in\mathsf{indeps}(\mathcal{Z})\\ (X,\{[x_{i_1}]\varepsilon,\dots,[x_{i_m}]\varepsilon\})\in\mathsf{indeps}(\mathcal{Z})\\ \left[\begin{array}{c}(\{[x_{i_1}]\varepsilon\},\{[x_{i_2}]\varepsilon,\dots,[x_{i_m}]\varepsilon\})\in\mathsf{indeps}(\mathcal{Z})\\ (\{[x_{i_2}]\varepsilon\},\{[x_{i_3}]\varepsilon,\dots,[x_{i_m}]\varepsilon\})\in\mathsf{indeps}(\mathcal{Z})\\ \cdots\cdots\cdots\cdots\cdots\cdots\\ (\{[x_{i_{m-1}}]\varepsilon\},\{[x_{i_m}]\varepsilon\})\in\mathsf{indeps}(\mathcal{Z})\end{array}\right]\\ x_{i_1},\dots,\widetilde{x_{i_m}}\in\mathsf{keys}(\mathcal{Z})\\ (X\cup Y)\cap(\mathbf{Var}_{\text{asgn}}\cup\{N\})=\emptyset\end{array}}{\begin{array}{c}\big(X\cup\{[x_{i_1}]\varepsilon,\dots,[x_{i_m}]\varepsilon\},\\ Y\cup\{[x_{i_{l+1}}]\varepsilon,\dots,[x_{i_m}]\varepsilon,y_{j_1},[y_{j_1}]\varepsilon,\dots,y_{j_r},[y_{j_r}]\varepsilon,[y_{j_{r+1}}]\varepsilon,\dots,[y_{j_s}]\varepsilon\}\big)\\ \in\mathsf{indeps}(\mathcal{Y})\end{array}} \tag{4.18}$$

$$\frac{\begin{array}{c}w\in\mathsf{keys}(\mathcal{Z})\\ w\notin\mathbf{Var}_{\text{asgn}}\end{array}}{w\in\mathsf{keys}(\mathcal{Y})} \tag{4.19}$$

$$\frac{\begin{array}{c}x_i\in\mathbf{Var}_{\text{asgn}}\\ x_i\in\mathsf{keys}(\mathcal{Z})\\ (\{[x_i]\varepsilon\},\{N\})\in\mathsf{indeps}(\mathcal{Z})\end{array}}{x_i\in\mathsf{keys}(\mathcal{Y})} \tag{4.20}$$

Figure 4.5: Definition of $merge(N,\mathbf{Var}_{\text{asgn}})(\mathcal{Z})$

this branch has been taken. For at least one of the branches the probability that the values of variables and black boxes in $Y \cup \{\ldots\}$ (the second element of the pair in the consequent) can be computed, is at least $\frac{1}{2}$ (i.e. non-negligible).

Also, someone that knows the values of variables and black boxes in $Y \cup \{\ldots\}$ (the second element of the pair in the antecedent broken over three lines) at the end of one of the branches, *knows, whether he is* able to compute the values of variables and black boxes in $Y \cup \{\ldots\}$ (the second element of the pair in the consequent) after the merge.

Therefore one can often compute the values of variables and black boxes in $Y \cup \{\ldots\}$ (the second element of the pair in the consequent) after the merge. The results of this computation will never be wrong. This is enough for the consequent of rule (4.18) to hold.

If both $m > 0$ and $s > 0$, then the variables x (with subscripts) are still independent of everything else, including the variables y (with subscripts). This allows us to put these two cases together.

Rule (4.19): This is just a variant of rule (4.14). The value of the variable w has not been changed in any of the branches P_1 and P_2, therefore $w \in \mathsf{keys}(\mathcal{Z})$ means that the value of w already was distributed as a key before the *if*-statement. As it has not been changed, it also is distributed as a key after the entire *if*-statement.

Rule (4.20): If x_i is distributed according to the same distribution in both branches of the *if*-statement (in rule (4.20), it is distributed as a key in both branches), then it is not necessarily distributed according to that distribution after that *if*-statement, as its value may have influenced, which of the branches was taken. However, if the distribution of x_i is same in both branches and is independent of the choice of the branches, then it is still distributed in the same way after the *if*-statement.

Loops

Let P be *while b do* P'. The abstract semantics of P is defined similarly to its concrete semantics given in Sec. 3.1.3. Again, $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ is defined as a solution to the equation

$$\mathbb{A}^{(\mathbf{Var})}[\![\mathit{while}\ b\ \mathit{do}\ \mathsf{P}']\!] = \mathbb{A}^{(\mathbf{Var})}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}'\ \mathit{else}\ \mathit{skip}; \mathit{while}\ b\ \mathit{do}\ \mathsf{P}']\!] \ .$$

We first define a function $G_{\mathsf{P}'}$, and then define the semantics $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ as a certain fixed point of $G_{\mathsf{P}'}$. We have

$$G_{\mathsf{P}'} : (\mathcal{PF}(\mathbf{Var}) \to \mathcal{PF}(\mathbf{Var})) \to (\mathcal{PF}(\mathbf{Var}) \to \mathcal{PF}(\mathbf{Var}))$$

$$G_{\mathsf{P}'}(f) := f \circ \mathbb{A}^{(\mathbf{Var})}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}'\ \mathit{else}\ \mathit{skip}]\!] \tag{4.21}$$

$$\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!] = \mathit{gfp}^{\mathbf{id}_{\mathcal{PF}(\mathbf{Var})}}\ G_{\mathsf{P}'} \ . \tag{4.22}$$

Here we have *gfp* instead of *lfp* because the order on $\mathcal{PF}(\mathbf{Var})$ is different (if we had taken $\mathcal{R}(\mathcal{PF}(\mathbf{Var}))$ as our domain, then we should have taken the least fixed point). We also have a different starting point of the iteration — namely $\mathbf{id}_{\mathcal{PF}(\mathbf{Var})}$. The reason for this is, that we cannot give a suitable definition of $\mathbb{A}^{(\mathbf{Var})}[\![\mathit{stuck}]\!]$.

The well-definedness of the semantics follows from the monotonicity of $G_{\mathsf{P}'}$, the following lemma and Proposition 2.7.

Lemma 4.2. $G_{\mathsf{P}'}(\mathbf{id}_{\mathcal{PF}(\mathbf{Var})}) \leq \mathbf{id}_{\mathcal{PF}(\mathbf{Var})}$.

Proof. We have $G_{\mathsf{P}'}(\mathbf{id}_{\mathcal{PF}(\mathbf{Var})}) = \mathbb{A}^{(\mathbf{Var})}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}'\ \mathit{else}\ \mathit{skip}]\!]$. From Fig. 4.5 one can easily see that $\mathit{merge}(N, \mathbf{Var}_{\mathrm{asgn}}) \leq \mathbf{id}_{\mathcal{PF}(\mathbf{Var})}$. Also, if N is a variable that is not a member of $\mathbf{Var}$ and if $\mathbf{Var}'$ denotes $\mathbf{Var} \uplus \{N\}$, then

$$\mathit{merge}(N, \mathbf{Var}_{\mathrm{asgn}}) \circ \mathbb{A}^{(\mathbf{Var}')}[\![N := b]\!] \leq \mathbf{id}_{\mathcal{PF}(\mathbf{Var})} \quad . \tag{4.23}$$

To verify this, replace N by b everywhere in Fig. 4.5.

By the definition of the abstract semantics for *if*-statements, the semantics $\mathbb{A}^{(\mathbf{Var})}[\![\mathit{if}\ b\ \mathit{then}\ \mathsf{P}'\ \mathit{else}\ \mathit{skip}]\!]$ is less or equal than the left hand side of (4.23), which corresponds to only the false-branch of this *if*-statement. □

The semantics $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ can be computed according to the formula given by Prop. 2.7. If we had defined $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ as the smallest solution to the equation

$$\mathbb{A}^{(\mathbf{Var})}[\![\mathit{while}\ b\ \mathit{do}\ \mathsf{P}']\!] = \mathbb{A}^{(\mathbf{Var})}[\![\mathit{if}\ b\ \mathit{then}\ (\mathsf{P}'; \mathit{while}\ b\ \mathit{do}\ \mathsf{P}')\ \mathit{else}\ \mathit{skip}]\!]$$

then the semantics would not necessarily have been computable. The number of variables (each nested *if*-statement adds one variable) necessary to compute the iterates would have been unbounded.

4.2.3 Discussion

We have given the abstract semantics and also the abstraction from $\mathbf{Distr}_{\mathsf{pol}}$ to $\mathcal{PF}(\mathbf{Var})$ before that. Given a program P and the sets of public and private variables $\mathbf{Var}_{\mathrm{P}}, \mathbf{Var}_{\mathrm{S}} \subseteq \mathbf{Var}$, we envision the abstract semantics to be used as follows:

1. Determine the probability distribution $D \in \mathbf{TerD}[\mathsf{P}]$ that the inputs to P are distributed according to. This distribution has to be found from the context where P is used. Actually, we do not need D, we only need a $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$, such that $\mathcal{X} \leq \beta^{\mathbb{KI}}_{\mathbf{Var}}(D)$. How this distribution D may be determined and this $\mathcal{X}$ may be found, *is not the topic of this thesis*. We believe that in practice, one can easily derive from the context of using P, which elements can be safely assumed to be in $\mathsf{keys}(\mathcal{X})$ and $\mathsf{indeps}(\mathcal{X})$.

2. Compute $\mathcal{Y} = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\mathcal{X})$.

3. Check whether $(\mathbf{Var}_{\mathrm{S}}, \mathbf{Var}_{\mathrm{P}}) \in \mathsf{indeps}(\mathcal{Y})$. If yes, then P has secure information flow, if its inputs are distributed according to D.

Looking at the domain of the abstract semantics (here we mostly have the set $\mathcal{F}(\mathbf{Var})$ in mind), we see that it is much more complex than the domains of the analyses proposed before. However, this compexity has evolved in a quite natural way. The analyses based on information-theoretic definitions of secure information flow have recorded, which variables are low-security (or high-security) variables (the program analysis based approaches [DD77] record it for each program point, the type-based approaches [VSI96] once for the whole program). Their domain of abstract semantics is therefore (more or less) $\mathcal{P}(\mathbf{Var})$.

In [Lau01] we noted that for handling the encryption it is not enough to record the secureness in per-variable basis. Indeed, a ciphertext alone does not reveal the plaintext, the same goes about the key alone. From the ciphertext and the plaintext together, however, the plaintext can be found. Therefore we recorded in [Lau01], which sets of variables are low-security sets of variables, the domain of the abstract semantics was $\mathcal{P}(\mathcal{P}(\mathbf{Var}))$. Recording a set of variables as a low-security set more or less meant that this set of variables was independent from the high security inputs of the program.

In this thesis we have noted that it is also not enough to record only the independece from the secret inputs. For example, rule (4.11) uses the information about the independence of some set of variables from the encrypting black box $[k]_{\mathcal{E}}$. Such information is not recorded by the analysis in [Lau01]. The domain $\mathcal{F}(\mathbf{Var})$ is isomorphic to $\mathcal{P}(\widetilde{\mathbf{Var}}) \rightarrow \mathcal{P}(\mathcal{P}(\widetilde{\mathbf{Var}}))$. The earlier analyses can be obtained by further abstracting (see e.g. [NNH99, Chapter 4]) the abstract semantics introduced here (and in case of [Lau01], putting some extra requirements on the analysed program).

4.3 Shape of the Correctness Proof

Our main correctness result is the following:

Theorem 4.3. *Let* P *be a program with the set of variables* $\mathbf{Var}$. *Then* $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ *abstracts the concrete semantics of* P.

When we substitute the definition of "abstracts" and the definition of $\beta^{\mathbb{KI}}_{\mathbf{Var}}$ to this theorem, then we get the following claim. Let $D \in \mathbf{TerD}[\mathsf{P}]$ and $X, Y \subseteq \mathbf{Var}$. If $(X, Y) \in \mathsf{indeps}(\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\beta^{\mathbb{KI}}_{\mathbf{Var}}(D)))$, then the values of the variables in sets X and Y are independent in the distribution $\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)$.

From this theorem and proposition 3.8 directly follows

Corollary 4.4. *Let* P *be a program with the set of variables* $\mathbf{Var}$, *whose set of private variables is* $\mathbf{Var}_{\mathrm{S}}$ *and whose set of public variables is* $\mathbf{Var}_{\mathrm{P}}$. *Assume that* P *does not assign to the variables in* $\mathbf{Var}_{\mathrm{S}}$. *Let* $D \in \mathbf{TerD}[\mathsf{P}]$. *If*

$$(\mathbf{Var}_{\mathrm{S}}, \mathbf{Var}_{\mathrm{P}}) \in \mathsf{indeps}(\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\beta^{\mathbb{KI}}_{\mathbf{Var}}(D))),$$

then P *with inputs distributed according to* D *has secure information flow.*

The rest of this chapter deals with proving Thm. 4.3.

4.3.1 Proof Idea

Let P be a program with the set of variables $\mathbf{Var}$, let $D \in \mathbf{TerD}[\mathsf{P}]$. In this chapter we mostly deal with showing that

$$\mathsf{indeps} \circ \beta^{\mathbb{KI}}_{\mathbf{Var}} \circ \mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!] \mathbin{\$} D \geq \mathsf{indeps} \circ \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!] \circ \beta^{\mathbb{KI}}_{\mathbf{Var}} \mathbin{\$} D \tag{4.24}$$

(compare that with (4.8)). Only at the end of the chapter, in Sec. 4.7, when we have already proved the above inequality, we show that it also holds when we replace indeps by keys.

If $(X, Y) \in \mathsf{indeps}(\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\beta^{\mathbb{KI}}_{\mathbf{Var}}(D)))$, then we have to show the indistinguishability of distributions

$$\{\!|(S_n \mathbin{\$} X, S_n \mathbin{\$} Y) \,:\, S_n \leftarrow [\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)]_n|\!\}_{n\in\mathbb{N}} \tag{4.25}$$

and

$$\{\!|(S_n \mathbin{\$} X, S'_n \mathbin{\$} Y) \,:\, S_n, S'_n \leftarrow [\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)]_n|\!\}_{n\in\mathbb{N}} \tag{4.26}$$

Similarly to the proof of Prop. 2.11, for each value of the security parameter n we want to construct a small number (i.e. polynomial in n) of hybrid distributions, such that the first and last hybrid are equal to (4.25) and (4.26), respectively, and the difference of neighbouring hybrids is small. In our case, the following differences are considered small:

- For each $k \in \mathsf{keys}(\beta^{\mathbb{KI}}_{\mathbf{Var}}(D))$, the difference between two distributions on (4.3).
- For each $(X, Y) \in \mathsf{indeps}(\beta^{\mathbb{KI}}_{\mathbf{Var}}(D))$, the difference between distributions (4.1) and (4.2).
- The difference of two distributions on (2.9) and the difference of two distributions on (2.10). On (2.9) and (2.10), we stated the security of the encryption system.
- No difference at all is a small difference, too. I.e. we may allow two neighbouring hybrids to be equal.

For each value of the security parameter n we have to construct some kind of structure (and this structure must have some parameters that we can change) and give a suitable interpretation to it, such that

- The interpretation of the structure is a probability distribution over the set $\mathbf{StrOut}^{X;Y}_n := (X \rightarrow \widetilde{\mathbf{Val}}_n) \times (Y \rightarrow \widetilde{\mathbf{Val}}_n)$ (note that the n-th components of families of distributions (4.25) and (4.26) are probability distributions over this set).
- For some values of the parameters of the structure, its interpretation must be equal to the n-th component of (4.25). For some other values of the parameters, the interpretation of the structure must be equal to the n-th component of (4.26).

- Certain changes of the parameters must correspond to the small differences listed above. Using such changes, it must be possible to quickly change ("quickly" means that the number of changes must be polynomial in n) the parameters for (4.25) to the parameters for (4.26).

The structure that we use is basically a flowchart of the program P, where the loops have been unrolled (we have fixed the security parameter, and P runs in expected polynomial time, therefore there is a limit on the number of steps that the program may do). The parameters influence, how the computation along the flowchart proceeds. For example, some parameters influence the distribution of inputs to the flowchart (they may be generated from a single sample of the distribution D_n, but also from multiple samples of it). Some parameters fix, whether at a certain encryption operation, the value to be encrypted is the input to this operation, or is $\mathbf{0}^{\ell(n)}$. Etc.

4.3.2 Roadmap

The proof proceeds as follows. In Sec. 4.4 we define the hybrid distributions. In Sec. 4.5 we define, which hybrids are neighbouring, and show that the distributions (4.25) and (4.26) really are connected through a short chain of neighbours (see comments after the proof of Prop. 2.11). In Sec. 4.6 we summarise the work done in Sec. 4.4 and Sec. 4.5 by assuming that there exists a PPT algorithm $\mathcal{A}$ distinguishing (4.25) and (4.26), and giving a number of PPT algorithms, each of them trying to distinguish a pair of distributions whose difference is considered to be small. The sum of advantages of all these algorithms that we give in Sec. 4.6 is equal to the advantage of algorithm $\mathcal{A}$, therefore at least one of these algorithms has a non-negligible advantage and the assumption about the smallness of the difference of the corresponding pair of distributions is wrong. Finally, in Sec. 4.7 we show that (4.24) also holds when we replace indeps by keys.

Sec. 4.4 has the following structure. Sec. 4.4.1 shows how to unroll a program. It also demonstrates that such unrolling does not (distinguishably) change the concrete semantics of the program and it either does not change the abstract semantics of the program or makes it more optimistic. In Sec. 4.4.2 we define the flowcharts of the program. We do not yet define the interpretation of the flowchart in Sec. 4.4.2, as we have not yet introduced any changeable parameters there. In Secs. 4.4.3–4.4.5 we define the *configurations* of the flowcharts that embody the changeable parameters. There we also define the interpretation of a flowchart together with a configuration. We do not introduce all changeable parameters together, but present them one by one and show for each new parameter, how the definition of interpretation changes. Finally, in Sec. 4.4.6 we fold all these changes together.

Sec. 4.5 has the following structure. In Sec. 4.5.1 we define, how one may change one configuration to another. There we also show that all these changes correspond to "small" steps listed above. In Secs. 4.5.2 and 4.5.3 we show that one can change configurations corresponding to (4.25) to configurations corresponding to (4.26).

More concretely, in Sec. 4.5.2 we prove the main result about the possibility of this changing, and in Sec. 4.5.3 tie up some loose ends. Finally, in Sec. 4.5.4 we show that this change only needs a "small" number of steps introduced in Sec. 4.5.1.

4.4 Structures for the Proof

4.4.1 Unrolling the Program

Let $p \in \mathbf{Pol}(\mathbb{Z})$ be an upper bound on the runtime of the program P, when its inputs are distributed according to D.

Similarly to the proof of Prop. 2.11, we start by fixing the security parameter n (in the proof of Prop. 2.11, the distributions $D_n^{(i)}$ were defined for a fixed n, they were not families of probability distributions). When n has been fixed, then the maximal running time of P has also been fixed — it is at most $p(n)$.

The unrolled program, denoted by $\mathsf{P}^{(n)}$, is basically a sequence of assignments. It is defined inductively over the program structure. If P is *skip* or a single assignment, then $\mathsf{P}^{(n)} = \mathsf{P}$. If $\mathsf{P} = \mathsf{P}_1; \mathsf{P}_2$, then $\mathsf{P}^{(n)} = \mathsf{P}_1^{(n)}; \mathsf{P}_2^{(n)}$.

If P is *if* b *then* P_1 *else* P_2, then $\mathsf{P}^{(n)}$ is defined as follows. Let $\mathbf{Var}_{\text{asgn}}$ be the set of variables that are assigned to in either P_1 or P_2. Define

$$\begin{aligned}
\mathbf{Var}_{\text{asgn}}^{\mathsf{true}} &:= \{x^{\mathsf{true}} : x \in \mathbf{Var}_{\text{asgn}}\} \\
\mathbf{Var}_{\text{asgn}}^{\mathsf{false}} &:= \{x^{\mathsf{false}} : x \in \mathbf{Var}_{\text{asgn}}\} \\
\mathbf{Var}_{\text{all}} &:= \mathbf{Var} \uplus \mathbf{Var}_{\text{asgn}}^{\mathsf{true}} \uplus \mathbf{Var}_{\text{asgn}}^{\mathsf{false}} \uplus \{N\} \ .
\end{aligned} \tag{4.27}$$

Let $\mathsf{P}_1^{\mathsf{true}}$ be identical to P_1, except that all occurrences of variables in $\mathbf{Var}_{\text{asgn}}$ are replaced with the corresponding variables in $\mathbf{Var}_{\text{asgn}}^{\mathsf{true}}$. Similarly, let $\mathsf{P}_2^{\mathsf{false}}$ be identical to P_2, except that all occurrences of variables in $\mathbf{Var}_{\text{asgn}}$ are replaced with the corresponding variables in $\mathbf{Var}_{\text{asgn}}^{\mathsf{false}}$. Consider the program P' in Fig. 4.6. We define $\mathsf{P}^{(n)} = (\mathsf{P}')^{(n)}$.

In Fig. 4.6, $\langle x_1, \ldots, x_k\rangle := N\,?\langle y_1, \ldots, y_k\rangle : \langle z_1, \ldots, z_k\rangle$, called the *vectorised choice*, is a new kind of statement, replacing the *merge*-operation. Its semantics (both concrete and abstract) is defined below. Intuitively, it simultaneously assigns to the variables $x_1, \ldots, x_k$ either the values of $y_1, \ldots, y_k$ or the values of $z_1, \ldots, z_k$, depending on whether the value of N is true or false.

The concrete semantics of the vectorised choice is the following:

$$\mathbb{C}_{\text{len}}[\![\langle x_1, \ldots, x_k\rangle := N\,?\langle y_1, \ldots, y_k\rangle : \langle z_1, \ldots, z_k\rangle]\!](S_n) =$$
$$\eta^{\mathcal{D}}\left((S_n\begin{bmatrix} x_1 \mapsto \langle\!\langle S_n(N)\ \mathbf{?}\ S_n(y_1)\ \mathbf{:}\ S_n(z_1)\rangle\!\rangle \\ \cdots\cdots\cdots\cdots\cdots\cdots \\ x_k \mapsto \langle\!\langle S_n(N)\ \mathbf{?}\ S_n(y_k)\ \mathbf{:}\ S_n(z_k)\rangle\!\rangle \end{bmatrix}, 0)\right),$$

where $S_n \in \mathbf{State}_n$ and $\langle\!\langle b\ \mathbf{?}\ x\ \mathbf{:}\ y\rangle\!\rangle$ is defined on Fig. 3.1. The concrete semantics therefore nicely corresponds to the intuition given above. For our purposes, it does

The program P':

$$
\begin{aligned}
&N := b;\\
&x_1^{\mathsf{true}} := x_1;\\
&\dots\dots\dots\dots\\
&x_m^{\mathsf{true}} := x_m;\\
&x_1^{\mathsf{false}} := x_1;\\
&\dots\dots\dots\dots\\
&x_m^{\mathsf{false}} := x_m;\\
&\mathsf{P}_1^{\mathsf{true}};\\
&\mathsf{P}_2^{\mathsf{false}};\\
&\langle x_1, \dots, x_m\rangle := N\,?\langle x_1^{\mathsf{true}}, \dots, x_m^{\mathsf{true}}\rangle : \langle x_1^{\mathsf{false}}, \dots, x_m^{\mathsf{false}}\rangle
\end{aligned}
$$

Here $\{x_1, \dots, x_m\} = \mathbf{Var}_{\mathrm{asgn}}$.

Figure 4.6: Transformed program *if* b *then* P_1 *else* P_2

not matter, how long the running time of the vectorised choice is; here, to just define it somehow, we have defined, that it takes no time.

The abstract semantics of the vectorised choice is the following. Let P be $\langle x_1, \dots, x_k\rangle := b\,?\langle y_1, \dots, y_k\rangle : \langle z_1, \dots, z_k\rangle$. Let $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$. We define $\mathcal{Y} = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\mathcal{X})$ as the smallest element of $\mathcal{PF}(\mathbf{Var})$, such that

- $\mathcal{Y}$ satisfies the rules in Fig. 4.7 and Fig. 4.8.
- $\mathsf{indeps}(\mathcal{Y})$ is symmetric and downwards closed.

As we see, the rules in Fig. 4.7 and Fig. 4.8 more or less repeat the rules in Fig. 4.5. The explanations of the rules on Fig. 4.5 therefore also explain the rules on Fig. 4.7–4.8.

Having introduced the vectorised choice statement, we also have to define its unrolled form. If P is a single vectorised choice, then we define $\mathsf{P}^{(n)}$ to be P.

Last, let P be *while* b *do* P'. Then $\mathsf{P}^{(n)}$ is defined as follows:

$$
\begin{aligned}
\mathsf{Q} &:= (\textit{if}\ b\ \textit{then}\ \mathsf{P}'\ \textit{else skip})^{(n)}\\
\mathsf{P}^{(n)} &:= \underbrace{\mathsf{Q}; \mathsf{Q}; \cdots \mathsf{Q}}_{p(n)\ \text{times}}\ .
\end{aligned}
$$

Note that the program $\mathsf{P}^{(n)}$ always terminates, no matter what its input is and what the original program P was. Also note that the size of the program $\mathsf{P}^{(n)}$ is polynomial in n.

The concrete and abstract semantices of P and $\mathsf{P}^{(n)}$ are related by the following two propositions:

Proposition 4.5. *The concrete semantices of* P *and* $\mathsf{P}^{(n)}$ *are related by*

$$
\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D) \approx \{\!|S_n : S_n \leftarrow [\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}^{(n)}]\!](D)]_n|\!\}_{n\in\mathbb{N}}\ .
$$

$$
\frac{\begin{array}{c}
(X,Y) \in \mathsf{indeps}(\mathcal{X}) \\
i_1,\ldots,i_l,i_{l+1},\ldots,i_m \in \{1,\ldots,k\} \\
(\{[y_{i_1}]_\mathcal{E},\ldots,[y_{i_m}]_\mathcal{E}\}, X \cup Y \cup \{b\}) \in \mathsf{indeps}(\mathcal{X}) \\
(\{[z_{i_1}]_\mathcal{E},\ldots,[z_{i_m}]_\mathcal{E}\}, X \cup Y \cup \{b\}) \in \mathsf{indeps}(\mathcal{X}) \\
\left[\begin{array}{c}
(\{[y_{i_1}]_\mathcal{E}\},\{[y_{i_2}]_\mathcal{E},\ldots,[y_{i_m}]_\mathcal{E}\}),(\{[z_{i_1}]_\mathcal{E}\},\{[z_{i_2}]_\mathcal{E},\ldots,[z_{i_m}]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{X}) \\
(\{[y_{i_2}]_\mathcal{E}\},\{[y_{i_3}]_\mathcal{E},\ldots,[y_{i_m}]_\mathcal{E}\}),(\{[z_{i_2}]_\mathcal{E}\},\{[z_{i_3}]_\mathcal{E},\ldots,[z_{i_m}]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{X}) \\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
(\{[y_{i_{m-1}}]_\mathcal{E}\},\{[y_{i_m}]_\mathcal{E}\}),(\{[z_{i_{m-1}}]_\mathcal{E}\},\{[z_{i_m}]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{X})
\end{array}\right] \\
\left[\begin{array}{c}
(\{[y_{i_{l+1}}]_\mathcal{E}\},\{[y_{i_{l+1}}]_\mathcal{E}\}),(\{[z_{i_{l+1}}]_\mathcal{E}\},\{[z_{i_{l+1}}]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{X}) \\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
(\{[y_{i_m}]_\mathcal{E}\},\{[y_{i_m}]_\mathcal{E}\}),(\{[z_{i_m}]_\mathcal{E}\},\{[z_{i_m}]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{X})
\end{array}\right] \\
y_{i_1},z_{i_1},\ldots,y_{i_m},z_{i_m} \in \mathsf{keys}(\mathcal{X}) \\
x_{i_1},[x_{i_1}]_\mathcal{E},\ldots,x_{i_m},[x_{i_m}]_\mathcal{E} \notin X \cup Y \\
y_1,[y_1]_\mathcal{E},z_1,[z_1]_\mathcal{E},\ldots,y_k[y_k]_\mathcal{E},z_k,[z_k]_\mathcal{E},b \notin X \cup Y
\end{array}}{(X \cup \{[x_{i_1}]_\mathcal{E},\ldots,[x_{i_m}]_\mathcal{E}\}, Y \cup \{[x_{i_{l+1}}]_\mathcal{E},\ldots,[x_{i_m}]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{Y})} \tag{4.28}
$$

$$
\frac{\begin{array}{c}
i_1,\ldots,i_l,i_{l+1},\ldots,i_m,j_1,\ldots,j_r,j_{r+1},\ldots,j_s \in \{1,\ldots,k\} \\
\big(X \cup \{[y_{i_1}]_\mathcal{E},\ldots,[y_{i_m}]_\mathcal{E}\}, \\
\quad Y \cup \{b,[y_{i_{l+1}}]_\mathcal{E},\ldots,[y_{i_m}]_\mathcal{E},y_{j_1},[y_{j_1}]_\mathcal{E},\ldots,y_{j_r},[y_{j_r}]_\mathcal{E},[y_{j_{r+1}}]_\mathcal{E},\ldots,[y_{j_s}]_\mathcal{E}\}\big) \\
\in \mathsf{indeps}(\mathcal{X}) \\
\big(X \cup \{[z_{i_1}]_\mathcal{E},\ldots,[z_{i_m}]_\mathcal{E}\}, \\
\quad Y \cup \{b,[z_{i_{l+1}}]_\mathcal{E},\ldots,[z_{i_m}]_\mathcal{E},z_{j_1},[z_{j_1}]_\mathcal{E},\ldots,z_{j_r},[z_{j_r}]_\mathcal{E},[z_{j_{r+1}}]_\mathcal{E},\ldots,[z_{j_s}]_\mathcal{E}\}\big) \\
\in \mathsf{indeps}(\mathcal{X}) \\
(X,\{[y_{i_1}]_\mathcal{E},\ldots,[y_{i_m}]_\mathcal{E}\}),(X,\{[z_{i_1}]_\mathcal{E},\ldots,[z_{i_m}]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{X}) \\
\left[\begin{array}{c}
(\{[y_{i_1}]_\mathcal{E}\},\{[y_{i_2}]_\mathcal{E},\ldots,[y_{i_m}]_\mathcal{E}\}),(\{[z_{i_1}]_\mathcal{E}\},\{[z_{i_2}]_\mathcal{E},\ldots,[z_{i_m}]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{X}) \\
(\{[y_{i_2}]_\mathcal{E}\},\{[y_{i_3}]_\mathcal{E},\ldots,[y_{i_m}]_\mathcal{E}\}),(\{[z_{i_2}]_\mathcal{E}\},\{[z_{i_3}]_\mathcal{E},\ldots,[z_{i_m}]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{X}) \\
\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
(\{[y_{i_{m-1}}]_\mathcal{E}\},\{[y_{i_m}]_\mathcal{E}\}),(\{[z_{i_{m-1}}]_\mathcal{E}\},\{[z_{i_m}]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{X})
\end{array}\right] \\
y_{i_1},z_{i_1},\ldots,y_{i_m},z_{i_m} \in \mathsf{keys}(\mathcal{X}) \\
x_{i_1},[x_{i_1}]_\mathcal{E},\ldots,x_{i_m},[x_{i_m}]_\mathcal{E},x_{j_1},[x_{j_1}]_\mathcal{E},\ldots,x_{j_s},[x_{j_s}]_\mathcal{E} \notin X \cup Y \\
y_1,[y_1]_\mathcal{E},z_1,[z_1]_\mathcal{E},\ldots,y_k[y_k]_\mathcal{E},z_k,[z_k]_\mathcal{E},b \notin X \cup Y
\end{array}}{\begin{array}{l}
\big(X \cup \{[x_{i_1}]_\mathcal{E},\ldots,[x_{i_m}]_\mathcal{E}\}, \\
\quad Y \cup \{[x_{i_{l+1}}]_\mathcal{E},\ldots,[x_{i_m}]_\mathcal{E},x_{j_1},[x_{j_1}]_\mathcal{E},\ldots,x_{j_r},[x_{j_r}]_\mathcal{E},[x_{j_{r+1}}]_\mathcal{E},\ldots,[x_{j_s}]_\mathcal{E}\}\big) \\
\hfill \in \mathsf{indeps}(\mathcal{Y})
\end{array}} \tag{4.29}
$$

Figure 4.7: The semantics $\mathbb{A}^{(\mathbf{Var})}[\![\langle x_1,\ldots,x_k\rangle := b\,?\langle y_1,\ldots,y_k\rangle : \langle z_1,\ldots,z_k\rangle]\!]$, indeps-part

$$\frac{\begin{array}{c} w \in \mathsf{keys}(\mathcal{X}) \\ w \notin \{x_1, \ldots, x_k, y_1, \ldots, y_k, z_1, \ldots, z_k\} \end{array}}{w \in \mathsf{keys}(\mathcal{Y})} \tag{4.30}$$

$$\frac{\begin{array}{c} i \in \{1, \ldots, k\} \\ y_i, z_i \in \mathsf{keys}(\mathcal{X}) \\ (\{[y_i]_{\mathcal{E}}\}, \{b\}), (\{[z_i]_{\mathcal{E}}\}, \{b\}) \in \mathsf{indeps}(\mathcal{X}) \end{array}}{x_i \in \mathsf{keys}(\mathcal{Y})} \tag{4.31}$$

Figure 4.8: The semantics $\mathbb{A}^{(\mathbf{Var})}[\![\langle x_1, \ldots, x_k \rangle := b\,?\langle y_1, \ldots, y_k \rangle : \langle z_1, \ldots, z_k \rangle]\!]$, keys-part

Note that in the right hand side of $\approx$, the n-th member of the family of probability distributions is defined using the program $\mathsf{P}^{(n)}$ (i.e. different programs for different members).

Proof. The proof is by induction over the program structure.

If P is an assignment, *skip* or vectorised choice, then the claim of the proposition follows immediately.

Also, if P is equal to $\mathsf{P}_1; \mathsf{P}_2$ then the claim follows immediately (by using Lemma 2.10).

If P is *if* b *then* P_1 *else* P_2, then the claim follows from the fact, that the programs $(\mathsf{P}_1^{\mathsf{true}})^{(n)}$ and $(\mathsf{P}_2^{\mathsf{false}})^{(n)}$ always terminate — they are just sequences of assignments and vectorised choices. Therefore eagerly computing both branches of an *if*-statement cannot lead to nontermination here.

If P is *while* b *do* P', then the distributions in the claim of the proposition are indistinguishable, because there is only a negligible chance that the number of iterations of the loop is greater than $p(n)$. We have therefore sufficiently unrolled P when creating $\mathsf{P}^{(n)}$. ∎

Proposition 4.6. *Let $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$. Then $\mathbb{A}^{(\mathbf{Var}'')}[\![\mathsf{P}^{(n)}]\!] \geq \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$ for each $n \in \mathbb{N}$, where $\mathbf{Var}''$ is the set of variables of the program $\mathsf{P}^{(n)}$.*

Note that the set $\mathbf{Var}''$ does not depend on n, it only depends on P.

Proof. The proof is by induction over the program structure.

If P is an assignment, *skip* or vectorised choice, then the claim of the proposition follows immediately.

Also, if P is equal to $\mathsf{P}_1; \mathsf{P}_2$ then the claim follows immediately from the definition of $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}_1; \mathsf{P}_2]\!]$ and the monotonicity of functional composition.

Suppose that P is *if* b *then* P_1 *else* P_2. By the induction assumption, the abstract semantics of $(\mathsf{P}_1^{\mathsf{true}})^{(n)}$ is greater that the abstract semantics of P_1, if we rename the variables in $\mathbf{Var}_{\mathrm{asgn}}^{\mathsf{true}}$ back to variables in $\mathbf{Var}_{\mathrm{asgn}}$; similar result holds for $(\mathsf{P}_2^{\mathsf{false}})^{(n)}$ and P_2.

Let $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$. We introduce the following quantities:

- Let the programs $\mathsf{Q}_1, \mathsf{Q}_2, \mathsf{Q}_3$ be the following:

$$\begin{aligned}&\mathsf{Q}_1 \equiv N := b; x_1^{\mathsf{true}} := x_1; \cdots x_m^{\mathsf{true}} := x_m; x_1^{\mathsf{false}} := x_1; \cdots x_m^{\mathsf{false}} := x_m\\&\mathsf{Q}_2 \equiv \mathsf{Q}_1; \mathsf{P}_1^{\mathsf{true}}\\&\mathsf{Q}_3 \equiv \mathsf{Q}_2; \mathsf{P}_2^{\mathsf{false}},\end{aligned}$$

  where $\{x_1, \ldots, x_m\} = \mathbf{Var}_{\mathrm{asgn}}$ (compare this with Fig. 4.6).

- Let $\mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3 \in \mathcal{PF}(\mathbf{Var}_{\mathrm{all}})$ be defined by $\mathcal{Z}_i = \mathbb{A}^{(\mathbf{Var}_{\mathrm{all}})}[\![\mathsf{Q}_i]\!](\mathcal{X})$.

- Let $\mathcal{Z}_i^{\mathsf{true}}$, where $i \in \{1,2,3\}$, be the projection of $\mathcal{Z}_i$ to the set of variables $(\mathbf{Var} \cup \{N\} \backslash \mathbf{Var}_{\mathrm{asgn}}) \cup \mathbf{Var}_{\mathrm{asgn}}^{\mathsf{true}}$. Projecting means removing from the keys- and indeps-components of $\mathcal{Z}_i$ all elements that contain something not in the set to be projected onto. Let $\mathcal{Z}_i^{\mathsf{false}}$ be defined similarly.

- Let $\hat{\mathcal{Z}}_i^{\mathsf{true}} \in \mathcal{PF}(\mathbf{Var} \cup \{N\})$ be equal to $\mathcal{Z}_i^{\mathsf{true}}$, where the variables in $\mathbf{Var}_{\mathrm{asgn}}^{\mathsf{true}}$ are renamed back to variables in $\mathbf{Var}_{\mathrm{asgn}}$. Let $\hat{\mathcal{Z}}_i^{\mathsf{false}}$ be defined similarly.

Then the following equalities hold (they follow from rules (4.9) and (4.10^2)):

$$\begin{gathered}\hat{\mathcal{Z}}_1^{\mathsf{true}} = \hat{\mathcal{Z}}_1^{\mathsf{false}} = \mathbb{A}^{(\mathbf{Var}\cup\{N\})}[\![N := b]\!](\mathcal{X})\\ \hat{\mathcal{Z}}_2^{\mathsf{true}} = \mathbb{A}^{(\mathbf{Var}\cup\{N\})}[\![N := b; \mathsf{P}_1]\!](\mathcal{X})\\ \mathcal{Z}_2^{\mathsf{false}} = \mathcal{Z}_1^{\mathsf{false}}\\ \mathcal{Z}_3^{\mathsf{true}} = \mathcal{Z}_2^{\mathsf{true}} \text{ and therefore } \hat{\mathcal{Z}}_3^{\mathsf{true}} = \mathbb{A}^{(\mathbf{Var}\cup\{N\})}[\![N := b; \mathsf{P}_1]\!](\mathcal{X})\\ \hat{\mathcal{Z}}_3^{\mathsf{false}} = \mathbb{A}^{(\mathbf{Var}\cup\{N\})}[\![N := b; \mathsf{P}_2]\!](\mathcal{X}) \ .\end{gathered}$$

The abstract semantics of *if b then* P_1 *else* P_2 is therefore defined by the application of the rules in Fig. 4.5 to $\hat{\mathcal{Z}}_3^{\mathsf{true}} \wedge \hat{\mathcal{Z}}_3^{\mathsf{false}}$. The abstract semantics of the program P' in Fig. 4.6 is defined by the application of the rules in Fig. 4.7 and Fig. 4.8 to the "union" of $\mathcal{Z}_3^{\mathsf{true}}$ and $\mathcal{Z}_3^{\mathsf{false}}$ (the pairs of sets of variables that belong to the indeps-component of neither $\mathcal{Z}_3^{\mathsf{true}}$ nor $\mathcal{Z}_3^{\mathsf{false}}$, are not used by the rules in Fig. 4.7–4.8). Therefore

$$\mathbb{A}^{(\mathbf{Var})}[\![\textit{if } b \textit{ then } \mathsf{P}_1 \textit{ else } \mathsf{P}_2]\!] = \mathbb{A}^{(\mathbf{Var}_{\mathrm{all}})}[\![\mathsf{P}']\!] \ .$$

Applying the induction assumption for P' gives $\mathbb{A}^{(\mathbf{Var}'')}[\![(\mathsf{P}')^{(n)}]\!] \geq \mathbb{A}^{(\mathbf{Var}_{\mathrm{all}})}[\![\mathsf{P}']\!]$. From the fact $\mathsf{P}^{(n)} = (\mathsf{P}')^{(n)}$ follows the claim of the proposition.

Last, if P is *while b do* P', then

$$\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!] = \bigwedge\{\left(\mathbb{A}^{(\mathbf{Var})}[\![\textit{if } b \textit{ then } \mathsf{P}' \textit{ else skip}]\!]\right)^k \ : \ k \in \mathbb{N}\}$$

and

$$\mathbb{A}^{(\mathbf{Var}'')}[\![\mathsf{P}^{(n)}]\!] = \left(\mathbb{A}^{(\mathbf{Var}'')}[\![\textit{if } b \textit{ then } \mathsf{P}' \textit{ else skip}]\!]\right)^{p(n)} \ .$$

Therefore $\mathbb{A}^{(\mathbf{Var}'')}[\![\mathsf{P}^{(n)}]\!] \geq \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$. □

$$
\begin{array}{l}
k_1 := \mathcal{G}en() \\
\textit{if } b \textit{ then} \\
\quad k_2 := k_1 \\
\textit{else} \\
\quad k_2 := \mathcal{G}en() \\
x_1 := \mathcal{E}nc(k_1, y_1) \\
x_2 := \mathcal{E}nc(k_2, y_2)
\end{array}
$$

b is independent of $\{x_1, x_2\}$ in the end.

Figure 4.9: The example program

From these two propositions follows that for showing (4.24), it is enough to prove the following proposition:

Proposition 4.7. *Let* P *be a program and let* $D \in \mathbf{TerD}[\mathsf{P}]$. *Let* $\mathbf{Var}$ *be the set of variables of* P *and let* $\mathbf{Var}'' \supseteq \mathbf{Var}$ *be the set of variables of the programs* $\mathsf{P}^{(n)}$. *Let* $\mathcal{X} = \beta^{\mathbb{KI}}_{\mathbf{Var}}(D)$,

$$D' = \{\!|S_n \,:\, S_n \leftarrow [\mathbb{C}_{\text{term}}[\![\mathsf{P}^{(n)}]\!](D)]_n|\!\}_{n\in\mathbb{N}}$$

and let $X, Y \subseteq \widetilde{\mathbf{Var}}$. *If for all* $n \in \mathbb{N}$ *holds* $(X, Y) \in \mathsf{indeps}(\mathbb{A}^{(\mathbf{Var}'')}[\![\mathsf{P}^{(n)}]\!](\mathcal{X}))$, *then* X *and* Y *are independent in the distribution* D'.

In the following we consider the program $\mathsf{P}^{(n)}$, where the security parameter has been fixed. It is a sequence of assignments and vectorised choices (we may assume that the *skip*-statements have been removed). Let the number of assignments be s and let $\mathsf{P}^{(n)} = \mathsf{A}_1; \cdots; \mathsf{A}_s$, where A_i are individual assignments.

The running example. We now introduce a program, on which we demonstrate the proof of correctness of the analysis. The program that we have in mind is presented in Fig. 4.9, and we are going to show, how the independence of the variable `b` from the variables `x1` and `x2` at the end of this program is proved. The first step of the proof is to unroll the program. There are no loops in the running example, therefore all unrolled programs, corresponding to different values of the security parameter, are equal. The unrolled program (for all security parameters) is presented in Fig. 4.10. We have numbered its lines for easier referencing.

Let us show, how $(\{b\}, \{x_1, x_2\}) \in \mathsf{indeps}(\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}^{(n)}]\!](\mathcal{X}))$ can be derived, where $\mathsf{P}^{(n)}$ is the program in Fig. 4.10 and $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$ is the abstraction of the input distribution. We need to make no assumptions about $\mathcal{X}$. Let $\mathcal{Y}_i$, where $i \in \{0, \ldots, 9\}$ be the value of the abstract semantics after the statements $1, \ldots, i$. I.e. $\mathcal{Y}_0 = \mathcal{X}$ and $\mathcal{Y}_i = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{A}_i]\!](\mathcal{Y}_{i-1})$, where A_i is the i-th assignment or vectorised choice in Fig. 4.10. We have

(i). $(\emptyset, \{b, y_1, y_2\}) \in \mathsf{indeps}(\mathcal{Y}_0)$, because the empty set is independent of every other set. Also $(\emptyset, \{b, y_1, y_2\}) \in \mathsf{indeps}(\mathcal{Y}_i)$ for $i \in \{1, \ldots, 9\}$ by rule (4.9).

$$
\begin{array}{l}
k_1 := \mathcal{G}en() \\
N := b \\
k_2^{\mathsf{true}} := k_2 \\
k_2^{\mathsf{false}} := k_2 \\
k_2^{\mathsf{true}} := k_1 \\
k_2^{\mathsf{false}} := \mathcal{G}en() \\
\langle k_2 \rangle := N \,?\, \langle k_2^{\mathsf{true}} \rangle : \langle k_2^{\mathsf{false}} \rangle \\
x_1 := \mathcal{E}nc(k_1, y_1) \\
x_2 := \mathcal{E}nc(k_2, y_2)
\end{array}
$$

b is independent of $\{x_1, x_2\}$ in the end.

Figure 4.10: The example program, unrolled

(ii). $(\{[k_1]_\mathcal{E}\}, \{b, y_1, y_2, [k_1]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{Y}_1)$ by rule (4.12). The conditions of using this rule are fulfilled by (i). Also, $(\{[k_1]_\mathcal{E}\}, \{b, y_1, y_2, [k_1]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{Y}_i)$ for $i \in \{2, \ldots, 9\}$ by rule (4.9).

(iii). $k_1 \in \mathsf{keys}(\mathcal{Y}_1)$ by rule (4.15). Also, $k_1 \in \mathsf{keys}(\mathcal{Y}_i)$ for $i \in \{2, \ldots, 9\}$ by rule (4.14).

(iv). $(\{[k_1]_\mathcal{E}\}, \{b, N, y_1, y_2, [k_1]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{Y}_2)$ by rule (4.10). The conditions of using this rule are fulfilled by (ii). Also, $(\{[k_1]_\mathcal{E}\}, \{b, N, y_1, y_2, [k_1]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{Y}_i)$ for $i \in \{3, \ldots, 6\}$ by rule (4.9).

(v). $(\{[k_2^{\mathsf{true}}]_\mathcal{E}\}, \{b, N, y_1, y_2, [k_1]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{Y}_5)$ by rule (4.10^2). The conditions of using this rule are fulfilled by (iv). Also, $(\{[k_2^{\mathsf{true}}]_\mathcal{E}\}, \{b, N, y_1, y_2, [k_1]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{Y}_6)$ by rule (4.9).

(vi). $k_2^{\mathsf{true}} \in \mathsf{keys}(\mathcal{Y}_5)$ by rule (4.14^1). The conditions of using this rule are fulfilled by (iii). Also, $k_2^{\mathsf{true}} \in \mathsf{keys}(\mathcal{Y}_5)$ by rule (4.14).

(vii). $(\{[k_2^{\mathsf{false}}]_\mathcal{E}\}, \{b, N, y_1, y_2, [k_1]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{Y}_6)$ by rule (4.10). The conditions of using this rule require in this case, that the empty set is independent of some other set. But this is always the case.

(viii). $k_2^{\mathsf{false}} \in \mathsf{keys}(\mathcal{Y}_6)$ by rule (4.15).

(ix). $(\{[k_2]_\mathcal{E}\}, \{b, y_2\}) \in \mathsf{indeps}(\mathcal{Y}_7)$ by rule (4.28). This rule is instantiated as follows:

- $X \longleftarrow \emptyset$, $Y \longleftarrow \{b, y_2\}$
- $l \longleftarrow 1$, $m \longleftarrow 1$
- $y_{i_1} \longleftarrow k_2^{\mathsf{true}}$, $z_{i_1} \longleftarrow k_2^{\mathsf{false}}$, $x_{i_1} \longleftarrow k_2$.

These instantiations make the two groups of antecedents of rule (4.28) empty. The other conditions of using this rule are fulfilled by (v), (vii), (vi) and (viii).

(x). $(\{[k_2]_\mathcal{E}, b, y_1, y_2\}, \{[k_1]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{Y}_7)$ by rule (4.28). This rule is instantiated as follows:

- $X \longleftarrow \{b, y_1, y_2\}$. $Y \longleftarrow \{[k_1]_\mathcal{E}\}$
- $l \longleftarrow 1$, $m \longleftarrow 1$
- $y_{i_1} \longleftarrow k_2^{\mathsf{true}}$, $z_{i_1} \longleftarrow k_2^{\mathsf{false}}$, $x_{i_1} \longleftarrow k_2$.

These instantiations make the two groups of antecedents of rule (4.28) empty. The other conditions of using this rule are fulfilled by (ii), (v), (vii), (vi) and (viii).

(xi). $k_2 \in \mathsf{keys}(\mathcal{Y}_7)$ by rule (4.31). The conditions of using this rule are fulfilled by (vi), (viii), (v) and (vii). Also, $k_2 \in \mathsf{keys}(\mathcal{Y}_8)$ by rule (4.14).

(xii). $(\{b\}, \{x_1\}) \in \mathsf{indeps}(\mathcal{Y}_8)$ by rule (4.11). The conditions of using this rule are fulfilled by (i), (iii) and (ii).

(xiii). $(\{[k_2]_\mathcal{E}\}, \{b, x_1, y_2\}) \in \mathsf{indeps}(\mathcal{Y}_8)$ by rule (4.11). The conditions of using this rule are fulfilled by (ix), (iii) and (x).

(xiv). $(\{b\}, \{x_1, x_2\}) \in \mathsf{indeps}(\mathcal{Y}_9)$ by rule (4.11). The conditions of using this rule are fulfilled by (xii), (xi) and (xiii).

4.4.2 The Flowchart of an Unrolled Program

The flowchart of a program (that is a sequence of assignments and vectorised choices) is a directed acyclic graph (DAG), whose nodes correspond to the operations of the program and whose edges carry the values from the operations that produced them to the operations that use them. There are also input and output edges. In general, there is one node per assignment, labelled with the operator of the assignment.

The variant of the construction of flowcharts that we present here gives extra attention to the encrypting black boxes. Namely:

- The encrypting black boxes are first class values. This means that the edges of the flowchart may carry either bit-strings or encrypting black boxes.

- The encrypting black boxes can be inputs to the flowchart or be produced by a node labelled with $\mathcal{G}en$.

- Each encrypting black box is used only once. A use of a black box can be either using it for encryption or outputting it. If we want to encrypt several times with the same key, then we introduce several copies of the corresponding black box. This convention is necessary for using the independence of black boxes from themselves in the arguments.

We now give a precise definition of the flowchart for a sequence of assignments and vectorised choices P, computing the values of $X \subseteq \widetilde{\mathbf{Var}}$. We denote this flowchart by $\mathsf{Chart}_{\mathsf{P};X}$.

A flowchart is a DAG $G = (N, E)$. Nodes are labelled with operators. Edges, incoming to a certain node, are ordered. Inputs to a flowchart are edges without sources. Flowchart's outputs are edges without targets. The sources and targets are labelled with the elements of $\widetilde{\mathbf{Var}}$. One edge may have several targets. Notation:

- $\mathsf{Nodes}(G) = N$;
- $\mathsf{Edges}(G) = E$;
- $\mathsf{inputs}(G) \subseteq E$ — inputs of G;
- $\mathsf{outputs}(G) \subseteq E$ — outputs of G;
- $\overrightarrow{\rho}(v) \in E^*$ — inputs to the node v;
- $\overleftarrow{\rho}(v) \in E$ — output from the node v;
- $\lambda_N(v) \in \mathbf{Op}'$ — the label of v;
- $\lambda_I(e) \in \widetilde{\mathbf{Var}}$ — the label of $e \in \mathsf{inputs}(G)$;
- $\lambda_O(e) \in \widetilde{\mathbf{Var}}$ — the label of $e \in \mathsf{outputs}(G)$.

Here $\mathbf{Op}' := \mathbf{Op} \uplus \{\mathcal{G}en_{\mathrm{val}}, Mk\mathcal{E}, \mathtt{?:}, [\mathtt{?:}]_{\mathcal{E}}\}$. The extra elements of $\mathbf{Op}'$ denote the following operations:

- $\mathcal{G}en_{\mathrm{val}}$ denotes the operation creating a new encryption key (i.e. a bit-string). We use $\mathcal{G}en$ to denote the operation that creates a new encrypting black box.
- $Mk\mathcal{E}$ denotes the operation of creating an encrypting black box, given the bit-string that is to be used as the key.
- $\mathtt{?:}$ is used for vectorised choices. A node labelled with $\mathtt{?:}$ has three inputs, corresponding to the boolean variable b and to some variables y_i and z_i (with the same subscript; see Fig. 4.7 and Fig. 4.8). The operator $[\mathtt{?:}]_{\mathcal{E}}$ is also used for vectorised choices, its three inputs correspond to the variable b and to black boxes $[y_i]_{\mathcal{E}}$ and $[z_i]_{\mathcal{E}}$.

Inputs to a node v are also called the *in-edges* of v. Output from the node v is also called the *out-edge* of v.

If P is the empty sequence of assignments and vectorised choices, (i.e. P is *skip*) then $\mathsf{Chart}_{skip;X}$ is a flowchart with zero nodes and $|X|$ edges, where all edges are both inputs and outputs of the flowchart. The edges are labelled with the elements of X.

We need an auxiliary notion while giving the definition of $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$, where A is an assignment or a vectorised choice and P is a sequence of assignments and vectorised choices. For $e \in \mathsf{inputs}(\mathsf{Chart}_{\mathsf{P};X})$, where $\lambda_I(e) = [x]_{\mathcal{E}}$ for some $x \in \mathbf{Var}$, let $\mathsf{bb\_use}(e)$ denote the node or output edge of $\mathsf{Chart}_{\mathsf{P};X}$ where the value on e is used. For $\mathsf{Chart}_{skip;X}$, $\mathsf{bb\_use}$ is the identity function (actually, a partial function, defined only for the edges that are labelled with some $[x]_{\mathcal{E}}$).

The shape of the flowchart $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ depends on whether A is an assignment or a vectorised choice, and on the operator of the assignment.

A is $x := o(x_1, \ldots, x_k)$

Here we have assumed that o is not $\mathcal{E}nc$ or $\mathcal{G}en$, nor is A a simple assignment $x := y$ (these cases are handled separately). If no input of $\mathsf{Chart}_{\mathsf{P};X}$ is labelled with x or $[x]_{\mathcal{E}}$, then $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ is equal to $\mathsf{Chart}_{\mathsf{P};X}$. Otherwise, it is constructed in the following way.

Let N and E be the set of nodes and the set of edges of the flowchart $\mathsf{Chart}_{\mathsf{P};X}$. Let the edge e_x (if it exists) be the input to $\mathsf{Chart}_{\mathsf{P};X}$ that is labelled with x. Let $E_{[x]_{\mathcal{E}}}$ be the set of inputs of $\mathsf{Chart}_{\mathsf{P};X}$ that are labelled with $[x]_{\mathcal{E}}$. Let $O \subseteq \{x_1, \ldots, x_k\}$ contain all x_i, such that no input to $\mathsf{Chart}_{\mathsf{P};X}$ is labelled with x_i. Let e_{x_i}, where $x_i \in \{x_1, \ldots, x_k\} \backslash O$ be the input to $\mathsf{Chart}_{\mathsf{P};X}$ that is labelled with x_i.

The flowchart $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$, depicted in Fig. 4.11, is defined as follows:

- The set of nodes: $N \uplus \{v_{\mathsf{A}}\} \uplus \{v_e \;:\; e \in E_{[x]_{\mathcal{E}}}\}$. We say that the node v_{A} *corresponds* to the assignment A in the program $\mathsf{A};\mathsf{P}$.
- The set of edges: $E \uplus \{e_y \;:\; y \in O\} \cup \{e_x\}$, i.e. the edge e_x is added only if it was not already in E.
- The inputs and outputs of the old nodes stay the same. The labels, inputs and outputs of new nodes are:
  - $\lambda_N(v_{\mathsf{A}}) = o$, $\overrightarrow{\rho}(v_{\mathsf{A}}) = e_{x_1} e_{x_2} \cdots e_{x_k}$, $\overleftarrow{\rho}(v_{\mathsf{A}}) = e_x$;
  - $\lambda_N(v_e) = Mk\mathcal{E}$, $\overrightarrow{\rho}(v_e) = e_x$, $\overleftarrow{\rho}(v_e) = e$ for each $e \in E_{[x]_{\mathcal{E}}}$.
- The outputs are the same as the outputs of $\mathsf{Chart}_{\mathsf{P};X}$.
- The set $\mathsf{inputs}(\mathsf{Chart}_{\mathsf{P};X}) \backslash (\{e_x\} \cup E_{[x]_{\mathcal{E}}}) \cup \{e_y \;:\; y \in O\}$ is the set of input edges to $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$. The edges in $\mathsf{inputs}(\mathsf{Chart}_{\mathsf{P};X})$ retain their labels and $\mathsf{bb\_use}$-values. For each $y \in O$, the edge e_y is labelled with y and $\mathsf{bb\_use}$ is undefined for them.

A is $x := \mathcal{E}nc(k, y)$

If no input of $\mathsf{Chart}_{\mathsf{P};X}$ is labelled with x or $[x]_{\mathcal{E}}$, then $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ is equal to $\mathsf{Chart}_{\mathsf{P};X}$. Otherwise, it is constructed in the following way.

Let N, E, $E_{[x]_{\mathcal{E}}}$, e_x be defined as in the previous case. Similarly to e_x, let e_y, if it exists, be the input to $\mathsf{Chart}_{\mathsf{P};X}$ that is labelled with y.The flowchart $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$, depicted in Fig. 4.12, is defined as follows:

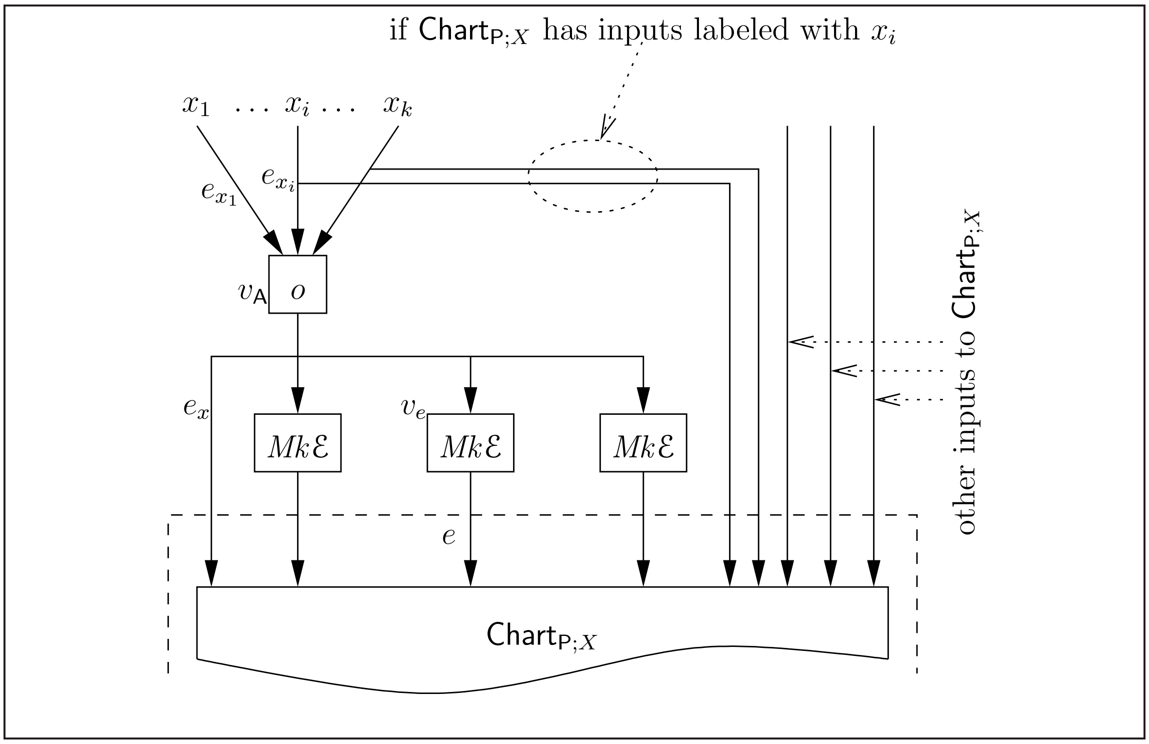


Figure 4.11: Adding $x := o(x_1, \ldots, x_k)$ to the flowchart

- The set of nodes: $N \uplus \{v_{\mathsf{A}}\} \uplus \{v_e \,:\, e \in E_{[x]_{\mathcal{E}}}\}$.
- The set of edges: $(E \cup \{e_x, e_y\}) \uplus \{e_{[k]_{\mathcal{E}}}\}$. Note that $e_{[k]_{\mathcal{E}}}$ is a new edge.
- The inputs and outputs of the old nodes stay the same. The labels, inputs and outputs of the new nodes are:
  - $\lambda_N(v_{\mathsf{A}}) = \mathcal{E}nc$, $\overrightarrow{\rho}(v_{\mathsf{A}}) = e_{[k]_{\mathcal{E}}} e_y$, $\overleftarrow{\rho}(v_{\mathsf{A}}) = e_x$;
  - $\lambda_N(v_e) = Mk\mathcal{E}$, $\overrightarrow{\rho}(v_e) = e_x$, $\overleftarrow{\rho}(v_e) = e$ for each $e \in E_{[x]_{\mathcal{E}}}$.
- The outputs are the same as the outputs of $\mathsf{Chart}_{\mathsf{P};X}$.
- The set $\mathsf{inputs}(\mathsf{Chart}_{\mathsf{P};X}) \backslash (\{e_x\} \cup E_{[x]_{\mathcal{E}}}) \cup \{e_{[k]_{\mathcal{E}}}, e_y\}$ is the set of input edges to $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$. The edges in $\mathsf{inputs}(\mathsf{Chart}_{\mathsf{P};X})$ retain their labels and $\mathsf{bb\_use}$-values. The edges $e_{[k]_{\mathcal{E}}}$ and e_y are labelled with $[k]_{\mathcal{E}}$ and y, respectively. $\mathsf{bb\_use}(e_{[k]_{\mathcal{E}}})$ is defined to be v_{A}.

A **is** $x := \mathcal{G}en()$

If no input of $\mathsf{Chart}_{\mathsf{P};X}$ is labelled with x or $[x]_{\mathcal{E}}$, then $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ is equal to $\mathsf{Chart}_{\mathsf{P};X}$. Otherwise, it is constructed in the following way.

Let N, E, e_x and $E_{[x]_{\mathcal{E}}}$ be defined as before. The flowchart $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$, depicted in Fig. 4.13, is defined as follows:

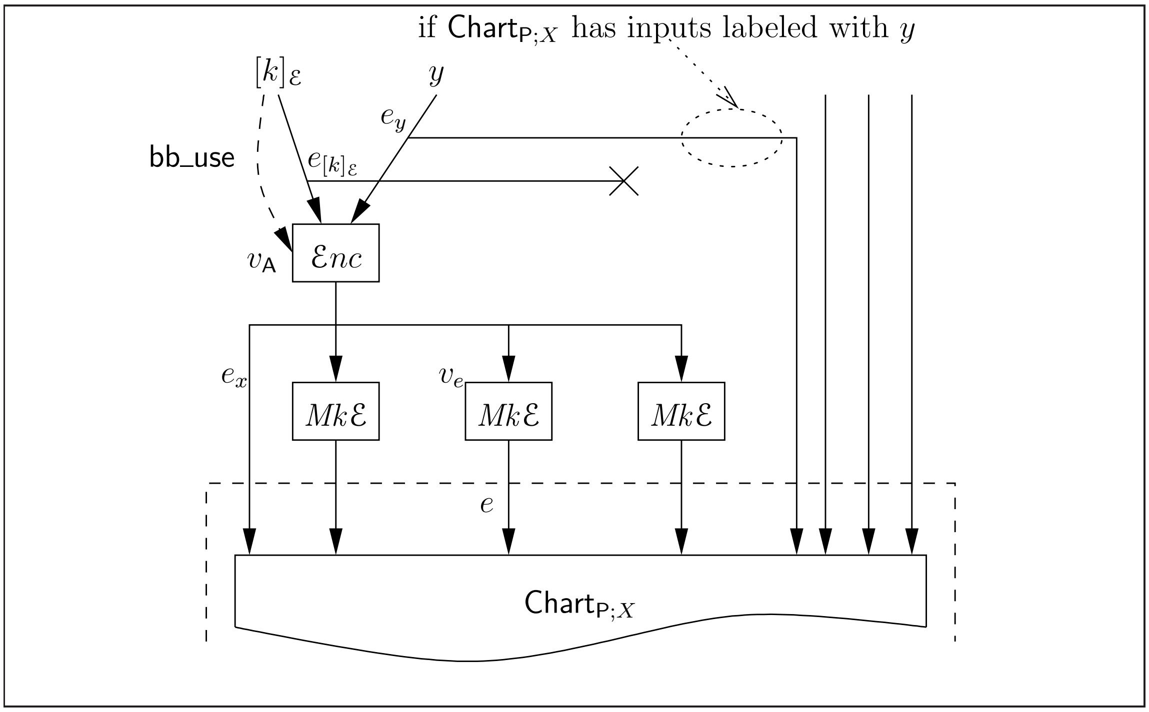


Figure 4.12: Adding $x := \mathcal{E}nc(k, y)$ to the flowchart

- The set of nodes is $N \uplus \{v_{\mathsf{A}}\} \uplus \{v_e : e \in E_{[x]_{\mathcal{E}}}\}$, if the edge e_x exists, and $N \uplus \{v_e : e \in E_{[x]_{\mathcal{E}}}\}$ otherwise.
- The set of edges is E.
- The inputs and outputs of the old nodes stay the same. The labels, inputs and outputs of the new nodes are:
  - If the edge e_x exists, then $\lambda_N(v_{\mathsf{A}}) = \mathcal{G}en_{\mathrm{val}}$, $\overrightarrow{\rho}(v_{\mathsf{A}}) = \epsilon$ (the empty sequence of edges), $\overleftarrow{\rho}(v_{\mathsf{A}}) = e_x$;
  - $\lambda_N(v_e) = \mathcal{G}en$, $\overrightarrow{\rho}(v_e) = \epsilon$, $\overleftarrow{\rho}(v_e) = e$ for each $e \in E_{[x]_{\mathcal{E}}}$.
- The outputs are the same as the outputs of $\mathsf{Chart}_{\mathsf{P};X}$.
- The set $\mathsf{inputs}(\mathsf{Chart}_{\mathsf{P};X}) \backslash (\{e_x\} \cup E_{[x]_{\mathcal{E}}})$ is the set of input edges to $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$. They retain their labels and bb_use-values.

Our intuition tells us, that if the flowchart contains several nodes labeled with $\mathcal{G}en$ and maybe also one that is labeled with $\mathcal{G}en_{\mathrm{val}}$, all corresponding to the same assignment $x := \mathcal{G}en$, then the interpretation should invoke the key generation operation $[\![\mathcal{G}en]\!]$ once for each node. Actually, this is not necessarily the case. The configurations defined in the next section contain a partition of the set of all these nodes and the key generation operation is invoked *once for each class* of the partition. For example, in the configurations corresponding to the distributions (4.25) and (4.26), this partition puts all nodes to the same class. In the intermediate configurations, the number of classes may be higher.

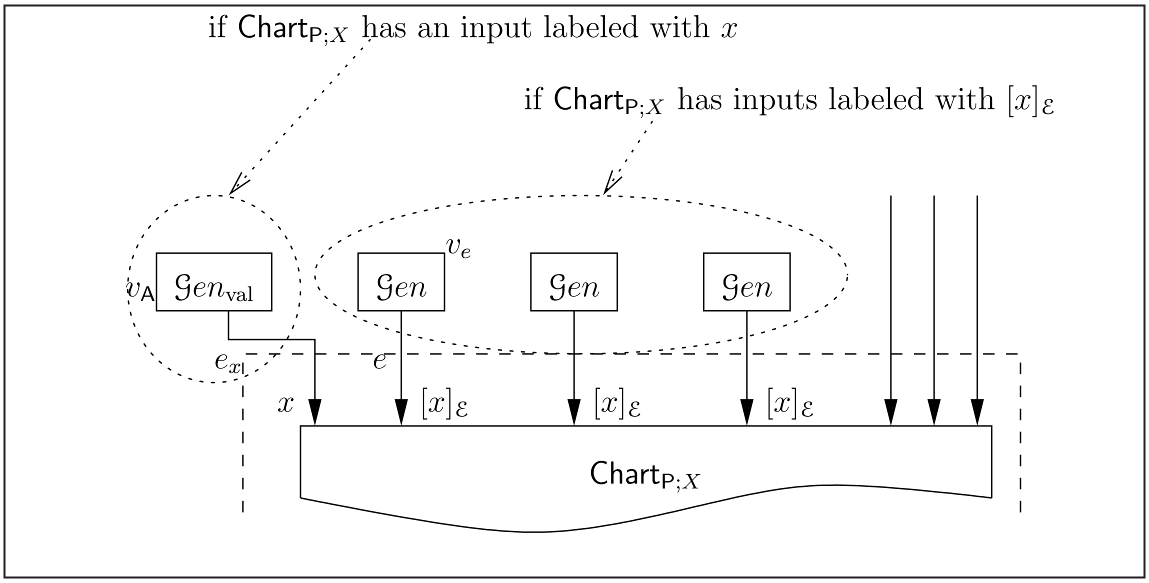


Figure 4.13: Adding $x := \mathcal{G}en()$ to the flowchart

A is $x := y$

If no input of $\mathsf{Chart}_{\mathsf{P};X}$ is labelled with x or $[x]_{\mathcal{E}}$, then $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ is equal to $\mathsf{Chart}_{\mathsf{P};X}$. Otherwise, it is constructed in the following way.

Let N, E, e_x and $E_{[x]_{\mathcal{E}}}$ be defined as before. Also, let e_y, if it exists, be the input to $\mathsf{Chart}_{\mathsf{P};X}$ that is labelled with y. Let $E_{\mathsf{bb\_use}} \subseteq E_{[x]_{\mathcal{E}}}$ be the set of such edges e labelled with $[x]_{\mathcal{E}}$, for which there exists an input e' of $\mathsf{Chart}_{\mathsf{P};X}$ that is labelled with $[y]_{\mathcal{E}}$ and satisfies $\mathsf{bb\_use}(e') = \mathsf{bb\_use}(e)$. The flowchart $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$, depicted in Fig. 4.14, is defined as follows:

- The set of nodes in N.
- If the chart $\mathsf{Chart}_{\mathsf{P};X}$ has an input edge that is labelled with either x or y then the set of edges is $E \backslash E_{\mathsf{bb\_use}} \backslash \{e_x\} \cup \{e_y\}$. Otherwise the set of edges is $E \backslash E_{\mathsf{bb\_use}}$.
- In the inputs of the nodes, e_x is replaced with e_y and each $e \in E_{\mathsf{bb\_use}}$ is replaced with the input e' that is labelled with $[y]_{\mathcal{E}}$ and has the same $\mathsf{bb\_use}$-value.
- The outputs of the flowchart are the same as the outputs of $\mathsf{Chart}_{\mathsf{P};X}$.
- If the chart $\mathsf{Chart}_{\mathsf{P};X}$ has an input edge that is labelled with either x or y then the set of inputs of the flowchart is $\mathsf{inputs}(\mathsf{Chart}_{\mathsf{P};X}) \backslash E_{\mathsf{bb\_use}} \backslash \{e_x\} \cup \{e_y\}$, otherwise the set of inputs is $\mathsf{inputs}(\mathsf{Chart}_{\mathsf{P};X}) \backslash E_{\mathsf{bb\_use}}$. The edge e_y is labelled with y. The input edges of $\mathsf{Chart}_{\mathsf{P};X}$ that were labelled with $[x]_{\mathcal{E}}$ will be labelled with $[y]_{\mathcal{E}}$. Other input edges retain their labels. Also, $\mathsf{bb\_use}$-values of the input edges stay the same.

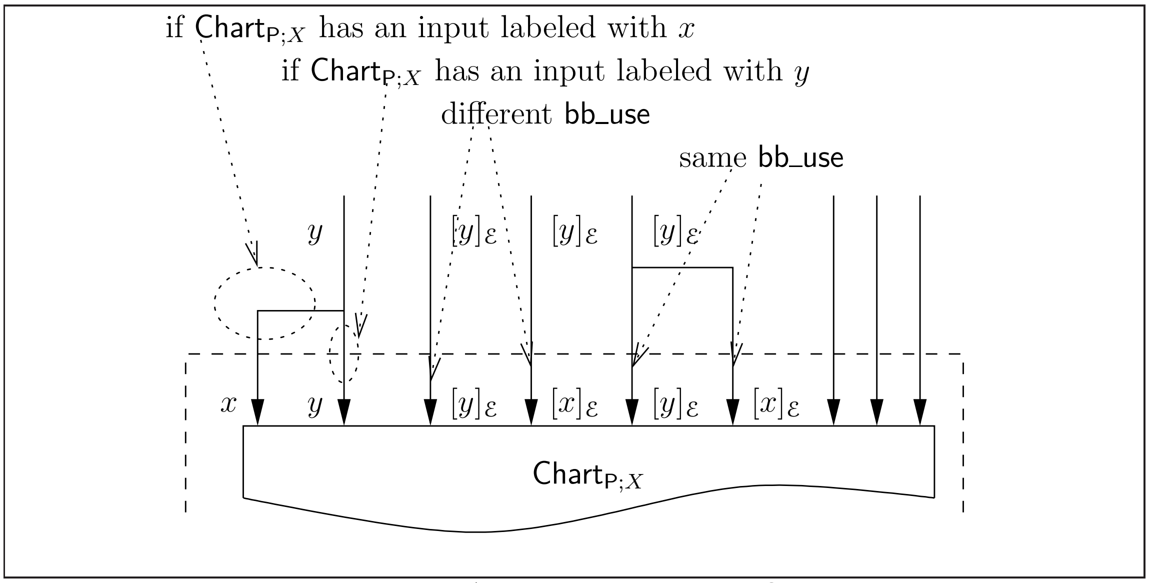


Figure 4.14: Adding $x := y$ to the flowchart

A **is** $\langle x_1, \ldots, x_k\rangle := b\,?\langle y_1, \ldots, y_k\rangle : \langle z_1, \ldots, z_k\rangle$

If no input of $\mathsf{Chart}_{\mathsf{P};X}$ is labelled with some x_i or $[x_i]_\mathcal{E}$ for $i \in \{1, \ldots, k\}$, then $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ is equal to $\mathsf{Chart}_{\mathsf{P};X}$. Otherwise, it is constructed in the following way.

Let N and E be defined as before. For each x_i, let e_{x_i} (if it exists) be the input edge of $\mathsf{Chart}_{\mathsf{P};X}$ that is labelled with x_i and let $E_{[x_i]_\mathcal{E}}$ be the set of input edges of $\mathsf{Chart}_{\mathsf{P};X}$ that are labelled with $[x_i]_\mathcal{E}$. Let $L \subseteq \{x_1, \ldots, x_k\}$ be the set of all x_i, such that e_{x_i} exists. Because of the shape of the unrolled programs, the flowchart $\mathsf{Chart}_{\mathsf{P};X}$ does not have any input edges that are labelled with b or with any y_i, $[y_i]_\mathcal{E}$, z_i, $[z_i]_\mathcal{E}$.

A part of the flowchart $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ is depicted on Fig. 4.15. Namely, this figure shows the additional nodes and edges (compared to $\mathsf{Chart}_{\mathsf{P};X}$) that go to the inputs of $\mathsf{Chart}_{\mathsf{P};X}$ labelled with x_i or $[x_i]_\mathcal{E}$. There is a similar construction for each index $i \in \{1, \ldots, k\}$. However, there is only a single input edge of $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ labelled with b. Additionally, the input edges of $\mathsf{Chart}_{\mathsf{P};X}$ that are labelled with something else than $x_1, \ldots, x_k$ and $[x_1]_\mathcal{E}, \ldots, [x_k]_\mathcal{E}$, are also the input edges of $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$.

Formally, the parts of $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ are the following:

- The set of nodes is $N \uplus \{v_{x_i} : x_i \in L\} \uplus \{v_{e_i} : i \in \{1, \ldots, k\}, e_i \in E_{[x_i]_\mathcal{E}}\}$.
- The set of edges is
$$E \uplus \{e_b\} \uplus \{e_{y_i}, e_{z_i} : x_i \in L\} \uplus \{e_{i,[y_i]_\mathcal{E}}, e_{i,[z_i]_\mathcal{E}} : i \in \{1, \ldots, k\}, e_i \in E_{[x_i]_\mathcal{E}}\} .$$
- The inputs and outputs of the old nodes stay the same. The labels, inputs and outputs of the new nodes are:
  - new nodes v_{x_i}, where $x_i \in L$, are labelled with $\mathsf{?:}$, all other new nodes are labelled with $[\mathsf{?:}]_\mathcal{E}$;

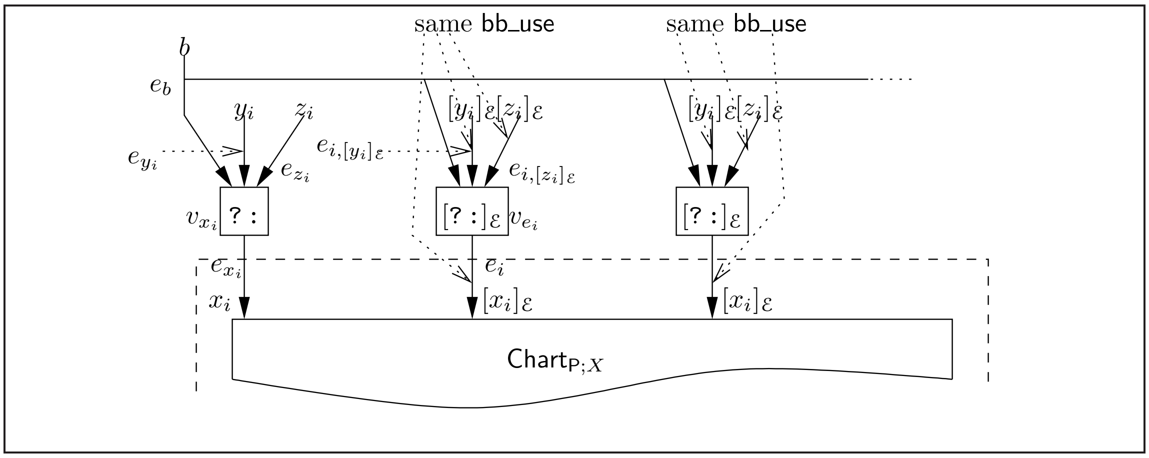


Figure 4.15: Adding $\langle x_1,\ldots,x_k\rangle := b\,?\langle y_1,\ldots,y_k\rangle : \langle z_1,\ldots,z_k\rangle$ to the flowchart (a part)

- $\overrightarrow{\rho}(v_{x_i}) = e_b e_{y_i} e_{z_i}$, $\overleftarrow{\rho}(v_{x_i}) = e_{x_i}$ for each $x_i \in L$;
- $\overrightarrow{\rho}(v_{e_i}) = e_b e_{i,[y_i]_\mathcal{E}} e_{i,[z]_\mathcal{E}}$, $\overleftarrow{\rho}(v_{e_i}) = e_i$ for each $i \in \{1,\ldots,k\}$ and $e_i \in E_{[x_i]_\mathcal{E}}$.

- The outputs of the flowchart are the same as the outputs of $\mathsf{Chart}_{\mathsf{P};X}$.
- The set of inputs of the flowchart is

$$\mathsf{inputs}(\mathsf{Chart}_{\mathsf{P};X})\backslash(\{e_{x_i} : x_i \in L\} \cup \bigcup_{i=1}^{k} E_{[x_i]_\mathcal{E}}) \cup \{e_b\} \cup \{e_{y_i}, e_{z_i} : x_i \in L\} \uplus \{e_{i,[y_i]_\mathcal{E}}, e_{i,[z_i]_\mathcal{E}} : i \in \{1,\ldots,k\}, e_i \in E_{[x_i]_\mathcal{E}}\}\ .$$

The edge e_b is labelled with b. The edges e_{y_i} and e_{z_i} are labelled with y_i and z_i, respectively. The edges $e_{i,[y_i]_\mathcal{E}}$ and $e_{i,[z_i]_\mathcal{E}}$ are labelled with $[y_i]_\mathcal{E}$ and $[z_i]_\mathcal{E}$, respectively. $\mathsf{bb\_use}(e_{i,[y_i]_\mathcal{E}})$ and $\mathsf{bb\_use}(e_{i,[z_i]_\mathcal{E}})$ are defined to be equal to $\mathsf{bb\_use}(e_i)$. Other input edges retain their labels and bb_use-values.

For each of the input edges of $\mathsf{Chart}_{\mathsf{P};X}$ there is a certain set of input edges of $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$, such that they "define" the value on that edge. Namely, for each e that is an input edge of $\mathsf{Chart}_{\mathsf{P};X}$ we let $\mathsf{origins}(e)$ be a certain subset of input edges of $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$. If e is also an input to $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$, then $\mathsf{origins}(e) = \{e\}$. Otherwise e is the out-edge of some node v of the chart $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$, we let $\mathsf{origins}(e)$ be the set of input edges to $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ that lead to v.

Given two flowcharts $\mathsf{Chart}_{\mathsf{P};X}$ and $\mathsf{Chart}_{\mathsf{P};Y}$ for the same sequence of assignments and vectorised choices P, we denote their union (i.e. one takes the union of the sets of nodes and the sets of edges; the sources and targets of edges and labels of nodes and inputs and outputs remain the same) by $\mathsf{2Chart}_{\mathsf{P};X,Y}$. It is again a flowchart. This flowchart is the basis of the structure that we mentioned in Sec. 4.3.1. In the following, we are going to define its interpretation (with respect to a certain set of parameters).

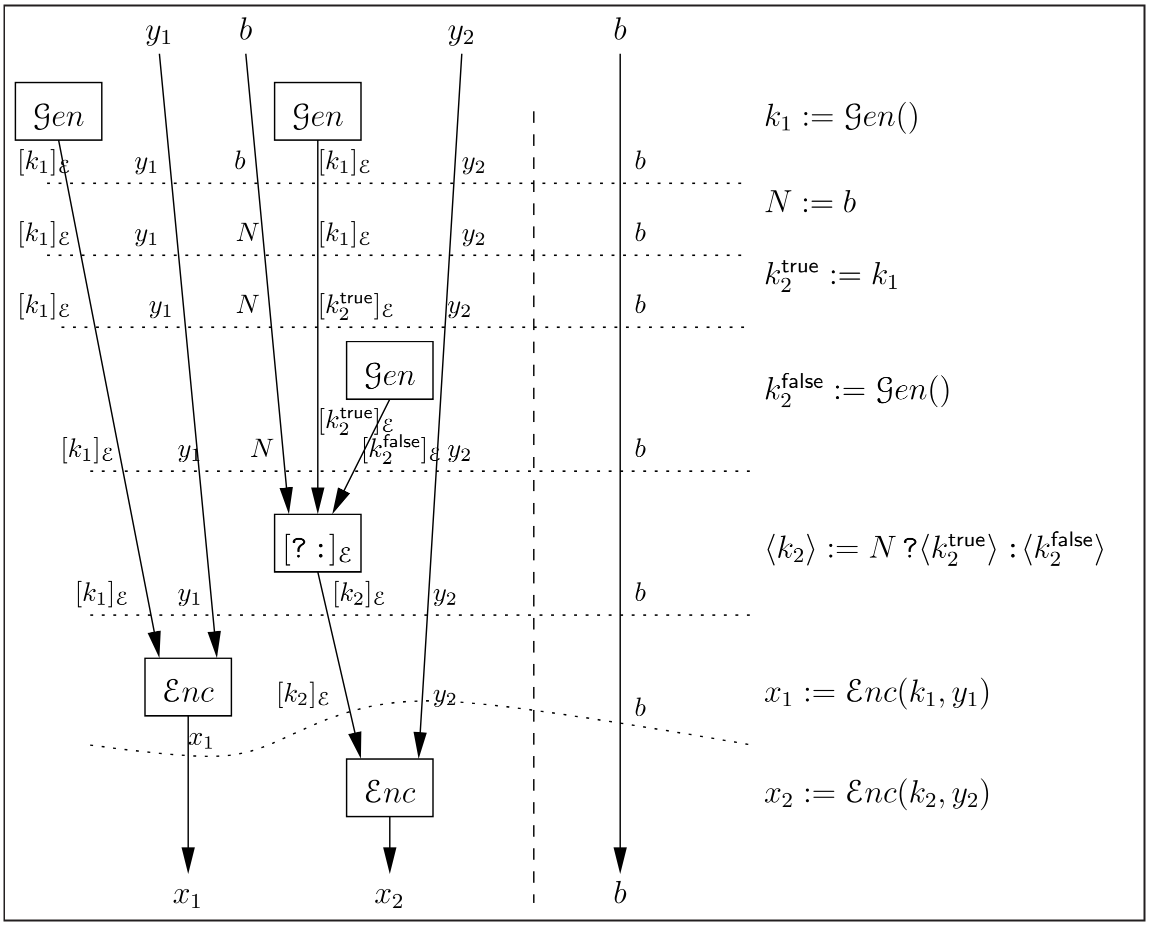


Figure 4.16: The flowchart of the running example

The running example. In the running example we take $X = \{x_1, x_2\}$ and $Y = \{b\}$. The flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$ of the unrolled program in Fig. 4.10 is given in Fig. 4.16. The chart $\mathsf{Chart}_{\mathsf{P};X}$ lies to the left of the dashed line, the chart $\mathsf{Chart}_{\mathsf{P};Y}$ (a single arrow) to the right. In the right edge of the figure the statements of the program P, corresponding to the nodes in the left, are given. The figure also shows, how the flowchart has been constructed, statement by statement. The construction starts with the last statement $x_2 := \mathcal{E}nc(k_2, y_2)$, the flowchart corresponding to this statement is below the lowermost dotted line. The labels of the input edges of this flowchart are given in smaller print. Each successive dotted line corresponds to adding a statement. Note that the flowchart does not change while adding the 4th and 3rd statements of the program in Fig. 4.10.

4.4.3 Configurations of a Flowchart

A configuration of the flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$ represents one particular value of the "changeable parameters" (see Sec. 4.3.1). Given a flowchart G and its configuration C, we will define the interpretation $[\![G, C]\!]$. This interpretation is a distribution over labellings of the output edges of G with the values from $\widetilde{\mathbf{Val}}_n$.

We present the structure of a configuration and the corresponding interpretation step by step — i.e. we start by giving as little of the structure as possible, but enough to be able to define the interpretation. We then introduce another components of the configuration and show how to change the interpretation to incorporate them.

A *configuration* of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ is a tuple with the following components:

- A partition $\mathsf{InpParts}(C)$ of the set of input edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$.
- For each assignment or vectorised choice A_i in P (here $i \in \{1,\ldots,s\}$, where s is the length of P), a partition $\mathsf{OpParts}_i(C)$ of the set of nodes in $\mathsf{2Chart}_{\mathsf{P};X,Y}$ that correspond to this assignment (the nodes labelled with $Mk\mathcal{E}$ are not considered here). More concretely, the nodes contained in the parts of $\mathsf{OpParts}_i(C)$ are the following nodes in Figs. 4.11–4.15:
  - On Fig. 4.11 and Fig. 4.12 — the node v_{A}.
  - On Fig. 4.13 and Fig. 4.15 — all nodes depicted on these figures.

Let $Mk\mathcal{E}\mathrm{Set}_i$ denote the set of nodes of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ that are labelled with $Mk\mathcal{E}$ and whose inputs are created at nodes corresponding to the assignment A_i. Fig. 4.17 presents a probabilistic algorithm that defines (a distribution of) the quantity λ_E — a labelling of the edges of the flowchart with the elements of $\widetilde{\mathbf{Val}}_n$. This distribution of λ_E, restricted to the output edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$, is defined to be the interpretation of the flowchart with the configuration C. The interpretation $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0$ (the subscript 0 denotes the preliminarity of this definition of interpretation of flowcharts) can therefore be considered to have the type $\mathbf{StrOut}_n^{X;Y}$.

The main element of the algorithm in Fig. 4.17 is the subroutine do_assignment. Notice, how the partition $\mathsf{OpParts}_i(C)$ is used — the outer loop (lines 1–23) of the subroutine cycles over all parts in $\mathsf{OpParts}_i(C)$ (and not over all nodes corresponding to the assignment or vectorised choice A_i). Inside this outer loop, a new value is computed and then this value is assigned to all elements inside that part in the inner loop (lines 15–22).

There exists a configuration C_{L}, such that the interpretation of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ with C_{L} is equal to the n-th component of (4.25). It is the following:

- $\mathsf{InpParts}(C_{\mathrm{L}})$ puts all input edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ to the same part.
- $\mathsf{OpParts}_i(C_{\mathrm{L}})$ puts all nodes in $\mathsf{2Chart}_{\mathsf{P};X,Y}$ corresponding to A_i to the same part.

There also exists a configuration C_{R}, such that the interpretation of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ with C_{R} is equal to the n-th component of (4.26). It is the following:

- $\mathsf{InpParts}(C_{\mathrm{R}})$ puts all input edges of $\mathsf{Chart}_{\mathsf{P};X}$ to one part and all input edges of $\mathsf{Chart}_{\mathsf{P};Y}$ to another.

algorithm defining λ_E:

call create_inputs
for $i = 1$ to s **do**
    call do_assignment(i)
end for

subroutine create_inputs

for all $E' \in \mathsf{InpParts}(C)$ **do**
    generate $S_n \in \mathbf{State}_n$ according to D_n
    for all $e \in E'$ **do**
        define $\lambda_E(e) = S_n(\lambda_I(e))$
    end for
end for

subroutine do_assignment(i)

for all $N' \in \mathsf{OpParts}_i(C)$ **do**
    let v be an element of N'
    let $e_1, \ldots, e_k$ be the in-edges of v (from left to right)
    if $\lambda_N(v) = \mathcal{E}nc$ **then**
        generate $x \in \mathbf{Val}_n$ by feeding $\lambda_E(e_2)$ to the black box $\lambda_E(e_1)$
    else if $\lambda_N(v) = \mathcal{G}en$ or $\lambda_N(v) = \mathcal{G}en_{\mathrm{val}}$ **then**
        generate $x \in \mathbf{Val}_n$ according to the distribution $[\![\mathcal{G}en]\!]_n$
    else if $\lambda_N(v) = \mathsf{?:}$ **then**
        let x be $\lambda_E(e_2)$, if $\lambda_E(e_1) = \mathsf{true}$, and $\lambda_E(e_3)$ otherwise
    else if $\lambda_N(v) = [\mathsf{?:}]_{\mathcal{E}}$ **then**
        let x be $\lambda_E(e_2)$, if $\lambda_E(e_1) = \mathsf{true}$, and $\lambda_E(e_3)$ otherwise
    else
        generate $x \in \mathbf{Val}_n$ according to $[\![\lambda_N(v)]\!]_n(\lambda_E(e_1), \ldots, \lambda_E(e_k))$
    end if
    for all $v \in N'$ **do**
        let $e = \overleftarrow{\rho}(v)$
        if $\lambda_N(v) \neq \mathcal{G}en$ **then**
            define $\lambda_E(e) := x$
        else
            define $\lambda_E(e)$ as a black box $[\![\mathcal{E}nc]\!]_n(x, \cdot)$
        end if
    end for
end for
for all $v \in Mk\mathcal{E}\mathrm{Set}_i$ **do**
    let e be the in-edge and e' be the out-edge of v
    define $\lambda_E(e')$ as the black box $[\![\mathcal{E}nc]\!]_n(\lambda_E(e), \cdot)$
end for

Figure 4.17: Interpretation of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ with the configuration C — initial version

- $\mathsf{OpParts}_i(C_{\mathrm{R}})$ puts all nodes in $\mathsf{Chart}_{\mathsf{P};X}$ corresponding to A_i to one part and all nodes in $\mathsf{Chart}_{\mathsf{P};Y}$ corresponding to A_i to another.

Note that the subroutine do_assignment in Fig. 4.17 is nondeterministic — on the 2nd line it chooses an element of the set of nodes N', but it is not fixed, which one. Different nodes may have different labels on their in-edges, therefore the interpretation of flowcharts has not generally been uniquely defined. For configurations C_{L} and C_{R} it has been uniquely defined — all nodes in the same set $N' \in \mathsf{OpParts}_i(C_{\mathrm{L}})$ or $N' \in \mathsf{OpParts}_i(C_{\mathrm{R}})$ have same labels on their in-edges. Generally, however, we have to put some *hygiene conditions* on the configurations of the flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$.

Given a configuration C, we define a partition $\mathsf{edgeParts}(C)$ over the edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$. It is defined in such way, that if two edges e, e' belong to the same part of $\mathsf{edgeParts}(C)$, then $\lambda_E(e) = \lambda_E(e')$. The partition $\mathsf{edgeParts}(C)$ contains the following parts:

- For each part $E' \in \mathsf{InpParts}(C)$ and each $x \in \widetilde{\mathbf{Var}}$, the set of all $e \in E'$, where $\lambda_I(e) = x$, is a part in $\mathsf{edgeParts}(C)$.
- For each $i \in \{1, \ldots, s\}$ and each $N' \in \mathsf{OpParts}_i(C)$, the set of out-edges of the nodes in N' constitute a part in $\mathsf{edgeParts}(C)$.

We have introduced the following notational convention — the names of base components of configuration begin with capital letters, the names of derived components begin with small letters.

The hygiene condition put on configurations C is the following:

- Let $i \in \{1, \ldots, s\}$, $N' \in \mathsf{OpParts}_i(C)$ and $v, v' \in N'$. Let e be an in-edge of v and e' be an in-edge of v', such that they have the same position in the sequences $\overrightarrow{\rho}(v)$ and $\overrightarrow{\rho}(v')$. Then e and e' must belong to the same part in $\mathsf{edgeParts}(C)$.

It is clear that C_{L} and C_{R} satisfy this condition.

We are now going to extend the configurations with more components. The components introduced so far do not yet allow us to introduce the necessary "small steps" for turning C_{L} to C_{R}.

$\mathsf{InpKeys}(C) \subseteq \mathsf{inputs}(\mathsf{2Chart}_{\mathsf{P};X,Y})$ is a component of C. All members of the set $\mathsf{InpKeys}(C)$ must be labelled with some $[k]_{\mathcal{E}}$ for $k \in \mathbf{Var}$ and for each $e \in \mathsf{InpKeys}(C)$ labelled with $[k]_{\mathcal{E}}$, the part in partition $\mathsf{InpParts}(C)$ that contains the edge e must only contain edges labelled with $[k]_{\mathcal{E}}$.

The construction of λ_E in Fig. 4.17 corresponds to having $\mathsf{InpKeys}(C) = \emptyset$. For the general case, Fig. 4.18 gives the necessary change to Fig. 4.17. We see that $\mathsf{InpKeys}(C)$ is somehow related to the keys-part of the elements of $\mathcal{PF}(\mathbf{Var})$ (similarly, $\mathsf{InpParts}(C)$ is somehow related to the indeps-part of these elements).

$\mathsf{BoxNull}(C) \subseteq \mathsf{InpKeys}(C) \cup \mathsf{Nodes}(\mathsf{2Chart}_{\mathsf{P};X,Y})$ is a component of C. All *nodes* in $\mathsf{BoxNull}(C)$ must be labelled with $\mathcal{G}en$.

Replace the subroutine create_inputs by

1: **for all** $E' \in \mathsf{InpParts}(C)$ **do**
2:   **if** $E' = \{e_1, \ldots, e_k\}$ and $e_1, \ldots, e_k \in \mathsf{InpKeys}(C)$ **then**
3:     generate $x \in \mathbf{Val}_n$ according to $[\![\mathcal{G}en]\!]_n$
4:     **for all** $e \in E'$ **do**
5:       define $\lambda_E(e)$ as a black box $[\![\mathcal{E}nc]\!]_n(x, \cdot)$
6:     **end for**
7:   **else**
8:     generate $S_n \in \mathbf{State}_n$ according to D_n
9:     **for all** $e \in E'$ **do**
10:       define $\lambda_E(e) = S_n(\lambda_I(e))$
11:     **end for**
12:   **end if**
13: **end for**

Figure 4.18: Adding $\mathsf{InpKeys}(C)$ to the interpretation of flowcharts

Replace the 20th line of the subroutine do_assignment in Fig. 4.17 by

if $v \in \mathsf{BoxNull}(C)$ **then**
  define $\lambda_E(e)$ as a black box $[\![\mathcal{E}nc]\!]_n(x, \mathbf{0}^{\ell(n)})$
else
  define $\lambda_E(e)$ as a black box $[\![\mathcal{E}nc]\!]_n(x, \cdot)$
end if

Also, replace the 5th line of the subroutine create_inputs on Fig. 4.18 with almost the same excerpt (only replace the check "$v \in \mathsf{BoxNull}(C)$" with the check "$e \in \mathsf{BoxNull}(C)$").

Figure 4.19: Adding $\mathsf{BoxNull}(C)$ to the interpretation of flowcharts

The construction of λ_E in Figures 4.17 and 4.18 corresponds to having $\mathsf{BoxNull}(C)$ equal the empty set. For the general case, Fig. 4.19 gives the necessary changes to these figures. We see that $\mathsf{BoxNull}(C)$ is somehow related to the repetition-concealedness of the encryption primitive.

In Fig. 4.19, the black box $[\![\mathcal{E}nc]\!]_n(x, \mathbf{0}^{\ell(n)})$ operates as follows. If an input is submitted to this black box, then it discards that input, encrypts the bit-string $\mathbf{0}^{\ell(n)}$ and returns the result of that encryption.

One can say that the parts $\mathsf{InpKeys}(C)$ and $\mathsf{BoxNull}(C)$ somehow control the *inputs* to the flowchart (the result of generating a new key can be seen as an input, too). The next two components, $\mathsf{EncNull}(C)$ and $\mathsf{IfNewKeys}(C)$, control some aspects of the computation.

$\mathsf{EncNull}(C) \subseteq \mathsf{Nodes}(\mathsf{2Chart}_{\mathsf{P};X,Y})$ is a component of the configuration C. All nodes in $\mathsf{EncNull}(C)$ must be labelled with $\mathcal{E}nc$. The construction of λ_E presented this far corresponds to having $\mathsf{EncNull}(C) = \emptyset$. The modifications for the general case are given on Fig. 4.20.

Before defining the component $\mathsf{IfNewKeys}(C)$ and explaining, how it affects the

Replace the 5th line of the subroutine do_assignment on Fig. 4.17 by
if $v \in \mathsf{EncNull}(C)$ **then**
  generate $x \in \mathbf{Val}_n$ by feeding $\mathbf{0}^{\ell(n)}$ to the black box $\lambda_E(e_1)$
else
  generate $x \in \mathbf{Val}_n$ by feeding $\lambda_E(e_2)$ to the black box $\lambda_E(e_1)$
end if

Figure 4.20: Adding $\mathsf{EncNull}(C)$ to the interpretation of flowcharts

Replace the 11th line of the subroutine do_assignment on Fig. 4.17 by
if $v \in \mathsf{IfNewKeys}(C)$ **then**
  generate $x' \in \mathbf{Val}_n$ according to the distribution $[\![\mathcal{G}en]\!]_n$
  if $\overleftarrow{\rho}(v) \in \mathsf{edgeNull}(C)$ **then**
    let x be the black box $[\![\mathcal{E}nc]\!]_n(x, \mathbf{0}^{\ell(n)})$
  else
    let x be the black box $[\![\mathcal{E}nc]\!]_n(x, \cdot)$
  end if
else
  let x be $\lambda_E(e_2)$, if $\lambda_E(e_1) = \mathsf{true}$, and $\lambda_E(e_3)$ otherwise
end if

Figure 4.21: Adding $\mathsf{IfNewKeys}(C)$ to the interpretation of flowcharts

interpretation of the flowchart, we have to define another derived component of the configuration C. Let $\mathsf{edgeNull}(C)$ be the smallest set of edges of the flowchart, such that the following conditions are satisfied.

- If $e \in \mathsf{BoxNull}(C)$ is an edge, then $e \in \mathsf{edgeNull}(C)$.
- If $v \in \mathsf{BoxNull}(C)$ is a node, then $\overleftarrow{\rho}(v) \in \mathsf{edgeNull}(C)$.
- Let v be a node of the flowchart that is labelled with $[\mathsf{?:}]_{\mathcal{E}}$. Let $\overrightarrow{\rho}(v) = e_b e_{\mathsf{true}} e_{\mathsf{false}}$. If both $e_{\mathsf{true}} \in \mathsf{edgeNull}(C)$ and $e_{\mathsf{false}} \in \mathsf{edgeNull}(C)$, then also $\overleftarrow{\rho}(v) \in \mathsf{edgeNull}(C)$.

The set of $\mathsf{edgeNull}(C)$ contains such edges e, where the computation of λ_E assigns a black box that encrypts $\mathbf{0}^{\ell(n)}$ to the edge e.

$\mathsf{IfNewKeys}(C) \subseteq \mathsf{Nodes}(\mathsf{2Chart}_{\mathsf{P};X,Y})$ is a component of the configuration C. All nodes in $\mathsf{IfNewKeys}(C)$ must be labelled with $[\mathsf{?:}]_{\mathcal{E}}$. The construction of λ_E presented this far corresponds to having $\mathsf{IfNewKeys}(C) = \emptyset$. The modifications for the general case are given in Fig. 4.21. We see that $\mathsf{IfNewKeys}(C)$ is somehow related to the which-key-concealedness of the encryption primitive.

Summary

The base components of a configuration C as used as follows:

- $\mathsf{InpParts}(C)$ determines the input partition to $\mathsf{2Chart}_{\mathsf{P};X,Y}$.
- $\mathsf{OpParts}_i(C)$ fixes, which nodes denote the same operation and which nodes denote different operations.
- If $e \in \mathsf{InpKeys}(C)$, then the value on e is a black box encrypting with a "real" key, not just with a value of a variable.
- If $e \in \mathsf{BoxNull}(C)$ or $v \in \mathsf{BoxNull}(C)$, then the encrypting black box that is generated at e or v does not encrypt its input, but encrypts the constant $\mathbf{0}^{\ell(n)}$.
- If $v \in \mathsf{EncNull}(C)$, then the operation $\mathcal{E}nc$ at v encrypts $\mathbf{0}^{\ell(n)}$, not its input.
- If $v \in \mathsf{IfNewKeys}(C)$, then the operation $[\mathsf{?}:]_{\mathcal{E}}$ at v generates a new black box as its output, instead of using its inputs.

4.4.4 Known and Unknown Values in a Flowchart

The configurations defined so far are still not enough for transforming the configuration C_{L} to the configuration C_{R}. When looking at the analysis rules (4.28) and (4.29), we see that their antecedents (pertaining to $\mathsf{indeps}(\mathcal{X})$) never contain both the variables y_i and z_i. Actually, in the explanation of rule (4.18) we said that for computing the values of the variables x_i after *merge* (i.e. after the vectorised choice), we only need to know the value of the guard variable and the values of the variables x_i in one of the branches (i.e. either the values of the variables y_i or the values of the variables z_i).

Here we want to introduce the interpretation of flowcharts, if we only give values to a subset of its input edges. These subsets are not arbitrary ones, as the first thing we are going to define the set of allowed subsets.

Given the program $\mathsf{P} = \mathsf{A}_1; \cdots; \mathsf{A}_s$ and the set $X \subseteq \widetilde{\mathbf{Var}}$, we define the sets $\mathsf{KV}_i^X \subseteq \mathcal{P}(\mathbf{Var})$, where $0 \leq i \leq s$. The set KV_s^X is equal to $\{X\}$, the set KV_0^X contains the possible sets of input variables that may be initialised.

The sets KV_i^X are defined inductively, in the order of decreasing i. As already mentioned, $\mathsf{KV}_s^X = \{X\}$ — to compute the values of the variables in X we need to know the values of the variables in X at the end of the program and nothing else.

Suppose that KV_i^X has been defined. If A_i is an assignment $x := o(x_1, \ldots, x_k)$ (including all possible "special cases"), then

$$\mathsf{KV}_{i-1}^X = \{\langle\!\langle x \overset{?}{\in} Z \;\mathbf{?}\; Z\backslash\{x\} \cup \{x_1, \ldots, x_k\} \;\mathbf{:}\; Z \rangle\!\rangle \;:\; Z \in \mathsf{KV}_i^X\} \ . \qquad (4.32)$$

I.e. if we can compute the values of X at the end of the program from the values of Z after the assignment A_i, then, if Z contains x, we need the variables at the right hand side of A_i before the assignment A_i.

If A_i is $\langle x_1, \ldots, x_k \rangle := b \,?\langle y_1, \ldots, y_k \rangle : \langle z_1, \ldots, z_k \rangle$, then

$$\mathsf{KV}_{i-1}^X = \big\{ Z \backslash \{x_1, \ldots, x_k\} \cup \bigcup_{j=1}^{k} \langle\!\langle x_j \overset{?}{\in} Z \;\textbf{?}\; \{b, y_j\} \;\textbf{:}\; \emptyset \rangle\!\rangle \;:\; Z \in \mathsf{KV}_i^X \big\} \cup$$
$$\big\{ Z \backslash \{x_1, \ldots, x_k\} \cup \bigcup_{j=1}^{k} \langle\!\langle x_j \overset{?}{\in} Z \;\textbf{?}\; \{b, z_j\} \;\textbf{:}\; \emptyset \rangle\!\rangle \;:\; Z \in \mathsf{KV}_i^X \big\} \;. \quad (4.33)$$

I.e. if we can compute the values of X at the end of the program from the values of Z after the vectorised choice A_i, then, if Z contains some x_j, we need the guard variable b and either the variables y_j or the variables z_j before the vectorised choice A_i.

Having defined the sets KV_i^X, we introduce a new component $\mathsf{KnownInps}(C)$ of the configuration C of the flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$. $\mathsf{KnownInps}(C)$ is a subset of the input edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$, such that

- There exists a set $S_X \in \mathsf{KV}_0^X$, such that an input edge of $\mathsf{Chart}_{\mathsf{P};X}$ belongs to $\mathsf{KnownInps}(C)$ iff it is labelled with x or $[x]_{\mathcal{E}}$ for some $x \in S_X$.
- There exists a set $S_Y \in \mathsf{KV}_0^Y$, such that an input edge of $\mathsf{Chart}_{\mathsf{P};Y}$ belongs to $\mathsf{KnownInps}(C)$ iff it is labelled with x or $[x]_{\mathcal{E}}$ for some $x \in S_Y$.

Computation of λ_E in Figures 4.17–4.21 is changed as described in Fig. 4.22.

The definition of $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$ (we now introduce the notation without the subscript 0; $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0$ still denotes the interpretation defined in Figures 4.17–4.21) also changes respectively. First, the type of the interpretation of the flowchart changes — the probability distribution is no longer over the set $\mathbf{StrOut}_n^{X;Y}$, but over the set $(\mathbf{StrOut}_n^{X;Y})_\perp$ — if λ_E assigns $\perp_{\mathrm{u}}$ to any of the output edges of the flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$, then the interpretation is considered to have picked $\perp$. But this is not the only case where $\perp$ appears. We want $\perp$ appear more often — often enough for the following claim to hold.

Let $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ be the set of all such configurations C of the flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$ that satisfy

- $\mathsf{InpParts}(C)$ puts all inputs of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ to the same part;
- $\mathsf{EncNull}(C) = \emptyset$;
- For each assignment A_i in P, $\mathsf{OpParts}_i(C)$ puts all nodes corresponding to A_i to the same part;
- $\mathsf{InpKeys}(C) = \emptyset$;
- $\mathsf{BoxNull}(C) = \emptyset$.

- In Fig. 4.18, $\lambda_E(e)$ is only defined for an edge e in the way given on the figure, if $e \in \mathsf{KnownInps}(C)$. If $e \notin \mathsf{KnownInps}(C)$, then $\lambda_E(e) := \bot_{\mathrm{u}}$. Here $\bot_{\mathrm{u}}$ is a new symbol, it denotes an unknown value.

- In subroutine do_assignment in Fig. 4.17, add between the 3rd and 4th lines:

  if $\lambda_N(v) \neq \mathsf{?:}$ and $\lambda_N(v) \neq [\mathsf{?:}]_{\mathcal{E}}$ **then**
    if $\lambda_E(e_i) = \bot_{\mathrm{u}}$ for any of the edges $e_1, \ldots, e_k$ **then**
      let x be $\bot_{\mathrm{u}}$
      go to line 15
    end if
  else
    if $\lambda_E(e_1) = \bot_{\mathrm{u}}$ or
        $\lambda_E(e_1) = \mathsf{true}$ and $\lambda_E(e_2) = \bot_{\mathrm{u}}$ or
        $\lambda_E(e_1) = \mathsf{false}$ and $\lambda_E(e_3) = \bot_{\mathrm{u}}$ **then**
      let x be $\bot_{\mathrm{u}}$
      go to line 15
    end if
  end if

- The subroutine do_assignment also constructs new black boxes. We define that attempting to create a black box that encrypts with $\bot_{\mathrm{u}}$ results again in $\bot_{\mathrm{u}}$.

Figure 4.22: Adding $\mathsf{KnownInps}(C)$ to the interpretation of flowcharts (1st part)

The only thing that can vary for the configurations $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ is the component $\mathsf{KnownInps}(C)$. It may be chosen freely (subject to the constraints that we gave when introducing the component $\mathsf{KnownInps}(C)$).

We want to define $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$ in such a way that the sum of distributions

$$\sum_{C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}} [\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!] \tag{4.34}$$

exists and is equal to the n-th component of (4.25). Recall that n is the security parameter that we have fixed at the beginning of Sec. 4.4.1.

Similarly, let $\mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$ be the set of all such configurations C of the flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$ that satisfy

- $\mathsf{InpParts}(C)$ puts all inputs of $\mathsf{Chart}_{\mathsf{P};X}$ to one part and all inputs of $\mathsf{Chart}_{\mathsf{P};Y}$ to another;

- $\mathsf{EncNull}(C) = \emptyset$;

- For each assignment A_i in P, $\mathsf{OpParts}_i(C)$ puts all nodes of $\mathsf{Chart}_{\mathsf{P};X}$ corresponding to A_i to the same part and all nodes of $\mathsf{Chart}_{\mathsf{P};Y}$ corresponding to A_i to another part;

- $\mathsf{InpKeys}(C) = \emptyset$;
- $\mathsf{BoxNull}(C) = \emptyset$.

Again, $\mathsf{KnownInps}(C)$ may vary freely. We want the sum of distributions

$$\sum_{C \in \mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}} [\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!] \tag{4.35}$$

to exist and to be equal to the n-th component of (4.26).

As an example, where the presented claims about $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ and $\mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$ do not hold, consider the program

$$\begin{array}{l} z_1 := y_1; z_2 := y_1; \\ \langle x_1, x_2 \rangle := b\,?\langle y_1, y_2 \rangle : \langle z_1, z_2 \rangle \end{array}$$

and suppose $X = \{x_1, x_2\}$ and $Y = \emptyset$. We then have $\mathsf{KV}^X_0 = \{\{b, y_1, y_2\}, \{b, y_1\}\}$. The flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$ has three input edges, labelled with b, y_1, y_2. The set $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ has two elements C_1 and C_2, corresponding to the two elements $\{b, y_1, y_2\}$ and $\{b, y_1\}$ of KV^X_0. If we denote $D_i = [\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C_i]\!]$, where $i \in \{1, 2\}$, then D_1 assigns no weight to $\perp$ and D_2 assigns as much weight to $\perp$ as D (the initial distribution) assigns to program states where the value of b is true. Therefore $D_1(\perp) + D_2(\perp)$ is smaller than 1 and $D_1 + D_2$ is undefined. In this example we would like D_1 to also assign some weight to $\perp$ — namely as much as D assigns to states where the value of b is false.

We now proceed to define for each $i \in \{1, \ldots, s\}$ the function $\mathbf{f}_{X;i}$ (if A_i is an assignment) or the functions $\mathbf{f}^{\mathsf{true}}_{X;i}$ and $\mathbf{f}^{\mathsf{false}}_{X;i}$ (if A_i is a vectorised choice) from the set KV^X_i to the set KV^X_{i-1}. With help of these functions, we are able to precisely specify, for which picks of λ_E the interpretation $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$ has to pick $\perp$.

Suppose that A_i is an assignment $x := o(x_1, \ldots, x_k)$. Let $Z \in \mathsf{KV}^X_i$. Then we define

$$\mathbf{f}_{X;i}(Z) := \langle\!\langle x \overset{?}{\in} Z \,\mathbf{?}\, Z \backslash \{x\} \cup \{x_1, \ldots, x_k\} \,\mathbf{:}\, Z \rangle\!\rangle \ .$$

Compare this definition with (4.32). We see that $\mathbf{f}_{X;i}(Z) \in \mathsf{KV}^X_{i-1}$.

Suppose that A_i is $\langle x_1, \ldots, x_k \rangle := b\,?\langle y_1, \ldots, y_k \rangle : \langle z_1, \ldots, z_k \rangle$. Let $Z \in \mathsf{KV}^X_i$. Then we define

$$\begin{aligned} \mathbf{f}^{\mathsf{true}}_{X;i}(Z) &:= Z \backslash \{x_1, \ldots, x_k\} \cup \bigcup_{j=1}^{k} \langle\!\langle x_j \overset{?}{\in} Z \,\mathbf{?}\, \{b, y_j\} \,\mathbf{:}\, \emptyset \rangle\!\rangle \\ \mathbf{f}^{\mathsf{false}}_{X;i}(Z) &:= Z \backslash \{x_1, \ldots, x_k\} \cup \bigcup_{j=1}^{k} \langle\!\langle x_j \overset{?}{\in} Z \,\mathbf{?}\, \{b, z_j\} \,\mathbf{:}\, \emptyset \rangle\!\rangle \ . \end{aligned}$$

Compare these definitions with (4.33). We see that $\mathbf{f}^{\mathsf{true}}_{X;i}(Z) \in \mathsf{KV}^X_{i-1}$ and $\mathbf{f}^{\mathsf{false}}_{X;i}(Z) \in \mathsf{KV}^X_{i-1}$. We also see that if none of the variables x_j are elements of Z, then $\mathbf{f}^{\mathsf{true}}_{X;i}(Z) = \mathbf{f}^{\mathsf{false}}_{X;i}(Z)$.

The algorithm $\mathsf{ValidChoice}(\mathsf{P}, X, B, S_0)$:
  return $\mathrm{VCStep}(s, X, B, S_0)$

subroutine $\mathrm{VCStep}(i, S, B, S_0)$
  if $i = 0$ **then**
    return $S \stackrel{?}{=} S_0$
  end if
  if $i \in \mathsf{VC}_\mathsf{P}$ and $B(i) = \mathsf{true}$ **then**
    return $\mathrm{VCStep}(i-1, \mathbf{f}^{\mathsf{true}}_{X;i}(S), B, S_0)$
  else if $i \in \mathsf{VC}_\mathsf{P}$ and $B(i) = \mathsf{false}$ **then**
    return $\mathrm{VCStep}(i-1, \mathbf{f}^{\mathsf{false}}_{X;i}(S), B, S_0)$
  else
    return $\mathrm{VCStep}(i-1, \mathbf{f}_{X;i}(S), B, S_0)$
  end if

Figure 4.23: Right choices for $S_0 \in \mathsf{KV}_0^X$

Let $\mathsf{VC}_\mathsf{P} \subseteq \{1, \ldots, s\}$ be the set of all indices i, such that A_i is a vectorised choice. Let $B : \mathsf{VC}_\mathsf{P} \to \mathbb{B}$. Consider the algorithm $\mathsf{ValidChoice}$ in Fig. 4.23. In this algorithm, S_0 is supposed to be an element of KV_0^X. This algorithm defines for each set of choices the "right" initial knowledge.

We can now finally integrate the component $\mathsf{KnownInps}(C)$ to the computation of the interpretation $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$. The interpretation is given in Fig. 4.24.

Basically, the algorithm given in Fig. 4.24 just calls the algorithm $\mathsf{ValidChoice}$ with the values of booleans at vectorised choices. The question here is, what to do if we do not know the value of the guard variable at some vectorised choice. The subroutine constructB then assumes that value to be true, but actually it does not matter at all. If the value of the guard variable is unknown at some vectorised choice A_i, then none of the variables in the left hand side of A_i are known afterwards and therefore $\mathbf{f}^{\mathsf{true}}_{X;i}$ and $\mathbf{f}^{\mathsf{false}}_{X;i}$ are equal for the set of known variables after A_i.

Now the property for $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ and $\mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$ that we mentioned before indeed holds, as witnessed by the following lemma.

Lemma 4.8. *Let C be a configuration of $\mathsf{2Chart}_{\mathsf{P};X,Y}$. Let $\mathbf{Conf}_{\mathsf{P};X,Y}$ be the set of configurations of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ that contains C and that also contains every other configuration C' where $\mathsf{KnownInps}(C) \neq \mathsf{KnownInps}(C')$ and where all other components of C and C' are (pairwise) equal. Then the following equality holds:*

$$[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0 = \sum_{C' \in \mathbf{Conf}_{\mathsf{P};X,Y}} [\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C']\!] \ .$$

Proof. Let $\mathbf{R}$ be the set of all possible random choices that are made by

- the PPT algorithm sampling the initial distribution D;
- the PPT algorithms at the nodes of $\mathsf{2Chart}_{\mathsf{P};X,Y}$, computing the semantics of operators;

Let $S_X \in \mathsf{KV}_0^X$ be the set of labels of input edges e of $\mathsf{Chart}_{\mathsf{P};X}$, such that $e \in \mathsf{KnownInps}(C)$ (see the def. of $\mathsf{KnownInps}(C)$). Let $S_Y \in \mathsf{KV}_0^Y$ be defined similarly.
Computation of $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$:

pick λ_E according to the computation in Figures 4.17–4.22
let $B_X :=$ constructB(λ_E, X)
let $B_Y :=$ constructB(λ_E, Y)
if $\mathsf{ValidChoice}(\mathsf{P}, X, B_X, S_X)$ and $\mathsf{ValidChoice}(\mathsf{P}, Y, B_Y, S_Y)$ **then**
    return $\lambda_E|_{\mathsf{outputs}(\mathsf{2Chart}_{\mathsf{P};X,Y})}$
else
    return $\perp$
end if

subroutine constructB(λ_E, Z)

let B be the partial function from VC_P to $\mathbb{B}$ that is nowhere defined
for all $i \in \mathsf{VC}_\mathsf{P}$ **do**
    if $\mathsf{Chart}_{\mathsf{P};Z}$ contains a node corresponding to A_i **then**
        let the edge e_b be the first input to the nodes corresponding to A_i in $\mathsf{Chart}_{\mathsf{P};Z}$
        if $\lambda_E(e_b) = \perp_\mathrm{u}$ **then**
            $B := B[i \mapsto \mathsf{true}]$
        else
            $B := B[i \mapsto \lambda_E(e_b)]$
        end if
    else
        $B := B[i \mapsto \mathsf{true}]$
    end if
end for
return B

Figure 4.24: Adding $\mathsf{KnownInps}(C)$ to the interpretation of flowcharts (2nd part)

- the black boxes, when they are used for encryption.

$\mathbf{R}$ can be seen as the set of bit-strings of certain, large enough length.

When we have fixed an element $\mathbf{r} \in \mathbf{R}$, then we have fixed all choices that are made by these algorithms. This means, that we have fixed the value of λ_E that is picked by $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0$. We denote the restriction of λ_E to the output edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ by $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0^{\mathbf{r}}$.

We may assume that the structure of $\mathbf{R}$ is such, that for a fixed $\mathbf{r} \in \mathbf{R}$, the values of λ_E picked by $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C']\!]$, where $C' \in \mathbf{Conf}_{\mathsf{P};X,Y}$, are all equal to the value of λ_E picked by $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0$ for the same $\mathbf{r}$ (except for the edges that are labelled with $\perp_{\mathrm{u}}$). Given the value of λ_E picked by $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C']\!]$ for a fixed $\mathbf{r} \in \mathbf{R}$, we denote the restriction of λ_E to the output edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ by $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C']\!]^{\mathbf{r}}$.

Fixing $\mathbf{r}$ fixes λ_E and therefore also the values of guard variables at vectorised choices. I.e. we have fixed $B_X, B_Y : \mathsf{VC}_{\mathsf{P}} \to \mathbb{B}$ in the calls to the algorithm $\mathsf{ValidChoice}$ at the 4th line on Fig. 4.24. This fixes also S_X and S_Y on Fig. 4.24 and finally the configuration $C' \in \mathbf{Conf}_{\mathsf{P};X,Y}$, such that $\mathsf{KnownInps}(C')$ corresponds to these sets S_X and S_Y.

We have seen that for each $\mathbf{r}$ there exists exactly one $C_{\mathbf{r}} \in \mathbf{Conf}_{\mathsf{P};X,Y}$ that does not pick the value $\perp$ for that set of random choices $\mathbf{r}$. Also, for that set of random choices, the values of $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0$ and $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C_{\mathbf{r}}]\!]$ are equal.

By the definition of $\mathbf{R}$ we have

$$
\begin{aligned}
[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0 &= \{\!|\, [\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0^{\mathbf{r}} \;:\; \mathbf{r} \leftarrow \mathbf{R} \,|\!\} \\
\sum_{C' \in \mathbf{Conf}_{\mathsf{P};X,Y}} [\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C']\!] &= \{\!|\, [\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C_{\mathbf{r}}]\!]^{\mathbf{r}} \;:\; \mathbf{r} \leftarrow \mathbf{R} \,|\!\} \;.
\end{aligned}
$$

From this the statement of the lemma immediately follows. □

When we apply this lemma for C_{L} and $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ then we get that the sum of all distributions $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$, where $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$, is equal to (4.25). Similarly, applying this lemma for C_{R} and $\mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$ gives that the sum of all distributions $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$, where $C \in \mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$, is equal to (4.26).

Summary

- If an input edge of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ is not a member of $\mathsf{KnownInps}(C)$, then the value on it is unknown, denoted by $\perp_{\mathrm{u}}$.
- All nodes, except those that are labelled with $\mathsf{?:}$ or $[\mathsf{?:}]_{\mathcal{E}}$ are *strict* with respect to $\perp_{\mathrm{u}}$ — if any of their inputs is $\perp_{\mathrm{u}}$, then their output is $\perp_{\mathrm{u}}$, too.
- A node labelled with $\mathsf{?:}$ or $[\mathsf{?:}]_{\mathcal{E}}$ certainly needs the value on its first input — if this value is $\perp_{\mathrm{u}}$, then also the output of the node is $\perp_{\mathrm{u}}$. Depending on the value on the first input, the node needs either the value on the second input (if the value on the first input is true) or the value on the third input (if the value on the first input is false).

- For each possible value of $\mathsf{KnownInps}(C)$, the first inputs to nodes corresponding to vectorised choices must have certain values. If they do not have such values, then the interpretation of the flowchart returns $\bot$.

4.4.5 Same Choices at Both Sides

The component $\mathsf{KnownInps}(C)$ is necessary for handling vectorised choices; without it we cannot hope to handle them so precisely as given by the rules in Fig. 4.7. But it also brings its own problems. Namely, let $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ and $C' \in \mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$ be such, that $\mathsf{KnownInps}(C) = \mathsf{KnownInps}(C')$. We would like to show the closeness of (4.34) and (4.35) by showing the closeness of all such pairs C and C'. However, the interpretation of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ with the configuration C can be rather different from the interpretation with the configuration C'.

For example, consider the program

$$\begin{aligned}&\langle x\rangle := b\,?\langle y\rangle : \langle z\rangle;\\ &u := \mathcal{E}nc(k,x)\end{aligned}$$

and let $X = \{x\}$ and $Y = \{u\}$ (if b, y and z are independent of $[k]_{\mathcal{E}}$ then x is independent of u). The set KV_0^X contains two sets of variables — $\{b,y\}$ and $\{b,z\}$. The set KV_0^Y also contains two sets of variables — $\{b,y,k\}$ and $\{b,z,k\}$. Assume that the configurations C and C' mentioned before are such, that $\mathsf{KnownInps}(C)$ and $\mathsf{KnownInps}(C')$ correspond to the sets $S_X = \{b,y\}$ and $S_Y = \{b,z,k\}$. Suppose that the initial probability distribution D is such, that the probability that the value of b is true is $\frac{1}{2}$. Now

- $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$ assigns to $\bot$ the weight 1. Indeed, not getting $\bot$ would need different values for the input labelled with b in $\mathsf{Chart}_{\mathsf{P};X}$ and $\mathsf{Chart}_{\mathsf{P};Y}$. But this is impossible by the definition of $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$.
- $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C']\!]$ assigns to $\bot$ the weight $\frac{3}{4}$. Here we can get different values for the inputs labelled with b in $\mathsf{Chart}_{\mathsf{P};X}$ and $\mathsf{Chart}_{\mathsf{P};Y}$.

This is a big difference.

In th example above, the value of the input labelled with b in $\mathsf{Chart}_{\mathsf{P};Y}$ does not really seem to matter that much. It is only used to choose either the value of y or the value of z as the value to be encrypted. As the encryption hides the information anyway, it seems not so important, whether the value of y or the value of z was encrypted.

Our solution to the problems with $\mathsf{KnownInps}(C)$ is similar — we are looking for guard variables whose value does not matter. We are going to add one more (base) component $\mathsf{SameIf}(C)$ to the configurations C.

We start by defining the derived components $\mathsf{fictUse}(C)$, $\mathsf{fictitious}(C)$ and $\mathsf{constEdge}(C)$.

A fictitious edge of the flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$ is an edge e whose value does not matter, when we are computing the interpretation $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$. We are free to change the label $\lambda_E(e)$, this will not change the values of λ_E on the output edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$.

$\mathsf{fictitious}(C) \subseteq \mathsf{Edges}(\mathsf{2Chart}_{\mathsf{P};X,Y})$, defined below, only contains fictitious edges. While defining it, we also define $\mathsf{fictUse}(C)$ — the set of fictitious uses (of the values carried by edges). The elements of $\mathsf{fictUse}(C)$ are uses of edges — basically pairs of a node v and a position in the sequence of edges $\overrightarrow{\rho}(v)$.

- Let v be a node. If $v \in \mathsf{EncNull}(C)$, then the second argument to v is fictitious — i.e. $(v, 2) \in \mathsf{fictUse}(C)$. Indeed, recall that $v \in \mathsf{EncNull}(C)$ means that the encryption operation at v encrypts not its second argument, but a fixed bit-string $\mathbf{0}^{\ell(n)}$.

- Let v be a node. If $v \in \mathsf{IfNewKeys}(C)$ (and then $\lambda_N(v) = [\mathsf{?:}]_{\mathcal{E}}$) then all (three) arguments to v are fictitious — $(v, 1), (v, 2), (v, 3) \in \mathsf{fictUse}(C)$. Indeed, if $v \in \mathsf{IfNewKeys}(C)$ then the choice operation at v generates a *new* encrypting black box and does not use its inputs at all.

- Let e be an edge and let U_e be the set of its uses, i.e.

  $$U_e = \{(v, i) \,:\, v \in \mathsf{Nodes}(\mathsf{2Chart}_{\mathsf{P};X,Y}), \overrightarrow{\rho}(v) = e_1, \ldots, e_k, e_i = e\} \ .$$

  If $U_e \subseteq \mathsf{fictUse}(C)$, then $e \in \mathsf{fictitious}(C)$.

- Let v be a node. If $\overleftarrow{\rho}(v) \in \mathsf{fictitious}(C)$, then all arguments to v are fictitious. I.e. if k is the number of inputs to v, then $(v, 1), \ldots, (v, k) \in \mathsf{fictUse}(C)$.

The derived component $\mathsf{constEdge}(C)$ records, which edges are labelled with constant values by λ_E. It has the following definition.

- Let e be an input edge of $\mathsf{2Chart}_{\mathsf{P};X,Y}$, labelled with $x \in \mathbf{Var}$. If $(\{x\}, \{x\}) \in \mathsf{indeps}(\beta^{\mathbb{KI}}_{\mathbf{Var}}(D))$, then $e \in \mathsf{constEdge}(C)$.

- Let v be a node of $\mathsf{2Chart}_{\mathsf{P};X,Y}$, labelled with $o \in \mathbf{Op}$. If o is a nullary operation and $[\![o]\!]$ is deterministic, then $\overleftarrow{\rho}(v) \in \mathsf{constEdge}(C)$.

We will now make an assumption about the initial probability distribution D. We already should have made this assumption in Prop. 4.5, but here is the first place where we are going to use it, therefore we make it here. We assume that if the value of some variable x is constant in the distribution D (i.e. (4.4) is negligible), then the probability (4.4) is actually 0, i.e. the value of the variable x really *is* a constant, without even a negligible probability of having some other value. This assumption does not restrict the generality, as it can change the distribution D only negligibly.

The component $\mathsf{SameIf}(C)$ is just a boolean value. The interpretations of the flowchart $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0$ and $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$ presented so far correspond to $\mathsf{SameIf}(C) = \mathsf{false}$. For the case $\mathsf{SameIf}(C) = \mathsf{true}$, let us state a hygiene condition first.

In subroutine do_assignment in Fig. 4.17, add after the 3rd line:
  let $Z \in \{X, Y\}$ be such that $v \in \mathsf{Nodes}(\mathsf{Chart}_{\mathsf{P};Z})$
  let W be Y, if Z is X. Otherwise let W be X.
  if $\mathsf{SameIf}(C)$ and $\left(\lambda_N(v) = \mathsf{?:} \text{ or } \lambda_N(v) = [\mathsf{?:}]_{\mathcal{E}}\right)$ **then**
    let v', if it exists, be a node in $\mathsf{Chart}_{\mathsf{P};W}$ corresponding to the same vectorised choice A_i as v
    if v' exists and v and v' do not belong to same part of $\mathsf{OpParts}_i(C)$ **then**
      let e'_1 be the first in-edge of v'
      if $e_1 \in \mathsf{fictitious}(C)$ and $e'_1 \notin \mathsf{fictitious}(C)$ **then**
        let $\lambda_E(e_1) := \lambda_E(e'_1)$ .... /* i.e. change $\lambda_E(e_1)$ */
      else if $e_1, e'_1 \in \mathsf{fictitious}(C)$ and not $e_1, e'_1 \in \mathsf{constEdge}(C)$ **then**
        if W is X **then**
          let $\lambda_E(e_1) := \lambda_E(e'_1)$
        end if
      end if
    end if
  end if

Figure 4.25: Adding $\mathsf{SameIf}(C)$ to the interpretation of flowcharts

- Let $\mathsf{SameIf}(C) = \mathsf{true}$. Let A_i be a vectorised choice and suppose that there exist a node v_X in $\mathsf{Chart}_{\mathsf{P};X}$ and a node v_Y in $\mathsf{Chart}_{\mathsf{P};Y}$ that both correspond to A_i. Let e_X and e_Y be the first inputs to v_X and v_Y, respectively (recall that the first input corresponds to the guard variable). Then at least one of the following cases must hold:
  - e_X and e_Y belong to the same part in $\mathsf{edgeParts}(C)$;
  - $e_X \in \mathsf{fictitious}(C)$ or $e_Y \in \mathsf{fictitious}(C)$;
  - $e_X \in \mathsf{constEdge}(C)$ and $e_Y \in \mathsf{constEdge}(C)$.

Fig. 4.25 gives the necessary change to the interpretations of flowcharts $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0$ and $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$ for the general case. As we see from this figure, we try to use the value on a non-fictitious edge. If both edges (e_1 and e'_1) are fictitious, then we just fix somehow, which of the values $\lambda_E(e_1)$ and $\lambda_E(e'_1)$ we use (we have fixed that the edge in $\mathsf{Chart}_{\mathsf{P};X}$ has priority).

We see that if the only difference of configurations C and C' is $\mathsf{SameIf}(C) \neq \mathsf{SameIf}(C')$, then $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0$ and $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C']\!]_0$ are equal. Indeed, the computation of these interpretations may differ only in values assigned to fictitious edges.

We define that $\mathsf{SameIf}(C) = \mathsf{true}$ for all $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ and $\mathsf{SameIf}(C) = \mathsf{false}$ for all $C \in \mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$. It is obvious that the hygiene condition holds for all $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ — the case "e_X and e_Y belong to the same part in $\mathsf{edgeParts}(C)$" applies for all relevant e_X and e_Y. As the introduction of $\mathsf{SameIf}(C)$ is orthogonal to the introduction of $\mathsf{KnownInps}(C)$, Lemma 4.8 still holds.

Adding the component $\mathsf{SameIf}(C)$ to the configuration C still does not help us to avoid the big difference between corresponding configurations in $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ and $\mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$, but it allows us to control, where it occurs. Let us first define an auxiliary notion.

Definition 4.3. Let C be a configuration of $\mathsf{2Chart}_{\mathsf{P};X,Y}$. We say that C *has no ties between sides*, if

- The partition $\mathsf{InpParts}(C)$ is such, that there are no such parts in $\mathsf{InpParts}(C)$ that contain edges from both $\mathsf{Chart}_{\mathsf{P};X}$ and $\mathsf{Chart}_{\mathsf{P};Y}$.
- For each $i \in \{1,\ldots,s\}$, the partition $\mathsf{OpParts}_i(C)$ has similar property — there are no such parts in $\mathsf{OpParts}_i(C)$ that contain nodes from both $\mathsf{Chart}_{\mathsf{P};X}$ and $\mathsf{Chart}_{\mathsf{P};Y}$.

The following lemma holds.

Lemma 4.9. *Let $\mathbf{C}$ be a set of configurations of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ with the following properties:*

- $\mathsf{SameIf}(C) = \mathsf{true}$ *for each* $C \in \mathbf{C}$.
- *For each $S_X \in \mathsf{KV}_0^X$ and $S_Y \in \mathsf{KV}_0^Y$ there exists exactly one $C \in \mathbf{C}$, such that* $\mathsf{KnownInps}(C)$ *is defined by S_X and S_Y.*
- *For each $C \in \mathbf{C}$, the configuration C has no ties between sides.*
- *Let $S_X \in \mathsf{KV}_0^X$. Consider an arbitrary configuration $C \in \mathbf{C}$, such that* $\mathsf{KnownInps}(C)$ *is defined by S_X (and by some element of KV_0^Y). The set S_X must uniquely determine all components of C, restricted to nodes and edges of $\mathsf{Chart}_{\mathsf{P};X}$. This means, that if C' is another configuration, such that* $\mathsf{KnownInps}(C')$ *is defined by S_X, then the parts in* $\mathsf{InpParts}(C)$ *and* $\mathsf{InpParts}(C')$ *that contain the inputs of $\mathsf{Chart}_{\mathsf{P};X}$, must be equal. Also, the intersection of* $\mathsf{EncNull}(C)$ *with the set of nodes of $\mathsf{Chart}_{\mathsf{P};X}$ must be equal to the intersection of* $\mathsf{EncNull}(C')$ *with the set of nodes of $\mathsf{Chart}_{\mathsf{P};X}$. Similar requirements hold for* $\mathsf{OpParts}_i(\cdot)$, $\mathsf{BoxNull}(\cdot)$ *and* $\mathsf{IfNewKeys}(\cdot)$.

  The set S_X therefore determines a configuration of the flowchart $\mathsf{Chart}_{\mathsf{P};X}$ (not $\mathsf{2Chart}_{\mathsf{P};X,Y}$). Let us denote it by $C[S_X]$.
- *There is similar requirement for the elements of KV_0^Y. They must also uniquely determine the components of configurations, restricted to nodes and edges of $\mathsf{Chart}_{\mathsf{P};Y}$. We let $C[S_Y]$ denote the configuration of $\mathsf{Chart}_{\mathsf{P};Y}$ determined by $S_Y \in \mathsf{KV}_0^Y$.*

Let $\mathbf{C}'$ be another set of configurations of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ that is obtained from $\mathbf{C}$ by setting $\mathsf{SameIf}(C)$ *to* false *for each $C \in \mathbf{C}$. Then*

$$\sum_{C\in\mathbf{C}} [\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!] = \sum_{C'\in\mathbf{C}'} [\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C']\!] \ . \tag{4.36}$$

Proof. This lemma is actually rather similar to Lemma 4.8. The difference is that here we do not have a single configuration C, such that the sum of the interpretations of configurations (4.36) is equal to $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]_0$. Nevertheless, we can still give a single computation that has the same output distribution as the right hand side of (4.36).

First note that the interpretation of flowcharts can also be defined for $\mathsf{Chart}_{\mathsf{P};X}$ (together with a suitable configuration) and $\mathsf{Chart}_{\mathsf{P};Y}$ (together with a suitable configuration) alone, not only for $\mathsf{2Chart}_{\mathsf{P};X,Y}$.

Consider the following computation:

1. Uniformly randomly choose S_X from the set KV_0^X.
2. Compute λ_E according to $[\![\mathsf{Chart}_{\mathsf{P};X}, C[S_X]]\!]_0$
3. Let $B = \text{constructB}(\lambda_E, X)$, where the algorithm constructB is given in Fig. 4.24.
4. Call $\mathsf{ValidChoice}(\mathsf{P}, X, B, S_X)$. If it returns true then output λ_E. If it returns false, then go back to 1st step.

This computation runs in expected polynomial time (in the size of P) and it returns the same distribution as the right hand side of (4.36), restricted to the outputs of $\mathsf{Chart}_{\mathsf{P};X}$. If we replace X by Y in this computation, then it returns the same distribution as the right hand side of (4.36), restricted to the outputs of $\mathsf{Chart}_{\mathsf{P};Y}$. Taking these two together gives us the whole distribution at the right hand side of (4.36), because the values on the outputs of $\mathsf{Chart}_{\mathsf{P};X}$ and the values on the outputs of $\mathsf{Chart}_{\mathsf{P};Y}$ are independent of each other.

For getting the distribution on the left hand side of (4.36), we have to change the values on some edges, while computing the interpretations $[\![\mathsf{Chart}_{\mathsf{P};X}, C[S_X]]\!]_0$ and $[\![\mathsf{Chart}_{\mathsf{P};Y}, C[S_Y]]\!]_0$. As these values are fictitious, changing of them does not change the result of the computation. □

Summary

If $\mathsf{SameIf}(C)$ is true, then the computation of the interpretation of the flowchart attempts to not use the values on fictitious edges as the guards of choices.

4.4.6 The Interpretation $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$ — Final Shape

Here we once more state the definition of the interpretation of flowcharts already given in Figures 4.17–4.25. This time, however, we will integrate the changes to the main algorithm.

■ The interpretation $[\![\mathsf{2Chart}_{\mathsf{P};X,Y}, C]\!]$ — sampling algorithm:

```
call create_inputs
for i = 1 to s do
    call do_assignment(i)
```

end for
let $B_X :=$ constructB(λ_E, X)
let $B_Y :=$ constructB(λ_E, Y)
let $S_X \in \mathsf{KV}_0^X$ and $S_Y \in \mathsf{KV}_0^Y$ be such, that the define $\mathsf{KnownInps}(C)$
if $\mathsf{ValidChoice}(\mathsf{P}, X, B_X, S_X)$ and $\mathsf{ValidChoice}(\mathsf{P}, Y, B_Y, S_Y)$ **then**
  return $\lambda_E|_{\mathsf{outputs}(\mathsf{2Chart}_{\mathsf{P};X,Y})}$
else
  return $\perp$
end if

■ subroutine create_inputs

for all $E' \in \mathsf{InpParts}(C)$ **do**
  if $E' = \{e_1, \ldots, e_k\}$ and $e_1, \ldots, e_k \in \mathsf{InpKeys}(C)$ **then**
    generate $x \in \mathbf{Val}_n$ according to $[\![\mathcal{G}en]\!]_n$
    for all $e_i \in E'$ **do**
      if $e_i \notin \mathsf{KnownInps}(C)$ **then**
        define $\lambda_E(e_i) := \perp_{\mathrm{u}}$
      else if $e_i \in \mathsf{BoxNull}(C)$ **then**
        define $\lambda_E(e_i)$ as a black box $[\![\mathcal{E}nc]\!]_n(x, \mathbf{0}^{\ell(n)})$
      else
        define $\lambda_E(e_i)$ as a black box $[\![\mathcal{E}nc]\!]_n(x, \cdot)$
      end if
    end for
  else
    generate $S_n \in \mathbf{State}_n$ according to D_n
    for all $e \in E'$ **do**
      if $e \in \mathsf{KnownInps}(C)$ **then**
        define $\lambda_E(e) := S_n(\lambda_I(e))$
      else
        define $\lambda_E(e) := \perp_{\mathrm{u}}$
      end if
    end for
  end if
end for

■ subroutine do_assignment(i)

for all $N' \in \mathsf{OpParts}_i(C)$ **do**
  let v be an element of N'
  let $e_1, \ldots, e_k$ be the in-edges of v (from left to right)
  if $\lambda_N(v) \neq \mathsf{?:}$ and $\lambda_N(v) \neq [\mathsf{?:}]_{\mathcal{E}}$ and $\exists j : \lambda_E(e_j) = \perp_{\mathrm{u}}$ **then**
    let x be $\perp_{\mathrm{u}}$
  else if $\lambda_N(v) = \mathsf{?:}$ or $\lambda_N(v) = [\mathsf{?:}]_{\mathcal{E}}$ **then**
    let x be do_choice(v, e_1, e_2, e_3)
  else if $\lambda_N(v) = \mathcal{E}nc$ **then**

    if $v \in \mathsf{EncNull}(C)$ **then**
      generate $x \in \mathbf{Val}_n$ by feeding $\mathbf{0}^{\ell(n)}$ to the black box $\lambda_E(e_1)$
    else
      generate $x \in \mathbf{Val}_n$ by feeding $\lambda_E(e_2)$ to the black box $\lambda_E(e_1)$
    end if
  else if $\lambda_N(v) = \mathcal{G}en$ or $\lambda_N(v) = \mathcal{G}en_{\mathrm{val}}$ **then**
    generate $x \in \mathbf{Val}_n$ according to the distribution $[\![\mathcal{G}en]\!]_n$
  else
    generate $x \in \mathbf{Val}_n$ according to $[\![\lambda_N(v)]\!]_n(\lambda_E(e_1), \ldots, \lambda_E(e_k))$
  end if
  for all $v \in N'$ **do**
    let $e = \overleftarrow{\rho}(v)$
    if $\lambda_N(v) \neq \mathcal{G}en$ **then**
      define $\lambda_E(e) := x$
    else if $v \in \mathsf{BoxNull}(C)$ **then**
      define $\lambda_E(e)$ as a black box $[\![\mathcal{E}nc]\!]_n(x, \mathbf{0}^{\ell(n)})$
    else
      define $\lambda_E(e)$ as a black box $[\![\mathcal{E}nc]\!]_n(x, \cdot)$
    end if
  end for
end for
for all $v \in Mk\mathcal{E}\mathrm{Set}_i$ **do**
  let e be the in-edge and e' be the out-edge of v
  if $\lambda_E(e) = \bot_{\mathrm{u}}$ **then**
    define $\lambda_E(e') := \bot_{\mathrm{u}}$
  else
    define $\lambda_E(e')$ as the black box $[\![\mathcal{E}nc]\!]_n(\lambda_E(e), \cdot)$
  end if
end for

■ subroutine do_choice(v, e_1, e_2, e_3)

let $Z \in \{X, Y\}$ be such that $v \in \mathsf{Nodes}(\mathsf{Chart}_{\mathsf{P};Z})$
let W be Y, if Z is X. Otherwise let W be X.
if $\mathsf{SameIf}(C)$ and $\big(\lambda_N(v) = \mathsf{?:}$ or $\lambda_N(v) = [\mathsf{?:}]_{\mathcal{E}}\big)$ **then**
  let v', if it exists, be a node in $\mathsf{Chart}_{\mathsf{P};W}$ corresponding to the same vectorised choice A_i as v
  if v' exists and v and v' do not belong to same part of $\mathsf{OpParts}_i(C)$ **then**
    let e'_1 be the first in-edge of v'
    if $e_1 \in \mathsf{fictitious}(C)$ and $e'_1 \notin \mathsf{fictitious}(C)$ **then**
      let $\lambda_E(e_1) := \lambda_E(e'_1)$ .... /* i.e. change $\lambda_E(e_1)$ */
    else if $e_1, e'_1 \in \mathsf{fictitious}(C)$ and not $e_1, e'_1 \in \mathsf{constEdge}(C)$ **then**
      if W is X **then**
        let $\lambda_E(e_1) := \lambda_E(e'_1)$
      end if

end if
end if
end if
if $\lambda_E(e_1) = \bot_u$ or
$\lambda_E(e_1) = \mathsf{true}$ and $\lambda_E(e_2) = \bot_u$ or
$\lambda_E(e_1) = \mathsf{false}$ and $\lambda_E(e_3) = \bot_u$ **then**
let x be $\bot_u$
else if $\lambda_N(v) = \mathsf{?:}$ **then**
let x be $\lambda_E(e_2)$, if $\lambda_E(e_1) = \mathsf{true}$, and $\lambda_E(e_3)$ otherwise
else if $\lambda_N(v) = [\mathsf{?:}]_{\mathcal{E}}$ **then**
if $v \in \mathsf{IfNewKeys}(C)$ **then**
generate $x' \in \mathbf{Val}_n$ according to the distribution $[\![\mathcal{G}en]\!]_n$
if $\overleftarrow{\rho}(v) \in \mathsf{edgeNull}(C)$ **then**
let x be the black box $[\![\mathcal{E}nc]\!]_n(x, \mathbf{0}^{\ell(n)})$
else
let x be the black box $[\![\mathcal{E}nc]\!]_n(x, \cdot)$
end if
else
let x be $\lambda_E(e_2)$, if $\lambda_E(e_1) = \mathsf{true}$, and $\lambda_E(e_3)$ otherwise
end if
end if
return x

■ subroutine $\mathsf{ValidChoice}(\mathsf{P}, X, B, S_0)$:

return $\mathrm{VCStep}(s, X, B, S_0)$

■ subroutine $\mathrm{VCStep}(i, S, B, S_0)$

if $i = 0$ **then**
return $S \stackrel{?}{=} S_0$
end if
if $i \in \mathsf{VC_P}$ and $B(i) = \mathsf{true}$ **then**
return $\mathrm{VCStep}(i-1, \mathbf{f}^{\mathsf{true}}_{X;i}(S), B, S_0)$
else if $i \in \mathsf{VC_P}$ and $B(i) = \mathsf{false}$ **then**
return $\mathrm{VCStep}(i-1, \mathbf{f}^{\mathsf{false}}_{X;i}(S), B, S_0)$
else
return $\mathrm{VCStep}(i-1, \mathbf{f}_{X;i}(S), B, S_0)$
end if

■ subroutine $\mathrm{constructB}(\lambda_E, Z)$

let B be the partial function from $\mathsf{VC_P}$ to $\mathbb{B}$ that is nowhere defined
for all $i \in \mathsf{VC_P}$ **do**
if $\mathsf{Chart}_{\mathsf{P};Z}$ contains a node corresponding to A_i **then**
let the edge e_b be the first input to the nodes corresponding to A_i in $\mathsf{Chart}_{\mathsf{P};Z}$
if $\lambda_E(e_b) = \bot_u$ **then**
$B := B[i \mapsto \mathsf{true}]$

        else
            $B := B[i \mapsto \lambda_E(e_b)]$
        end if
    else
        $B := B[i \mapsto \mathsf{true}]$
    end if
end for
return B

The presented algorithm obviously runs in polynomial time. We assume that a representation for the encrypting black boxes has been fixed. This representation must also accommodate the case, where some encrypting black box really *is* a black box — i.e. when it can be accessed only through the oracle interface.

4.5 Changing the Structures

4.5.1 Ways for Turning One Configuration to Another

Let C be a configuration of $G = \mathsf{2Chart}_{\mathsf{P};X,Y}$. We let $\mathsf{matters}(C)$ denote the set of inputs of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ whose values are really used during the computation of $[\![G, C]\!]$. Formally, $\mathsf{matters}(C) := \mathsf{KnownInps}(C)\backslash\mathsf{fictitious}(C)$.

A configuration C of $G = \mathsf{2Chart}_{\mathsf{P};X,Y}$ can be changed to another configuration C' (denoted $C \leftrightarrow C'$; the reflexive transitive closure of the relation $\leftrightarrow$ is denoted by $\overset{*}{\leftrightarrow}$) in any of the ways described below. Some of these ways (**1**, **2**, **6**, **9**) do not change the distribution $[\![G, C]\!]$ at all. The others change it, but "only a little", in the sense of short steps in the proof of Prop. 2.11. Let $\mathcal{X} = \beta^{\mathbb{KI}}_{\mathbf{Var}}(D)$.

1. The partition $\mathsf{InpParts}(C)$ may be changed — it does not matter, which parts the fictitious and unknown inputs belong to. We define $C \leftrightarrow C'$ and $C' \leftrightarrow C$, if there exists a set of edges $E_f \subseteq \mathsf{inputs}(G)\backslash\mathsf{matters}(C)$, such that

- there exists a part $E' \in \mathsf{InpParts}(C)$, such that $E_f \subseteq E'$ and $E_f \neq E'$;
- $\mathsf{InpParts}(C)$ and $\mathsf{InpParts}(C')$ differ in the following way: the part $E' \in \mathsf{InpParts}(C)$ is replaced by two parts in $\mathsf{InpParts}(C')$ — by E_f and $E'\backslash E_f$;
- other components of C and C' are equal.

2. Similarly, $\mathsf{OpParts}_i(C)$ may be changed — it does not matter, which parts the fictitious nodes belong to. Actual definition of $C \leftrightarrow C'$ and $C' \leftrightarrow C$ is similar to way **1**.

3. If the partition $\mathsf{InpParts}(C)$ contains a part $E' = \{e_1, \ldots, e_l\}$, where $\lambda_I(e_1) = \cdots = \lambda_I(e_l) = [k]_{\mathcal{E}}$ for some $k \in \mathsf{keys}(\mathcal{X})$, then one may change $\mathsf{InpKeys}(C)$ by adding all elements of E' to it or removing all elements of E' from it.

4. Let E' be defined as in the description of way **3**. If all elements of E' are in $\mathsf{InpKeys}(C)$, then one may add them all to or remove them all from $\mathsf{BoxNull}(C)$.

5. Let $\bar{E} \in \mathsf{InpParts}(C)$ and suppose that there exists $(\bar{X}, \bar{Y}) \in \mathsf{indeps}(\mathcal{X})$ such that the labels of the edges in $\bar{E}$ all belong to $\bar{X} \cup \bar{Y}$. In this case $\bar{E}$ may be partitioned to sets $\bar{E}_X, \bar{E}_Y \subseteq \mathsf{inputs}(G)$, such that the labels of the edges in $\bar{E}_X$ [resp. in $\bar{E}_Y$] all belong to $\bar{X}$ [resp. to $\bar{Y}$]. I.e. the part $\bar{E}$ in $\mathsf{InpParts}(C)$ may be replaced with $\bar{E}_X$ and $\bar{E}_Y$ in $\mathsf{InpParts}(C')$. The same transformation of C may also be done in the other direction.

6. If the encrypting black boxes reaching a node v that is labelled with $\mathcal{E}nc$ all originate (origination is given by $\mathsf{bb\_use}$) from nodes and inputs in $\mathsf{BoxNull}(C)$, then one may change $\mathsf{EncNull}(C)$ by adding v to it or removing v from it.

7. Let A_i be an assignment of the form $x := \mathcal{G}en()$. Let $V \in \mathsf{OpParts}_i(C)$ be such, that all nodes in V are labelled with $\mathcal{G}en$ (i.e. none are labelled with $\mathcal{G}en_{\mathrm{val}}$). Then the nodes in V may all be added to or all be removed from $\mathsf{BoxNull}(C)$.

8. Let A_i be an assignment of the form $x := \mathcal{G}en()$. Let $V \in \mathsf{OpParts}_i(C)$ and let $\{V_1, V_2\}$ be a partition of V. If all nodes in V are labelled with $\mathcal{G}en$, then one may replace the part V in $\mathsf{OpParts}_i(C)$ with parts V_1, V_2. The same transformation of C may also be done in the other direction.

9. Let A_i be an assignment of the form $x := o()$, where $[\![o]\!]$ is deterministic. Then $\mathsf{OpParts}_i(C)$ may be freely changed (as long as the hygiene conditions of the configuration remain satisfied). The same holds for the vectorised choices, because they are deterministic as well.

10. Let v be a node labelled with $[\mathsf{?}:]_{\mathcal{E}}$. If the values on its two input edges that carry encrypting black boxes originate only from

- input edges from $\mathsf{InpKeys}(C)$ or
- nodes v labelled with $\mathcal{G}en$, such that the part $V \in \mathsf{OpParts}_i(C)$ that contains v does not contain nodes labelled with $\mathcal{G}en_{\mathrm{val}}$,

then one may change $\mathsf{IfNewKeys}(C)$ by adding v to it or removing v from it.

Note that it is impossible to change $\mathsf{KnownInps}(C)$ or $\mathsf{SameIf}(C)$.

Let us define a partial order on the set of configurations of the flowchart G.

Definition 4.4. Let C_1, C_2 be two configurations of the same flowchart, such that $\mathsf{KnownInps}(C_1) = \mathsf{KnownInps}(C_2)$ and $\mathsf{SameIf}(C_1) = \mathsf{SameIf}(C_2)$. We say that C_1 is *finer grained* than C_2 (denoted $C_1 \sqsubseteq C_2$), iff

- $\mathsf{InpParts}(C_1)$ is a finer partition than $\mathsf{InpParts}(C_2)$;
- $\mathsf{OpParts}_i(C_1)$ is a finer partition than $\mathsf{OpParts}_i(C_2)$ for all $i \in \{1, \ldots, s\}$;
- $\mathsf{EncNull}(C_1) \supseteq \mathsf{EncNull}(C_2)$;

- $\mathsf{InpKeys}(C_1) \supseteq \mathsf{InpKeys}(C_2)$;
- $\mathsf{BoxNull}(C_1) \supseteq \mathsf{BoxNull}(C_2)$;
- $\mathsf{IfNewKeys}(C_1) \supseteq \mathsf{BoxNull}(C_2)$.

From the definition follows that if $C_1 \leftrightarrow C_2$ then either $C_1 \sqsubseteq C_2$ or $C_2 \sqsubseteq C_1$. It also follows that $C_1 \sqsubseteq C_2$ implies $\mathsf{fictitious}(C_1) \supseteq \mathsf{fictitious}(C_2)$.

For each of the ways that change the distribution $[\![G, C]\!]$ (i.e. ways **3**, **4**, **5**, **7**, **8** and **10**), let us give an algorithm that makes the step between the distributions $[\![G, C]\!]$ and $[\![G, C']\!]$. This algorithm is analogous to the algorithm $\mathcal{B}$ in Fig. 2.3. This time we will not have just one algorithm $\mathcal{B}$, but several (indexed by the number of the way and, in case of using $\mathcal{X}$ (this happens by way **3** and way **5**), also by an element of $\mathsf{keys}(\mathcal{X})$ or $\mathsf{indeps}(\mathcal{X})$); this makes the whole proof more complex, but not significantly.

The algorithms $\mathcal{B}$ take the following arguments:

- The security parameter $\mathbf{1}^n$.
- The flowchart G and its two configurations C and C'. These three arguments correspond to the argument i in Fig. 2.3. We assume w.l.o.g. that C' is finer grained than C.

Additionally, the algorithms $\mathcal{B}$ take some arguments corresponding to the argument x in Fig. 2.3. The nature of these arguments depends on the way of changing the configurations.

Algorithm $\mathcal{B}_{\mathbf{3};[k]_\mathcal{E}}$

We start with the algorithm $\mathcal{B}_{\mathbf{3};[k]_\mathcal{E}}$. Here $[k]_\mathcal{E}$ is the label of the edges mentioned in the description of way **3**. As way **3** is about $\mathsf{keys}(\mathcal{X})$, the algorithm $\mathcal{B}_{\mathbf{3};[k]_\mathcal{E}}$ additionally takes as its argument an encrypting black box (that it accesses through the oracle interface). This black box can be distributed according to the n-th component of either the left or the right family of distributions on (4.3).

The algorithm $\mathcal{B}^{(\cdot)}_{\mathbf{3};[k]_\mathcal{E}}(\mathbf{1}^n, G, C, C')$ works by sampling $[\![G, C]\!]$ with the following changes to the algorithm presented in Sec. 4.4.6. Let E' be the part in $\mathsf{InpParts}(C)$ from the description of way **3** (the algorithm $\mathcal{B}_{\mathbf{3};[k]_\mathcal{E}}$ can find this part from C and C').

- In the subroutine create_inputs, check whether the variable E' of the subroutine is equal to E' from the description of way **3**. If this is the case, then for each $e_i \in E'$ do not compute $\lambda_E(e_i)$ as given in the subroutine create_inputs, but let $\lambda_E(e_i)$ be equal to the oracle that is given to $\mathcal{B}_{\mathbf{3};[k]_\mathcal{E}}$.

It is obvious that if the oracle given to $\mathcal{B}_{\mathbf{3};[k]_\mathcal{E}}$ is equal to the n-th component of the family of distributions in the left hand side of (4.3), then $\mathcal{B}_{\mathbf{3};[k]_\mathcal{E}}$ samples $[\![G, C]\!]$. If the oracle given to $\mathcal{B}_{\mathbf{3};[k]_\mathcal{E}}$ is equal to the n-th component of the family of distributions in the right hand side of (4.3), then $\mathcal{B}_{\mathbf{3};[k]_\mathcal{E}}$ samples $[\![G, C']\!]$.

Algorithm $\mathcal{B}_{\mathbf{4}}$

The additional argument to $\mathcal{B}_{\mathbf{4}}$ is again an encrypting black box (accessed through oracle interface). This black box can be distributed according to the n-th component of either the left or the right family of distributions in (2.9).

The description of the algorithm $\mathcal{B}_{\mathbf{4}}^{(\cdot)}(\mathbf{1}^n, G, C, C')$ is the same as the description of the algorithm $\mathcal{B}_{\mathbf{3};[k]_\mathcal{E}}$. The subroutine create_inputs is changed in exactly the same way. The comments about sampling either $[\![G, C]\!]$ or $[\![G, C']\!]$ also stay the same, one only has to replace $\mathcal{B}_{\mathbf{3};[k]_\mathcal{E}}$ by $\mathcal{B}_{\mathbf{4}}$ and (4.3) by (2.9).

Algorithm $\mathcal{B}_{\mathbf{5};\bar{X},\bar{Y}}$

The additional arguments to $\mathcal{B}_{\mathbf{5};\bar{X},\bar{Y}}$ are two functions. First of them, v_X, is a mapping from $\bar{X}$ to $\widetilde{\mathbf{Val}}_n$. Second of them, v_Y, is a mapping from $\bar{Y}$ to $\widetilde{\mathbf{Val}}_n$. The pair (v_X, v_Y) can be distributed either according to the n-th component of (4.1) or (4.2).

The algorithm $\mathcal{B}_{\mathbf{5};\bar{X},\bar{Y}}(\mathbf{1}^n, G, C, C', v_X, v_Y)$ works by sampling $[\![G, C]\!]$ with the following changes to the algorithm presented in Sec. 4.4.6. Let $\bar{E}$, $\bar{E}_X$ and $\bar{E}_Y$ be as in the description of way **5** (the algorithm $\mathcal{B}_{\mathbf{5};\bar{X},\bar{Y}}$ can find them from C and C').

- In the subroutine create_inputs, check whether the variable E' of the subroutine is equal to $\bar{E}$ from the description of way **5**. If this is the case, then compute $\lambda_E(e)$, where $e \in \bar{E}$, not as in the subroutine create_inputs, but as follows:
  - If $e \in \bar{E}_X$ then $\lambda_E(e) := v_X(\lambda_I(e))$.
  - If $e \in \bar{E}_Y$ then $\lambda_E(e) := v_Y(\lambda_I(e))$.

Obviously, if (v_X, v_Y) is distributed according to the n-th component of (4.1), then $\mathcal{B}_{\mathbf{5};\bar{X},\bar{Y}}$ samples $[\![G, C]\!]$. If (v_X, v_Y) is distributed according to the n-th component of (4.2), then $\mathcal{B}_{\mathbf{5};\bar{X},\bar{Y}}$ samples $[\![G, C']\!]$.

Algorithm $\mathcal{B}_{\mathbf{7}}$

The additional argument to $\mathcal{B}_{\mathbf{7}}$ is an encrypting black box (accessed through oracle interface). This black box can be distributed according to the n-th component of either the left or the right family of distributions in (2.9).

The algorithm $\mathcal{B}_{\mathbf{7}}^{(\cdot)}$ works by sampling $[\![G, C]\!]$ with the following changes to the algorithm presented in Sec. 4.4.6. Let $V \in \mathsf{OpParts}_i(C)$ be as in the description of way **7** (the algorithm $\mathcal{B}_{\mathbf{7}}$ can compute it from C and C').

- In the subroutine do_assignment, check whether the variable N' is equal to V. If this is the case, then do not compute the values $\lambda_E(\overleftarrow{\rho}(v))$ for $v \in V$ as in the subroutine do_assignment, but let $\lambda_E(\overleftarrow{\rho}(v))$ be equal to the oracle given to $\mathcal{B}_{\mathbf{7}}$.

Similarly to previous cases, $\mathcal{B}_\mathbf{7}$ samples either $[\![G, C]\!]$ (if the black box given to it is distributed according to the n-th component of the left family of distributions in (2.9)) or $[\![G, C]\!]$ (if the black box given to it is distributed according to the n-th component of the right family of distributions in (2.9)).

Algorithm $\mathcal{B}_\mathbf{8}$

The additional arguments to $\mathcal{B}_\mathbf{8}$ are two encrypting black boxes (accessed through oracle interface). This pair of black boxes can be distributed according to the n-th component of either the left or the right family of distributions in (2.10).

The algorithm $\mathcal{B}_\mathbf{8}^{(\cdot),(\cdot)}$ works by sampling $[\![G, C]\!]$ with the following changes to the algorithm presented in Sec. 4.4.6. Let $V \in \mathsf{OpParts}_i(C)$, $V_1, V_2 \in \mathsf{OpParts}_i(C')$ be as in the description of way **8** (the algorithm $\mathcal{B}_\mathbf{8}$ can compute them from C and C').

- In the subroutine do_assignment, check whether the variable N' is equal to V. If this is the case, then do not compute the values $\lambda_E(\overleftarrow{\rho}(v))$ for $v \in V$ as in the subroutine do_assignment, but
  - If $v \in V_1$ then let $\lambda_E(\overleftarrow{\rho}(v))$ be either the left oracle given to $\mathcal{B}_\mathbf{8}$ (if $v \notin \mathsf{BoxNull}(C)$) or the left oracle given to $\mathcal{B}_\mathbf{8}$, where the argument to the oracle has been fixed as $\mathbf{0}^{\ell(n)}$ (if $v \in \mathsf{BoxNull}(C)$).
  - If $v \in V_2$, then define $\lambda_E(\overleftarrow{\rho}(v))$ similarly, but replace the left oracle with the right oracle.

Similarly to previous cases, $\mathcal{B}_\mathbf{8}$ samples either $[\![G, C]\!]$ (if the pair of black boxes given to it is distributed according to the n-th component of the left family of distributions in (2.10)) or $[\![G, C']\!]$ (if the pair of black boxes given to it is distributed according to the n-th component of the right family of distributions in (2.10)).

Algorithm $\mathcal{B}_\mathbf{10}$

The additional arguments to $\mathcal{B}_\mathbf{10}$ are four encrypting black boxes (accessed through oracle interface). This pair of black boxes can be distributed either according to the n-th component of

$$\{\!|(\mathcal{E}(\mathbf{1}^n, k, \cdot), \mathcal{E}(\mathbf{1}^n, k', \cdot), \mathcal{E}(\mathbf{1}^n, k, \cdot), \mathcal{E}(\mathbf{1}^n, k', \cdot)) \,:\, k, k' \leftarrow \mathcal{G}(\mathbf{1}^n)|\!\}_{n\in\mathbb{N}} \tag{4.37}$$

or according to the n-th component of

$$\{\!|(\mathcal{E}(\mathbf{1}^n, k, \cdot), \mathcal{E}(\mathbf{1}^n, k', \cdot), \mathcal{E}(\mathbf{1}^n, k'', \cdot), \mathcal{E}(\mathbf{1}^n, k'', \cdot)) \,:\, k, k', k'' \leftarrow \mathcal{G}(\mathbf{1}^n)|\!\}_{n\in\mathbb{N}}, \tag{4.38}$$

where $\mathcal{E}$ is the algorithm implementing $[\![\mathcal{E}nc]\!]$ and $\mathcal{G}$ is the algorithm implementing $[\![\mathcal{G}en]\!]$.

This is also a valid "small step". Indeed, the following lemma holds.

Lemma 4.10. *If* $(\llbracket \mathcal{G}en \rrbracket, \llbracket \mathcal{E}nc \rrbracket)$ *is a which-key concealing encryption scheme, then the families of distributions (4.37) and (4.38) are indistinguishable.*

Proof. This can be shows by several applications of (2.10). Indeed,

$$\begin{aligned}
&\{\!|(\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k',\cdot),\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k',\cdot))\ :\ k,k'\leftarrow\mathcal{G}(\mathbf{1}^n)|\!\}_{n\in\mathbb{N}}\approx\\
&\quad\{\!|(\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k,\cdot))\ :\ k\leftarrow\mathcal{G}(\mathbf{1}^n)|\!\}_{n\in\mathbb{N}}\approx\\
&\quad\{\!|(\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k'',\cdot),\mathcal{E}(\mathbf{1}^n,k'',\cdot))\ :\ k,k''\leftarrow\mathcal{G}(\mathbf{1}^n)|\!\}_{n\in\mathbb{N}}\approx\\
&\quad\{\!|(\mathcal{E}(\mathbf{1}^n,k,\cdot),\mathcal{E}(\mathbf{1}^n,k',\cdot),\mathcal{E}(\mathbf{1}^n,k'',\cdot),\mathcal{E}(\mathbf{1}^n,k'',\cdot))\ :\ k,k',k''\leftarrow\mathcal{G}(\mathbf{1}^n)|\!\}_{n\in\mathbb{N}},
\end{aligned}$$

where each $\approx$ follows from (2.10). □

The algorithm $\mathcal{B}_{\mathbf{10}}^{(\cdot),(\cdot),(\cdot),(\cdot)}(\mathbf{1}^n, G, C, C')$ works as follows:

1. Let v be the node from the description of way **10**. The algorithm $\mathcal{B}_{\mathbf{10}}$ can find it from C and C'.
2. Uniformly randomly choose W_{true} from

$$\{E' \in \mathsf{InpParts}(C)\ :\ E' \subseteq \mathsf{InpKeys}(C)\} \cup \\ \{V \in \mathsf{OpParts}_i(C)\ :\ \mathsf{A}_i \text{ is } x := \mathcal{G}en(), \nexists v \in V : \lambda_N(v) = \mathcal{G}en_{\mathrm{val}}\}\ .$$

3. Uniformly randomly choose W_{false} from the same set.
4. Execute the algorithm sampling $\llbracket G, C \rrbracket$ with the following modifications:
   - In the subroutine create_inputs, check whether the variable E' of the subroutine is equal to W_{true}. If this is the case, then for each $e_i \in E'$ do not compute $\lambda_E(e_i)$ as given in the subroutine create_inputs, but let $\lambda_E(e_i)$ be equal either to the *first* oracle given to $\mathcal{B}_{\mathbf{10}}$ (if $e_i \notin \mathsf{BoxNull}(C)$) or to the *first* oracle given to $\mathcal{B}_{\mathbf{10}}$ where the argument to the oracle has been fixed as $\mathbf{0}^{\ell(n)}$ (if $e_i \in \mathsf{BoxNull}(C)$).
   - In the subroutine do_assignment, check whether the variable N' is equal to W_{true}. If this is the case, then do not compute the values $\lambda_E(\overleftarrow{\rho}(v'))$ for $v' \in W_{\mathsf{true}}$ as in the subroutine do_assignment, but let $\lambda_E(\overleftarrow{\rho}(v'))$ be either the *first* oracle given to $\mathcal{B}_{\mathbf{10}}$ (if $v' \notin \mathsf{BoxNull}(C)$) or the *first* oracle given to $\mathcal{B}_{\mathbf{10}}$, where the argument to the oracle has been fixed as $\mathbf{0}^{\ell(n)}$ (if $v' \in \mathsf{BoxNull}(C)$).
   - The previous two modifications are also done for W_{false}, but the *first* oracle is replaced with the *second* oracle.
   - Let $e_1e_2e_3 = \overrightarrow{\rho}(v)$, where v is the node mentioned in the description of way **10**. When the subroutine do_assignment is considering the node v, check whether $\lambda_E(e_2)$ and $\lambda_E(e_3)$ are the first and the second oracle of $\mathcal{B}_{\mathbf{10}}$, respectively[1]. If either $\lambda_E(e_2)$ or $\lambda_E(e_3)$ is something different, and they are not $\perp_{\mathrm{u}}$, then abort the computation sampling $\llbracket G, C \rrbracket$.

[1]The flow of these oracles through the flowchart G has to be kept track of.

- The value $\lambda_E(\overleftarrow{\rho}(v))$ is defined as follows:
  - If $\lambda_E(e_1) = \perp_{\mathrm{u}}$, then $\lambda_E(\overleftarrow{\rho}(v)) = \perp_{\mathrm{u}}$.
  - If $\lambda_E(e_1) = \mathsf{true}$, then $\lambda_E(\overleftarrow{\rho}(v))$ is equal to either the *third* oracle of $\mathcal{B}_{\mathbf{10}}$ (if $W_{\mathsf{true}} \not\subseteq \mathsf{BoxNull}(C)$ or $W_{\mathsf{false}} \not\subseteq \mathsf{BoxNull}(C)$) or the *third* oracle of $\mathcal{B}_{\mathbf{10}}$ where the argument to the oracle has been fixed as $\mathbf{0}^{\ell(n)}$ (if $W_{\mathsf{true}}, W_{\mathsf{false}} \subseteq \mathsf{BoxNull}(C)$).
  - If $\lambda_E(e_1) = \mathsf{false}$, then $\lambda_E(\overleftarrow{\rho}(v))$ is equal to either the *fourth* oracle of $\mathcal{B}_{\mathbf{10}}$ (if $W_{\mathsf{true}} \not\subseteq \mathsf{BoxNull}(C)$ or $W_{\mathsf{false}} \not\subseteq \mathsf{BoxNull}(C)$) or the *fourth* oracle of $\mathcal{B}_{\mathbf{10}}$ where the argument to the oracle has been fixed as $\mathbf{0}^{\ell(n)}$ (if $W_{\mathsf{true}}, W_{\mathsf{false}} \subseteq \mathsf{BoxNull}(C)$).

5. If the computation sampling $[\![G, C]\!]$ was aborted then go back to the 2nd step. Otherwise output the result of sampling.

The algorithm $\mathcal{B}_{\mathbf{10}}$ samples either $[\![G, C]\!]$ or $[\![G, C']\!]$. This is so, because the choices of W_{true} and W_{false} are independent from the random choices done during the sampling of $[\![G, C]\!]$. Note also that the expected running time of $\mathcal{B}_{\mathbf{10}}$ is polynomial in the size of the flowchart ($\mathcal{B}_{\mathbf{10}}$ has non-negligible chance of choosing W_{true} and W_{false} correctly, as there are only polynomially many possible choices), therefore also polynomial in the security parameter n.

4.5.2 Changing the Configurations in $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$

Obviously, our reason for introducing the ways of changing the configurations was to use them for changing the configurations in $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ to configurations in $\mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$. More concretely, we plan to turn every configuration $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ to a configuration C' where C' has no ties between sides (Def. 4.3).

Well, not actually every last configuration in $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$. There may be configurations $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$, such that $[\![G, C]\!] = \eta^{\mathcal{D}}(\perp)$ and for every configuration C', such that $C \overset{*}{\leftrightarrow} C'$, also $[\![G, C']\!] = \eta^{\mathcal{D}}(\perp)$.

When computing $[\![G, C]\!]$, we construct the functions $B_X, B_Y : \mathsf{VC}_{\mathsf{P}} \to \mathbb{B}$ (lines 5 and 6 in the "main program" in Sec. 4.4.6). If $\mathsf{SameIf}(C) = \mathsf{true}$, then necessarily $B_X = B_Y$.

Definition 4.5. Let $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ and let $S_X \in \mathsf{KV}^X_0$ and $S_Y \in \mathsf{KV}^Y_0$ be such, that $\mathsf{KnownInps}(C)$ is defined through S_X and S_Y. We say that C is *valid*, if there exists a function $B : \mathsf{VC}_{\mathsf{P}} \to \mathbb{B}$, such that $\mathsf{ValidChoice}(\mathsf{P}, X, B, S_X)$ and $\mathsf{ValidChoice}(\mathsf{P}, Y, B, S_Y)$ hold.

If $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ is not valid, then $[\![G, C]\!] = \eta^{\mathcal{D}}(\perp)$. Also, if $C \overset{*}{\leftrightarrow} C'$, then $[\![G, C']\!] = \eta^{\mathcal{D}}(\perp)$, because we cannot change $\mathsf{KnownInps}(C)$ and $\mathsf{SameIf}(C)$ while changing configurations.

It is easy to find out, whether $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ is valid or not. One can compute the set $\mathit{ValidPair}_0 \subseteq \mathsf{KV}^X_0 \times \mathsf{KV}^Y_0$, such that C is valid iff $\mathsf{KnownInps}(C)$ is defined by

$S_X \in \mathsf{KV}_0^X$ and $S_Y \in \mathsf{KV}_0^Y$, such that $(S_X, S_Y) \in \mathit{ValidPair}_0$. For each $i \in \{0, \ldots, s\}$ we define the sets $\mathit{ValidPair}_i \subseteq \mathsf{KV}_i^X \times \mathsf{KV}_i^Y$. The definition inductive, in the reverse order over i. If $i = s$ then

$$\mathit{ValidPair}_s = \{(X, Y)\} \ .$$

Let $\mathit{ValidPair}_i$ be defined. If A_i is an assignment then

$$\mathit{ValidPair}_{i-1} = \{(\mathbf{f}_{X;i}(S_X), \mathbf{f}_{Y;i}(S_Y)) \, : \, (S_X, S_Y) \in \mathit{ValidPair}_i\} \ .$$

If A_i is a vectorised choice then

$$\begin{aligned}\mathit{ValidPair}_{i-1} = \{(\mathbf{f}^{\mathsf{true}}_{X;i}(S_X), \mathbf{f}^{\mathsf{true}}_{Y;i}(S_Y)) \, : \, (S_X, S_Y) \in \mathit{ValidPair}_i\} \cup \\ \{(\mathbf{f}^{\mathsf{false}}_{X;i}(S_X), \mathbf{f}^{\mathsf{false}}_{Y;i}(S_Y)) \, : \, (S_X, S_Y) \in \mathit{ValidPair}_i\} \ .\end{aligned}$$

We now state the main result of the current Section.

Proposition 4.11. *If $(X, Y) \in \mathsf{indeps}(\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\mathcal{X}))$ then for each valid $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ there exists a configuration C', such that $C \overset{*}{\leftrightarrow} C'$ and C' has no ties between sides.*

The proposition is proved by the induction over the length of P.

Base: the program P is *skip*. Let $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ and let the configuration C' be equal to C, except that $\mathsf{InpParts}(C')$ contains two parts — the set of edges of $\mathsf{Chart}_{skip;X}$ and the set of edges of $\mathsf{Chart}_{skip;Y}$ — instead of one. We have $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\mathcal{X}) = \mathcal{X}$ an therefore $(X, Y) \in \mathcal{X}$. Thus $C \leftrightarrow C'$ by the way **5** of changing the configurations.

The rest of this section deals with the induction step. Let P be a program and A be an assignment or a vectorised choice and consider the program $\mathsf{A};\mathsf{P}$. Let $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{A};\mathsf{P};X,Y}$. Setting the stage, we define the following quantities with respect to the program $\mathsf{A};\mathsf{P}$:

- Let s be the length of the program $\mathsf{A};\mathsf{P}$ and let the sets KV_i^X and relations $\mathbf{f}_{X;i}$, $\mathbf{f}^{\mathsf{true}}_{X;i}$ and $\mathbf{f}^{\mathsf{false}}_{X;i}$, where $0 \le i \le s$, be given with respect to the program $\mathsf{A};\mathsf{P}$ (in this case, the corresponding sets and relations for the program P are simply the same ones, where $1 \le i \le s$).

- Let $S_X \in \mathsf{KV}_0^X$ and $S_Y \in \mathsf{KV}_0^Y$ be such, that $\mathsf{KnownInps}(C)$ is defined through them.

We have to relate the configuration C with some configuration $\bar{C} \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$. If A is an assignment then there exist $\bar{S}_X \in \mathsf{KV}_1^X$ and $\bar{S}_Y \in \mathsf{KV}_1^Y$, such that $\mathbf{f}_{X;1}(\bar{S}_X) = S_X$ and $\mathbf{f}_{Y;1}(\bar{S}_Y) = S_Y$. Let $\bar{C}$ be such, that $\mathsf{KnownInps}(\bar{C})$ is defined through $\bar{S}_X$ and $\bar{S}_Y$ (this uniquely determines $\bar{C}$).

If A is a vectorised choice, then there exist $\bar{S}_X \in \mathsf{KV}_1^X$ and $\bar{S}_Y \in \mathsf{KV}_1^Y$, such that either $\mathbf{f}^{\mathsf{true}}_{X;1}(\bar{S}_X) = S_X$ and $\mathbf{f}^{\mathsf{true}}_{Y;1}(\bar{S}_Y) = S_Y$ or $\mathbf{f}^{\mathsf{false}}_{X;1}(\bar{S}_X) = S_X$ and $\mathbf{f}^{\mathsf{false}}_{Y;1}(\bar{S}_Y) = S_Y$.

Assume w.l.o.g. that $\mathbf{f}^{\mathsf{true}}_{X;1}(\bar{S}_X) = S_X$ and $\mathbf{f}^{\mathsf{true}}_{Y;1}(\bar{S}_Y) = S_Y$. Let $\bar{C}$ be such, that $\mathsf{KnownInps}(\bar{C})$ is defined through $\bar{S}_X$ and $\bar{S}_Y$.

The induction assumption gives us the configuration $\bar{C}'$, such that $\bar{C} \overset{*}{\leftrightarrow} \bar{C}'$ and $\bar{C}'$ has no ties between sides. Let us expand the meaning of $\bar{C} \overset{*}{\leftrightarrow} \bar{C}'$ a bit and define the following quantities:

- Let $\bar{\mathcal{X}} = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{A}]\!](\mathcal{X})$. When talking about changing the configurations of the program P, the ways **3** and **5** refer to $\bar{\mathcal{X}}$.
- Let $\bar{C}_0, \bar{C}_1, \ldots \bar{C}_t$ be configurations of the program P, such that
  - $\bar{C}_0 = \bar{C}$ and $\bar{C}_t = \bar{C}'$;
  - $\bar{C}_0 \leftrightarrow \bar{C}_1 \leftrightarrow \cdots \leftrightarrow \bar{C}_t$.

We assume that the configurations $\bar{C}_j$, where $0 \le j \le t$, satisfy certain properties. If P is the program *skip*, then these properties are trivially satisfied. Otherwise, these properties follow from the construction of these configurations (the construction is given below) and the induction over the length of P.

P1. If $e \in \mathsf{InpKeys}(\bar{C}_j)$, where the input edge e of the flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$ is labelled with $[k]_{\mathcal{E}}$, then $k \in \mathsf{keys}(\bar{\mathcal{X}})$.

P2. $\bar{C}_0 \sqsupseteq \bar{C}_1 \sqsupseteq \cdots \sqsupseteq \bar{C}_t$.

As the next step, we define configurations C_j, where $0 \le j \le t$, of the program $\mathsf{A};\mathsf{P}$. The configuration C_t will be the configuration C' that we are looking for. Let

- $\mathsf{KnownInps}(C_j) = \mathsf{KnownInps}(C)$ and $\mathsf{SameIf}(C_j) = \mathsf{SameIf}(C)$.
- $\mathsf{OpParts}_i(C_j) = \mathsf{OpParts}_i(\bar{C}_j)$ for $2 \le i \le s$. $\mathsf{OpParts}_1(C_j)$ is defined by $\mathsf{InpParts}(\bar{C}_j)$ — two nodes of $\mathsf{2Chart}_{\mathsf{A};\mathsf{P};X,Y}$ that correspond to A belong to the same part in $\mathsf{OpParts}_1(C_j)$ iff the out-edges of these nodes (or the out-edges of the following nodes labelled by $Mk\mathcal{E}$) belong to the same part in $\mathsf{InpParts}(\bar{C}_j)$.

The elements $\mathsf{InpParts}(C_j)$, $\mathsf{EncNull}(C_j)$, $\mathsf{InpKeys}(C_j)$, $\mathsf{IfNewKeys}(C_j)$ and $\mathsf{BoxNull}(C_j)$ must be defined, too. The partition $\mathsf{InpParts}(C_0)$ puts all inputs of $\mathsf{inputs}(\mathsf{2Chart}_{\mathsf{A};\mathsf{P};X,Y})$ to the same part. The sets $\mathsf{EncNull}(C_0)$, $\mathsf{InpKeys}(C_0)$, $\mathsf{IfNewKeys}(C_0)$ and $\mathsf{BoxNull}(C_0)$ are all empty. Thus $C_0 = C$.

The elements $\mathsf{InpParts}(C_j)$, $\mathsf{EncNull}(C_j)$, $\mathsf{InpKeys}(C_j)$, $\mathsf{IfNewKeys}(C_j)$ and $\mathsf{BoxNull}(C_j)$, where $1 \le j \le t$, depend on the configuration C_{j-1} and the way of changing the configurations that turned $\bar{C}_{j-1}$ to $\bar{C}_j$. Together with defining the elements of C_j, we also show that $C_{j-1} \overset{*}{\leftrightarrow} C_j$.

Before enumerating the ways of changing $\bar{C}_{j-1}$ to $\bar{C}_j$ we give a *partial* specification of $\mathsf{InpParts}(C_j)$. Namely, there has to exist a function $\mathsf{descend}_j : \mathsf{inputs}(\mathsf{2Chart}_{\mathsf{A};\mathsf{P};X,Y}) \to \mathsf{InpParts}(\bar{C}_j)$, such that:

1. For each $e \in \mathsf{inputs}(\mathsf{2Chart}_{\mathsf{A};\mathsf{P};X,Y})$ there exists $e' \in \mathsf{descend}_j(e)$, such that $e \in \mathsf{origins}(e')$.

2. For each $e_1, e_2 \in \mathsf{inputs}(\mathsf{2Chart}_{\mathsf{A};\mathsf{P};X,Y})$, if $\mathsf{descend}_j(e_1) \neq \mathsf{descend}_j(e_2)$, then e_1 and e_2 belong to different parts of $\mathsf{InpParts}(C_j)$. In other words, if e_1 and e_2 belong to the same part of $\mathsf{InpParts}(C_j)$, then $\mathsf{descend}_j(e_1) = \mathsf{descend}_j(e_2)$.

3. If A is $\langle x_1, \ldots, x_k\rangle := b\,?\langle y_1, \ldots, y_k\rangle : \langle z_1, \ldots, z_k\rangle$, e is an input edge of $\mathsf{2Chart}_{\mathsf{A};\mathsf{P};X,Y}$ labelled with b and for all $e' \in \mathsf{descend}_j(e)$, where $e \in \mathsf{origins}(e')$, the edge e' is labelled with $[k]_{\mathcal{E}}$, then either $e \notin \mathsf{matters}(C_j)$ or for all $e' \in \mathsf{inputs}(\mathsf{2Chart}_{\mathsf{P};X,Y})$, where $e \in \mathsf{origins}(e')$, the edge e' is labelled with $[k]_{\mathcal{E}}$.

   In other words, if there is a node labelled with $? :$ (and not $[? :]_{\mathcal{E}}$) corresponding to A, and if this node is not fictitious (i.e. its out-edge is not fictitious), then $\mathsf{descend}_j(e)$ must contain the out-edge of this node or some other node having the same properties (label and fictitiousness).

Because of the 1st and 2nd properties, if $\bar{E}$ is a part in $\mathsf{InpParts}(\bar{C}_j)$, then $\mathsf{descend}_j^{-1}(\bar{E})$ is a union of parts in $\mathsf{InpParts}(C_j)$. Also, because of these properties and by the definition of $\mathsf{OpParts}_1(C_t)$, the configuration $C_t = C'$ satisfies the requirements put on it by the proposition.

We also partially specify $\mathsf{InpKeys}(C_j)$ and $\mathsf{BoxNull}(C_j)$:

P3. Let $\bar{e} \in \mathsf{inputs}(\mathsf{2Chart}_{\mathsf{P};X,Y})$. If $\bar{e}$ is labelled with $[k]_{\mathcal{E}}$ for some $k \in \mathbf{Var}$ and $\bar{e} \in \mathsf{InpKeys}(\bar{C}_j)$ [resp. $\bar{e} \in \mathsf{BoxNull}(\bar{C}_j)$], then for each $e \in \mathsf{origins}(\bar{e})$, where e is labelled with $[k']_{\mathcal{E}}$ for some $k' \in \mathbf{Var}$, the inclusion $e \in \mathsf{InpKeys}(C_j)$ [resp. $e \in \mathsf{BoxNull}(C_j)$] holds.

Consider now the possible ways of turning $\bar{C}_{j-1}$ to $\bar{C}_j$.

Way 1

Let A be the assignment $x := o(x_1, \ldots, x_k)$ or the vectorised choice $\langle x_1, \ldots, x_k\rangle := b\,?\langle y_1, \ldots, y_k\rangle : \langle z_1, \ldots, z_k\rangle$. The components $\mathsf{InpKeys}(C_j)$, $\mathsf{IfNewKeys}(C_j)$, $\mathsf{BoxNull}(C_j)$ and $\mathsf{EncNull}(C_j)$ are equal to the corresponding components of C_{j-1}. Let $\bar{E} \in \mathsf{InpParts}(\bar{C}_{j-1})$ be the part that was split to two parts $\bar{E}_f, \bar{E}_{nf} \in \mathsf{InpParts}(\bar{C}_j)$, where $\bar{E}_f$ contains only fictitious inputs. Let $E_f = E_1 \cup E_2$, where E_1 contains all those edges that are labelled with variables (or black boxes of these variables) in the left hand side of the assignment or vectorised choice A and E_2 contains all edges that are labelled with something else. Let V be the set of nodes corresponding to A, such that the out-edges of the nodes of V are in E_1. All nodes in V are fictitious. The first step in changing the configuration C_{j-1} to C_j is done by way **2**, making V an extra part of $\mathsf{OpParts}_1(C_j)$.

Let $E_{\mathsf{A}} = \mathsf{descend}_{j-1}^{-1}(\bar{E})$. $\mathsf{InpParts}(C_j)$ contains the following parts:

- For each $E \in \mathsf{InpParts}(C_{j-1})$, where $E \cap E_{\mathsf{A}} = \emptyset$, $\mathsf{InpParts}(C_j)$ contains E.

- For each $E \in \mathsf{InpParts}(C_{j-1})$, where $E \cap E_{\mathsf{A}} \neq \emptyset$, let E_f contain all edges $e \in E_{\mathsf{A}} \cap E$, where

– for all input edges e' of $\mathsf{2Chart}_{\mathsf{P};X,Y}$, if $e \in \mathsf{origins}(e')$, then $e' \in \mathsf{fictitious}(\bar{C}_j)$;

– there exists an input edge e' of $\mathsf{2Chart}_{\mathsf{P};X,Y}$, such that $e \in \mathsf{origins}(e')$ and $e' \in \bar{E}_f$.

If $E_f = \emptyset$ then $\mathsf{InpParts}(C_j)$ contains E. If $E_f \neq \emptyset$ then $\mathsf{InpParts}(C_j)$ contains E_f and $E \backslash E_f$. As the edges in E_f are all fictitious, way **1** can be used to split the set E in two, when changing the configuration C_{j-1} to C_j.

It should be rather obvious, how $\mathsf{descend}_j$ is defined. For these sets E in $\mathsf{InpParts}(C_{j-1})$, where $E \cap E_{\mathsf{A}} = \emptyset$, the equality $\mathsf{descend}_{j-1}(e) = \mathsf{descend}_j(e)$ holds for all $e \in E$. For these sets E in $\mathsf{InpParts}(C_{j-1})$ that were split to two sets E_f and $E \backslash E_f$ in $\mathsf{InpParts}(C_j)$, the equality $\mathsf{descend}_{j-1}(e) = \mathsf{descend}_j(e)$ holds for all $e \in E \backslash E_f$ with the exception that $\bar{E}_{nf}$ is substituted for $\bar{E}$. For the edges $e \in E_f$ we define $\mathsf{descend}_j(e) = \bar{E}_f$.

Ways 2, 6, 7, 8, 9 and 10

The yet undefined components of C_j are equal to the corresponding components of C_{j-1}, except for

- For way **6**, $\mathsf{EncNull}(C_j) = \mathsf{EncNull}(C_{j-1}) \cup \mathsf{EncNull}(\bar{C}_j)$.
- For way **7**, $\mathsf{BoxNull}(C_j) = \mathsf{BoxNull}(C_{j-1}) \cup \mathsf{BoxNull}(\bar{C}_j)$.
- For way **10**, $\mathsf{IfNewKeys}(C_j) = \mathsf{IfNewKeys}(C_{j-1}) \cup \mathsf{IfNewKeys}(\bar{C}_j)$.

The function $\mathsf{descend}_j$ is equal to $\mathsf{descend}_{j-1}$. The same change that turned $\bar{C}_{j-1}$ to $\bar{C}_j$ also turns C_{j-1} to C_j (this follows from the property P3).

Way 3

The components $\mathsf{IfNewKeys}(C_j)$, $\mathsf{BoxNull}(C_j)$ and $\mathsf{EncNull}(C_j)$ are equal to the corresponding components of C_{j-1}. The set $\mathsf{InpKeys}(C_j)$ contains at least as much elements as $\mathsf{InpKeys}(C_{j-1})$; below we add more elements to it. Let $\bar{e}_1, \ldots, \bar{e}_l$ and k be defined as in the wording of the way **3** of changing the configurations. Let $\bar{E} = \{\bar{e}_1, \ldots, \bar{e}_l\}$. Consider the rule that was used to derive $k \in \mathsf{keys}(\bar{\mathcal{X}})$; this consideration allows us to make certain assumptions about $\mathcal{X}$.

Rule (4.14) or Rule (4.30) The partitions $\mathsf{InpParts}(C_{j-1})$ and $\mathsf{InpParts}(C_j)$ are equal and $\mathsf{descend}_j = \mathsf{descend}_{j-1}$. Let $E_{\mathsf{A}} = \mathsf{descend}_{j-1}^{-1}(\bar{E})$. The edges in E_{A} are all labelled with $[k]_{\mathcal{E}}$. The set E_{A} is a union of some parts $E \in \mathsf{InpParts}(C_{j-1})$. The way **3** of changing the configurations is applied to C_{j-1} with respect to these sets E. The elements in all these sets E are added to $\mathsf{InpKeys}(C_j)$.

Rule (4.14^1) Let A be $k := k'$. This case is similar to the rule (4.14). The only difference is that the edges in E_{A} are labelled with $[k']_{\mathcal{E}}$.

Rule (4.15) Here we just take $C_j = C_{j-1}$ and $\mathsf{descend}_j = \mathsf{descend}_{j-1}$.

Rule (4.31) The function $\mathsf{descend}_j$ equals $\mathsf{descend}_{j-1}$. Let the vectorised choice A be $\langle x_1,\ldots,x_m\rangle := b\,?\langle y_1,\ldots,y_m\rangle : \langle z_1,\ldots,z_m\rangle$. Because of the assumed relationship between $\mathsf{KnownInps}(C)$ and $\mathsf{KnownInps}(\bar{C})$, the input edges of $\mathsf{2Chart}_{\mathsf{A};\mathsf{P};X,Y}$ labelled with $z_1,\ldots,z_m$ or $[z_1]_{\mathcal{E}},\ldots,[z_m]_{\mathcal{E}}$ are not elements of $\mathsf{matters}(C_{j-1})$. The key k that we fixed before enumerating the possible rules for deriving $k \in \mathsf{keys}(\bar{\mathfrak{X}})$, is some x_r for $r \in \{1,\ldots,m\}$.

Let $E_{\mathsf{A}} = \mathsf{descend}_{j-1}^{-1}(\bar{E})$. Let $E^{(1)},\ldots,E^{(l)} \in \mathsf{InpParts}(C_{j-1})$ be such, that $E_{\mathsf{A}} = E^{(1)} \cup \cdots \cup E^{(l)}$. Each such set E_i only contains edges labelled with b, $[y_r]_{\mathcal{E}}$ or $[z_r]_{\mathcal{E}}$. Let $E_b^{(i)}$, $E_{[y_r]_{\mathcal{E}}}^{(i)}$ and $E_{[z_r]_{\mathcal{E}}}^{(i)}$ be the subsets of $E^{(i)}$ containing all those edges that are labelled with b, $[y_r]_{\mathcal{E}}$ or $[z_r]_{\mathcal{E}}$, respectively. We add the elements of $E_{[y_r]_{\mathcal{E}}}^{(i)}$ and $E_{[z_r]_{\mathcal{E}}}^{(i)}$ to $\mathsf{InpKeys}(C_j)$. The difference between partitions $\mathsf{InpParts}(C_{j-1})$ and $\mathsf{InpParts}(C_j)$ is, that instead of including the part $E^{(i)}$, the partition $\mathsf{InpParts}(C_j)$ includes the parts $E_b^{(i)}$, $E_{[y_r]_{\mathcal{E}}}^{(i)}$ and $E_{[z_r]_{\mathcal{E}}}^{(i)}$. For each set $E^{(i)}$, we have to make a number of steps to turn C_{j-1} to C_j:

1. If neither $E_{[z_r]_{\mathcal{E}}}^{(i)}$ nor $E_b^{(i)} \cup E_{[y_r]_{\mathcal{E}}}^{(i)}$ are empty, then we make a step using the way **1** and split the set $E^{(i)}$ to sets $E_{[z_r]_{\mathcal{E}}}^{(i)}$ and $E_b^{(i)} \cup E_{[y_r]_{\mathcal{E}}}^{(i)}$.

2. If neither $E_b^{(i)}$ nor $E_{[y_r]_{\mathcal{E}}}^{(i)}$ are empty, then we make a step using the way **5** and split the set $E_b^{(i)} \cup E_{[y_r]_{\mathcal{E}}}^{(i)}$ to sets $E_b^{(i)}$ and $E_{[y_r]_{\mathcal{E}}}^{(i)}$. We can use that way, because $(\{[y_r]_{\mathcal{E}}\},\{b\}) \in \mathsf{indeps}(\mathfrak{X})$ — see the antecedents of rule (4.31).

3. If $E_{[y_r]_{\mathcal{E}}}^{(i)}$ is not empty, then we add its elements to $\mathsf{InpKeys}(C_j)$ using the way **3**. If $E_{[z_r]_{\mathcal{E}}}^{(i)}$ is not empty, then we add its elements to $\mathsf{InpKeys}(C_j)$ using the way **3**.

In the rest of the proof, while we are making the steps to turn C_{j-1} to C_j, we will no longer add the qualifiers “if the set …is not empty”. We will implicitly assume that they are there. Neither will we explicitly refer to the antecedents of the rules, we will just say “split the set …to sets …and …by using way **5**”.

Way 4

The components $\mathsf{InpKeys}(C_j)$, $\mathsf{IfNewKeys}(C_j)$, $\mathsf{InpParts}(C_j)$ and $\mathsf{EncNull}(C_j)$ are equal to the corresponding components of C_{j-1}. The set $\mathsf{BoxNull}(C_j)$ contains at least as much elements as $\mathsf{BoxNull}(C_{j-1})$, below we will add more elements to it. The function $\mathsf{descend}_j$ is equal to $\mathsf{descend}_{j-1}$.

Let $\bar{E} \in \mathsf{InpParts}(\bar{C}_{j-1})$ be the set, where $\bar{E} = \mathsf{BoxNull}(\bar{C}_j)\backslash\mathsf{BoxNull}(\bar{C}_{j-1})$. Let $[k]_{\mathcal{E}}$ be the label of the edges in $\bar{E}$.

If A is an assignment $k := \mathcal{G}en()$, then we obtain $\mathsf{BoxNull}(C_j)$ by adding to $\mathsf{BoxNull}(C_{j-1})$ all nodes whose out-edges are in $\bar{E}$. The configuration C_{j-1} can be turned to C_j by using way **7** of changing the configurations.

If A has some other form, then the properties of $\mathsf{descend}_{j-1}$ and property P3 give us that the set

$$E_{\mathsf{A}} = \{e : \bar{e} \in \bar{E}, e \in \mathsf{origins}(\bar{E}), e \text{ is labelled with some } [k']_{\mathcal{E}}\}$$

is a subset of $\mathsf{InpKeys}(C_j)$. The set $\mathsf{BoxNull}(C_j)$ is obtained from $\mathsf{BoxNull}(C_{j-1})$ by adding the elements of E_{A} to it. The configuration C_{j-1} can be turned to C_j by using the way **4** of changing the configurations (once for each $E^{(i)} \in \mathsf{InpParts}(C_{j-1})$, where $E^{(i)} \subseteq E_{\mathsf{A}}$).

Way 5

Let $\bar{E}, \bar{X}, \bar{Y}, \bar{E}_X, \bar{E}_Y$ be defined as in the wording of the way **5** of the configurations. Consider the rule that was used to derive $(\bar{X}, \bar{Y}) \in \mathsf{indeps}(\bar{\mathcal{X}})$.

Rule (4.9) The sets $\mathsf{InpKeys}(C_j)$, $\mathsf{EncNull}(C_j)$, $\mathsf{IfNewKeys}(C_j)$ and $\mathsf{BoxNull}(C_j)$ are equal to the corresponding components of C_{j-1}. Let A be $x := o(x_1, \ldots, x_k)$ and let $E_{\mathsf{A}} = \mathsf{descend}_{j-1}^{-1}(\bar{E})$. No edge in $\bar{E}$ is labelled with x or $[x]_{\mathcal{E}}$, therefore the edges in E_{A} are only labelled with such elements of $\widetilde{\mathbf{Var}}$ that are also the labels of some edge in $\bar{E}$. For each input edge e of $\mathsf{2Chart}_{\mathsf{A;P};X,Y}$ in the set E_{A} there is a corresponding input edge $\bar{e}$ of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ in the set $\bar{E}$; because of the details of the construction of flowcharts given in Sec. 4.4.2, the edges e and $\bar{e}$ are the same.

Let $E^{(1)}, \ldots, E^{(l)} \in \mathsf{InpParts}(C_{j-1})$ be such, that E_{A} is equal to their union. Let $E_X^{(i)} = E^{(i)} \cap \bar{E}_X$ and $E_Y^{(i)} = E^{(i)} \cap \bar{E}_Y$ for all $i \in \{1, \ldots, l\}$. The partition $\mathsf{InpParts}(C_j)$ is obtained from the partition $\mathsf{InpParts}(C_{j-1})$ by removing the sets $E^{(i)}$ from it and adding the (nonempty) sets $E_X^{(i)}, E_Y^{(i)}$ to it. Splitting each of the sets $E^{(i)}$ to sets $E_X^{(i)}$ and $E_Y^{(i)}$ turns the configuration C_{j-1} to the configuration C_j, these splittings can be done by the way **5** of changing the configurations. The function $\mathsf{descend}_j$ is equal to $\mathsf{descend}_{j-1}$ everywhere except on the elements of E_{A}. Applying $\mathsf{descend}_j$ to $e \in E_{\mathsf{A}}$ gives either $\bar{E}_X$ or $\bar{E}_Y$, depending on whether $e \in E_X^{(i)}$ or $e \in E_Y^{(i)}$ for some $i \in \{1, \ldots, l\}$.

Rule (4.10) The sets $\mathsf{InpKeys}(C_j)$, $\mathsf{EncNull}(C_j)$, $\mathsf{IfNewKeys}(C)$ and $\mathsf{BoxNull}(C_j)$ are equal to the corresponding components of C_{j-1}. Let A be $x := o(x_1, \ldots, x_k)$ and let $E_{\mathsf{A}} = \mathsf{descend}_{j-1}^{-1}(\bar{E})$. Let $E^{(1)}, \ldots, E^{(l)} \in \mathsf{InpParts}(C_{j-1})$ be the parts whose union is E_{A}. Only one of the sets $\bar{E}_X$ and $\bar{E}_Y$ may contain edges labelled with x or $[x]_{\mathcal{E}}$, assume w.l.o.g. that $\bar{E}_X$ contains no such edges. Let V be the set of nodes labelled with o whose outputs are the members of $\bar{E}$ (also including such a node whose output is an input to a node that is labelled with $Mk\mathcal{E}$ and whose output is a member of $\bar{E}$). The possible elements of the set E_{A} are the elements of $\bar{E}$ (that are not labelled with x or $[x]_{\mathcal{E}}$) and the inputs to the nodes in V.

Define now the sets $E_Y^{(i)}$, where $1 \le i \le l$, as follows: $E_Y^{(i)}$ contains all such elements $e \in E^{(i)}$ where

- either $e \in \bar{E}_Y$,
- or e is an input to a node $v \in V$ and the requirements on $\mathsf{descend}_j$ state that $\mathsf{descend}_j(e)$ must include the out-edge of v
- or e is an input to a node $v \in V$ and $e \notin \bar{E}_X$.

Let $E_X^{(i)} = E^{(i)} \backslash E_Y^{(i)}$. The partition $\mathsf{InpParts}(C_j)$ is obtained from $\mathsf{InpParts}(C_{j-1})$ by removing the sets $E^{(i)}$ from it and adding the (nonempty) sets $E_X^{(i)}, E_Y^{(i)}$ to it. Splitting each of the sets $E^{(i)}$ to sets $E_X^{(i)}$ and $E_Y^{(i)}$ turns the configuration C_{j-1} to the configuration C_j, these splittings can be done by the way **5** of changing the configurations. Indeed, the edges in sets $E_X^{(i)}$ may be labelled only with the elements of $\bar{X}$ and the edges in sets $E_Y^{(i)}$ by elements of $\bar{Y} \cup \{x_1, \ldots, x_k\}$. But the pair $(\bar{X}, \bar{Y} \cup \{x_1, \ldots, x_k\})$ is a member of $\mathsf{indeps}(\mathfrak{X})$ by the rule (4.10).

The function $\mathsf{descend}_j$ is equal to $\mathsf{descend}_{j-1}$ everywhere except on the elements of E_{A}. Applying $\mathsf{descend}_j$ to $e \in E_{\mathsf{A}}$ gives either $\bar{E}_X$ or $\bar{E}_Y$, depending on whether $e \in E_X^{(i)}$ or $e \in E_Y^{(i)}$ for some $i \in \{1, \ldots, l\}$.

Rule (4.10^1) This rule is similar to rule (4.10). The only difference is, that some of the relevant input edges of $\mathsf{2Chart}_{\mathsf{A};\mathsf{P};X,Y}$ are labelled with a black box, not with a variable. The argumentation for rule (4.10) can be turned to the argumentation for rule (4.10^1) by replacing the set $\{x_1, \ldots, x_k\}$ with the set $\{[k]_{\mathcal{E}}, y\}$.

Rule (4.10^2) Define $\mathsf{InpKeys}(C_j)$, $\mathsf{IfNewKeys}(C_j)$, $\mathsf{EncNull}(C_j)$, $\mathsf{BoxNull}(C_j)$, E_{A}, $E^{(1)}, \ldots, E^{(l)}$ as in the argumentation for rule (4.10). Let the assignment A be $x := y$.

We define the sets $E_X^{(i)}$ and $E_Y^{(i)}$ ($i \in \{1, \ldots, l\}$) as follows. Let $e \in E^{(i)}$.

- If e is not labelled with y or $[y]_{\mathcal{E}}$, then $e \in E_X^{(i)}$ iff $e \in \bar{E}_X$. Otherwise $e \in E_Y^{(i)}$.
- If e is labelled with y or $[y]_{\mathcal{E}}$, then e may count as an input to $\mathsf{2Chart}_{\mathsf{P};X,Y}$ twice — once labelled with y or $[y]_{\mathcal{E}}$ and once labelled with x or $[x]_{\mathcal{E}}$. If both instances of e belong to $\bar{E}_X$ or both belong to $\bar{E}_Y$, then e also belongs to corresponding $E_X^{(i)}$ or $E_Y^{(i)}$. If one of the instances belongs to $\bar{E}_X$ and the other one belongs to $\bar{E}_Y$ then e belongs to $E_X^{(i)}$ iff the instance of e labelled with x or $[x]_{\mathcal{E}}$ belongs to $\bar{E}_X$, otherwise $e \in E_Y^{(i)}$. If e only counts once as an input to $\mathsf{2Chart}_{\mathsf{P};X,Y}$ then we apply the previous case.

The partition $\mathsf{InpParts}(C_j)$ and the function $\mathsf{descend}_j$ are defined as in the argumentation for rule (4.10). The configuration C_{j-1} is changed to C_j by splitting the sets $E^{(i)}$ to sets $E_X^{(i)}$ and $E_Y^{(i)}$, using way **5**.

Rule (4.11) Let A be $x := \mathcal{E}nc(k,y)$ and let $E_{\mathsf{A}} = \mathsf{descend}_{j-1}^{-1}(\bar{E})$. Let $E^{(1)},\dots,E^{(l)} \in \mathsf{InpParts}(C_{j-1})$ be the parts whose union is E_{A}. Only one of the sets $\bar{E}_X$ and $\bar{E}_Y$ may contain edges labelled with x or $[x]_{\mathcal{E}}$, assume w.l.o.g. that $\bar{E}_X$ contains no such edges. Let V be the set of nodes labelled with o whose outputs are the members of $\bar{E}$ (also including such a node whose output is an input to a node that is labelled with $Mk\mathcal{E}$ and whose output is a member of $\bar{E}$). Because of the structure of the analysis $\mathbb{A}^{(\mathbf{Var})}[\![\cdot]\!]$ and the construction of the flowcharts, the set V has at most two elements: there are at most one node in $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ and at most one node in $\mathsf{Chart}_{\mathsf{A};\mathsf{P};Y}$ that correspond to the assignment A. Denote these nodes by v_X and v_Y (if they exist) and denote their inputs that are labelled with y by $e_{y;X}$ and $e_{y;Y}$ and their inputs that are labelled with $[k]_{\mathcal{E}}$ by $e_{[k]_{\mathcal{E}};X}$ and $e_{[k]_{\mathcal{E}};Y}$. The possible elements of the set E_{A} are the elements of $\bar{E}$ (that are not labelled with x or $[x]_{\mathcal{E}}$) and the inputs to the nodes in V.

Consider a set $E^{(i)}$. The edge $e_{y;Z} \in E^{(i)}$, where Z is either X or Y, satisfies exactly one of the following statements:

ES1. either $e_{y;Z}$ is not an input to $\mathsf{2Chart}_{\mathsf{P};X,Y}$ or $e_{y;Z} \notin \mathsf{matters}(\bar{C}_{j-1})$;

ES2. $e_{y;Z} \in \mathsf{matters}(\bar{C}_{j-1})$ and $e_{y;Z} \in \bar{E}_X$;

ES3. $e_{y;Z} \in \mathsf{matters}(\bar{C}_{j-1})$ and $e_{y;Z} \in \bar{E}_Y$;

There is one more case, if $e_{y;Z} \notin E_{\mathsf{A}}$:

ES4. $e_{y;Z} \in \mathsf{matters}(\bar{C}_{j-1})$ and $e_{y;Z} \notin \bar{E}$.

Again, we partition the set $E^{(i)}$. This partition includes the sets $E_X^{(i)}$ and $E_Y^{(i)}$, but it may include more parts. An edge $e \in E^{(i)}$ that is neither $e_{y;X}$, $e_{y;Y}$, $e_{[k]_{\mathcal{E}};X}$ nor $e_{[k]_{\mathcal{E}};Y}$ belongs to $E_X^{(i)}$ iff it belongs to $\bar{E}_X$, otherwise it belongs to $E_Y^{(i)}$. An edge $e_{y;Z} \in E^{(i)}$, where Z is either X or Y, belongs to either $E_y^{(i)}$, $E_X^{(i)}$ or $E_Y^{(i)}$, depending on whether the statement ES1, ES2 or ES3 is true for $e_{y;Z}$. If ES1 is true, then the part $E_y^{(i)}$ exists. If $e_{[k]_{\mathcal{E}};Z} \in E^{(i)}$ then there is a part $E_{[k]_{\mathcal{E}}}^{(i)}$ that contains $e_{[k]_{\mathcal{E}};Z}$. Similarly to previous cases, we have now defined $\mathsf{InpParts}(C_j)$: it is equal to $\mathsf{InpParts}(C_{j-1})$, except that the parts $E^{(i)}$ are replaced with parts $E_X^{(i)}$, $E_Y^{(i)}$, $E_y^{(i)}$ and $E_{[k]_{\mathcal{E}}}^{(i)}$.

The mapping $\mathsf{descend}_j$ is equal to $\mathsf{descend}_{j-1}$, except that $\mathsf{descend}_j(e) = \bar{E}_X$ for $e \in E_X^{(i)}$ and $\mathsf{descend}_j(e) = \bar{E}_Y$ for $e \in E_Y^{(i)}$, $e \in E_y^{(i)}$ and $e \in E_{[k]_{\mathcal{E}}}^{(i)}$.

Adding the edges $e_{[k]_{\mathcal{E}};X}$ and $e_{[k]_{\mathcal{E}};Y}$ to the set $\mathsf{InpKeys}(C_{j-1})$ gives us set $\mathsf{InpKeys}(C_j)$ and adding them to the set $\mathsf{BoxNull}(C_{j-1})$ gives us the set $\mathsf{BoxNull}(C_j)$. The set $\mathsf{EncNull}(C_j)$ equals $\mathsf{EncNull}(C_{j-1}) \cup \{v_X, v_Y\}$.

Several steps are necessary for turning the configuration C_{j-1} to C_j:

Step 1. For each $i \in \{1,\dots,l\}$, where $E_{[k]_{\mathcal{E}}}^{(i)}$ exists, split the set $E^{(i)}$ to sets $E_{[k]_{\mathcal{E}}}^{(i)}$ and $E_X^{(i)} \cup E_Y^{(i)} \cup E_y^{(i)}$, using the way **5** of changing the configurations.

Step 2. For each $i \in \{1,\dots,l\}$, where $E_{[k]_{\mathcal{E}}}^{(i)}$ exists, add the element of $E_{[k]_{\mathcal{E}}}^{(i)}$ to $\mathsf{InpKeys}(C_j)$ by using the way **3** of changing the configurations.

Step 3. Add the elements in the part(s) containing $e_{[k]_\mathcal{E};X}$ and $e_{[k]_\mathcal{E};Y}$ to $\mathsf{BoxNull}(C_j)$, using the way **4** of changing the configurations.

Step 4. We can now use the way **6** of changing the configurations to add v_X and v_Y to the set $\mathsf{EncNull}(C_j)$. We do it.

Step 5. For each $i \in \{1,\dots,l\}$, where $E_y^{(i)}$ exists, the edge $e_{y;Z} \in E_y^{(i)}$ is fictitious. We split the set $E_X^{(i)} \cup E_Y^{(i)} \cup E_y^{(i)}$ to sets $E_X^{(i)} \cup E_Y^{(i)}$ and $E_y^{(i)}$, using the way **1** of changing the configurations.

Step 6. For each $i \in \{1,\dots,l\}$, we split the set $E_X^{(i)} \cup E_Y^{(i)}$ to sets $E_X^{(i)}$ and $E_Y^{(i)}$, using the way **5** of changing the configurations. We have shown $C_{j-1} \overset{*}{\leftrightarrow} C_j$.

Rule (4.12) This rule is handled similarly to the rule (4.9). In addition to splitting the sets $E_X^{(i)}$ and $E_Y^{(i)}$ we also have to use the way **8** of changing the configurations to turn $\mathsf{OpParts}_1(C_{j-1})$ to $\mathsf{OpParts}_1(C_j)$.

Rule (4.13) This rule is again handled similarly to the rule (4.9). In addition to splitting the sets $E_X^{(i)}$ and $E_Y^{(i)}$ we also have to use the way **9** of changing the configurations to turn $\mathsf{OpParts}_1(C_{j-1})$ to $\mathsf{OpParts}_1(C_j)$.

Rule (4.28) Let A be $\langle x_1,\dots,x_k\rangle := b\,?\langle y_1,\dots,y_k\rangle : \langle z_1,\dots,z_k\rangle$. The sets $\mathsf{EncNull}(C_j)$ and $\mathsf{BoxNull}(C_j)$ are equal to the corresponding components of C_{j-1}. Because of the assumed relationship of $\mathsf{KnownInps}(C)$ and $\mathsf{KnownInps}(\bar{C})$, the input edges labelled with $z_1,\dots,z_k$ or $[z_1]_\mathcal{E},\dots,[z_k]_\mathcal{E}$ are not elements of $\mathsf{matters}(C_{j-1})$. Let $E_\mathsf{A} = \mathsf{descend}_{j-1}^{-1}(\bar{E})$ and let $E^{(1)},\dots,E^{(l)} \in \mathsf{InpParts}(C_{j-1})$ be the sets whose union is E_A. Let V be the set of nodes labelled with $[?:]_\mathcal{E}$ whose outputs are the members of $\bar{E}$. The edges in E_A may be labelled with b, $[y_i]_\mathcal{E}$, $[z_i]_\mathcal{E}$ or the labels of elements of $\bar{E}$ (except $[x_i]_\mathcal{E}$).

For each edge $e_b \in E_\mathsf{A}$, labelled with b, at least one of the following claims holds:

- e_b is fictitious in C_{j-1};
- the set V contains all non-fictitious nodes that have e_b as one of their inputs.

Indeed, suppose that the edge e_b is not used only at the nodes that correspond to A and whose outputs are labelled with some $[x_i]_\mathcal{E}$. I.e. e_b is also used at nodes whose outputs are labelled with some x_i. This case is not possible, because the consequent of rule (4.28) does not contain any of the variables x_i.

Suppose now that the edge e_b is only used at the nodes corresponding to A and whose outputs are labelled with some $[x_i]_\mathcal{E}$. Suppose that j is the smallest number where the configuration $\bar{C}_{j-1}$ is turned to $\bar{C}_j$ using way **5** of changing the configurations, such that the necessary condition $(\bar{X},\bar{Y}) \in \mathsf{indeps}(\bar{\mathfrak{X}})$ is derived using either the rule (4.28) or (4.29). Then the set $\bar{E}$ must contain all (non-fictitious) input edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ that are labelled with $[x_i]_\mathcal{E}$. If j is not the smallest number with such property, then either $e_b \notin E_\mathsf{A}$ or the edge e_b is fictitious, as shown by the following construction of C_j (and also by the construction of C_j at the handling of rule (4.29) below).

We start with the function $\mathsf{descend}_j$. It is equal to $\mathsf{descend}_{j-1}$, except for edges $e \in E_{\mathsf{A}}$. On these edges e it is equal to either $\bar{E}_X$ or $\bar{E}_Y$, depending on which of these two sets contains an edge e', such that $e \in \mathsf{origins}(e')$. If both sets contain such edge e' (this may only happen if e is labelled with b), then the value $\mathsf{descend}_j(e)$ may be freely chosen.

For a set $E^{(i)}$, where $i \in \{1, \dots, l\}$, we define the following subsets:

- The sets $E^{(i)}_{b;X}$, $E^{(i)}_{b;Y}$, $E^{(i)}_{[y_{i_1}]_\mathcal{E};X}, \dots, E^{(i)}_{[y_{i_m}]_\mathcal{E};X}$, $E^{(i)}_{[y_{i_{l+1}}]_\mathcal{E};Y}, \dots, E^{(i)}_{[y_{i_m}]_\mathcal{E};Y}$, $E^{(i)}_{[z_{i_1}]_\mathcal{E};X}, \dots, E^{(i)}_{[z_{i_m}]_\mathcal{E};X}$ and $E^{(i)}_{[z_{i_{l+1}}]_\mathcal{E};Y}, \dots, E^{(i)}_{[z_{i_m}]_\mathcal{E};Y}$. Each of those sets contains all edges $e \in E^{(i)}$, such that the label $\lambda_I(e)$ and the index of $\mathsf{descend}_j(e)$ (either X or Y) are equal to the subscript(s) of the set.

- The sets $E^{(i)}_X = E^{(i)} \cap \bar{E}_X$ and $E^{(i)}_Y = E^{(i)} \cap \bar{E}_Y$.

All these defined sets together constitute a partition of $E^{(i)}$. The partition $\mathsf{InpParts}(C_j)$ is obtained from the partition $\mathsf{InpParts}(C_{j-1})$ by replacing the parts $E^{(i)}$ with all the parts defined above.

Several steps are necessary for turning the configuration C_{j-1} to C_j. During the description of turning C_{j-1} to C_j we also define the sets $\mathsf{InpKeys}(C_j)$ and $\mathsf{IfNewKeys}(C_j)$. These sets contain at least as many elements as their counterparts in C_{j-1}, but may also contain more elements.

Step 1. Using the way **9** of changing the configurations, change $\mathsf{OpParts}_1(C_{j-1})$ to $\mathsf{OpParts}_1(C_j)$.

Step 2. For each $i \in \{1, \dots, l\}$, split $E^{(i)}$ to sets $E^{(i)}_{[z_{i_1}]_\mathcal{E};X}$, $\dots$, $E^{(i)}_{[z_{i_m}]_\mathcal{E};X}$, $E^{(i)}_{[z_{i_{l+1}}]_\mathcal{E};Y}$, $\dots$, $E^{(i)}_{[z_{i_m}]_\mathcal{E};Y}$ and

$$E^{(i)}_{b;X} \cup E^{(i)}_{b;Y} \cup E^{(i)}_X \cup E^{(i)}_Y \cup \bigcup_{h=1}^{m} E^{(i)}_{[y_{i_h}]_\mathcal{E};X} \cup \bigcup_{h=l+1}^{m} E^{(i)}_{[y_{i_h}]_\mathcal{E};Y} \tag{4.39}$$

using the way **1** (several times) of changing the configurations.

Step 3. For each $i \in \{1, \dots, l\}$, split (4.39) to $E^{(i)}_{b;X} \cup E^{(i)}_{b;Y} \cup E^{(i)}_X \cup E^{(i)}_Y$ and

$$\bigcup_{h=1}^{m} E^{(i)}_{[y_{i_h}]_\mathcal{E};X} \cup \bigcup_{h=l+1}^{m} E^{(i)}_{[y_{i_h}]_\mathcal{E};Y} \tag{4.40}$$

using the way **5** of changing the configurations.

Step 4. For each $i \in \{1, \dots, l\}$, split (4.40) to sets $E^{(i)}_{[y_{i_1}]_\mathcal{E};X}$, $\dots$, $E^{(i)}_{[y_{i_m}]_\mathcal{E};X}$, $E^{(i)}_{[y_{i_{l+1}}]_\mathcal{E};Y}$, $\dots$, $E^{(i)}_{[y_{i_m}]_\mathcal{E};Y}$, using the way **5** (several times, the necessary elements of $\mathsf{indeps}(\mathcal{X})$ are given by the two groups of antecedents of rule (4.28)) of changing the configurations.

Step 5. For each $i \in \{1,\ldots,l\}$, $h \in \{1,\ldots,m\}$ and $Z \in \{X,Y\}$ add the edges in $E^{(i)}_{[y_{i_h}]_{\mathcal{E}};Z}$ and $E^{(i)}_{[z_{i_h}]_{\mathcal{E}};Z}$ to $\mathsf{InpKeys}(C_j)$, using the way **3** of changing the configurations.

Step 6. Add the nodes in V to $\mathsf{IfNewKeys}(C_j)$, using the way **10** of changing the configurations. After this step, the sets $E^{(i)}_{b;X}$ and $E^{(i)}_{b;Y}$ are either empty or contain only a fictitious edge.

Step 7. For each $i \in \{1,\ldots,l\}$, split $E^{(i)}_{b;X} \cup E^{(i)}_{b;Y} \cup E^{(i)}_{X} \cup E^{(i)}_{Y}$ to $E^{(i)}_{b;X}$, $E^{(i)}_{b;Y}$ and $E^{(i)}_{X} \cup E^{(i)}_{Y}$ using the way **1** (twice) of changing the configurations.

Step 8. For each $i \in \{1,\ldots,l\}$, split $E^{(i)}_{X} \cup E^{(i)}_{Y}$ to $E^{(i)}_{X}$ and $E^{(i)}_{Y}$, using the way **5** of changing the configurations.

Rule (4.29) Let A be $\langle x_1,\ldots,x_k\rangle := b\,?\langle y_1,\ldots,y_k\rangle : \langle z_1,\ldots,z_k\rangle$. The sets $\mathsf{EncNull}(C_j)$ and $\mathsf{BoxNull}(C_j)$ are equal to the corresponding components of C_{j-1}. Because of the assumed relationship of $\mathsf{KnownInps}(C)$ and $\mathsf{KnownInps}(\bar{C})$, the input edges labelled with $z_1,\ldots,z_k$ or $[z_1]_{\mathcal{E}},\ldots,[z_k]_{\mathcal{E}}$ are not elements of $\mathsf{matters}(C_{j-1})$. Let $E_{\mathsf{A}} = \mathsf{descend}^{-1}_{j-1}(\bar{E})$ and let $E^{(1)},\ldots,E^{(l)} \in \mathsf{InpParts}(C_{j-1})$ be the sets whose union is E_{A}. Let V be the set of nodes labelled with $[?:]_{\mathcal{E}}$ whose outputs are the members of $\bar{E}$ that are labelled with $[x_1]_{\mathcal{E}},\ldots,[x_m]_{\mathcal{E}}$. Let W be the set of nodes labelled with $?:$ or $[?:]_{\mathcal{E}}$ whose outputs are the other members of $\bar{E}$. The edges in E_{A} may be labelled with b, y_i, z_i, $[y_i]_{\mathcal{E}}$, $[z_i]_{\mathcal{E}}$ or the labels of elements of $\bar{E}$ (except $[x_i]_{\mathcal{E}}$).

For each edge $e_b \in E_{\mathsf{A}}$, labelled with b, at least one of the following claims holds:

- The set W contains a node that has e_b as its input.
- e_b is fictitious in C_{j-1};
- the set V contains all non-fictitious nodes that have e_b as one of their inputs.

Indeed, suppose that the edge e_b is not used only at the nodes that correspond to A and whose outputs are labelled with some $[x_i]_{\mathcal{E}}$ I.e. e_b is also used at nodes whose outputs are labelled with some x_i. Then there exists an edge $\bar{e}_b \in \bar{E}$ that is labelled with some x_i. The node whose out-edge is x_i is an element of W.

Suppose now that the edge e_b is only used at the nodes corresponding to A and whose outputs are labelled with some $[x_i]_{\mathcal{E}}$. Suppose that j is the smallest number where the configuration $\bar{C}_{j-1}$ is turned to $\bar{C}_j$ using way **5** of changing the configurations, such that the necessary condition $(\bar{X},\bar{Y}) \in \mathsf{indeps}(\bar{\mathfrak{X}})$ is derived using either the rule (4.28) or (4.29). Then the set $\bar{E}$ must contain all (non-fictitious) input edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ that are labelled with $[x]_{\mathcal{E}}$. If j is not the smallest number with such property, then either $e_b \notin E_{\mathsf{A}}$ or the edge e_b is fictitious, as shown by the following construction of C_j.

We start with the function $\mathsf{descend}_j$. It is equal to $\mathsf{descend}_{j-1}$, except for edges $e \in E_{\mathsf{A}}$. On these edges e it is equal to either $\bar{E}_X$ or $\bar{E}_Y$, depending on which of these two sets contains an edge e', such that $e \in \mathsf{origins}(e')$. If both sets contain such edge e' (this may only happen if e is labelled with b), then the value $\mathsf{descend}_j(e)$

must be chosen so that the requirements put on $\mathsf{descend}_j$ still hold. I.e. if possible, $\mathsf{descend}_j(e)$ must contain an out-edge of W.

For a set $E^{(i)}$, where $i \in \{1, \dots, l\}$, we define the following subsets:

- The sets $E^{(i)}_{b;X}$ and $E^{(i)}_{b;Y}$ contain such edges $e \in E^{(i)}$, that
  - e is labelled with b;
  - there are no nodes in W that have e as their in-edge.

  The set $E^{(i)}_{b;X}$ contains edges e where $\mathsf{descend}_j(e) = \bar{E}_X$, and the set $E^{(i)}_{b;Y}$ contains edges e where $\mathsf{descend}_j(e) = \bar{E}_Y$.
- The sets $E^{(i)}_{[y_{i_1}]_{\mathcal{E}};X}, \dots, E^{(i)}_{[y_{i_m}]_{\mathcal{E}};X}$, $E^{(i)}_{[y_{i_{l+1}}]_{\mathcal{E}};Y}, \dots, E^{(i)}_{[y_{i_m}]_{\mathcal{E}};Y}$, $E^{(i)}_{[z_{i_1}]_{\mathcal{E}};X}, \dots, E^{(i)}_{[z_{i_m}]_{\mathcal{E}};X}$ and $E^{(i)}_{[z_{i_{l+1}}]_{\mathcal{E}};Y}, \dots, E^{(i)}_{[z_{i_m}]_{\mathcal{E}};Y}$. Each of those sets contains all edges $e \in E^{(i)}$, such that the label $\lambda_I(e)$ and the index of $\mathsf{descend}_j(e)$ (either X or Y) are equal to the subscript(s) of the set.
- The sets $E^{(i)}_{z;X}$ and $E^{(i)}_{z;Y}$. They contain all edges in E_{A} that are labelled with $z_{j_1}, \dots, z_{j_s}$ or $[z_{j_{r+1}}]_{\mathcal{E}}, \dots, [z_{j_s}]_{\mathcal{E}}$. The set $E^{(i)}_{z;X}$ contains all such edges e, where $\mathsf{descend}_j(e) = \bar{E}_X$, and the set $E^{(i)}_{z;Y}$ contains all such edges e, where $\mathsf{descend}_j(e) = \bar{E}_Y$.
- The sets $E^{(i)}_X$ and $E^{(i)}_Y$. They contain all remaining edges $e \in E_{\mathsf{A}}$, where $\mathsf{descend}_j(e) = \bar{E}_X$ or $\mathsf{descend}_j(e) = \bar{E}_Y$, respectively.

All these defined sets together constitute a partition of $E^{(i)}$. The partition $\mathsf{InpParts}(C_j)$ is obtained from the partition $\mathsf{InpParts}(C_{j-1})$ by replacing the parts $E^{(i)}$ with all the parts defined above.

Several steps are necessary for turning the configuration C_{j-1} to C_j. During the description of turning C_{j-1} to C_j we also define the sets $\mathsf{InpKeys}(C_j)$ and $\mathsf{IfNewKeys}(C_j)$. These sets contain at least as many elements as their counterparts in C_{j-1}, but may also contain more elements.

Step 1. Using the way **9** of changing the configurations, change $\mathsf{OpParts}_1(C_{j-1})$ to $\mathsf{OpParts}_1(C_j)$.

Step 2. For each $i \in \{1, \dots, l\}$, split $E^{(i)}$ to sets $E^{(i)}_{z;X}$, $E^{(i)}_{z;Y}$, $E^{(i)}_{[z_{i_1}]_{\mathcal{E}};X}, \dots,$ $E^{(i)}_{[z_{i_m}]_{\mathcal{E}};X}$, $E^{(i)}_{[z_{i_{l+1}}]_{\mathcal{E}};Y}, \dots, E^{(i)}_{[z_{i_m}]_{\mathcal{E}};Y}$ and

$$E^{(i)}_{b;X} \cup E^{(i)}_{b;Y} \cup E^{(i)}_X \cup E^{(i)}_Y \cup \bigcup_{h=1}^{m} E^{(i)}_{[y_{i_h}]_{\mathcal{E}};X} \cup \bigcup_{h=l+1}^{m} E^{(i)}_{[y_{i_h}]_{\mathcal{E}};Y} \tag{4.41}$$

using the way **1** (several times) of changing the configurations.

Step 3. For each $i \in \{1, \dots, l\}$, split (4.41) to

$$E^{(i)}_X \cup \bigcup_{h=1}^{m} E^{(i)}_{[y_{i_h}]_{\mathcal{E}};X} \cup \bigcup_{h=l+1}^{m} E^{(i)}_{[y_{i_h}]_{\mathcal{E}};Y} \tag{4.42}$$

and $E^{(i)}_{b;X} \cup E^{(i)}_{b;Y} \cup E^{(i)}_{Y}$ using the way **5** of changing the configurations.

Step 4. For each $i \in \{1,\dots,l\}$, split (4.42) to sets $E^{(i)}_{X}$, $E^{(i)}_{[y_{i_1}]_\mathcal{E};X}$, …, $E^{(i)}_{[y_{i_m}]_\mathcal{E};X}$, $E^{(i)}_{[y_{i_{l+1}}]_\mathcal{E};Y}$, …, $E^{(i)}_{[y_{i_m}]_\mathcal{E};Y}$, using the way **5** (several times) of changing the configurations.

Step 5. For each $i \in \{1,\dots,l\}$, $h \in \{1,\dots,m\}$ and $Z \in \{X,Y\}$ add the edges in $E^{(i)}_{[y_{i_h}]_\mathcal{E};Z}$ and $E^{(i)}_{[z_{i_h}]_\mathcal{E};Z}$ to $\mathsf{InpKeys}(C_j)$, using the way **3** of changing the configurations.

Step 6. Add the nodes in V to $\mathsf{IfNewKeys}(C_j)$, using the way **10** of changing the configurations. After this step, the sets $E^{(i)}_{b;X}$ and $E^{(i)}_{b;Y}$ are either empty or contain only a fictitious edge.

Step 7. For each $i \in \{1,\dots,l\}$, split $E^{(i)}_{b;X} \cup E^{(i)}_{b;Y} \cup E^{(i)}_{Y}$ to $E^{(i)}_{b;X}$, $E^{(i)}_{b;Y}$ and $E^{(i)}_{Y}$ using the way **1** (twice) of chining the configurations.

This completes the proof of Proposition 4.11.

4.5.3 Paths between $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ and $\mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$

Proposition 4.11 gives for each (valid) $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ a configuration C' with "severed ties" between $\mathsf{Chart}_{\mathsf{P};X}$ and $\mathsf{Chart}_{\mathsf{P};Y}$. The set of all these configurations C' cannot yet serve as the set $\mathbf{C}$ of configurations in Lemma 4.9, as the configurations $C[S_X]$ and $C[S_Y]$ for $S_X \in \mathsf{KV}^X_0$ and $S_Y \in \mathsf{KV}^Y_0$ are not yet uniquely determined. However, it is easy to change the configurations C' further, so that they become uniquely determined.

If C is a configuration of $2\mathsf{Chart}_{\mathsf{P};X,Y}$, then let $C|_X$ denote the "restriction" of C to $\mathsf{Chart}_{\mathsf{P};X}$, i.e.

- $\mathsf{InpParts}(C|_X) = \{E \cap \mathsf{inputs}(\mathsf{Chart}_{\mathsf{P};X}) \,:\, E \in \mathsf{InpParts}(C)\}$;
- $\mathsf{OpParts}_i(C|_X) = \{N \cap \mathsf{Nodes}(\mathsf{Chart}_{\mathsf{P};X}) \,:\, N \in \mathsf{OpParts}_i(C)\}$;
- $\mathsf{KnownInps}(C|_X) = \mathsf{KnownInps}(C) \cap \mathsf{inputs}(\mathsf{Chart}_{\mathsf{P};X})$;
- the sets $\mathsf{EncNull}(C|_X)$, $\mathsf{BoxNull}(C|_X)$, $\mathsf{InpKeys}(C|_X)$ and $\mathsf{IfNewKeys}(C|_X)$ are equal to the corresponding components of C intersected with either the set of edges or the set of nodes of $\mathsf{Chart}_{\mathsf{P};X}$.

The configuration $C|_Y$ of $\mathsf{Chart}_{\mathsf{P};Y}$ is defined similarly.

Lemma 4.12. *If C, C_1, C_2 are configurations of some flowchart G, such that $C \leftrightarrow C_1$, $C \leftrightarrow C_2$, $C \sqsupseteq C_1$ and $C \sqsupseteq C_2$, then there exists a configuration C_3, such that $C_1 \overset{*}{\leftrightarrow} C_3$, $C_2 \overset{*}{\leftrightarrow} C_3$, $C_1 \sqsupseteq C_3$ and $C_2 \sqsupseteq C_3$.*

Proof. The configuration C_3 is obtained by applying to C_1 the same change of configurations that changed C to C_2. For all pairs of ways of changing C to C_1 and changing C to C_2 this is possible. Alternatively, C_3 is obtained by applying to C_2 the same change of configurations that changed C to C_1. ☐

Lemma 4.13. *Let C and C' be configurations of $\mathsf{2Chart}_{\mathsf{P};X,Y}$, such the components $\mathsf{SameIf}(C)$ and $\mathsf{SameIf}(C')$ are equal and both C and C' have no ties between sides. If $C|_X \overset{*}{\leftrightarrow} C'|_X$ and $C|_Y \overset{*}{\leftrightarrow} C'|_Y$, then $C \overset{*}{\leftrightarrow} C'$.*

Proof. Indeed, if one can apply a change of configurations to $C|_X$ or $C|_Y$, then one can also apply the same change of configurations to C (because the other side will not interfere). Applying all the changes that change $C|_X$ to $C'|_X$ and $C|_Y$ to $C'|_Y$, changes C to C'. □

If $C_1, C_2 \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ and $\mathsf{KnownInps}(C_1|_X) = \mathsf{KnownInps}(C_2|_X)$, then $C_1|_X = C_2|_X$. Proposition 4.11 gives us the configurations C_1' and C_2', such that $C_1 \overset{*}{\leftrightarrow} C_1'$ and $C_2 \overset{*}{\leftrightarrow} C_2'$. Obviously also $C_1|_X \overset{*}{\leftrightarrow} C_1'|_X$ and $C_2|_X \overset{*}{\leftrightarrow} C_2'|_X$. By Lemma 4.12, there exists a configuration C_3 of $\mathsf{Chart}_{\mathsf{P};X}$, such that $C_3 \sqsubseteq C_1'|_X$, $C_3 \sqsubseteq C_2'|_X$, $C_1'|_X \overset{*}{\leftrightarrow} C_3$ and $C_2'|_X \overset{*}{\leftrightarrow} C_3$. Applying this construction repeatedly for all (valid) $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$, where $\mathsf{KnownInps}(C)$ is defined through a certain $S_X \in \mathsf{KV}_0^X$ (and through some element of KV_0^Y), gives us a configuration $C[S_X]$ of $\mathsf{Chart}_{\mathsf{P};X}$, such that $C'|_X \overset{*}{\leftrightarrow} C[S_X]$ for all (valid) $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$, where $\mathsf{KnownInps}(C)$ is defined through S_X.

We construct the configurations $C[S_Y]$ of $\mathsf{Chart}_{\mathsf{P};Y}$ similarly. We can now change each valid $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ to some configuration C' with severed ties between sides, and change this configuration further to a configuration C'', such that $C''|_X = C[S_X]$ and $C''|_Y = C[S_Y]$, where $S_X \in \mathsf{KV}_0^X$ and $S_Y \in \mathsf{KV}_0^Y$ are such, that $\mathsf{KnownInps}(C)$ is defined through S_X and S_Y. The relationship $C' \overset{*}{\leftrightarrow} C''$ follows from Lemma 4.13. The set of all such configurations C'' can serve as the set $\mathbf{C}$ of configurations in Lemma 4.9.

For each $\tilde{C} \in \mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$ we also need a configuration $\tilde{C}''$ of $\mathsf{2Chart}_{\mathsf{P};X,Y}$, such that the set of all these configurations $\tilde{C}''$ could be the corresponding set $\mathbf{C}'$ of configurations in Lemma 4.9. These configurations $\tilde{C}''$ are already simple to construct. Let $\mathsf{KnownInps}(\tilde{C})$ be defined through $S_X \in \mathsf{KV}_0^X$ and $S_Y \in \mathsf{KV}_0^Y$. Let $C_1, C_2 \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ be two valid configurations, such that $C_1|_X = \tilde{C}|_X$ and $C_2|_Y = \tilde{C}|_Y$. We have $C_1|_X \overset{*}{\leftrightarrow} C[S_X]$ and $C_2|_Y \overset{*}{\leftrightarrow} C[S_Y]$. We define $\tilde{C}''$ so, that $\tilde{C}''|_X = C[S_X]$ and $\tilde{C}''|_Y = C[S_Y]$. By Lemma 4.13, $\tilde{C} \overset{*}{\leftrightarrow} \tilde{C}''$, because the elements of $\mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$ have no ties between sides.

We have now given two sets of configurations $\mathbf{C}$ and $\mathbf{C}'$, such that the sums of interpretations of the flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$ together with the elements of these sets are equal. We also have shown how to change a configuration $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ to a configuration $C'' \in \mathbf{C}$ or a configuration $\tilde{C} \in \mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$ to a configuration $\tilde{C}'' \in \mathbf{C}'$ in a number of *short* steps.

4.5.4 Short Paths

Only one thing is still missing. We have not yet shown that the number of these short steps is *small*. We want an algorithm running in polynomial time in s (which is polynomial in the security parameter n) to be able to find for each configuration $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$ the corresponding configuration C'' and also the sequence of configurations $C = C_0 \leftrightarrow C_1 \leftrightarrow \cdots \leftrightarrow C_t = C''$. If the number of configurations were polynomial in s, then such an algorithm would clearly exist. The algorithm would just search for a path in a graph whose nodes are configurations and edges are given by the relation $\leftrightarrow$. We have shown before that such a path exists.

However, the number of configurations is superpolynomial. Nevertheless, the construction of the sequence of configurations given in the proof of Proposition 4.11 still produces a configuration of polynomial length. Also, the construction can be executed in polynomial time.

The construction in the proof of Proposition 4.11 produces a sequence of configurations $C_0 \leftrightarrow \cdots \leftrightarrow C_t$, such that $C_0 \sqsupseteq \cdots \sqsupseteq C_t$. The length of this sequence (i.e. the number of different elements in it) cannot be greater than the height of the partially ordered set of configurations. Let us estimate this height. A configuration is a tuple of its components, therefore the set of configurations is the Cartesian product of the sets of possible values of components. Consider the components of a configuration C of $\mathsf{2Chart}_{\mathsf{P};X,Y}$.

- The component $\mathsf{InpParts}(C)$ is an element of the partially ordered set of partitions over the set of input edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$. The height of this partially ordered set is equal to the number of input edges to $\mathsf{2Chart}_{\mathsf{P};X,Y}$ (by Lemma 2.1). This number is polynomial in s.

- The component $\mathsf{OpParts}_i(C)$ is an element of the partially ordered set of partitions over the set of nodes of $\mathsf{2Chart}_{\mathsf{P};X,Y}$ corresponding to the assignment or vectorised choice A_i. The height of this partially ordered set is equal to the number of these nodes, the number of these nodes is polynomial in s. There are altogether s such components — i.e. the number of such components is polynomial in s.

- The components $\mathsf{EncNull}(C)$, $\mathsf{BoxNull}(C)$, $\mathsf{InpKeys}(C)$ and $\mathsf{IfNewKeys}(C)$ are all elements of the partially ordered set of subsets of the set of nodes and edges of $\mathsf{2Chart}_{\mathsf{P};X,Y}$. Its height is one more than the number of nodes and edges in the flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$, i.e. it is polynomial in s.

- The components $\mathsf{KnownInps}(C)$ and $\mathsf{SameIf}(C)$ are elements of unordered sets. The height of unordered sets is 1.

The height of the partially ordered set of configurations is not greater than the sum of the heights of the partially ordered sets of its components. There is a polynomial number (in s) of components, each having a polynomial height. Therefore the height of the partially ordered set of configurations is polynomial in s.

The construction given in the proof of Proposition 4.11 allows us to construct this sequence of configurations $C_0 \leftrightarrow \cdots \leftrightarrow C_t$ in time polynomial to s. Indeed, we have to do s induction steps and in each of the steps construct such a sequence of polynomial length. The amount of work necessary for constructing such a sequence in one of the induction steps is proportional to the length of the sequence constructed in the previous induction step and is therefore polynomial in s.

We have now given a proof of Proposition 4.7, because we have presented all components of the hybrid argument that proves it. Hence we have also shown that (4.24) holds. The next section can be considered to be a summary of this proof.

4.6 The Attacker(s)

Let us summarise the proof of (4.24) by giving the analogue of the algorithm $\bar{\mathcal{A}}$ in the proof of Proposition 2.11.

Suppose that $(X, Y) \in \mathsf{indeps}(\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\mathcal{X}))$, but there exists an algorithm $\mathcal{A}$ that can distinguish the distributions (4.25) and (4.26). The analogue is the following: for each algorithm $\mathcal{B}$ (with subscripts) that we defined in Sec. 4.5.1 we define an algorithm $\bar{\mathcal{A}}$ (with the same subscripts). These algorithms take the following inputs:

- The security parameter $\mathbf{1}^n$.
- The same (specific) inputs that the algorithm $\mathcal{B}$ with same subscripts takes.

The specific inputs of an algorithm $\mathcal{B}$ are all its inputs, except the security parameter, the flowchart G, and two configurations C, C'.

The specific inputs of an algorithm $\mathcal{B}$ could be distributed by two possible distributions. The corresponding algorithm $\bar{\mathcal{A}}$ attempts to distinguish these two distributions. The algorithms $\bar{\mathcal{A}}$ are constructed so that the sum of their advantages is equal to the advantage of $\mathcal{A}$. Therefore at least one of the algorithms $\bar{\mathcal{A}}$ has non-negligible advantage. But this violates our assumption that all these pairs of distributions, namely

- the two distributions on (4.3), where $k \in \mathsf{keys}(\mathcal{X})$;
- the two distributions on (2.9);
- the distribution (4.1) and the distribution (4.2), where $(X, Y) \in \mathsf{indeps}(\mathcal{X})$;
- the two distributions on (2.10);
- the distribution (4.37) and the distribution (4.38)

are indistinguishable.

The algorithm $\bar{\mathcal{A}}$ (with subscripts) works as follows:

1. Construct the unrolled program $\mathsf{P}^{(n)}$ and the flowchart $\mathsf{2Chart}_{\mathsf{P}^{(n)};X,Y}$.
2. Construct the sets of configurations $\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P}^{(n)};X,Y}$ and $\mathbf{Conf}^{\mathrm{R}}_{\mathsf{P}^{(n)};X,Y}$.

3. For each $C \in \mathbf{Conf}^{\mathrm{L}}_{\mathsf{P}^{(n)};X,Y}$ construct the corresponding configuration C'', as defined in Sec. 4.5.3. Also, construct the derivation sequence $C = C_0 \leftrightarrow C_1 \leftrightarrow \cdots \leftrightarrow C_t = C''$. Let $\mathcal{S}(C)$ be the set of all such pairs of configurations (C_{i-1}, C_i), where $i \in \{1, \ldots, t\}$ and $C_{i-1} \leftrightarrow C_i$ by one of the ways **3**, **4**, **5**, **7**, **8**, **10**.

4. Similarly, for each $\tilde{C} \in \mathbf{Conf}^{\mathrm{R}}_{\mathsf{P}^{(n)};X,Y}$ construct the corresponding configuration $\tilde{C}''$, the derivation sequence showing $\tilde{C} \overset{*}{\leftrightarrow} \tilde{C}''$, and the set $\mathcal{S}(\tilde{C})$ of pairs of configurations from that derivation sequence, where one configuration has been changed to another by one of the ways listed before.

5. Let $\mathcal{S}$ the *multiset* of pairs of configurations that is the *multiset union* of all sets $\mathcal{S}(C)$ and $\mathcal{S}(\tilde{C})$ that have been constructed. I.e. if the same pair (C_1, C_2) occurs in more than one of the sets $\mathcal{S}(C)$ and $\mathcal{S}(\tilde{C})$, then the multiplicity of this pair is greater than 1 in the multiset $\mathcal{S}$.

6. Randomly uniformly choose one pair (C, C') from the set $\mathcal{S}$.

7. If $C \leftrightarrow C'$ by a way that does not correspond to the subscripts of $\bar{\mathcal{A}}$ (it must be done by the right way and, if it is done by way **3** or **5**, then the element of $\mathsf{keys}(\mathcal{X})$ or $\mathsf{indeps}(\mathcal{X})$ also must be the right one) then output "failure"[2].

8. Otherwise call the algorithm $\mathcal{B}$ (with the same subscripts as $\bar{\mathcal{A}}$) with arguments $\mathbf{1}^n$, $\mathsf{2Chart}_{\mathsf{P}^{(n)};X,Y}$, C, C' and the specific inputs that were given to $\bar{\mathcal{A}}$.

9. If $\mathcal{B}$ returns $\perp$, then output "failure". Otherwise call $\mathcal{A}$ with the value outputted by $\mathcal{B}$ (recall that this value has the type $(X \to \widetilde{\mathbf{Val}_n}) \times (Y \to \widetilde{\mathbf{Val}_n})$) and output, whatever $\mathcal{A}$ outputs.

4.7 Correctness of the Abstraction of Keys

For fully proving Theorem 4.3, we still have show that (4.24) also holds, when we replace indeps by keys in it. We are going to show it here.

Let P be a program with the set of variables $\mathbf{Var}$ and let $D \in \mathbf{TerD}[\mathsf{P}]$. Let $\mathcal{X} = \beta^{\mathbb{KI}}_{\mathbf{Var}}(D)$ and $\mathcal{Y} = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\mathcal{X})$. Suppose that $k \in \mathsf{keys}(\mathcal{Y})$ for some $k \in \mathbf{Var}$. We are going to show that in the distribution $\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)$, the variable k is distributed as a key, i.e. $k \in \mathsf{keys}(\beta^{\mathbb{KI}}_{\mathbf{Var}}(\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)))$. Note that we have already shown that (4.24) holds.

Let b be a variable that is not a member of $\mathbf{Var}$ and let Q be the program

$$b := \mathsf{FlipCoin}(); \textit{if } b \textit{ then } \mathsf{P} \textit{ else } k := \mathcal{G}en(),$$

[2] In the definition of indistinguishability (Def. 2.4) we check whether $\bar{\mathcal{A}}$ outputs 1 or not. In the terms of this definition, "failure" is just something that is not 1.

where $\mathsf{FlipCoin} \in \mathbf{Op}$ is such, that $[\![\mathsf{FlipCoin}]\!]_n$ returns true with probability $\frac{1}{2}$ and returns false with the same probability.

Let $\mathbf{Var}' := \mathbf{Var} \uplus \{b\}$ and $\mathbf{Var}'' := \mathbf{Var}' \uplus \{N\}$, where N is another new variable (which we are going to use in computing the abstract semantics of the *if*-statement). Define the following quantities:

$$\begin{aligned}
\mathcal{X}' &:= \mathbb{A}^{(\mathbf{Var}')}[\![b := \mathsf{FlipCoin}()]\!](\mathcal{X}) \\
\mathcal{X}'' &:= \mathbb{A}^{(\mathbf{Var}'')}[\![N := b]\!](\mathcal{X}') \\
\mathcal{Z}_1 &:= \mathbb{A}^{(\mathbf{Var}'')}[\![\mathsf{P}]\!](\mathcal{X}'') \\
\mathcal{Z}_2 &:= \mathbb{A}^{(\mathbf{Var}'')}[\![k := \mathcal{G}en()]\!](\mathcal{X}'') \\
\mathcal{Z} &:= \mathcal{Z}_1 \wedge \mathcal{Z}_2 \ .
\end{aligned}$$

Then $\mathbb{A}^{(\mathbf{Var}')}[\![\mathsf{Q}]\!](\mathcal{X})$, that we denote with $\mathcal{Y}'$, is equal to $merge(N, \mathbf{Var})(\mathcal{Z})$.

The abstract semantics of programs is defined so, that the equalities

$$\begin{aligned}
\mathsf{indeps}(\mathcal{X}') &= \{(X, Y \cup \{b\}) \,:\, (X,Y) \in \mathsf{indeps}(\mathcal{X})\} \\
\mathsf{indeps}(\mathcal{X}'') &= \{(X, Y \cup \{N, b\}) \,:\, (X,Y) \in \mathsf{indeps}(\mathcal{X})\}
\end{aligned}$$

hold, this follows from the rule (4.10). Nothing changes the variables N and b in the program P, therefore they are just brought along in the computation of the abstract semantics, such that in the end the equality

$$\mathsf{indeps}(\mathcal{Z}_1) = \{(X, Y \cup \{N, b\}) \,:\, (X,Y) \in \mathsf{indeps}(\mathcal{Y})\}$$

holds. Taking here $X = \{[k]_\mathcal{E}\}$ and $Y = \emptyset$, we get that $(\{[k]_\mathcal{E}, \{N, b\}) \in \mathsf{indeps}(\mathcal{Z}_1)$. We also have $\mathsf{keys}(\mathcal{Z}_1) = \mathsf{keys}(\mathcal{Y})$ and therefore $k \in \mathsf{keys}(\mathcal{Z}_1)$.

In the other branch of Q, we also have $k \in \mathsf{keys}(\mathcal{Z}_2)$ and $(\{[k]_\mathcal{E}\}, \{N, b\}) \in \mathsf{indeps}(\mathcal{Z}_2)$. Therefore

$$k \in \mathsf{keys}(\mathcal{Z}) \wedge (\{[k]_\mathcal{E}\}, \{N, b\}) \in \mathsf{indeps}(\mathcal{Z})$$

and by rule (4.18), $(\{[k]_\mathcal{E}\}, \{b\}) \in \mathsf{indeps}(\mathcal{Y}')$.

The inequality (4.24) gives us

$$\begin{aligned}
\{\!|(S_n([k]_\mathcal{E}), S_n(b)) \,:\, S_n \leftarrow \mathbb{C}_{\mathrm{term}}[\![\mathsf{Q}]\!](D)|\!\}_{n\in\mathbb{N}} &\approx \\
&\{\!|(S_n([k]_\mathcal{E}), S'_n(b)) \,:\, S_n, S'_n \leftarrow \mathbb{C}_{\mathrm{term}}[\![\mathsf{Q}]\!](D)|\!\}_{n\in\mathbb{N}} \ . \quad (4.43)
\end{aligned}$$

This contradicts the assumption $k \notin \mathsf{keys}(\beta^{\mathbb{KI}}_{\mathbf{Var}}(\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)))$, i.e.

$$\{\!|S_n([k]_\mathcal{E}) \,:\, S_n \leftarrow \mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)|\!\}_{n\in\mathbb{N}} \not\approx \{\!|[\![\mathcal{E}nc]\!]_n(k', \cdot) \,:\, k' \leftarrow [\![\mathcal{G}en]\!]_n|\!\}_{n\in\mathbb{N}}, \quad (4.44)$$

as we show below. Let $\mathcal{A}^{(\cdot)}$ be an algorithm distinguishing the distributions (4.44) (the algorithm $\mathcal{A}$ accesses the encrypting black box through the oracle interface). Let $\mathcal{B}^{(\cdot)}$ be an algorithm that takes as its inputs the security parameter $\mathbf{1}^n$, a boolean value b_{val} and an encrypting black box (accessed through the oracle interface). Let $\mathcal{B}^{(\cdot)}$ work as follows:

1. Call $\mathcal{A}^{(\cdot)}$, giving it the same oracle as $\mathcal{B}^{(\cdot)}$ has. Let c be the returned value.
2. If $\big(c = 1$ and $b_{\mathrm{val}} = \mathsf{true}\big)$ or $\big(c \neq 1$ and $b_{\mathrm{val}} = \mathsf{false}\big)$ then return 1. Otherwise return 2.

The algorithm $\mathcal{B}$ can distinguish the distributions in (4.43). Indeed, let D^{P} denote the distribution $\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!](D)$ and D^{Q} denote the distribution $\mathbb{C}_{\mathrm{term}}[\![\mathsf{Q}]\!](D)$. We have to show that the difference of

$$\Pr\big[\mathcal{B}^{\mathcal{E}(\mathbf{1}^n,S_n(k),\cdot)}(\mathbf{1}^n, S_n(b)) = 1 \,:\, S_n \leftarrow D^{\mathsf{Q}}_n\big] \tag{4.45}$$

and

$$\Pr\big[\mathcal{B}^{\mathcal{E}(\mathbf{1}^n,S_n(k),\cdot)}(\mathbf{1}^n, S'_n(b)) = 1 \,:\, S_n, S'_n \leftarrow D^{\mathsf{Q}}_n\big] \tag{4.46}$$

is not negligible. Here we let $\mathcal{E}$ be the algorithm implementing $[\![\mathcal{E}nc]\!]$ and $\mathcal{G}$ be the algorithm implementing $[\![\mathcal{G}en]\!]$. Let us compute both (4.45) and (4.46).

First, the probability (4.46) is $\frac{1}{2}$. The encrypting black box $\mathcal{E}(\mathbf{1}^n, S_n(k), \cdot)$ is picked independently of the bit $S'_n(b)$. Therefore for each possible value returned by the algorithm $\mathcal{A}$ (either 1 or 2) the probability that $\mathcal{B}$ returns 1 is $\frac{1}{2}$ and the probability that $\mathcal{B}$ returns 2 is $\frac{1}{2}$.

The probability (4.45) is

$$\begin{aligned}
&\Pr\big[\mathcal{B}^{\mathcal{E}(\mathbf{1}^n,S_n(k),\cdot)}(\mathbf{1}^n, S_n(b)) = 1 \,:\, S_n \leftarrow D^{\mathsf{Q}}_n\big] = \\
&\Pr\big[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,S_n(k),\cdot)}(\mathbf{1}^n) \neq 1 \wedge S_n(b) \,:\, S_n \leftarrow D^{\mathsf{Q}}_n\big] + \\
&\qquad\qquad \Pr\big[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,S_n(k),\cdot)}(\mathbf{1}^n) = 1 \wedge \neg S_n(b) \,:\, S_n \leftarrow D^{\mathsf{Q}}_n\big] = \\
&\frac{1}{2}\big(\Pr\big[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,S_n(k),\cdot)}(\mathbf{1}^n) = 1 \,:\, S_n \leftarrow D^{\mathsf{P}}_n\big] + \\
&\qquad\qquad 1 - \Pr\big[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k',\cdot)}(\mathbf{1}^n) = 1 \,:\, k' \leftarrow \mathcal{G}(\mathbf{1}^n)\big]\big) = \\
&\frac{1}{2} + \frac{1}{2}\big(\Pr\big[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,S_n(k),\cdot)}(\mathbf{1}^n) = 1 \,:\, S_n \leftarrow D^{\mathsf{P}}_n\big] - \\
&\qquad\qquad \Pr\big[\mathcal{A}^{\mathcal{E}(\mathbf{1}^n,k',\cdot)}(\mathbf{1}^n) = 1 \,:\, k' \leftarrow \mathcal{G}(\mathbf{1}^n)\big]\big) \ .
\end{aligned}$$

Therefore the difference of (4.45) and (4.46) is half of the advantage of $\mathcal{A}$ in distinguishing the distributions in (4.44). It is therefore non-negligible.

Chapter 5

Implementation

The presented analysis has a rather large domain, therefore one may ask whether the results given in the previous chapter are also usable in practice. In this chapter we show that a suitable abstraction of the given analysis can be implemented and it does not suffer from any loss of precision. Indeed, we have implemented the analysis in the form described in this chapter; the implementation is available under `http://www.cs.uni-sb.de/~laud/csif/`.

We start this chapter by presenting a traditional form of program analysis — data flow analysis — in Sec. 5.1 and showing how our analysis can be represented as a data flow analysis. There exist well-known generic algorithms for computing the solution(s) of data flow analysis, we want to make use of them.

This representation as a data flow analysis is rather straightforward and uses the same domain $\mathcal{PF}(\mathbf{Var})$ that we used for the analysis in Chapter 4. In Sec. 5.2 we show how to define a suitable subset of the domain, called $\mathcal{P}\hat{\mathfrak{F}}(\mathbf{Var})$, such that the elements of that domain may be efficiently represented. In our implementation of the analysis, we only consider such elements of $\mathcal{PF}(\mathbf{Var})$ that are elements of $\mathcal{P}\hat{\mathfrak{F}}(\mathbf{Var})$. The cardinality of the set $\mathcal{P}\hat{\mathfrak{F}}(\mathbf{Var})$ is small enough for an efficient implementation. In Sec. 5.3 we give a detailed account on the *transfer functions* of our data flow analysis, in the form that we have implemented them. Basically, transfer functions are the abstract semantics of a single assignment statement or *merge*. Transfer functions are all that one needs to use the generic algorithms for computing the solutions of data flow analysis.

In Sec. 5.4 we give a small example of the analysis in action.

This chapter is also a suitable place to show how our earlier results [Lau01] relate to this dissertation. There we gave another, weaker data flow analysis for secure information flow. Sec. 5.5 restates these results in the terms used here.

5.1 Formulation as Data Flow Analysis

Data Flow Analysis (DFA) is the traditional form of program analysis (see for example [Mar99], [NNH99, Sec. 1.3 and Chap. 2], [WM92, Sec. 9.4]). One of the most typical applications of DFA is to compute for each point in the given program, whether some particular properties hold there. DFA is therefore useful for abstracting some operational semantics of the program language, but we can also use it for our analysis (which is an abstraction of a denotational semantics). DFA will still give us results for each program point, but we will only use the result for the exit point. Our reason for using DFA is, that there exist well-known, generic algorithms for doing the necessary computation.

The representation of the program, used by DFA, is the *control flow graph* (CFG).

Definition 5.1. A *Control Flow Graph* (*CFG*) is a tuple $(V, E, \sigma, \tau, \lambda_V)$, where

- V is the finite set of *nodes*;
- E is the finite set of *edges*;
- $\sigma : E \to V \uplus \{\lozenge\}$ gives for each edge e its source node $\sigma(e)$;
- $\tau : E \to V \uplus \{\blacklozenge\}$ gives for each edge e its target node $\tau(e)$;
- $\lambda_V : V \to L_V$ gives to each node its *label*, L_V is the set of possible node labels.

There must exist exactly one edge whose source is $\lozenge$ (we call this edge the *start edge*), and exactly one edge whose target is $\blacklozenge$ (this edge is called the *end edge*).

We will now explain, how we define the control flow graph G_P of the program P. During this explanation, we also introduce the set of possible node labels L_V. Table 5.1 gives an overview of the definition of G_P.

- If P is *skip* then G_P has no nodes at all. It also has a single edge. The source of this edge must be $\lozenge$ and the target must be $\blacklozenge$.
- If P is $x := o(x_1, \ldots, x_k)$ then G_P has a single node, this node is labelled with $x := o(x_1, \ldots, x_k)$ (i.e. all such assignments are members of L_V). The CFG G_P has two edges, connecting $\lozenge$ to the node and the node to $\blacklozenge$.
- If P is $\mathsf{P}_1; \mathsf{P}_2$ then G_P is just the "composition" of graphs $\mathsf{G}_{\mathsf{P}_1}$ and $\mathsf{G}_{\mathsf{P}_2}$. More precisely, the set of nodes of G_P is the (disjoint) union of the sets of nodes of $\mathsf{G}_{\mathsf{P}_1}$ and $\mathsf{G}_{\mathsf{P}_2}$ (that retain their labels) and the set of edges of G_P is the union of the sets of edges of $\mathsf{G}_{\mathsf{P}_1}$ and $\mathsf{G}_{\mathsf{P}_2}$, where the end edge of $\mathsf{G}_{\mathsf{P}_1}$ is identified with the start edge of $\mathsf{G}_{\mathsf{P}_2}$.
- If P is *if* b *then* P_1 *else* P_2 then G_P contains the graphs $\mathsf{G}_{\mathsf{P}_1}$ and $\mathsf{G}_{\mathsf{P}_2}$, two extra nodes v_{if} and v_{merge} and an additional start edge (whose target is v_{if}) and end edge (whose source is v_{merge}). The source of start edges of $\mathsf{G}_{\mathsf{P}_1}$ and $\mathsf{G}_{\mathsf{P}_2}$ is v_{if} and the target of their end edges is v_{merge}. The nodes v_{if} and v_{merge} have the following labels:

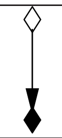
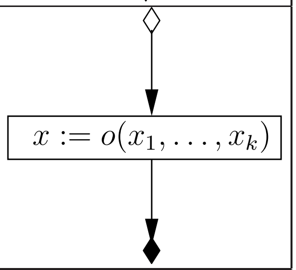
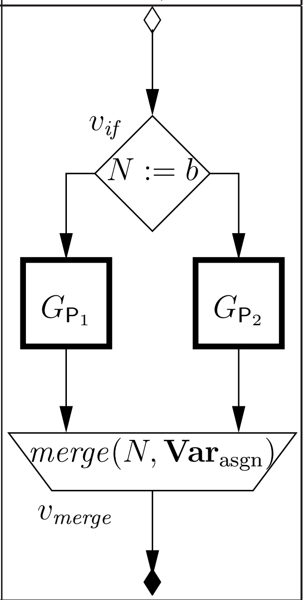
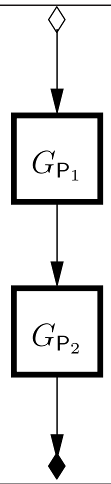
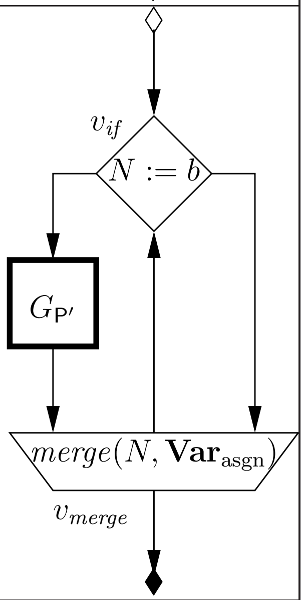
P	$\mathsf{G_P}$
skip	
$x := o(x_1, \ldots, x_k)$	
if b *then* P_1 *else* P_2	
$\mathsf{P}_1; \mathsf{P}_2$	
while b *do* P'	

Table 5.1: CFG of the program P

– The label of v_{if} is $N := b$, where N is a variable that does not occur in P. Such labels are elements of L_V, as already mentioned while explaining the control flow graphs of programs consisting of a single assignment.

– Let $\mathbf{Var}_{\mathrm{asgn}}$ be defined as in Sec. 4.2.2. The label of the node v_{merge} is $merge(N, \mathbf{Var}_{\mathrm{asgn}})$. All such labels $merge(N, Z)$, where $N \in \mathbf{Var}$ and $Z \subseteq \mathbf{Var}$, are members of L_V.

The shape of the nodes v_{if} and v_{merge} in Table 5.1 has no formal meaning.

- If P is *while b do* P' then G_P is almost the same as the CFG of the program *if b then* P' *else skip*. The difference is, that G_P has one more edge, from the node v_{merge} back to the node v_{if}.

The definition of the control flow graph for *while b do* P' is non-traditional. However, it corresponds nicely to the concrete and abstract semantices of loops, as defined in Sec. 3.1.3 and Sec. 4.2.2.

We now associate each node label (and, through λ_V, also each node of a CFG) with a *transfer function*. Generally, the transfer function of a node describes, how the operation that this node corresponds to, changes the properties, assigned to program points (the program points are actually the edges of the CFG), that we are interested in. In our case, the transfer function $F(l)$ of $l \in L_V$ is a function from $\mathcal{PF}(\mathbf{Var})$ to $\mathcal{PF}(\mathbf{Var})$. It is defined as follows:

- $F(x := o(x_1, \ldots, x_k)) = \mathbb{A}^{(\mathbf{Var})}[\![x := o(x_1, \ldots, x_k)]\!]$.
- $F(merge(N, Z))$, where $Z \subseteq \mathbf{Var}$, applied to $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$ returns $\mathcal{Y} \in \mathcal{PF}(\mathbf{Var})$ that is the smallest element, such that

  – $\mathcal{Y}$ satisfies the rules given in Fig. 4.5;

  – $\mathsf{indeps}(\mathcal{Y})$ is symmetric and downwards closed.

Generally, the transfer functions can be over any complete lattice. In the following we always use $\mathcal{PF}(\mathbf{Var})$ as the domain and range of transfer functions, as this is the domain that our analysis is using. The theory of DFA, however, works for any complete lattice.

We now describe, how the transfer functions are used. Let $\mathsf{G} = (V, E, \sigma, \tau, \lambda_V)$ be a CFG and let $\Lambda : E \to \mathcal{PF}(\mathbf{Var})$ be a labelling of the edges of G with the elements of $\mathcal{PF}(\mathbf{Var})$. Let Λ' and $\mathcal{V}(\Lambda)$ be the following labellings of the nodes and $\mathcal{L}(\Lambda)$ be the following labelling of the edges with elements of $\mathcal{PF}(\mathbf{Var})$:

$$\begin{aligned}
\Lambda' \,\$\, v &:= \bigwedge\{\Lambda(e') \,:\, e' \in E, \tau(e') = v\} \\
\mathcal{V}(\Lambda) \,\$\, v &:= F(\lambda_V(v)) \,\$\, \Lambda'(v) \\
\mathcal{L}(\Lambda) \,\$\, e &:= \Lambda(e) \wedge \big(\mathcal{V}(\Lambda) \,\$\, \sigma(e)\big),
\end{aligned}$$

where $v \in V$, $e \in E$ and $\mathcal{V}(\Lambda) \$ \Diamond$ is defined to be $\top$ — the greatest element of the lattice $\mathcal{PF}(\mathbf{Var})$. Basically, the task of DFA is computing fixed points (less or equal to some given mapping Λ_0) of the function $\mathcal{L}$.

The label $\mathcal{V}(\Lambda) \$ v$ describes the properties after the node v, if the properties before the node v are given by $\Lambda'(v)$. "Properties before a node" are defined as the summarisation of the properties of all the edges going to that node. Finally, $\mathcal{L}$ is defined so, that $\mathcal{L}(\Lambda) \leq \Lambda$. This guarantees the existence of fixed points of $\mathcal{L}$.

Given $\Lambda_0 : E \to \mathcal{PF}(\mathbf{Var})$, the data flow analysis computes

$$\bigwedge\{\mathcal{L}^n(\Lambda_0) \ : \ n \in \mathbb{N}\} \ . \tag{5.1}$$

There are algorithms for computing (5.1) that are in general more efficient than successively computing $\mathcal{L}(\Lambda_0), \mathcal{L}^2(\Lambda_0),$ etc. See [Mar99, NNH99]. For example, the algorithm given in [NNH99, Sec. 2.4.1] computes exactly the quantity (5.1).

If all transfer functions are monotone (in our case, they are) then the function $\mathcal{L}$ is monotone, too. Hence in this case, by Proposition 2.7, DFA computes $\mathit{gfp}^{\Lambda_0}\,\mathcal{L}$. If additionally the mapping Λ_0 has the property that for each edge e of the CFG and $v = \sigma(e)$,

$$F(\lambda_V(v)) \$ \Lambda'_0(v) \leq \Lambda_0(e), \tag{5.2}$$

then $\mathcal{L}(\Lambda_0)$ also satisfies the same property and moreover, $\mathcal{L}(\Lambda_0) = \mathcal{L}'(\Lambda_0)$, where

$$\mathcal{L}'(\Lambda) \$ e := \mathcal{V}(\Lambda) \$ \sigma(e) \ .$$

By induction, $\mathcal{L}^n(\Lambda_0) = \mathcal{L}'^n(\Lambda_0)$ for all $n \in \mathbb{N}$ and therefore $\mathit{gfp}^{\Lambda_0}\,\mathcal{L} = \mathit{gfp}^{\Lambda_0}\,\mathcal{L}'$.

Given $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$, let $\Lambda_0^{\mathcal{X}}$ be the following labelling of edges:

$$\Lambda_0^{\mathcal{X}}(e) = \begin{cases} \mathcal{X}, & \text{if } e \text{ is the start edge} \\ \top, & \text{otherwise.} \end{cases}$$

$\Lambda_0^{\mathcal{X}}$ is a typical input to DFA. It expresses that we know that at the start of the program, properties described by $\mathcal{X}$ hold. We do not yet know, what holds at the other program points, but we *optimistically* assume, that everything may hold there. DFA, applied to $\Lambda_0^{\mathcal{X}}$, finds out what actually holds in each program point, if properties described by $\mathcal{X}$ hold at the beginning. Let $\Lambda^{\mathcal{X}}$ be the output (5.1) of DFA for $\Lambda_0^{\mathcal{X}}$. The quantity $\Lambda^{\mathcal{X}}$ is called the *solution* of DFA for the initial value $\mathcal{X}$.

Let $\mathrm{DF}_{\mathsf{G}}(\mathcal{X}) \in \mathcal{PF}(\mathbf{Var})$ be the value assigned to the end edge of G by $\Lambda^{\mathcal{X}}$. The following theorem is the main result of this chapter.

Theorem 5.1. *Let* P *be a program with the set of variables* $\mathbf{Var}$. *Let* G_P *be the CFG on the program* P. *Then* $\mathrm{DF}_{\mathsf{G}_\mathsf{P}} = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$.

Proof. The proof is by induction over program structure.

If P is *skip*, then $\mathrm{DF}_{\mathsf{G}_\mathsf{P}}$ is the identity function on $\mathcal{PF}(\mathbf{Var})$, as the CFG for P only has a single edge, this edge is both the start and end edge. Therefore $\mathrm{DF}_{\mathsf{G}_\mathsf{P}} = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$.

If P is $x := o(x_1, \ldots, x_k)$ and $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$, then the solution of DFA for $\mathcal{X}$ assigns $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\mathcal{X})$ to the end edge of the CFG G_P. Hence $\mathrm{DF}_{\mathsf{G}_\mathsf{P}} = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$.

If P is $\mathsf{P}_1; \mathsf{P}_2$ and $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$, then the solution of DFA $\Lambda^\mathcal{X}$ for the initial value $\mathcal{X}$ is the following:

- $\Lambda^\mathcal{X}$ assigns to the edges of $\mathsf{G}_{\mathsf{P}_1}$ the same values as the solution of DFA for the CFG $\mathsf{G}_{\mathsf{P}_1}$ and the initial value $\mathcal{X}$. Hence the value assigned to the end edge of $\mathsf{G}_{\mathsf{P}_1}$ is $\mathrm{DF}_{\mathsf{G}_{\mathsf{P}_1}}(\mathcal{X})$.

- $\Lambda^\mathcal{X}$ assigns to the start edge of $\mathsf{G}_{\mathsf{P}_2}$ the value $\mathrm{DF}_{\mathsf{G}_{\mathsf{P}_1}}(\mathcal{X})$, because the start edge of $\mathsf{G}_{\mathsf{P}_2}$ and the end edge of $\mathsf{G}_{\mathsf{P}_1}$ are the same. $\Lambda^\mathcal{X}$ therefore assigns to the edges of $\mathsf{G}_{\mathsf{P}_2}$ the same values as the solution of DFA for the CFG $\mathsf{G}_{\mathsf{P}_2}$ and the initial value $\mathrm{DF}_{\mathsf{G}_{\mathsf{P}_1}}(\mathcal{X})$. In particular, it assigns the value $\mathrm{DF}_{\mathsf{G}_{\mathsf{P}_2}}(\mathrm{DF}_{\mathsf{G}_{\mathsf{P}_1}}(\mathcal{X}))$ to the end edge of $\mathsf{G}_{\mathsf{P}_2}$. This edge is also the end edge of G_P.

We have shown that $\mathrm{DF}_{\mathsf{G}_\mathsf{P}} = \mathrm{DF}_{\mathsf{G}_{\mathsf{P}_2}} \circ \mathrm{DF}_{\mathsf{G}_{\mathsf{P}_1}}$. The claim of the theorem now follows from the application of the induction hypothesis for programs P_1 and P_2.

Suppose that P is *if* b *then* P_1 *else* P_2. The solution of DFA $\Lambda^\mathcal{X}$ for the initial value $\mathcal{X}$ has the following properties:

- $\Lambda^\mathcal{X}$ assigns to the end edge of $\mathsf{G}_{\mathsf{P}_1}$ (this is one of the two edges whose target is v_{merge}) the value $\mathcal{Z}_1 := \mathrm{DF}_{\mathsf{G}_{N:=b;\mathsf{P}_1}}(\mathcal{X})$. By the induction assumption, this value is equal to $\mathbb{A}^{(\mathbf{Var}\uplus\{N\})}[\![N := b; \mathsf{P}_1]\!](\mathcal{X})$.

- $\Lambda^\mathcal{X}$ assigns to the end edge of $\mathsf{G}_{\mathsf{P}_2}$ the value $\mathcal{Z}_2 := \mathbb{A}^{(\mathbf{Var}\uplus\{N\})}[\![N := b; \mathsf{P}_2]\!](\mathcal{X})$. Again, this follows from the induction assumption.

The initial value $\Lambda_0^\mathcal{X}$ has the property (5.2), therefore $\Lambda^\mathcal{X} = gfp^{\Lambda_0^\mathcal{X}}\, \mathcal{L}'$. By definition of $\mathcal{L}'$, the labelling $\Lambda^\mathcal{X}$ assigns to the end edge of G_P the value $merge(N, \mathbf{Var}_{\mathrm{asgn}})(\mathcal{Z}_1 \wedge \mathcal{Z}_2)$. This equals $\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!](\mathcal{X})$.

Suppose that P is *while* b *do* P'. By the construction of G_P, the function $\mathrm{DF}_{\mathsf{G}_\mathsf{P}}$ must satisfy

$$\mathrm{DF}_{\mathsf{G}_\mathsf{P}}(\mathcal{X}) \leq (\mathrm{DF}_{\mathsf{G}_\mathsf{P}} \circ \mathrm{DF}_{\mathsf{G}_{\textit{if } b \textit{ then } \mathsf{P}' \textit{ else skip}}})(\mathcal{X}) \enspace .$$

By the induction assumption,

$$\mathrm{DF}_{\mathsf{G}_\mathsf{P}}(\mathcal{X}) \leq (\mathrm{DF}_{\mathsf{G}_\mathsf{P}} \circ \mathbb{A}^{(\mathbf{Var})}[\![\textit{if } b \textit{ then } \mathsf{P}' \textit{ else skip}]\!])(\mathcal{X}) \enspace .$$

As DFA is computing the greatest fixed points, $\mathrm{DF}_{\mathsf{G}_\mathsf{P}} = \mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$. □

The rest of this chapter (except Sec. 5.5) deals with implementing the data flow analysis that we presented.

5.1.1 Discussion

In most contemporary literature about data flow analysis, smaller abstract values (where "smaller" is given by the order on the complete lattice, serving as the domain of DFA) are considered to be more optimistic and greater abstract values are considered to be more conservative. I.e. the most optimistic value is $\perp$, this value is used to denote that we do not yet know what holds at a certain program point. In the presentation that we just gave, smaller values are more conservative and greater values are more optimistic. This is no significant difference, because of duality (See the end of Sec. 2.1.1). We chose such presentation because of the direction of the natural order on $\mathcal{PF}(\mathbf{Var})$. If we had let $\mathcal{R}(\mathcal{PF}(\mathbf{Var}))$ be the domain of our data flow analysis, then our treatment would have been in line with the usual conventions.

Computing (5.1) may in general case require quite a lot of effort. The worst-case running time of algorithms computing (5.1) is proportional to the product of the number of vertices on the control flow graph and the height of the underlying lattice (in our case $\mathcal{PF}(\mathbf{Var})$, the height of which is exponential in the number of variables occurring in the program). In practice, the computation of solutions of data flow analyses converges much faster. Obviously, it is still linear to the size of the CFG, but usually the computation does not have to step through all "'levels"' of the underlying lattice. In estimating the efficiency of a particular implementation, experimental results may be much more interesting. Our implementation, which is described below, can analyse small examples more or less instantly (i.e. in a fraction of second) on a 500 MHz Pentium III.

5.2 Simplified Abstract Domain

The algorithms for computing solutions of DFA require one to keep in memory a data structure that records the "current" label of each edge of the CFG. These labels are updated during the computation.

The set $\mathcal{PF}(\mathbf{Var}) = \mathcal{P}(\mathbf{Var}) \times \mathcal{F}(\mathbf{Var})$ is too large for assigning one element of it to each edge of the control flow graph. We therefore must abstract $\mathcal{PF}(\mathbf{Var})$; we do this by defining a certain subset $\hat{\mathcal{F}}_K(\mathbf{Var}) \subseteq \mathcal{F}(\mathbf{Var})$ for each $K \subseteq \mathbf{Var}$ and consider only such elements $\mathcal{X} \in \mathcal{PF}(\mathbf{Var})$, where $\mathsf{indeps}(\mathcal{X}) \in \hat{\mathcal{F}}_{\mathsf{keys}(\mathcal{X})}(\mathbf{Var})$. The transfer functions remain the same, they are still given by the figures 4.2, 4.3 and 4.5, except that $\mathsf{indeps}(\mathcal{X})$ [resp. $\mathsf{indeps}(\mathcal{Y})$] on these figures is now considered to be an element of $\hat{\mathcal{F}}_{\mathsf{keys}(\mathcal{X})}(\mathbf{Var})$ [resp. $\hat{\mathcal{F}}_{\mathsf{keys}(\mathcal{Y})}(\mathbf{Var})$]. We denote

$$\mathcal{P}\hat{\mathcal{F}}(\mathbf{Var}) := \{(K, \mathcal{I}) \,:\, K \subseteq \mathbf{Var}, \mathcal{I} \in \hat{\mathcal{F}}_K(\mathbf{Var})\} \ .$$

Such change makes an analysis more conservative — we are simply not recording the independence of certain pairs of sets of variables. Also, the modified analysis cannot derive the independence of some pair of sets of variables that the original analysis would not have derived. Hence the modified analysis, working on $\mathcal{P}\hat{\mathfrak{F}}(\mathbf{Var})$, is correct.

The set $\hat{\mathfrak{F}}_K(\mathbf{Var})$ itself depends on the set of private input variables $\mathbf{Var}_\mathrm{S} \subseteq \mathbf{Var}$. After this set has been fixed, define

$$\begin{aligned} \mathcal{SV}_K(\mathbf{Var}) &= \mathcal{P}([K]_\mathcal{E}) \uplus \{\mathbf{Var}_\mathrm{S}\} \\ \hat{\mathfrak{F}}_K(\mathbf{Var}) &= \mathcal{P}(\mathcal{SV}_K(\mathbf{Var}) \times \mathcal{P}(\widetilde{\mathbf{Var}})), \end{aligned}$$

where $[K]_\mathcal{E}$ denotes the set $\{[k]_\mathcal{E} : k \in K\}$. The elements of the set $\hat{\mathfrak{F}}_K(\mathbf{Var})$ can be represented as a pair of boolean functions. Indeed,

$$\begin{aligned} \hat{\mathfrak{F}}_K(\mathbf{Var}) = \mathcal{P}(\mathcal{SV}_K(\mathbf{Var}) \times \mathcal{P}(\widetilde{\mathbf{Var}})) \cong \mathcal{SV}_K(\mathbf{Var}) \to \mathcal{P}(\mathcal{P}(\widetilde{\mathbf{Var}})) = \\ (\mathcal{P}([K]_\mathcal{E}) \uplus \{\mathbf{Var}_\mathrm{S}\}) \to \mathcal{P}(\mathcal{P}(\widetilde{\mathbf{Var}})) \cong \\ \big(\mathcal{P}([K]_\mathcal{E}) \to \mathcal{P}(\mathcal{P}(\widetilde{\mathbf{Var}}))\big) \times \big(\{\mathbf{Var}_\mathrm{S}\} \to \mathcal{P}(\mathcal{P}(\widetilde{\mathbf{Var}}))\big) \cong \\ \mathcal{P}(\mathcal{P}([K]_\mathcal{E} \uplus \widetilde{\mathbf{Var}})) \times \mathcal{P}(\mathcal{P}(\widetilde{\mathbf{Var}})) \cong (\mathbb{B}^{[K]_\mathcal{E} \uplus \widetilde{\mathbf{Var}}} \to \mathbb{B}) \times (\mathbb{B}^{\widetilde{\mathbf{Var}}} \to \mathbb{B}) \ . \end{aligned}$$

Elementwise, to each $\mathfrak{I} \in \hat{\mathfrak{F}}_K(\mathbf{Var})$ corresponds a pair $(B^\mathrm{key}, B^\mathrm{sec})$ of boolean functions, such that

- If $X \subseteq [K]_\mathcal{E}$, then $(X, Y) \in \mathfrak{I}$ iff $B^\mathrm{key}(\mathcal{X}_{X \uplus Y}) = \mathsf{true}$ (here $\mathcal{X}_{X \uplus Y} : ([K]_\mathcal{E} \uplus \widetilde{\mathbf{Var}}) \to \mathbb{B}$ is the characteristic function of $X \uplus Y$).
- $(\mathbf{Var}_\mathrm{S}, Y) \in \mathfrak{I}$ iff $B^\mathrm{sec}(\mathcal{X}_Y) = \mathsf{true}$.

In our implementation of the analysis, we have used binary decision diagrams (BDDs) to represent boolean functions. In the usual implementation of BDDs, different BDDs may share common subterms. Therefore the memory requirements of the representation of elements of $\mathcal{P}\hat{\mathfrak{F}}(\mathbf{Var})$ are not prohibitively large.

5.3 Implementing Transfer Functions

Here we explain, how the transfer functions look like, if they are considered to work over the set $\mathcal{P}\hat{\mathfrak{F}}(\mathbf{Var})$. Moreover, the indeps-component of the elements of $\mathcal{P}\hat{\mathfrak{F}}(\mathbf{Var})$ is considered to be a pair of boolean functions. The main purpose of this section is to give a detailed account, how the transfer functions change these pairs of boolean functions.

More concretely, we are given the following:

- a node label l; it can be either $x := o(x_1, \ldots, x_k)$ or $\mathit{merge}(N, Z)$;
- a set of variables $K_\circ \subseteq \mathbf{Var}$;
- a pair of boolean functions $B_\circ = (B_\circ^\mathrm{key}, B_\circ^\mathrm{sec})$, where the arguments of the function $B_\circ^\mathrm{key}$ are named with the elements of $[K_\circ]_\mathcal{E} \uplus \widetilde{\mathbf{Var}}$ and the arguments of the function $B_\circ^\mathrm{sec}$ are named with the elements of $\mathbf{Var}$.

Hence the pair $\mathcal{X} = (K_\circ, B_\circ)$ is a member of $\mathcal{P}\hat{\mathcal{F}}(\mathbf{Var})$. We are looking for

- a set of variables $K_\bullet \subseteq \mathbf{Var}$;
- a pair of boolean functions $B_\bullet = (B_\bullet^{\mathrm{key}}, B_\bullet^{\mathrm{sec}})$, where the arguments of the function $B_\bullet^{\mathrm{key}}$ are named with the elements of $[K_\bullet]_\mathcal{E} \uplus \widetilde{\mathbf{Var}}$ and the arguments of the function $B_\bullet^{\mathrm{sec}}$ are named with the elements of $\mathbf{Var}$,

such that $\mathcal{Y} = (K_\bullet, B_\bullet)$ is equal to $F(l)\ \$\ \mathcal{X}$.

The arguments of the functions $B_\circ^{\mathrm{key}}$ and $B_\bullet^{\mathrm{key}}$ come from some set $[K]_\mathcal{E} \uplus \widetilde{\mathbf{Var}}$. In the following we also have to spell out the names of the arguments of these functions. To distinguish the elements $[k]_\mathcal{E}$ belonging to the first component of $[K]_\mathcal{E} \uplus \widetilde{\mathbf{Var}}$ from the elements $[k]_\mathcal{E}$ belonging to the second component of $[K]_\mathcal{E} \uplus \widetilde{\mathbf{Var}}$ we rename the elements belonging to the first component of $[K]_\mathcal{E} \uplus \widetilde{\mathbf{Var}}$. Let

$$[K]_\mathcal{E} \uplus \widetilde{\mathbf{Var}} := \{(k)_\mathcal{E} \,:\, k \in K\} \cup \mathbf{Var} \cup \{[x]_\mathcal{E} \,:\, x \in \mathbf{Var}\},$$

i.e. $(k)_\mathcal{E}$ belongs to the first component of $[K]_\mathcal{E} \uplus \widetilde{\mathbf{Var}}$ and $[k]_\mathcal{E}$ belongs to the second component.

We make a simplifying assumption, which is really a very natural one and certainly cannot be considered to be a constraint. We assume that the variables in $\mathbf{Var}_\mathrm{S}$ do not occur at the left hand sides of assignments. We did not need this assumption before, that's why we did not state it earlier.

The way of computing $K_\bullet$ and $B_\bullet$ from $K_\circ$ and $B_\circ$ obviously depends on the label l.

5.3.1 Transfer Functions for Assignments

The transfer function depends on the operator o of the assignment. Four possibilities are distinguished — whether the operator has no special properties, or the assignment is a simple assignment, or the operator is $\mathcal{E}nc$, or the operator is $\mathcal{G}en$.

We introduce the following notation. Let $f : \mathbb{B}^{\widetilde{\mathbf{Var}}} \to \mathbb{B}$ be a boolean function whose arguments are labelled with the elements of $\widetilde{\mathbf{Var}}$. Let $x_1, \ldots, x_k \in \widetilde{\mathbf{Var}}$ and let $f_1, \ldots, f_k : \mathbb{B}^{\widetilde{\mathbf{Var}}} \to \mathbb{B}$. Then $f[{}^{x_1}/_{f_1}, \ldots, {}^{x_k}/_{f_k}]$, denoting the *simultaneous substitution of x_i with f_i* for $i \in \{1, \ldots, k\}$ in f, is again a boolean function whose arguments are labelled with the elements of $\widetilde{\mathbf{Var}}$. Its definition is the following. Let $\mathcal{X} \in \mathbb{B}^{\widetilde{\mathbf{Var}}}$. We now define

$$f[{}^{x_1}/_{f_1}, \ldots, {}^{x_k}/_{f_k}](\mathcal{X}) = f(\mathcal{X}[x_1 \mapsto f_1(\mathcal{X}), \ldots, x_k \mapsto f_k(\mathcal{X})]),$$

where $\mathcal{X}[x \mapsto b]$ denotes a tuple that is obtained from $\mathcal{X}$ be setting its component x to $b \in \mathbb{B}$.

If $x \in \widetilde{\mathbf{Var}}$, then x also denotes a boolean function with the type $\mathbb{B}^{\widetilde{\mathbf{Var}}} \to \mathbb{B}$. This value of this function is equal to the value of its argument named x.

We also use the notation introduced here if the domain of boolean functions under consideration is $\mathbb{B}^{[K]_\mathcal{E} \uplus \widetilde{\mathbf{Var}}}$.

o is a "usual" operator

Let the label l be $x := o(x_1, \ldots, x_k)$. Obviously $K_\bullet = K_\circ \backslash \{x\}$ by rule (4.14).

The pair of functions $B_\bullet$ is found from $B_\circ$ with the help of rules (4.9) and (4.10). We have

$$\begin{aligned} B_\bullet^{\mathrm{key}} &= \neg(x)_\mathcal{E} \wedge \left((B_\circ^{\mathrm{key}} \wedge \neg(x \vee [x]_\mathcal{E})) \vee B_\circ^{\mathrm{key}}[{}^{x}/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}, {}^{x_1}/_{\mathsf{true}}, \ldots, {}^{x_k}/_{\mathsf{true}}]\right) \\ B_\bullet^{\mathrm{sec}} &= (B_\circ^{\mathrm{sec}} \wedge \neg(x \vee [x]_\mathcal{E})) \vee B_\circ^{\mathrm{sec}}[{}^{x}/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}, {}^{x_1}/_{\mathsf{true}}, \ldots, {}^{x_k}/_{\mathsf{true}}] \ . \end{aligned}$$

In both equations the first disjunct (for $B_\bullet^{\mathrm{key}}$, the disjunction itself is under the conjunction) comes from the rule (4.9) and the second disjunct from the rule (4.10).

Let us take a closer look at the second disjunct in the definition of $B_\bullet^{\mathrm{sec}}$ (similar things hold for the definition of $B_\bullet^{\mathrm{key}}$). It says that for any $Y \subseteq \widetilde{\mathbf{Var}}$ the following holds:

$$(\mathbf{Var}_\mathrm{S}, Y) \in \mathsf{indeps}(\mathcal{Y}) \Longleftarrow (\mathbf{Var}_\mathrm{S}, Y \backslash \{x, [x]_\mathcal{E}\} \cup \{x_1, \ldots, x_k\}) \in \mathsf{indeps}(\mathcal{X}) \ . \quad (5.3)$$

Indeed, applying $B_\circ^{\mathrm{sec}}[{}^{x}/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}, {}^{x_1}/_{\mathsf{true}}, \ldots, {}^{x_k}/_{\mathsf{true}}]$ to the characteristic function of Y is the same as applying $B_\circ^{\mathrm{sec}}$ to the characteristic function of the set $Y \backslash \{x, [x]_\mathcal{E}\} \cup \{x_1, \ldots, x_k\}$.

The implication (5.3) is exactly rule (4.10).

The assignment is a simple assignment

Let the label l be $x := y$. Then $K_\bullet$ is equal to $K_\circ \cup \{x\}$, if $y \in K_\circ$. If $y \notin K_\circ$, then $K_\bullet = K_\circ \backslash \{x\}$.

The pair of functions $B_\bullet$ is found from $B_\circ$ with the help of rule (4.10²). This rule subsumes (i.e. is at least as optimistic as) both rules (4.9) and (4.10). We have

$$\begin{aligned} B_\bullet^{\mathrm{key}} &= B_\circ^{\mathrm{key}}[{}^{(y)_\mathcal{E}}/_{(y)_\mathcal{E} \vee (x)_\mathcal{E}}, {}^{y}/_{y \vee x}, {}^{[y]_\mathcal{E}}/_{[y]_\mathcal{E} \vee [x]_\mathcal{E}}, {}^{(x)_\mathcal{E}}/_{\mathsf{false}}, {}^{x}/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}] \\ B_\bullet^{\mathrm{sec}} &= B_\circ^{\mathrm{sec}}[{}^{y}/_{y \vee x}, {}^{[y]_\mathcal{E}}/_{[y]_\mathcal{E} \vee [x]_\mathcal{E}}, {}^{x}/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}] \ . \end{aligned}$$

I.e. both the arguments x and y of $B_\bullet^{\mathrm{sec}}$ [resp. $B_\bullet^{\mathrm{key}}$] correspond to the argument y of $B_\circ^{\mathrm{sec}}$ [resp. $B_\circ^{\mathrm{key}}$]. The argument x of $B_\circ^{\mathrm{sec}}$ [resp. $B_\circ^{\mathrm{key}}$] is irrelevant.

o is the operator $\mathcal{E}nc$

Let the label l be $x := \mathcal{E}nc(k, y)$. Then $K_\bullet = K_\circ \backslash \{x\}$.

The rules (4.9), (4.10¹) and, if $k \in K_\circ$, then also (4.11) have to be taken into account when computing $B_\bullet$ from $B_\circ$. We consider the following cases.

- If $k \notin K_\circ$, then

$$\begin{aligned} B_\bullet^{\mathrm{key}} &= \neg(x)_\mathcal{E} \wedge \left((B_\circ^{\mathrm{key}} \wedge \neg(x \vee [x]_\mathcal{E})) \vee B^{\mathrm{key}}[{}^{x}/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}, {}^{[k]_\mathcal{E}}/_{\mathsf{true}}, {}^{y}/_{\mathsf{true}}]\right) \\ B_\bullet^{\mathrm{sec}} &= (B_\circ^{\mathrm{sec}} \wedge \neg(x \vee [x]_\mathcal{E})) \vee B^{\mathrm{sec}}[{}^{x}/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}, {}^{[k]_\mathcal{E}}/_{\mathsf{true}}, {}^{y}/_{\mathsf{true}}] \ . \end{aligned}$$

  This is similar to the case where o is a "usual" operator.

- If $k \in K_\circ$, then

$$B_\bullet^{\text{key}} = \neg(x)_\mathcal{E} \wedge \Big((B_\circ^{\text{key}} \wedge \neg(x \vee [x]_\mathcal{E})) \vee B^{\text{key}}[{}^x/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}, {}^{[k]_\mathcal{E}}/_{\mathsf{true}}, {}^y/_{\mathsf{true}}] \vee \big(B_\circ^{\text{key}}[{}^x/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}] \wedge B_\circ^{\text{key}}[{}^{(k)_\mathcal{E}}/_{\mathsf{true}}, {}^{[k]_\mathcal{E}}/_{\mathsf{false}}, \langle {}^{(k')_\mathcal{E}}/_{\mathsf{false}}, {}^{[k']_\mathcal{E}}/_{(k')_\mathcal{E} \vee [k']_\mathcal{E}} \rangle_{k' \in K_\circ \setminus \{k\}}, {}^x/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}, {}^y/_{\mathsf{true}}]\big)\Big)$$

$$B_\bullet^{\text{sec}} = (B_\circ^{\text{sec}} \wedge \neg(x \vee [x]_\mathcal{E})) \vee B^{\text{sec}}[{}^x/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}, {}^{[k]_\mathcal{E}}/_{\mathsf{true}}, {}^y/_{\mathsf{true}}] \vee \big(B_\circ^{\text{sec}}[{}^x/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}] \wedge B_\circ^{\text{key}}[{}^{(k)_\mathcal{E}}/_{\mathsf{true}}, {}^{[k]_\mathcal{E}}/_{\mathsf{false}}, \langle {}^{(k')_\mathcal{E}}/_{\mathsf{false}} \rangle_{k' \in K_\circ \setminus \{k\}}, {}^x/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}, \langle {}^z/_{\mathsf{true}} \rangle_{z \in \mathbf{Var}_\mathrm{S}}, {}^y/_{\mathsf{true}}]\big) \ .$$

  Here $\langle {}^x/_{f_x} \rangle_{x \in X}$ denotes that all elements x of X have to be substituted with the corresponding f_x. Note that substituting is simultaneous, therefore the implicit order of components of the tuple $\langle {}^x/_{f_x} \rangle_{x \in X}$ is not important.

In the last case, the three disjuncts exactly correspond to the rules (4.9), (4.10^1) and $\widetilde{(4.11)}$. For example, the last disjunct says that (if $k \in K_\circ$, then) for any $Y \subseteq \mathbf{Var}$ and $X \in \mathcal{SV}_{K_\circ}(\mathbf{Var})$ the following holds:

$$(X, Y) \in \mathsf{indeps}(\mathcal{Y}) \Longleftarrow (X, Y \setminus \{x, [x]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{X}) \wedge (\{[k]_\mathcal{E}\}, Y \cup X \cup \{y\} \setminus \{x, [x]_\mathcal{E}\}) \in \mathsf{indeps}(\mathcal{X}) \ .$$

o is the operator $\mathcal{G}en$

Let the label l be $x := \mathcal{G}en()$. Then $K_\bullet = K_\circ \cup \{x\}$.

The pair of functions $B_\bullet$ is found from $B_\circ$ with the help of rules (4.10) and (4.12). Rule (4.9) is subsumed by rule (4.10), because the operator $\mathcal{G}en$ takes no arguments. We have

$$B_\bullet^{\text{key}} = \big(\neg(x)_\mathcal{E} \wedge B_\circ^{\text{key}}[{}^x/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}]\big) \vee \big(\neg x \wedge B_\circ^{\text{key}}[{}^{(x)_\mathcal{E}}/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}]\big)$$
$$B_\bullet^{\text{sec}} = B_\circ^{\text{sec}}[{}^x/_{\mathsf{false}}, {}^{[x]_\mathcal{E}}/_{\mathsf{false}}]$$

The first disjunct of $B_\bullet^{\text{key}}$ and the entire $B_\bullet^{\text{sec}}$ comes from rule (4.10). The second disjunct of $B_\bullet^{\text{key}}$ comes from rule (4.12).

5.3.2 Transfer Function for *merge*s

We start the definition of these transfer functions by restating the rules (4.17) and (4.18) once more, this time in a form that is more convenient for the explanation of the definitions of transfer functions. The rules are given in Fig. 5.1.

The sets X_o, X_c and X_k in Fig. 5.1 have the following meaning:

- X_o contains all those variables and black boxes that have not been changed in any of the branches coming to that *merge*-statement.

Let $X, Y \subseteq \widetilde{\mathbf{Var}}$. Define

$$
\begin{aligned}
X_{\mathrm{k}} &= X \cap [\mathsf{keys}(\mathcal{X})]_{\mathcal{E}} \cap [Z]_{\mathcal{E}} \\
X_{\mathrm{c}} &= X \cap ((Z \cup [Z]_{\mathcal{E}}) \backslash [\mathsf{keys}(\mathcal{X})]_{\mathcal{E}}) \\
X_{\mathrm{o}} &= X \backslash (Z \cup [Z]_{\mathcal{E}})
\end{aligned}
$$

and define Y_{o}, Y_{c}, Y_{k} similarly.

$$
\frac{\begin{array}{c}
(X_{\mathrm{o}}, Y_{\mathrm{o}}) \in \mathsf{indeps}(\mathcal{X}) \\
(X_{\mathrm{k}} \cup Y_{\mathrm{k}}, X_{\mathrm{o}} \cup Y_{\mathrm{o}} \cup \{N\}) \in \mathsf{indeps}(\mathcal{X}) \\
X_{\mathrm{c}} = Y_{\mathrm{c}} = \emptyset \\
\text{let } \{x_1, \ldots, x_m\} = X_{\mathrm{k}} \cup Y_{\mathrm{k}}, \text{ then} \\
\forall i \in \{1, \ldots, m-1\} : \ (\{[x_{i+1}]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}, \{[x_i]_{\mathcal{E}}\}) \in \mathsf{indeps}(\mathcal{X}) \\
\forall k \in Y_{\mathrm{k}} : \ (\{[k]_{\mathcal{E}}\}, \{[k]_{\mathcal{E}}\}) \in \mathsf{indeps}(\mathcal{X})
\end{array}}{(X, Y) \in \mathsf{indeps}(\mathcal{Y})} \tag{5.4}
$$

$$
\frac{\begin{array}{c}
(X, Y \cup \{N\}) \in \mathsf{indeps}(\mathcal{X}) \\
(X_{\mathrm{o}}, X_{\mathrm{k}}) \in \mathsf{indeps}(\mathcal{X}) \\
X_{\mathrm{c}} = \emptyset \\
\text{let } \{x_1, \ldots, x_m\} = X_{\mathrm{k}}, \text{ then} \\
\forall i \in \{1, \ldots, m-1\} : \ (\{[x_{i+1}]_{\mathcal{E}}, \ldots, [x_m]_{\mathcal{E}}\}, \{[x_i]_{\mathcal{E}}\}) \in \mathsf{indeps}(\mathcal{X})
\end{array}}{(X, Y) \in \mathsf{indeps}(\mathcal{Y})} \tag{5.5}
$$

Figure 5.1: Simplified rules for computing $merge(N, Z)(\mathcal{X})$

- X_{k} contains all black boxes $[k]_{\mathcal{E}}$, such that k may have been changed in one of the branches coming to that *merge*-statement and k is distributed as a key.

- X_{c} contains all variables that may have been changed in one of the branches coming to that *merge*-statement. Additionally, it also contains such black boxes $[x]_{\mathcal{E}}$ where x is not distributed as a key (and x may have been changed in one of the branches).

Let the label l be $merge(N, Z)$, where N is a variable and $Z \subseteq \mathbf{Var}$. Then $K_\bullet \subseteq K_\circ$ contains by rule (4.19) all elements of $K_\circ \backslash Z$ and by rule (4.20) all $k \in K_\circ \cap Z$, such that

$$B_\circ^{\mathrm{key}}(\mathcal{X}_{\{(k)_{\mathcal{E}}, N\}}) = \mathsf{true} \enspace .$$

I.e. the arguments of $B_\circ^{\mathrm{key}}$ are all false, except the arguments that are named $(k)_{\mathcal{E}}$ and N.

The pair of functions $B_\bullet$ is found from $B_\circ$ with the help of rules (5.4) and (5.5). At first we are going to define the functions B_1^{key} and B_1^{sec} corresponding to the application of rule (5.4) to $B_\circ^{\mathrm{key}}$ and $B_\circ^{\mathrm{sec}}$, and the functions B_2^{key} and B_2^{sec} corresponding to the application of rule (5.5) to $B_\circ^{\mathrm{key}}$ and $B_\circ^{\mathrm{sec}}$. Their disjunction then gives $B^{\mathrm{key}}_{\bullet}$ and $B^{\mathrm{sec}}_{\bullet}$. Let there be defined a total order on the set $K_\circ$ (this corresponds to giving the variables of $X_{\mathrm{k}} \cup Y_{\mathrm{k}}$ the names $x_1, \ldots, x_m$). Let this order be such, that all elements of Z are greater than all elements of $K_\circ \backslash Z$.

When defining the function B^{key}_1 we note that we are going to apply rule (5.4) only for such pairs $(X, Y) \in \mathcal{F}(\mathbf{Var})$, where the set X_{c} is empty and the set X_{o} contains only black boxes. We have

$$\begin{aligned}
B^{\mathrm{key}}_1 = {} & B^{\mathrm{key}}_\circ[\langle{}^{(k)_\mathcal{E}}/_{\mathsf{false}}, {}^{[k]_\mathcal{E}}/_{\mathsf{false}}\rangle_{k\in K_\circ\cap Z}] \wedge \\
& B^{\mathrm{key}}_\circ[\langle{}^{(k)_\mathcal{E}}/_{(k)_\mathcal{E}\vee[k]_\mathcal{E}}, {}^{[k]_\mathcal{E}}/_{\mathsf{false}}\rangle_{k\in K_\circ\cap Z}, \langle{}^{(k)_\mathcal{E}}/_{\mathsf{false}}, {}^{[k]_\mathcal{E}}/_{(k)_\mathcal{E}\vee[k]_\mathcal{E}}\rangle_{k\in K_\circ\backslash Z}, {}^{N}/_{\mathsf{true}}] \wedge \\
& \bigwedge_{z\in Z} \neg z \wedge \bigwedge_{z\in Z\backslash K_\circ} \neg[z]_\mathcal{E} \wedge \\
& \bigwedge_{k\in K_\circ\cap Z} B^{\mathrm{key}}_\circ[\langle{}^{(k')_\mathcal{E}}/_{\mathsf{false}}\rangle_{k'\leq k}, \langle{}^{(k')_\mathcal{E}}/_{(k')_\mathcal{E}\vee[k']_\mathcal{E}}\rangle_{k'>k}, {}^{[k]_\mathcal{E}}/_{(k)_\mathcal{E}\vee[k]_\mathcal{E}}, \langle{}^{x}/_{\mathsf{false}}\rangle_{x\in\widetilde{\mathbf{Var}}\backslash\{[k]_\mathcal{E}\}}] \wedge \\
& \bigwedge_{k\in K_\circ\cap Z} B^{\mathrm{key}}_\circ[{}^{(k)_\mathcal{E}}/_{[k]_\mathcal{E}}, \langle{}^{x}/_{\mathsf{false}}\rangle_{x\in[K_\circ]_\mathcal{E}\uplus\widetilde{\mathbf{Var}}\backslash\{(k)_\mathcal{E},[k]_\mathcal{E}\}}],
\end{aligned}$$

where each row corresponds to one antecedent of rule (5.4). At the first row we say that the black boxes $[k]_\mathcal{E}$, where $k \in X_{\mathrm{k}}$ or $k \in Y_{\mathrm{k}}$, do not matter. At the second row we construct the union $X_{\mathrm{k}} \cup Y_{\mathrm{k}}$ at the left side (where the black boxes are named $(k)_\mathcal{E}$) and the union $X_{\mathrm{o}} \cup Y_{\mathrm{o}}$ at the right side (where black boxes are named $[k]_\mathcal{E}$). At the third row we demand that $Y_{\mathrm{c}} = \emptyset$. At the fourth row we check whether the black boxes in X_{k} and Y_{k} are all independent of each other. Here the variable k' is assumed to range over $K_\circ$. At the fifth row we check whether the black boxes in Y_{k} are all independent of themselves.

The definition of B^{sec}_1 is a bit simpler. Here the sets X_{c} and X_{k} are both empty, because the variables in $\mathbf{Var}_{\mathrm{S}}$ are never assigned to. We have

$$\begin{aligned}
B^{\mathrm{sec}}_1 = {} & B^{\mathrm{sec}}_\circ[\langle{}^{[k]_\mathcal{E}}/_{\mathsf{false}}\rangle_{k\in K_\circ\cap Z}] \wedge \\
& B^{\mathrm{key}}_\circ[\langle{}^{(k)_\mathcal{E}}/_{[k]_\mathcal{E}}, {}^{[k]_\mathcal{E}}/_{\mathsf{false}}\rangle_{k\in K_\circ\cap Z}, \langle{}^{(k)_\mathcal{E}}/_{\mathsf{false}}\rangle_{k\in K_\circ\backslash Z}, \langle{}^{x}/_{\mathsf{true}}\rangle_{x\in\mathbf{Var}_{\mathrm{S}}}, {}^{N}/_{\mathsf{true}}] \wedge \\
& \bigwedge_{z\in Z} \neg z \wedge \bigwedge_{z\in Z\backslash K_\circ} \neg[z]_\mathcal{E} \wedge \\
& \bigwedge_{k\in K_\circ\cap Z} B^{\mathrm{key}}_\circ[\langle{}^{(k')_\mathcal{E}}/_{\mathsf{false}}\rangle_{k'\leq k}, \langle{}^{(k')_\mathcal{E}}/_{[k']_\mathcal{E}}\rangle_{k'>k}, \langle{}^{x}/_{\mathsf{false}}\rangle_{x\in\widetilde{\mathbf{Var}}\backslash\{[k]_\mathcal{E}\}}] \wedge \\
& \bigwedge_{k\in K_\circ\cap Z} B^{\mathrm{key}}_\circ[{}^{(k)_\mathcal{E}}/_{[k]_\mathcal{E}}, \langle{}^{x}/_{\mathsf{false}}\rangle_{x\in[K_\circ]_\mathcal{E}\uplus\widetilde{\mathbf{Var}}\backslash\{(k)_\mathcal{E},[k]_\mathcal{E}\}}] \ .
\end{aligned}$$

Again, each row corresponds to one antecedent of rule (5.4).

The function B^{key}_2 is the following:

$$\begin{aligned}
B^{\mathrm{key}}_2 = {} & B^{\mathrm{key}}_\circ[{}^{N}/_{\mathsf{true}}] \wedge \\
& B^{\mathrm{key}}_\circ[\langle{}^{[k]_\mathcal{E}}/_{\mathsf{false}}\rangle_{k\in K_\circ\backslash Z}, \langle{}^{(k)_\mathcal{E}}/_{\mathsf{false}}, {}^{[k]_\mathcal{E}}/_{(k)_\mathcal{E}}\rangle_{k\in K_\circ\cap Z}, \langle{}^{x}/_{\mathsf{false}}\rangle_{x\in\mathbf{Var}}, \langle{}^{[x]_\mathcal{E}}/_{\mathsf{false}}\rangle_{x\in\mathbf{Var}\backslash K_\circ}] \wedge \\
& \bigwedge_{k\in K_\circ\cap Z} B^{\mathrm{key}}_\circ[\langle{}^{(k')_\mathcal{E}}/_{\mathsf{false}}\rangle_{k'\leq k}, {}^{[k]_\mathcal{E}}/_{(k)_\mathcal{E}}, \langle{}^{x}/_{\mathsf{false}}\rangle_{x\in\widetilde{\mathbf{Var}}\backslash\{[k]_\mathcal{E}\}}] \ .
\end{aligned}$$

Here the rows correspond to the first, second and fourth antecedent of rule (5.5), the third antecedent is trivially true. In the second antecedent, the only variables that are not set to false are

- $(k)_\mathcal{E}$, where $k \in K_\circ$ and $k \notin Z$. They are left unchanged. These variables correspond to the elements of X_o.
- $[k]_\mathcal{E}$, where $k \in K_\circ \cap Z$. Their value is defined to be the original value of $(k)_\mathcal{E}$. These variables $(k)_\mathcal{E}$ correspond to the elements of X_k.

When defining B_2^sec, the second and fourth antecedents of rule (5.5) are trivially true because $X_\mathrm{k} = \emptyset$. Therefore

$$B_2^\mathrm{sec} = B_\circ^\mathrm{sec}[{}^{N}/_{\mathsf{true}}] \enspace .$$

Finally, we define

$$\begin{aligned} B_\bullet^\mathrm{key} &= (B_1^\mathrm{key} \vee B_2^\mathrm{key}) \wedge \bigwedge_{k \in K_\circ \setminus K_\bullet} \neg(k)_\mathcal{E} \\ B_\bullet^\mathrm{sec} &= B_1^\mathrm{sec} \vee B_2^\mathrm{sec} \enspace . \end{aligned}$$

5.4 An Example

We continue with our running example presented in Fig. 4.9. We also showed the same program in [Lau01], when we discussed the shortcomings of the analysis presented there. This program allows us to show, how our analysis handles the information flow through the interaction of control flow and encryption.

Let us assume that the variable `b` is private and the variables `y1` and `y2` are public. Let us also assume that the value of the variable `b` is independent of the values of the variables `y1` and `y2`. The initial analysis information (K_0, B_0), where $B_0 = (B_0^\mathrm{key}, B_0^\mathrm{sec})$, being an abstraction of the initial probability distribution of the program, is therefore the following:

- $K_0 = \emptyset$;
- $B_0^\mathrm{key} = \mathsf{true}$, because there are no variables having the form $(k)_\mathcal{E}$;
- $B_0^\mathrm{sec} = \neg\mathsf{b} \wedge \neg[\mathsf{b}]_\mathcal{E}$.

The set of keys at the end of the program is obviously $\{\mathtt{k1}, \mathtt{k2}\}$. The values of B^key and B^sec at the end of the program are given in Fig. 5.2 and Fig. 5.3, respectively. In these figures, (k)E denotes $(k)_\mathcal{E}$ and [k]E denotes $[k]_\mathcal{E}$. Also, 1 and 0 denote true and false, respectively.

From Fig. 5.2 we see, that both keys may be dependent on both black boxes. Indeed, a key k is never independent of $[k]_\mathcal{E}$ and any $[k']_\mathcal{E}$, where it is possible that $k = k'$. At the end of the program in Fig. 4.9, `k1` may be equal to `k2`.

The quantity B^sec is more interesting. We see that

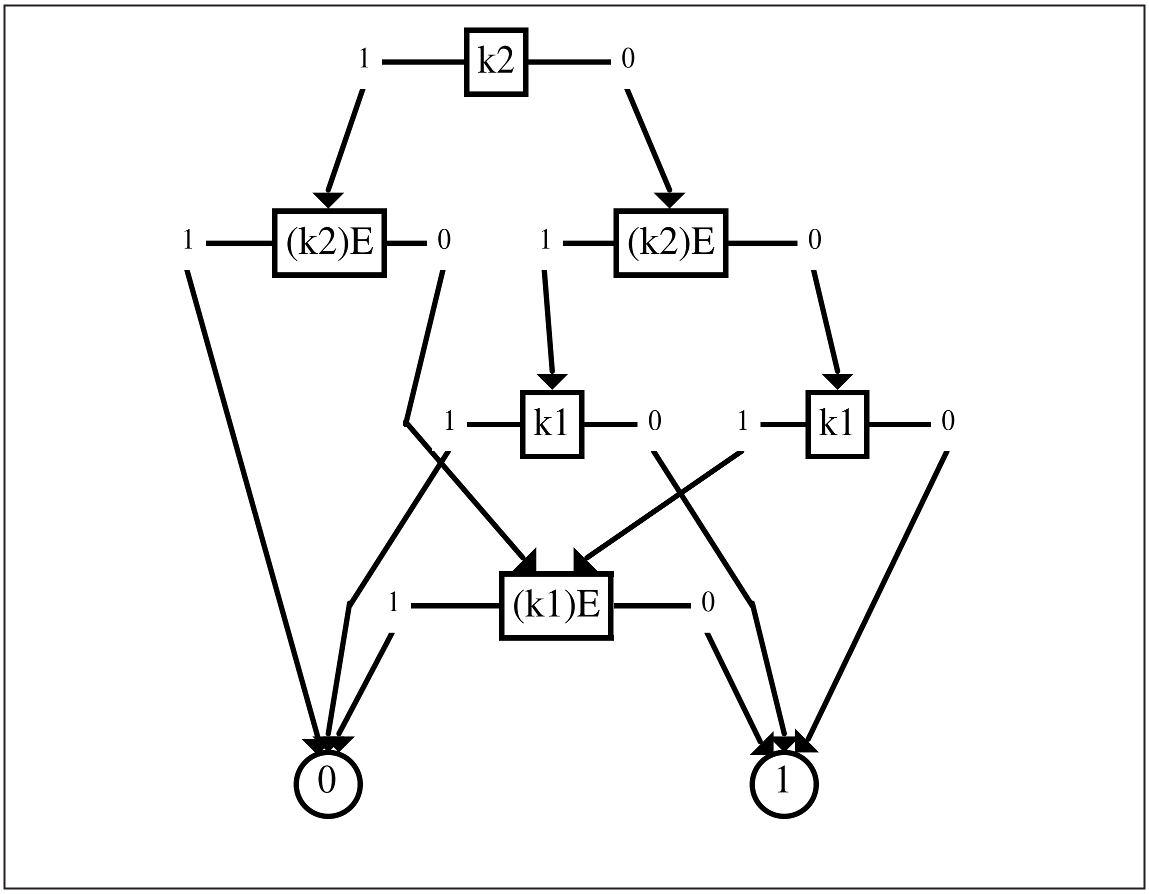


Figure 5.2: BDD representing B^{key} at the end of the program

- The variable `k2` is considered to be potentially dependent of the secret input (which was `b`). Indeed, the transfer function for *merge*-s considers, that there is a potential flow of information from the guard of the corresponding *if*-statement to all variables that have been assigned to at one of the branches.

  In our case, the variable `k2` is actually independent of `b` — no matter what the value of `b` is, `k2` is still distributed like a key. However, the value of `k2` together with the value of `x1` or with the value of $[\mathtt{k1}]_{\mathcal{E}}$ is dependent of `b` — if we have the values of both `x1` and `k2`, then we can check whether `k2` is usable for decrypting `x1` or not. But this depends on the value of `b`. Similarly, we can check whether the cryptotexts created by the black box $[\mathtt{k1}]_{\mathcal{E}}$ are decryptable by `k2` or not.

- Both `k1` and $[\mathtt{k2}]_{\mathcal{E}}$ (separately) are independent of the secret input `b`, but together they are not independent of it. We have already explained, why this is the case. Our analysis correctly (and precisely) reflects this.

- Similarly, `x2` alone is independent of the secret input, but `x2` and `k1` together depend on it.

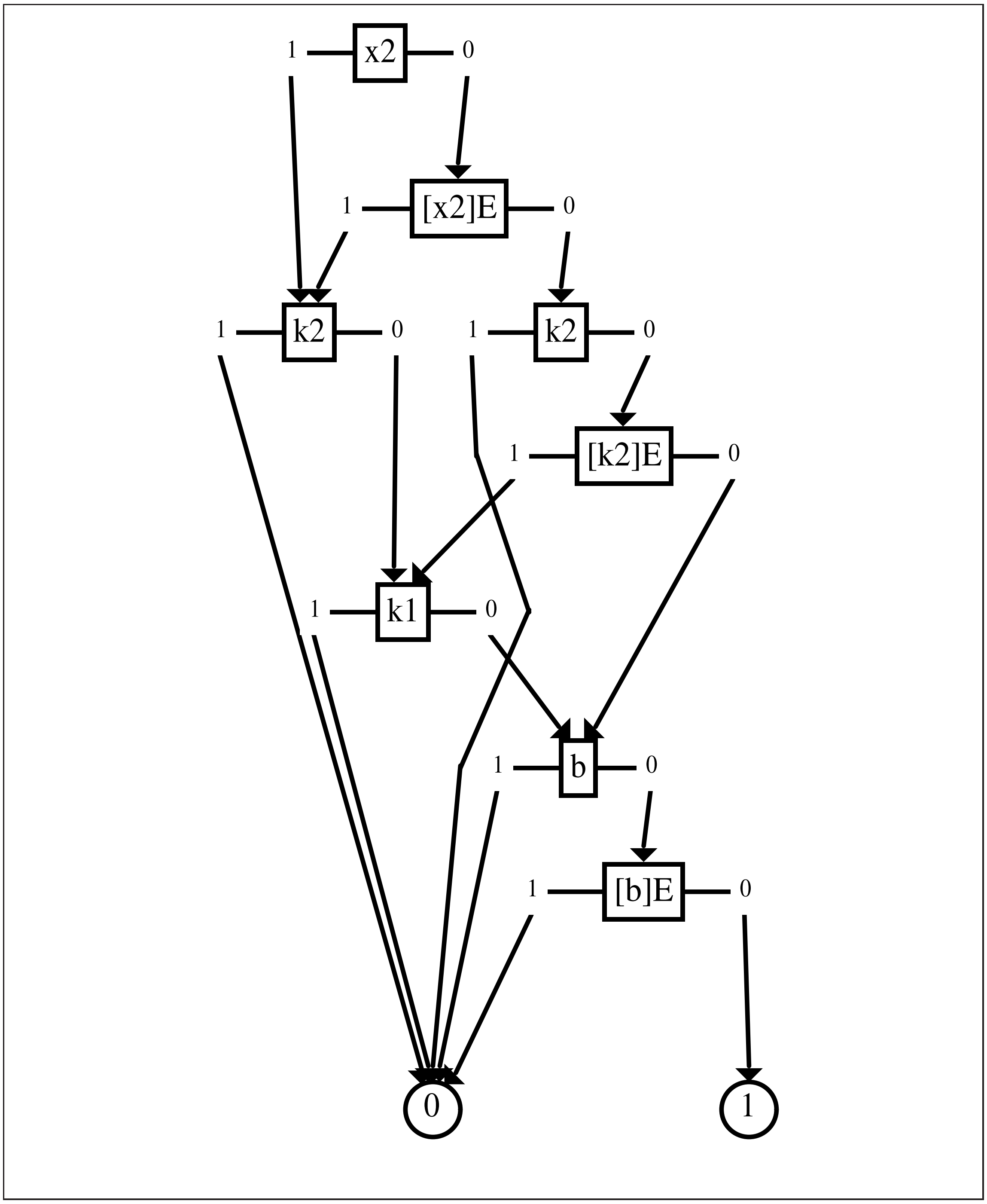


Figure 5.3: BDD representing B^{sec} at the end of the program

5.5 Putting [Lau01] to Context

In [Lau01] we gave a data flow analysis for secure information flow. Compared with the analysis in this thesis, it had a much simpler structure. This was achieved by putting several constraints on the program. Particularly, the usage of encryption keys was constrained in the following way:

- Each variable had to have one of the types “key” and “non-key”. Each operator had fixed types for its arguments and value and it was not allowed to substitute one type for another (particularly, one could not use a “key” in place of a “non-key”). The types of arguments and results of operations were the following:
  - The output of $\mathcal{G}en$ had the type “key”.
  - The first argument of $\mathcal{E}nc$ had the type “key” and the second argument had the type “non-key”. The output of $\mathcal{E}nc$ had the type “non-key”.
  - Simple assignments were polymorphic — at the statement $x := y$, the variables x and y merely had to have equal types — either “key” or “non-key”.
  - The arguments and outputs of other operations all had the type “non-key”.
- At each program point, and for each two variables of type “key”, it had to be known beforehand, whether the values of these two variables are equal or not at this program point. The option “sometimes equal and sometimes not” (where both cases have non-negligible probability) was not allowed at all. Note that this is also a constraint on the distribution of inputs of the program.

The knowledge about “keys” and ”non-keys” and about the equality of keys has to be available to the analysis presented in [Lau01]. If the program satisfies the constraints, then this knowledge can be obtained with simple and well-known methods. Particularly,

- determining the types of variables is a simple matter of type inference;
- the equality of keys can be determined using the methods of alias analysis, see, for example [Lan92, Sec. 4.2.1 and 4.2.2].

The analysis in [Lau01] was a DFA, this meant that an abstract value was assigned to each program point. Because of all the constraints put on the programs, much of the information that the implementation of the analysis presented in this chapter has to keep track of, was statically known. Namely, let $(K, (B^{\mathrm{key}}, B^{\mathrm{sec}}))$ be an abstract value of the data flow analysis given in this chapter. If the program satisfies all the constraints presented here, then

- The set K is already known before the analysis.
- The boolean function B^{key} is known, too. Namely, for each $k \in K$ let $\mathsf{E}(k) \subseteq K$ denote the set of keys that are known to be equal to the key k. Then

$$B^{\mathrm{key}} = \bigwedge_{k \in K} \neg\Big((k)_{\mathcal{E}} \wedge \bigvee_{k' \in \mathsf{E}(k)} k'\Big) \enspace .$$

  I.e. an encrypting black box $(k)_{\mathcal{E}}$ is independent of everything, except the keys that are known to be equal to k.
- The encrypting black boxes are independent of $\mathbf{Var}_{\mathrm{S}}$. In particular, the boolean function

$$B^{\mathrm{sec}} \rightarrow B^{\mathrm{sec}}[\langle{}^{[k]_{\mathcal{E}}}/_{\mathsf{true}}\rangle_{k \in K}],$$

  where $\rightarrow$ denotes the implication in the propositional calculus, is the constant function true, at least when the set $\mathbf{Var}_{\mathrm{S}}$ is assumed to not contain keys. This assumption is not essential, but it makes the presentation easier.

The abstract values of the analysis in [Lau01] did not include K or B^{key}. Also, there was no concept of encrypting black boxes in [Lau01]. The abstract values of this analysis were sets of subsets of $\mathbf{Var}$ — these are isomorphic to the boolean functions of type $\mathbb{B}^{\mathbf{Var}} \rightarrow \mathbb{B}$.

The transfer functions of the analysis in [Lau01] were special cases of transfer functions presented in Sec. 5.3. Let us present them. We still denote the abstract value of the analysis by B^{sec} and assume that it is a boolean function of type $\mathbb{B}^{\mathbf{Var}} \rightarrow \mathbb{B}$.

Assignments — usual operators

The transfer function of a node labelled with $x := o(x_1, \ldots, x_k)$, where o is not $\mathcal{E}nc$, is the following:

$$B^{\mathrm{sec}}_{\bullet} = (B^{\mathrm{sec}}_{\circ} \wedge \neg x) \vee B^{\mathrm{sec}}_{\circ}[{}^{x}/_{\mathsf{false}}, {}^{x_1}/_{\mathsf{true}}, \ldots, {}^{x_k}/_{\mathsf{true}}] \enspace .$$

We see that there is almost no difference with the transfer function presented in Sec. 5.3.

The transfer functions for simple assignments $x := y$ and the key generation operations $x := \mathcal{G}en()$ also had the same shape. In the analysis given in this thesis, simple assignments and key generation operations are special cases because they handle *encrypting black boxes* more optimistically than the general case.

Assignments — encryptions

The transfer function of a node labelled with $x := \mathcal{E}nc(k, y)$ is the following:

$$B^{\mathrm{sec}}_{\bullet} = (B^{\mathrm{sec}}_{\circ} \wedge \neg x) \vee B^{\mathrm{sec}}_{\circ}[{}^{x}/_{\mathsf{false}}, {}^{k}/_{\mathsf{true}}, {}^{y}/_{\mathsf{true}}] \vee \Big(B^{\mathrm{sec}}_{\circ}[{}^{x}/_{\mathsf{false}}] \wedge \neg \bigvee_{k' \in \mathsf{E}(k)} k'\Big) \enspace .$$

Here the three disjuncts correspond to the three disjuncts in the definition of $B^{\text{sec}}_{\bullet}$ for encryptions in Sec. 5.3. Note that we have substituted the value of $B^{\text{key}}_{\circ}$ to the formula.

merge-**s**

The transfer function of a node labelled with $\mathit{merge}(N, Z)$ is the following:

$$B^{\text{sec}}_{\bullet} = (B^{\text{sec}}_{\circ} \wedge \bigwedge_{z \in Z} \neg z) \vee B^{\text{sec}}_{\circ}[{}^{N}/_{\mathsf{true}}] \ .$$

Here the two disjuncts correspond to B^{sec}_{1} and B^{sec}_{2} in Sec. 5.3. Two conjuncts of the first disjunct here correspond to the conjuncts in the first and third row of the definition of B^{sec}_{1} in Sec. 5.3. Other conjuncts in the definition of B^{sec}_{1} are trivially true.

Chapter 6

Pseudorandom Permutations

In the preceding chapters, we have given a secure information flow analysis for programs containing encryption as the primitive operation. The semantics of the encryption operation had to be a repetition-concealing and which-key concealing encryption system.

We would also like to have results for the case, where the encryption primitive is only a pseudorandom permutation, i.e. we do not require any stronger security properties of it. Our interest is motivated by the following considerations:

- A PRP is easier to implement: it is believed that block ciphers like DES [DES99], AES [AES01], etc. are PRPs. Especially, one probably does not need a random number generator to implement a PRP, while random numbers are necessary for achieving repetition-concealedness.

- PRPs are more "primitive" than the primitives satisfying stronger security properties. Indeed, PRPs are usually used as a building block in constructing stronger primitives. For example, the modes of operation of block ciphers (see [Sch96, Chapter 9] for definitions and [BDJR97] for proofs of security) aim at achieving repetition-concealedness or even stronger properties.

We would like to have a program analysis similar to the one presented before. This analysis should be correct whenever the semantics of the encryption operator is a PRP. It also should precisely model the security offered by a PRP.

However, we are unable to give such an analysis. A PRP does not hide the identity of plaintexts, thus the analysis should model the possible (in)equality of each pair of values that may be encrypted with the same key. The number of such values may be unbounded, though. Also, it is not clear how to name them, not much precision can be achieved when talking about the (in)equality of anonymous values. However, the number of values would be finite and they all would have well-defined names, if we left the loops out of the programming language.

A programming language without loops has the computation power equal to that of *formal expressions* that are used, for example, to model the messages in the analysis of cryptographic protocols. Formal expressions are the object of our research in this chapter, we prefer to use an existing formalism, rather than to define a new programming language. This makes it easier to compare our results with those of other authors.

In this chapter we start by defining the language of formal expressions in Sec. 6.1 and giving a semantics for them — the *computational interpretation* in Sec. 6.2. The definition of formal expressions given here has similar expressiveness as the corresponding definition in [AR00]. It contains only the most basic means — tupling and encryption — to construct more complex expressions from simpler ones[1]. Similarly to [AR00, AJ01], we look for sufficient conditions on formal expressions for the indistinguishability of their interpretations. We concentrate on the indistinguishability of the interpretations of expressions, rather than on the independence of subexpressions of an expression, to make the comparison of results easier.

To compare the strength of PRPs to that of which-key and repetition concealing encryption, we prove a very similar result to [AR00] in Sec. 6.3. Similarly to them, we define an equivalence relation on formal expressions, such that two formal expressions have indistinguishable interpretations whenever they are equivalent. The only visible difference between our result and the result of [AR00] is the power of encryption (and our equivalence relation is a bit finer-grained).

There are some important less-visible differences, though. They become apparent if we try to increase the expressiveness of formal expressions, as we do in Sec. 6.4. Before this change, the computational interpretation given in Sec. 6.2 was *injective* — each bit-string could be the interpretation of at most one formal expression. When the interpretation is no longer injective, but the encryption system is which-key and encryption concealing, then the results of [AR00] are still useful — they would be rather easy to generalise for a richer language of formal expressions. If the encryption system is a pseudorandom permutation then these results do not generalise. We have to put more work to tracking the possible equality of the interpretations of subexpressions.

In Sec. 6.5 we give our analysis for checking, whether the interpretations of two formal expressions are indistinguishable. We also give its proof of correctness in this section. It is much easier to prove correct than the analysis in Chapter 4, because we have no loops here and therefore no fixed points that need approximation. The main idea in the proof of correctness of the analysis (or at least the part of analysis that deals with encryption) is to assume, that instead of an encryption operation we have an application of a random function. This assumption is obviously justified by us requiring, that the encryption operation is a pseudorandom permutation and therefore also a pseudorandom function. This idea first appears in [BR93a].

We demonstrate the power of our analysis in Sec. 6.6 where we use it to derive the security for the modes of operation of block ciphers. Basically, we are going to automatically derive the results of Bellare et al. [BKR94, BDJR97] (only for the asymptotic case, though).

[1]However, these means are sufficient to express many cryptographic protocols

6.1 Formal Expressions

The formal expressions are members of a certain formal language. One can compare them to programs, which are also just members of a certain language. Later we are going to give a computational interpretation for formal expressions; this interpretation might be compared to the semantics of programs.

Let **Keys** be a fixed, nonempty set. Formally, its elements do not have any further structure. Informally, they represent encryption keys (this informal interpretation is later formalised by the computational interpretation).

Definition 6.1. The set of *formal expressions*, denoted by **Exp**, is the set defined by the following grammar:

$$\begin{array}{rcll} E & ::= & K & \text{key (for } K \in \mathbf{Keys}) \\ & | & (E_1, E_2) & \text{pair (for } E_1, E_2 \in \mathbf{Exp}) \\ & | & \{E'\}_K & \text{encryption (for } E' \in \mathbf{Exp},\ K \in \mathbf{Keys}) \end{array}$$

When a formal expression describes a message that a party in a cryptographic protocol sends to another one, then a pair (E_1, E_2) denotes the pairing of messages that are described by formal expressions E_1 and E_2. It could be implemented by concatenation plus markers, one possible implementation is given in Sec. 6.2. Similarly, $\{E\}_K$ denotes the encryption of the message described by E with a key described by K.

These meanings are all informal, though. Formal meaning is given by the computational interpretation in Sec. 6.2.

Someone having received a message, may attempt to analyse it. The following *defines*, how a message, described by the formal expression E, may be analysed:

Definition 6.2. Let $E, E' \in \mathbf{Exp}$. Expression E' can be *obtained* from E, if $E \vdash E'$, where the relation $\vdash$ is given by the following inductive definition:

1. $E \vdash E$;
2. if $E \vdash E_1$ and $E \vdash E_2$ then $E \vdash (E_1, E_2)$;
3. if $E \vdash (E_1, E_2)$ then $E \vdash E_1$ and $E \vdash E_2$;
4. if $E \vdash E'$ and $E \vdash K$ then $E \vdash \{E'\}_K$;
5. if $E \vdash \{E'\}_K$ and $E \vdash K$ then $E \vdash E'$.

(here $E, E', E_1, E_2 \in \mathbf{Exp}$ and $K \in \mathbf{Keys}$)

The definition is a claim, that the only "useful" messages that can be derived from a message described by E, are described by formal expressions E', such that $E \vdash E'$. This claim must be a consequence of the properties of the interpretation of formal expressions. There exists no satisfactory proof for that claim, though. The rest of this chapter could be seen to be a partial justification for it. We use the relation $\vdash$ in Sec. 6.3.

Let us define some more useful notions, concerning the structure of formal expressions. Let $E_1 \sqsubseteq E_2$ denote that E_1 is a subexpression of E_2. Here we do not consider a key K to be a subexpression of $\{E\}_K$. The exact definition of $\sqsubseteq$ is given by the following:

- $E \sqsubseteq E$;
- if $E \sqsubseteq E'$ then $E \sqsubseteq (E', E'')$ and $E \sqsubseteq (E'', E')$;
- if $E \sqsubseteq E'$ then $E \sqsubseteq \{E'\}_K$.

In this definition we do not attempt to reflect the fact that the encryption is supposed to hide the structure of the encrypted text. The definition of $\sqsubseteq$ is also usable, if the key K is known.

Let E be an expression and let K, K' be two keys that occur in E. We say that K *encrypts* K' in E, if there exists an expression E', such that $K' \sqsubseteq E'$ and $\{E'\}_K \sqsubseteq E$. We say that E has an *encryption cycle*, if there exist keys $K_1, \ldots, K_l$, such that K_i encrypts K_{i+1} in E for all $i \in \{1, \ldots, l-1\}$ and K_l encrypts K_1 in E. The results in this chapter only apply for expressions that have no encryption cycles. The security definitions of encryption systems do not cover the case, where E has encryption cycles (see [AR00, Sec. 4.2]).

6.2 Interpretation of Expressions

The interpretation of formal expressions that we are going to define here is more or less the same as the interpretation given by Abadi and Rogaway [AR00].

Let $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ be an encryption system that is a pseudorandom permutation (Def. 2.8). Let the length of keys, corresponding to the security parameter n, be $\ell(n)$. Let $\mathbf{KeyVal}_n$ denote the set $\{\mathbf{0}, \mathbf{1}\}^{\ell(n)}$ and let $\mathbf{Val}_n$ denote the entire set of bit-strings $\{\mathbf{0}, \mathbf{1}\}^*$. We have defined $\mathbf{Val}_n$ to be able to talk about the family of sets $\mathbf{Val} = \{\mathbf{Val}_n\}_{n \in \mathbb{N}}$ and about families of probability distributions over it. If E is a formal expression, then the interpretation of E, denoted $[\![E]\!]$, is a family of probability distributions over the (family of) set(s) of bit-strings $\mathbf{Val}$.

The interpretation of formal expressions first assigns a value to each atomic expression and then computes a value for all other expressions. For the expressions defined in Sec. 6.1, the atomic expressions are keys. Let $\mathbf{Init}_n = \mathbf{Keys} \rightarrow \mathbf{Val}_n$ and let $\mathbf{Init}$ be the set of countable tuples $\prod_{n \in \mathbb{N}} \mathbf{Init}_n$. The mapping *init* that assigns the values to keys has the type $\mathbf{Init}$. Later we fix the family of probability distributions over $\mathbf{Init}$, according to which *init* is picked.

Let $\tau : \{\mathbf{0}, \mathbf{1}\}^* \times \{\mathbf{0}, \mathbf{1}\}^* \rightarrow \{\mathbf{0}, \mathbf{1}\}^*$ be a fixed polynomial-time computable *injective* function. Moreover, for $x, y \in \{\mathbf{0}, \mathbf{1}\}^*$, let $|\tau(x, y)|$ depend only on $|x|$ and $|y|$. The function τ is used to define the interpretation of messages whose outermost constructor is a pairing. Although we do not require that τ is efficiently invertible, it is more intuitive to assume that (consider the item 3 in Def. 6.2).

Such τ could be defined as follows. Let $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be defined as $f(i,j) = \frac{(i+j)(i+j+1)}{2} + i$. Then f is (a rather well-known example of) an injective function from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$. Let $x, y \in \{\mathbf{0},\mathbf{1}\}^*$. Let $i_x, i_y \in \mathbb{N}$ be such, that their binary representations are $\mathbf{1}x$ and $\mathbf{1}y$, respectively. Let $j = f(i_x, i_y)$. Let $\tau(x,y)$ be defined so, that it is the binary representation of j that has been padded from left with bits $\mathbf{0}$ to be as long as the binary representation of $f(2^{|x|+2} - 1, 2^{|y|+2} - 1)$. Then τ is injective, because the construction of i_x (and i_y) is an injective operation from $\{\mathbf{0},\mathbf{1}\}^*$ to $\mathbb{N}$. The padding ensures that the length of $\tau(x,y)$ only depends on the lengths of x and y.

For a given mapping *init*, the mapping conv gives for each formal expression the corresponding bit-string.

$$\begin{aligned}
\mathrm{conv}_n(\mathit{init}, K) &= \mathit{init}_n(K) \\
\mathrm{conv}_n(\mathit{init}, (E_1, E_2)) &= \tau(\mathrm{conv}_n(\mathit{init}, E_1), \mathrm{conv}_n(\mathit{init}, E_2)) \\
\mathrm{conv}_n(\mathit{init}, \{E\}_K) &= \mathcal{E}(\mathbf{1}^n, \mathit{init}_n(K), \mathrm{conv}_n(\mathit{init}, E)) \ .
\end{aligned} \tag{6.1}$$

For defining the computational interpretation $[\![E]\!] \in \mathcal{D}^{\mathbb{N}}(\mathbf{Val})$ of the formal expression E, we have to bind *init*. Define

$$[\![E]\!] = \{\!|\mathrm{conv}_n(\mathit{init}, E) \ : \ \left\langle \mathit{init}_n(K) \leftarrow \mathcal{G}(\mathbf{1}^n) \right\rangle_{K \in \mathbf{Keys}} |\!\}_{n \in \mathbb{N}} \ .$$

We have fixed the distribution of *init* by defining that each single key has the distribution given by $\mathcal{G}$ and the distributions of different keys are completely independent.

In this chapter, we are looking for sufficient conditions, expressible in terms of formal expressions $E, E' \in \mathbf{Exp}$, for $[\![E]\!] \approx [\![E']\!]$.

6.3 Explicit Interpretation of Constructors

The interpretation given in the previous section does not in general allow to determine, whether a bit-string has been created by pairing or by encryption. Whether this is desirable or not, depends on the application.

If the application is a cryptographic protocol where the expressions play the role of messages exchanged between participants, and the honest participants analyse the structure of the messages that they have received, then each expression has to be tagged with the identity of its outermost constructor. Basically, everything that is defined to be obtainable from an expression in the formal setting (Def. 6.2) must remain obtainable in the computational setting.

Adding tags is straightforward. Let key, pair and $\mathsf{ciphertext}$ be three fixed bit-strings (they have to be different from everything else) and redefine the mapping conv by

$$\begin{aligned}
\mathrm{conv}_n(\mathit{init}, K) &= \tau(\mathit{init}_n(K), \mathsf{key}) \\
\mathrm{conv}_n(\mathit{init}, (E_1, E_2)) &= \tau(\tau(\mathrm{conv}_n(\mathit{init}, E_1), \mathrm{conv}_n(\mathit{init}, E_2)), \mathsf{pair}) \\
\mathrm{conv}_n(\mathit{init}, \{E\}_K) &= \tau(\mathcal{E}(\mathbf{1}^n, \mathit{init}_n(K), \mathrm{conv}_n(\mathit{init}, E)), \mathsf{ciphertext}) \ .
\end{aligned}$$

The definition of $[\![E]\!]$ remains literally the same.

Instead of introducing tags, we could have introduced constants to the language defining **Exp**. In this case we could already have tagged the formal expressions. Not tagging the computational interpretations of messages can thus be considered more general and in most of this chapter, onward from Sec. 6.4, we are going to present the results about this case. In this section, however, we are going to use tagged interpretations to give an analogue to the result of [AR00] — in Sec. 6.3.1 we are going to define an equivalence relation $\cong$ on **Exp**, such that $E \cong E'$ is sufficient for $[\![E]\!] \approx [\![E']\!]$, as we prove in Sec. 6.3.2. Introducing tags has made the function $\mathrm{conv}(init, \cdot)$ injective and we need this property here.

6.3.1 The Equivalence Relation $\cong$

For the purposes of defining the relation $\cong$, we define the language **Pat** of *patterns* as an extension of the language of expressions as follows:

$$\begin{array}{rcll} P & ::= & E & \text{an expression} \\ & | & \Box_K^E & \text{undecryptable with identity } (K, E) \in \mathbf{Keys} \times \mathbf{Exp} \end{array}$$

Intuitively, $\Box_K^E$ denotes the same expression as $\{E\}_K$ for someone who cannot obtain the key K. For someone who knows the keys in $\mathbf{K} \subseteq \mathbf{Keys}$ we now define, how he sees the expression E. It is given by the pattern $pat(E, \mathbf{K})$ that is defined by

$$\begin{aligned} pat(K, \mathbf{K}) &:= K \\ pat((E_1, E_2), \mathbf{K}) &:= (pat(E_1, \mathbf{K}), pat(E_2, \mathbf{K})) \\ pat(\{E\}_K, \mathbf{K}) &:= \begin{cases} \{pat(E, \mathbf{K})\}_K & \text{if } K \in \mathbf{K} \\ \Box_K^E & \text{if } K \notin \mathbf{K} \end{cases} . \end{aligned}$$

For each expression E, there is an obvious choice for the set of known keys $\mathbf{K}$. This is the set of those keys that can be obtained from the expression E itself (see Def. 6.2). Let us define $pattern(E) := pat(E, \{K \in \mathbf{Keys} : E \vdash K\})$. This describes, how an expression looks like for someone that has no prior knowledge about the keys, but obtains the keys from the expression E itself.

Let us see examples of $pattern(E)$. In the following, K and K_i, where $i \in \mathbb{N}$, denote keys.

- $pattern(K) = K$
- $pattern((\{K_2\}_{K_1}, \{K_3\}_{K_1})) = (\Box_{K_1}^{K_2}, \Box_{K_1}^{K_3})$
- $pattern((\{\{K_1\}_{K_2}\}_{K_3}, K_3)) = (\{\Box_{K_2}^{K_1}\}_{K_3}, K_3)$
- $pattern((\{\{(K_1, K_1)\}_{K_2}\}_{K_3}, K_3)) = (\{\Box_{K_2}^{(K_1,K_1)}\}_{K_3}, K_3)$

Before defining the relation $\cong$, let us define the meaning of applying certain bijections to patterns. Recall that $\mathcal{S}(X)$ denoted the set of permutations of the set X.

For $P \in \mathbf{Pat}$ and $\sigma_{\mathrm{K}} \in \mathcal{S}(\mathbf{Keys})$ let $P\sigma_{\mathrm{K}}$ be a pattern where each $K \in \mathbf{Keys}$ that occurs as a subpattern (i.e. subexpression) of P or as an encryption key, is replaced with $\sigma_{\mathrm{K}}(K)$. However, the indices of undecryptables are not permuted by σ_{K}. Formally

$$\begin{aligned} K\sigma_{\mathrm{K}} &= \sigma_{\mathrm{K}}(K) \\ (P_1, P_2)\sigma_{\mathrm{K}} &= (P_1\sigma_{\mathrm{K}}, P_2\sigma_{\mathrm{K}}) \\ \{P\}_K\sigma_{\mathrm{K}} &= \{P\sigma_{\mathrm{K}}\}_{\sigma_{\mathrm{K}}(K)} \\ \Box_K^E\sigma_{\mathrm{K}} &= \Box_K^E \ . \end{aligned}$$

For $P \in \mathbf{Pat}$ and $\sigma_\Box \in \mathcal{S}(\mathbf{Keys} \times \mathbf{Exp})$ let $P\sigma_\Box$ be a pattern where each subpattern $\Box_K^E$ is replaced by $\Box_{K'}^{E'}$, where $(K', E') = \sigma_\Box(K, E)$.

We now define $E_1 \cong E_2$ iff there exist $\sigma_{\mathrm{K}} \in \mathcal{S}(\mathbf{Keys})$ and $\sigma_\Box \in \mathcal{S}(\mathbf{Keys} \times \mathbf{Exp})$, such that $\mathit{pattern}(E_1) = \mathit{pattern}(E_2)\sigma_{\mathrm{K}}\sigma_\Box$ and $\sigma_\Box$ preserves the lengths of the interpretations of expressions, i.e. for each $K \in \mathbf{Keys}$, $E \in \mathbf{Exp}$ and $(K', E') = \sigma_\Box(K, E)$, the equality $|[\![E]\!]| = |[\![E']\!]|$ holds. We have defined the interpretation of expressions in such a way, that it is easy to compute the length of the interpretation of a formal expression from the structure of this expression.

Some examples of $\cong$:

- $K \cong K'$ for all $K, K' \in \mathbf{Keys}$. We can choose the permutation σ_{K} so, that $\sigma_{\mathrm{K}}(K') = K$.
- $(\{K_2\}_{K_1}, \{K_3\}_{K_1}) \cong (\{K_2\}_{K_1}, \{K_3\}_{K_4})$. Indeed, the corresponding patterns are $(\Box_{K_1}^{K_2}, \Box_{K_1}^{K_3})$ and $(\Box_{K_1}^{K_2}, \Box_{K_4}^{K_3})$, the permutation $\sigma_\Box$ therefore has to map (K_4, K_3) to (K_1, K_3) and (K_1, K_2) to (K_1, K_2). Such mapping preserves the lengths of the interpretations of expressions.
- $(\{K_2\}_{K_1}, \{K_3\}_{K_1}) \cong (\{K_2\}_{K_1}, \{K_2\}_{K_4})$. The corresponding patterns are $(\Box_{K_1}^{K_2}, \Box_{K_1}^{K_3})$ and $(\Box_{K_1}^{K_2}, \Box_{K_4}^{K_2})$ and the permutation $\sigma_\Box$ can be defined. Its length-preservingness follows from the obvious equality $|[\![K_2]\!]| = |[\![K_3]\!]|$.
- $(\{K_2\}_{K_1}, \{K_2\}_{K_1}) \not\cong (\{K_2\}_{K_1}, \{K_3\}_{K_1})$. The corresponding patterns are $(\Box_{K_1}^{K_2}, \Box_{K_1}^{K_2})$ and $(\Box_{K_1}^{K_2}, \Box_{K_1}^{K_3})$ and $\sigma_\Box$ would have to map both (K_1, K_2) and (K_1, K_3) to (K_1, K_2). But in this case $\sigma_\Box$ would not be a permutation.
- $(\{\{K_1\}_{K_2}\}_{K_3}, K_3) \not\cong (\{\{(K_1, K_1)\}_{K_2}\}_{K_3}, K_3)$. The corresponding patterns (already given before) are $(\{\Box_{K_2}^{K_1}\}_{K_3}, K_3)$ and $(\{\Box_{K_2}^{(K_1,K_1)}\}_{K_3}, K_3)$. Here the permutation σ_{K} must map K_3 to K_3 and the permutation $\sigma_\Box$ must map $(K_2, (K_1, K_1))$ to (K_2, K_1). But such permutation $\sigma_\Box$ would not be length-preserving.

We have the following result:

Theorem 6.1. *Let E_1 and E_2 be formal expressions. If E_1 and E_2 contain no encryption cycles and $E_1 \cong E_2$, then $[\![E_1]\!] \approx [\![E_2]\!]$.*

6.3.2 Proof of Indistinguishability

This subsection deals with proving the theorem 6.1. It follows the proof by Abadi and Rogaway [AR00, Sec. 5.2].

Extending the interpretation to $\Box_K^E$

We start by defining the computational interpretation for all patterns, not only expressions. We have to extend the mappings *init* and conv to undecryptables and fix the distribution of such an extended *init*.

The mapping *init* now assigns the values not only to keys but also to undecryptables. We change the definition of $\mathbf{Init}_n$ to the type $(\mathbf{Keys} \uplus \mathbf{Keys} \times \mathbf{Exp}) \to \mathbf{Val}_n$. As before, $\mathbf{Init} = \prod_{n \in \mathbb{N}} \mathbf{Init}_n$ and $init \in \mathbf{Init}$. We extend the function conv by

$$\mathrm{conv}_n(i, \Box_K^E) = \tau(init_n(K, E), \mathsf{ciphertext}) \ .$$

The interpretation $[\![P]\!] \in \mathcal{D}^{\mathbb{N}}(\mathbf{Val})$ is defined by

$$[\![P]\!] = \{\!|\mathrm{conv}_n(init, P) \ : \ init \leftarrow D_n^{\mathbf{Init}}|\!\}_{n \in \mathbb{N}}$$

where the distribution $D_n^{\mathbf{Init}}$ is such, that

- different arguments of *init* are independent of each other;
- $init_n(K)$ is distributed according to $\mathcal{G}(\mathbf{1}^n)$;
- $init_n(K, E)$ is uniformly distributed over the bit-strings of length $|[\![E]\!]_n|$.

Key renaming

Another step (independent of the previous one) is to topologically sort the keys in E_1 and E_2. Let $Keys(E) \subseteq \mathbf{Keys}$ be the set of all keys that occur in the expression E (either as a subexpression of E or as an encryption key). We divide this set into two parts — the keys that an adversary can recover and the keys that it cannot find out:

$$\begin{aligned} recoverable(E) &= \{K \ : \ K \in \mathbf{Keys}, E \vdash K\} \\ hidden(E) &= Keys(E) \backslash recoverable(E) \end{aligned}$$

Let the cardinality of the set $hidden(E_j)$ be l_j, where $j \in \{1, 2\}$. Let the elements of the set $hidden(E_j)$ be called $K_j^{(1)}, \ldots, K_j^{(l_j)}$, where the indexes $1, \ldots, l_j$ are assigned to the keys in such a way that if $K_j^{(s)}$ encrypts $K_j^{(t)}$ in E_j, then $s > t$. The keys can be ordered in such a way, because E_j has no encryption cycles. Intuitively, the order on $hidden(E_j)$ is such that the key $K_j^{(1)}$ is “the deepest” — there is no subexpression $\{E'\}_{K_j^{(1)}}$ of E_j, such that any of the hidden keys of E_j are subexpressions of E'.

Constructing and comparing hybrids

We use the hybrid argument to show that $[\![E_1]\!] \approx [\![E_2]\!]$. We start by defining the "steps" between $[\![E_1]\!]$ and $[\![E_2]\!]$.

For $j \in \{1, 2\}$ and $s \in \{0, \ldots, l_j\}$ define the pattern $E_j^{(s)}$ by

$$E_j^{(s)} = \mathit{pat}(E_j, \mathit{recoverable}(E_j) \cup \{K_j^{(1)}, \ldots, K_j^{(s)}\}) \ .$$

These patterns satisfy the following properties:

- $E_j^{(l_j)} = E_j$. Indeed, all keys are known when defining $E_j^{(l_j)}$. Hence no subexpressions E_j are replaced with undecryptables.
- $E_j^{(s)}$ does not contain the keys $K_j^{(s+1)}, \ldots, K_j^{(l_j)}$ (except in the indexes of undecryptables). This follows from the acyclicity of the "encrypts"-relation in E_j.

Consider the following sequence of distributions:

$$[\![E_1^{(l_1)}]\!], [\![E_1^{(l_1-1)}]\!], \ldots, [\![E_1^{(0)}]\!], [\![E_2^{(0)}\sigma_K\sigma_\Box]\!], [\![E_2^{(0)}]\!], [\![E_2^{(1)}]\!], \ldots, [\![E_2^{(l_2)}]\!] \tag{6.2}$$

Our task is to show that the two outermost distributions ($[\![E_1^{(l_1)}]\!]$ and $[\![E_2^{(l_2)}]\!]$) are indistinguishable. We do it by showing that any two neighbouring distributions in the sequence (6.2) are indistinguishable. The rest follows then from the transitivity of indistinguishability (lemma 2.9).

The assumption gives us $E_1^{(0)} = E_2^{(0)}\sigma_K\sigma_\Box$ and thus also $[\![E_1^{(0)}]\!] = [\![E_2^{(0)}\sigma_K\sigma_\Box]\!]$. By the properties of σ_K and $\sigma_\Box$ and the interpretation of keys and undecryptables, $[\![E_2^{(0)}\sigma_K\sigma_\Box]\!] = [\![E_2^{(0)}]\!]$. Indeed, the interpretations of all keys are equal (as distributions) and the interpretations of undecryptables of the same length are also equal.

Breaking the encryption

Assume that there exists a $j \in \{1, 2\}$ and $s \in \mathbb{N}$, $0 \le s < l_j$, such that $[\![E_j^{(s)}]\!] \not\approx [\![E_j^{(s+1)}]\!]$. Denote the algorithm distinguishing those two distributions by $\mathcal{B}$. In this case the algorithm $\mathcal{A}_{E_j,s}^{(\cdot)}$ given in Fig. 6.1 shows that the encryption system $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ that we are using in computationally interpreting formal expressions, is not a pseudorandom function. Indeed, the following lemma holds:

Lemma 6.2. *If the oracle given to $\mathcal{A}_{E_j,s}^{(\cdot)}(\mathbf{1}^n)$ is a random function, then the output of* CONV(E_j) *has a distribution that is indistinguishable from $[\![E_j^{(s)}]\!]$.*

Proof. It is sufficient to show that after fixing *init* in the algorithm $\mathcal{A}_{E,s}^{(\cdot)}$, $E_1 \neq E_2$ almost always implies[2] CONV$(E_1) \neq$ CONV(E_2) for all $E_1, E_2 \sqsubseteq E$, where

[2] the opposite case has negligible probability

Let E have the hidden keys $K^{(1)}, \ldots, K^{(l)}$, sorted by the "encrypts"-relation.
Let $\mathcal{PRF}_{s+2}, \ldots, \mathcal{PRF}_l$ be different instances of the stateful oracle $\mathcal{PRF}$ (see Fig. 2.4), i.e. they have separate internal states.

The algorithm $\mathcal{A}_{E,s}^{f(\cdot)}(\mathbf{1}^n)$:

for all $K \in \mathbf{Keys}$, $K \sqsubseteq E$ **do**
 $\mathit{init}(K) \leftarrow \mathcal{G}(\mathbf{1}^n)$
end for
Initialise the internal states of $\mathcal{PRF}_{s+2}, \ldots, \mathcal{PRF}_l$
Initialise *encode* (see below; may be done by tabulating its values)
$x \leftarrow \mathrm{CONV}(E)$
$\mathit{result} \leftarrow \mathcal{B}(\mathbf{1}^n, x)$
return *result*

The algorithm CONV(E):

if $E = K$, $K \in \mathbf{Keys}$ **then**
 $\mathit{result} := \tau(\mathit{init}(K), \mathsf{key})$
else if $E = (E_1, E_2)$ **then**
 $x_1 \leftarrow \mathrm{CONV}(E_1)$
 $x_2 \leftarrow \mathrm{CONV}(E_2)$
 $\mathit{result} := \tau(\tau(x_1, x_2), \mathsf{pair})$
else if $E = \{E'\}_K$ **then**
 $x' \leftarrow \mathrm{CONV}(E')$
 if $K \in \mathit{recoverable}(E) \cup \{K^{(1)}, \ldots, K^{(s)}\}$ **then**
  $x = \mathcal{E}(\mathbf{1}^n, \mathit{init}(K), x')$
 else if $K = K^{(s+1)}$ **then**
  $x \leftarrow f(x')$
 else if $K = K^{(t)}$, $s + 2 \leq t \leq l$ **then**
  $x \leftarrow \mathcal{PRF}_t(\mathit{encode}(E'))$
 end if
 $\mathit{result} := \tau(x, \mathsf{ciphertext})$
end if
return *result*

encode : $\{E' \in \mathbf{Exp} : E' \sqsubseteq E\} \to \{\mathbf{0}, \mathbf{1}\}^*$ is an injective function, such that for all $E' \sqsubseteq E$, the length of $\mathit{encode}(E')$ is equal to the length of $[\![E']\!]_n$.

Figure 6.1: The Adversary $\mathcal{A}_{E,s}^{(\cdot)}$

the probability is taken over the distribution of *init* and over the choices of the oracles $\mathcal{PRF}_t$. Indeed, the only reason why we cannot say that the output of $\mathrm{CONV}(E_j)$ is distributed *identically* to $[\![E_j^{(s)}]\!]$, is the following: assume that the expression E has subexpressions $\{E_1\}_{K^{(s+1)}}$ and $\{E_2\}_{K^{(s+1)}}$, where $E_1 \neq E_2$. In $[\![E_j^{(s)}]\!]$, these subexpressions have been converted to $\square^{E_1}_{K^{(s+1)}}$ and $\square^{E_2}_{K^{(s+1)}}$, respectively, and their interpretations are two *independent* random bit-strings. The results of $\mathrm{CONV}(\{E_1\}_{K^{(s+1)}})$ and $\mathrm{CONV}(\{E_2\}_{K^{(s+1)}})$ are random bit-strings, too (generated at the 12th line of the algorithm CONV by invoking the random function f), but there is a small chance that they are equal, not independent. This happens iff $\mathrm{CONV}(E_1) = \mathrm{CONV}(E_2)$ (see the 8th line of the algorithm CONV). Hence it is sufficient to show that this case occurs only negligibly often.

Let $E_1, E_2 \sqsubseteq E$ such that $E_1 \neq E_2$. The proof that $\mathrm{CONV}(E_1) = \mathrm{CONV}(E_2)$ only has negligible probability, is by induction on the structure of E_1 and E_2.

Base: E_1 or E_2 is a key. if the other one is not a key, then $\mathrm{CONV}(E_1)$ and $\mathrm{CONV}(E_2)$ are different by the injectivity of τ. If both E_1 and E_2 are keys then they must be different. But in this case the probability of $\mathit{init}(E_1) = \mathit{init}(E_2)$ is negligible, otherwise the encryption system would not be secure (it would be too easy to guess the keys).

Step: neither E_1 nor E_2 is a key. If one of E_1 or E_2 is a subexpression of the other then $\mathrm{CONV}(E_1)$ and $\mathrm{CONV}(E_2)$ have different lengths and are therefore unequal. Assume that $E_1 \not\sqsubseteq E_2$ and $E_2 \not\sqsubseteq E_1$.

- If E_1 is a pair and E_2 an encryption, then $\mathrm{CONV}(E_1) = \tau(\ldots, \mathsf{pair})$ and $\mathrm{CONV}(E_2) = \tau(\ldots, \mathsf{ciphertext})$; they are thus different (because τ is injective).
- The argument is the same when E_1 is an encryption and E_2 is a pair.
- If $E_1 = (E_1', E_1'')$ and $E_2 = (E_2', E_2'')$ then either $E_1' \neq E_2'$ or $E_1'' \neq E_2''$. Assume w.l.o.g. that $E_1' \neq E_2'$. From the induction assumption follows that the probability for $\mathrm{CONV}(E_1') = \mathrm{CONV}(E_2')$ is negligible. The rest follows from the injectivity of τ.
- If $E_1 = \{E_1'\}_{K_1}$ and $E_2 = \{E_2'\}_{K_2}$, then
  - if $K_1 = K_2$ then $E_1' \neq E_2'$ and thus the probability that $\mathrm{CONV}(E_1') = \mathrm{CONV}(E_2')$ is negligible. If $K_1 \in \mathit{recoverable}(E) \cup \{K^{(1)}, \ldots, K^{(s)}\}$, then $\mathrm{CONV}(E_i)$ is computed from $\mathrm{CONV}(E_i')$ by applying an injective function to it. If $K_1 \in \{K^{(s+1)}, \ldots, K^{(l)}\}$, then $\mathrm{CONV}(E_1)$ and $\mathrm{CONV}(E_2)$ are two random numbers (tagged with $\mathsf{ciphertext}$) that are almost always independent and therefore almost always different.
  - if $K_1 \neq K_2$ then assume w.l.o.g. that K_1 does not encrypt K_2 in E. Consider all subexpressions of E_1 that have the form $\{E'\}_{K_2}$. $E_2 \not\sqsubseteq E_1$, hence $E_2' \neq E'$. From the induction assumption we derive that almost always $\mathrm{CONV}(E_2') \neq \mathrm{CONV}(E')$. Similarly, for all subexpressions of E_2' with the form $\{E'\}_{K_2}$ we almost always have $\mathrm{CONV}(E_2') \neq \mathrm{CONV}(E')$.

Suppose that $K_2 \in \mathit{recoverable}(E) \cup \{K^{(1)}, \ldots, K^{(s)}\}$. The key K_2 only occurs in E_1 and E_2 as the key in encryption operations (because of the nonexistence of encryption cycles). This means that for computing $\mathrm{CONV}(E_1)$ and $\mathrm{CONV}(E_2)$ we do not need the value of $\mathit{init}(K_2)$, it is sufficient to have an oracle that encrypts with $\mathit{init}(K_2)$. This oracle may be replaced by a random function without changing the distribution of $\mathrm{CONV}(E_1)$ and $\mathrm{CONV}(E_2)$ in a distinguishable way.

If $K_2 \in \{K^{(s+1)}, \ldots, K^{(l)}\}$, then the algorithm CONV already uses a random function for implementing the encryption with K_2.

Consider the process of computing $\mathrm{CONV}(E_1)$ and $\mathrm{CONV}(E_2)$ (in that order), where the encryption with K_2 has been implemented by a random function. The last step of the computation is invoking that random function on $\mathrm{CONV}(E'_2)$. But $\mathrm{CONV}(E'_2)$ is almost always different from anything else that has been given as the argument of this random function before. The result of this invocation is thus a random number that is almost always independent from anything else, and therefore almost always different from anything else.

$\square$

Also, if the oracle given to $\mathcal{A}^{(\cdot)}_{E_j,s}(\mathbf{1}^n)$ is $\mathcal{E}(\mathbf{1}^n, k, \cdot)$, where k is distributed according to $\mathcal{G}(\mathbf{1}^n)$, then the output of $\mathrm{CONV}(E_j)$ is distributed *identically* to $[\![E_j^{(s+1)}]\!]$. But we have now contradicted our assumption that the used encryption system was a PRP.

6.4 More Operators

The set **Exp** as defined above is not very expressive, as the only ways to combine expressions are pairing and encryption. Here we extend **Exp**, such that we can express general computations (that do not require loops) in it. We do it by introducing more constructors of expressions. Let **Op** be the set of operators, the elements of these set are the names of constructors. Each element o of **Op** has an associated arity $\mathrm{ar}(o) \in \mathbb{N}$.

We also extend the set **Exp** by introducing *inputs* to expressions, similar to the inputs to a program. Let **Inps** be the set of inputs. Formally, **Inps** is just a set. We can now extend the set of expressions by

$$
\begin{array}{rcll}
E & ::= & \ldots & \\
 & \mid & o(E_1, \ldots, E_{\mathrm{ar}(o)}) & \text{computation (for } o \in \mathbf{Op},\ E_1, \ldots, E_{\mathrm{ar}(o)} \in \mathbf{Exp}) \\
 & \mid & V & \text{input parameter } (V \in \mathbf{Inps}).
\end{array}
$$

For giving the computational interpretation we first need the semantics of operators. The semantics of an operator o is a family of polynomial-time computable functions $[\![o]\!] : \mathbf{Val}^{\mathrm{ar}(o)} \stackrel{\mathbb{N}}{\to} \mathbf{Val}$. This time, we require that the semantics of operators is *deterministic*, this is helpful for tracking the equality of interpretations of messages. In the absence of loops, a deterministic semantics is as general as a probabilistic semantics. If we would like to have an operator o to have probabilistic semantics, then we could replace o by another operator o' with arity $\mathrm{ar}(o)+1$. The algorithm computing $[\![o']\!]$ works in the same way as the algorithm computing $[\![o]\!]$, but instead of getting random bits by throwing coins, it uses the bits of its extra argument.

We will now extend the function conv. The mapping *init* must give values to keys and inputs. We therefore change the definition of $\mathbf{Init}_n$, letting it be $(\mathbf{Keys} \uplus \mathbf{Inps}) \to \mathbf{Val}_n$. As before, $\mathbf{Init} = \prod_{n\in\mathbb{N}} \mathbf{Init}_n$ and $\mathit{init} \in \mathbf{Init}$. The function conv (the tagless version, as defined in (6.1)) is extended by

$$\begin{aligned}\mathrm{conv}_n(\mathit{init}, o(E_1,\ldots,E_{\mathrm{ar}(o)}) &:= [\![o]\!]_n(\mathrm{conv}_n(\mathit{init},E_1),\ldots,\mathrm{conv}_n(\mathit{init},E_{\mathrm{ar}(o)}))\\ \mathrm{conv}_n(\mathit{init}, V) &:= \mathit{init}_n(V)\ .\end{aligned}$$

$[\![E]\!]$ is defined again as $\mathrm{conv}(\mathit{init}, E)$, where *init* is distributed according to the distribution $D^{\mathbf{Init}} \in \mathcal{D}^{\mathbb{N}}(\mathit{init})$. The distribution $D^{\mathbf{Init}}$ must be such that the values $\mathit{init}(K)$, where $K \in \mathbf{Keys}$, are distributed as keys and are independent of everything else. There are no further constraints on the distribution — different inputs to the expression do not have to be independent of each other, for example. The distribution $D^{\mathbf{Init}}$ corresponds to the distribution of inputs to a program.

As before, we are interested in sufficient conditions for the indistinguishability of interpretations of formal expressions. We could try to define an equivalence relation on formal messages, similar to the previous section, but this equivalence would no longer be sufficient for the indistinguishability of interpretations. The lemma 6.2 would no longer hold as the algorithm CONV on Fig. 6.1 no longer necessarily maps different expressions to different bit-strings.

6.5 Analysis

Our approach is to define a suitable formal language **PC** of *claims* to express the properties of the interpretation $[\![\cdot]\!]$. One of the properties that this language can express is, that the interpretations of two expressions are indistinguishable.

We will define, when a claim $C \in \mathbf{PC}$ holds for the interpretation $[\![\cdot]\!]$. For example, the claim "expressions E_1 and E_2 have indistinguishable interpretations" holds iff $[\![E_1]\!] \approx [\![E_2]\!]$. We denote "claim C holds for the interpretation $[\![\cdot]\!]$" by $[\![\cdot]\!] \models C$.

We also give a number of "rules" in the form of $C_1,\ldots,C_k \Rightarrow C_{k+1}$, where $C_1,\ldots,C_{k+1} \in \mathbf{PC}$. For each of these rules we prove that if $[\![\cdot]\!] \models C_i$ for all $i \in \{1,\ldots,k\}$, then also $[\![\cdot]\!] \models C_{k+1}$.

We can use the following method to determine whether $[\![E_1]\!] \approx [\![E_2]\!]$:

- Start with a set $\mathcal{C}_0 \subset \mathbf{PC}$ that describes the properties of interpretations of formal keys and input parameters. The set $\mathcal{C}_0$ corresponds to the abstraction of initial program state in chapter 4. Let $\mathcal{C} := \mathcal{C}_0$.
- Using given rules derive from the elements of $\mathcal{C}$ new claims that also hold for interpretations of formal expressions. Add them to $\mathcal{C}$.
- Try to derive the claim that has the meaning $[\![E_1]\!] \approx [\![E_2]\!]$.

6.5.1 The Language of Claims

The language **PC** is given by the following formal grammar:

$$\begin{array}{rcl} C & ::= & \mathsf{Uniform}(E) \\ & | & \mathsf{Constant}(E) \\ & | & \mathsf{Random}(E) \\ & | & \mathsf{Uneq}(E_1, E_2) \\ & | & \mathsf{Indep}(\{E_1, \ldots, E_k\}, \{E'_1, \ldots, E'_l\}) \\ & | & \mathsf{SameDist}(\{(E_1, E'_1), \ldots, (E_k, E'_k)\}) \end{array}$$

that intuitively have the following meaning:

- $\mathsf{Uniform}(E)$ means that the interpretation of the formal expression E is indistinguishable from the uniform distribution of bit-strings of the same length.
- $\mathsf{Constant}(E)$ means that the interpretation of E is a distribution that puts all its weight onto a single bit-string.
- $\mathsf{Random}(E)$ means that the interpretation of E is a distribution that only lays negligible weight onto each bit-string.
- $\mathsf{Uneq}(E_1, E_2)$ means that when we choose *init* once (according to the initial probability distribution) and compute $\mathrm{conv}(\mathit{init}, E_1)$ and $\mathrm{conv}(\mathit{init}, E_2)$, then the probability that they are equal is negligible.
- $\mathsf{Indep}(\{E_1, \ldots, E_k\}, \{E'_1, \ldots, E'_l\})$ denotes the independence of sets of expressions, similarly to Sec. 4.1.
- $\mathsf{SameDist}(\{(E_1, E'_1), \ldots, (E_k, E'_k)\})$ means that the expressions $(E_1, \ldots, E_k)$ and $(E'_1, \ldots, E'_k)$ have indistinguishable interpretations. We could have defined that the argument of $\mathsf{SameDist}$ is a pair of expressions, not a set of pairs of expressions, but the definition that we have given seems to be more intuitive.

Also, there are restrictions on encryption cycles: in the following we only handle such elements C of **PC**, where the restrictions given in Table 6.1 hold.

The meaning of claims is formalised by defining the relation $[\![\cdot]\!] \vDash C$ for $C \in \mathbf{PC}$. The definition is given in Fig. 6.2. This definition also explains the restrictions on

$C \in \mathbf{PC}$ is	no encryption cycles in
$\mathsf{Uniform}(E)$	E
$\mathsf{Constant}(E)$	E
$\mathsf{Random}(E)$	E
$\mathsf{Uneq}(E_1, E_2)$	(E_1, E_2)
$\mathsf{Indep}(\{E_1, \ldots, E_k\}, \{E'_1, \ldots, E'_l\})$	$(E_1, \ldots, E_k, E'_1, \ldots, E'_l)$
$\mathsf{SameDist}(\{(E_1, E'_1), \ldots, (E_k, E'_k)\})$	$(E_1, \ldots, E_k)$ and $(E'_1, \ldots, E'_k)$

Table 6.1: Prohibited encryption cycles for claims

$[\![\cdot]\!] \models \mathsf{Uniform}(E)$ iff

$$[\![E]\!] \approx \mathcal{U}(\{\mathbf{0},\mathbf{1}\}^{|[\![E]\!]|}) \ .$$

$[\![\cdot]\!] \models \mathsf{Constant}(E)$ iff there exists a polynomial-time algorithm $\mathcal{A}$, such that

$$\Pr\big[x \neq y \ : \ x \leftarrow [\![E]\!]_n, y \leftarrow \mathcal{A}(\mathbf{1}^n)\big]$$

is negligible.
$[\![\cdot]\!] \models \mathsf{Random}(E)$ iff for all PPT algorithms $\mathcal{A}$ the probability

$$\Pr\big[x = y \ : \ x \leftarrow [\![E]\!]_n, y \leftarrow \mathcal{A}(\mathbf{1}^n)\big]$$

is negligible.
$[\![\cdot]\!] \models \mathsf{Uneq}(E_1, E_2)$ iff the probability

$$\Pr\big[x = y \ : \ \tau(x, y) \leftarrow [\![(E_1, E_2)]\!]_n\big]$$

is negligible.
$[\![\cdot]\!] \models \mathsf{Indep}(\{E_1, \ldots, E_k\}, \{E'_1, \ldots, E'_l\})$ iff

$$[\![((E_1, \ldots, E_k), (E'_1, \ldots, E'_l))]\!] \approx \tau([\![(E_1, \ldots, E_k)]\!], [\![(E'_1, \ldots, E'_l)]\!]) \ .$$

$[\![\cdot]\!] \models \mathsf{SameDist}(\{(E_1, E'_1), \ldots, (E_k, E'_k)\})$ iff

$$[\![(E_1, \ldots, E_k)]\!] \approx [\![(E'_1, \ldots, E'_k)]\!] \ .$$

Figure 6.2: The relation $\models$

encryption cycles in formal expressions — expressions E, for which $[\![E]\!]$ is computed, must have no encryption cycles.

We are going to state a number of rules in the form of implications between different claims. These implications are correct for every possible interpretation. We prove each rule immediately after stating it.

6.5.2 General Rules

Here we give the rules whose correctness does not depend on any extra assumptions about the properties of the semantics of operators.

Analysis Rule G.1. *If* $K_1, K_2 \in \mathbf{Keys}$ *then* $\mathsf{SameDist}(\{(K_1, K_2)\})$.

Proof. By the distribution of *init*. □

Analysis Rule G.2. *If* $K_1, K_2 \in \mathbf{Keys}$ *and* $K_1 \neq K_2$ *then* $\mathsf{Indep}(\{K_1\}, \{K_2\})$.

Proof. By the distribution of *init*. □

Analysis Rule G.3. *The predicates* $\mathsf{SameDist}$*,* Indep *and* Uneq *are all symmetric. Also, the predicate* Indep *is monotone — if* $\mathsf{Indep}(X, Y)$ *holds and* $X' \subseteq X$ *and* $Y' \subseteq Y$*, then also* $\mathsf{Indep}(X', Y')$ *holds. The predicate* $\mathsf{SameDist}$ *is monotone, too — if* $X \subseteq \mathbf{Exp} \times \mathbf{Exp}$*,* $X' \subseteq X$ *and* $\mathsf{SameDist}(X)$ *holds, then* $\mathsf{SameDist}(X')$ *holds, too.*

Proof. Symmetry follows directly from the definition of $\models$. The monotonicity of Indep has been explained in Sec. 4.1. The monotonicity of $\mathsf{SameDist}$ is similar, if

$$[\![(E_1, \ldots, E_k)]\!] \approx [\![(E'_1, \ldots, E'_k)]\!]$$

and $1 \leq i_1 < i_2 < \cdots < i_l \leq k$, then also

$$[\![(E_{i_1}, \ldots, E_{i_l})]\!] \approx [\![(E'_{i_1}, \ldots, E'_{i_l})]\!] \enspace .$$

Indeed, an algorithm that could distinguish the latter two distributions could also distinguish the former two. □

Analysis Rule G.4. *For all* $X \subseteq \mathbf{Exp}$ *the claim* $\mathsf{Indep}(\emptyset, X)$ *holds.*

Proof. Follows directly from the definition of $\models$. □

Analysis Rule G.5. *For all* $E_1, \ldots, E_k \in \mathbf{Exp}$ *the claim* $\mathsf{SameDist}(\{(E_1, E_1), \ldots, (E_k, E_k)\})$ *holds.*

Proof. Follows directly from the definition of $\models$. □

Analysis Rule G.6. *If* $E, E' \in \mathbf{Exp}$*, where* $|[\![E]\!]| = |[\![E']\!]|$ *(recall that the equality of lengths of interpretations of expressions should be checkable directly from these expressions) and the claims* $\mathsf{Uniform}(E)$ *and* $\mathsf{Uniform}(E')$ *hold, then the claim* $\mathsf{SameDist}(\{(E, E')\})$ *also holds.*

Proof. Follows directly from the definition of $\models$. $\square$

Analysis Rule G.7. *Let $X, Y \subset \mathbf{Exp}$ and let $E_1, \ldots, E_k \in X$. If the claim $\mathsf{Indep}(X, Y)$ holds, then the claim $\mathsf{Indep}(X \cup \{o(E_1, \ldots, E_k)\}, Y)$ also holds, where $o \in \mathbf{Op}$ is any k-ary operator.*

Proof. This is a restatement of rule (4.10).

Indeed, if we have an algorithm $\mathcal{A}$ that can distinguish the distributions

$$[\![((\langle E\rangle_{E\in X}, o(E_1, \ldots, E_k)), (\langle E\rangle_{E\in Y}))]\!]$$

and

$$\tau([\![(\langle E\rangle_{E\in X}, o(E_1, \ldots, E_k))]\!], [\![(\langle E\rangle_{E\in Y})]\!]),$$

then an algorithm that first computes $o(E_1, \ldots, E_k)$ and then invokes $\mathcal{A}$ can distinguish the distributions

$$[\![((\langle E\rangle_{E\in X}), (\langle E\rangle_{E\in Y}))]\!]$$

and

$$\tau([\![(\langle E\rangle_{E\in X})]\!], [\![(\langle E\rangle_{E\in Y})]\!]) \enspace .$$

We have shown that $[\![\cdot]\!] \not\models \mathsf{Indep}(X \cup \{o(E_1, \ldots, E_k)\}, Y)$ implies $[\![\cdot]\!] \not\models \mathsf{Indep}(X, Y)$. $\square$

Analysis Rule G.8. *Let $E_1, \ldots, E_k, E'_1, \ldots, E'_k \in \mathbf{Exp}$. Let o be an l-ary operator and let $i_1, \ldots, i_l \in \{1, \ldots, k\}$. If the claim*

$$\mathsf{SameDist}(\{(E_1, E'_1), \ldots, (E_k, E'_k)\}) \tag{6.3}$$

holds, then the claim

$$\mathsf{SameDist}(\{(E_1, E'_1), \ldots, (E_k, E'_k), (o(E_{i_1}, \ldots, E_{i_l}), o(E'_{i_1}, \ldots, E'_{i_l}))\}) \tag{6.4}$$

also holds.

Proof. This rule is similar to the previous rule. If we have an algorithm $\mathcal{A}$ that can distinguish the two distributions associated with claim (6.4) (see Fig. 6.2), then the algorithm that first computes the value of the operation o on expressions in the i_1-th,...,i_l-th position in the tuple and then invokes $\mathcal{A}$ can distinguish the two distributions associated with claim (6.3). $\square$

Analysis Rule G.9. *Let $E \in \mathbf{Exp}$ and $X, Y \subseteq \mathbf{Exp}$. If the claims $\mathsf{Constant}(E)$ and $\mathsf{Indep}(X, Y)$ hold, then the claim $\mathsf{Indep}(X \cup \{E\}, Y \cup \{E\})$ also holds.*

Proof. This is a restatement of rule (4.13). The interpretation of the expression E puts almost all of its weight onto a single value, therefore it does not matter, whether $[\![E]\!]$ has been sampled once or twice. $\square$

Analysis Rule G.10. *Let $K \in \mathbf{Keys}$. The claim $\mathsf{Random}(K)$ holds.*

Proof. By the distribution of *init*. □

Analysis Rule G.11. *Let* $E \in \mathbf{Exp}$. *If the claim* $\mathsf{Uniform}(E)$ *holds, then the claim* $\mathsf{Random}(E)$ *also holds.*

Proof. The distribution $[\![E]\!]$ puts only negligible weight onto each bit-string of length $|[\![E]\!]|$, otherwise it would be distinguishable from an uniform distribution. I.e. $\mathsf{Random}(E)$ holds. □

Analysis Rule G.12. *Let* $E, E' \in \mathbf{Exp}$. *If the claims* $\mathsf{Random}(E)$ *and* $\mathsf{Indep}(\{E\}, \{E'\})$ *hold, then the claim* $\mathsf{Uneq}(E, E')$ *also holds.*

Proof. We have to show that $\Pr\big[x = y \,:\, \tau(x, y) \leftarrow [\![(E, E')]\!]_n\big]$ is negligible in n. By the independence of E and E' we have

$$\Pr\big[x = y \,:\, \tau(x, y) \leftarrow [\![(E, E')]\!]_n\big] = \Pr\big[x = y \,:\, x \leftarrow [\![E]\!]_n, y \leftarrow [\![E']\!]_n\big] + \alpha(n),$$

where α is some negligible function. But, by the definition of $\vDash$, the probability $\Pr\big[x = y \,:\, x \leftarrow [\![E]\!]_n, y \leftarrow [\![E']\!]_n\big]$ is negligible as well, because E is random and $[\![E']\!]$ is polynomial-time constructible. □

Analysis Rule G.13. *Let* $E_1, \ldots, E_k, E'_1, \ldots, E'_k, F_1, \ldots, F_l, F'_1, \ldots, F'_l \in \mathbf{Exp}$. *Then the following implication between claims holds:*

$$\begin{aligned}
&\mathsf{Indep}(\{E_1, \ldots, E_k\}, \{F_1, \ldots, F_l\}) \wedge \mathsf{SameDist}(\{(E_1, E'_1), \ldots, (E_k, E'_k)\}) \wedge \\
&\mathsf{SameDist}(\{(F_1, F'_1), \ldots, (F_l, F'_l)\}) \wedge \mathsf{Indep}(\{E'_1, \ldots, E'_k\}, \{F'_1, \ldots, F'_l\}) \Rightarrow \\
&\qquad\qquad \mathsf{SameDist}(\{(E_1, E'_1), \ldots, (E_k, E'_k), (F_1, F'_1), \ldots, (F_l, F'_l)\}) \ .
\end{aligned}$$

This rule is used to derive bigger claims $\mathsf{SameDist}(\ldots)$ from smaller ones.

Proof. We have to show that the two distributions in Fig. 6.2, associated with the claim on the right hand side of the implication, are indistinguishable. We have

$$\begin{aligned}
&[\![((E_1, \ldots, E_k), (F_1, \ldots, F_l))]\!] \approx^{\mathsf{Indep}(\{E_1, \ldots, E_k\}, \{F_1, \ldots, F_l\})} \\
&\qquad \tau([\![(E_1, \ldots, E_k)]\!], [\![(F_1, \ldots, F_l)]\!]) \approx^{\mathsf{SameDist}(\{(E_1, E'_1), \ldots, (E_k, E'_k)\})} \\
&\qquad \tau([\![(E'_1, \ldots, E'_k)]\!], [\![(F_1, \ldots, F_l)]\!]) \approx^{\mathsf{SameDist}(\{(F_1, F'_1), \ldots, (F_l, F'_l)\})} \\
&\qquad \tau([\![(E'_1, \ldots, E'_k)]\!], [\![(F'_1, \ldots, F'_l)]\!]) \approx^{\mathsf{Indep}(\{E'_1, \ldots, E'_k\}, \{F'_1, \ldots, F'_l\})} \\
&\qquad\qquad [\![((E'_1, \ldots, E'_k), (F'_1, \ldots, F'_l))]\!],
\end{aligned}$$

where at each $\approx$ we have marked the claim that gives the indistinguishability of these two distributions. □

6.5.3 Rules for Encryption

Let $\mathsf{NewEnc}(K, E, E')$, where $K \in \mathbf{Keys}$ and $E, E' \in \mathbf{Exp}$, be synonymous to the following statements about the structure of E' and the claims holding for the subexpressions of E, E':

- $K \not\sqsubseteq E'$
- For all such $E'' \in \mathbf{Exp}$ that $\{E''\}_K \sqsubseteq E'$ the claim $\mathsf{Uneq}(E, E'')$ holds.

We use the assumptions $\mathsf{NewEnc}(K, E, E')$ when we want to introduce a claim that includes $\{E\}_K$. From the condition $K \not\sqsubseteq E'$ follows that for computing $[\![E']\!]$ we do not need the value of the key K, we may only need an oracle that encrypts with (the value of) K. Same holds for computing $[\![E]\!]$, because otherwise we would have an encryption cycle. In the following, we may replace the oracle encrypting with K by the oracle $\mathcal{PRF}$.

Consider now computing the interpretation of E', followed by computing the interpretation of $\{E\}_K$. Because of the claims $\mathsf{Uneq}(E, E'')$ for every subexpression E'' of E' that is encrypted with K, when we submit the value of E to the oracle to obtain the value of $\{E\}_K$, then we are submitting to it a value that is different from all values that may have been submitted to the oracle while computing the value of E'. Therefore the value of $\{E\}_K$ will look just like a random bit-string to an observer that only has the value of E'.

Analysis Rule E.1. *Let $E \in \mathbf{Exp}$ and $K \in \mathbf{Keys}$. If $K \not\sqsubseteq E$, then the claim* $\mathsf{Random}(\{E\}_K)$ *holds.*

Proof. We do not need the value of K for computing the value of $\{E\}_K$, we only need an oracle that encrypts with its value. Therefore $\mathrm{conv}_n(\mathit{init}, \{E\}_K)$ is a value returned by the oracle $\mathcal{PRF}$, therefore it is a random bit-string. □

Analysis Rule E.2. *Let $E \in \mathbf{Exp}$ and $K \in \mathbf{Keys}$. If all the claims given by* $\mathsf{NewEnc}(K, E, E)$ *hold, then the claim* $\mathsf{Uniform}(\{E\}_K)$ *also holds.*

Proof. Additionally to the previous rule, $\mathrm{conv}_n(\mathit{init}, \{E\}_K)$ is a value returned by $\mathcal{PRF}$ for a query that has not been made before while computing $\mathrm{conv}_n(\mathit{init}, E)$. Therefore it is uniformly distributed. □

Analysis Rule E.3. *Let $E, E' \in \mathbf{Exp}$ and $K \in \mathbf{Keys}$. If all the claims given by* $\mathsf{NewEnc}(K, E, E')$ *hold, then the claim* $\mathsf{Uneq}(\{E\}_K, E')$ *also holds.*

Proof. Follows from the arguments made to describe NewEnc. The value of $\{E\}_K$ is a random number that has not been used when computing the value of E'. □

Analysis Rule E.4. *Let $E \in \mathbf{Exp}$, $X, Y \subseteq \mathbf{Exp}$ and $K \in \mathbf{Keys}$. If the claim* $\mathsf{Indep}(X, Y)$ *holds and if for all $E' \in X \cup Y \cup \{E\}$ the claims given by* $\mathsf{NewEnc}(K, E, E')$ *hold, then the claim* $\mathsf{Indep}(X \cup \{\{E\}_K\}, Y)$ *also holds.*

Proof. Consider the procedure that computes the values of all expressions in $X \cup Y$, then computes the value of the expression E and as the last step encrypts it under K. This procedure does not need the key K, it only needs an oracle encrypting with K; we may assume this oracle is $\mathcal{PRF}$. When the procedure submits the value of E to the oracle for encryption, then the same value has not been submitted to the oracle before (this follows from all the claims $\mathsf{NewEnc}(K, E, E')$). Therefore the value returned by the oracle is a newly generated random number, it is not dependent of anything. Therefore $\mathsf{Indep}(\{\{E\}_K\}, X \cup Y)$ holds. By Lemma 4.1, the claim $\mathsf{Indep}(X \cup \{\{E\}_K\}, Y)$ also holds. □

6.5.4 Rules for Group Operations

In the following we give some rules for dealing with binary operators $\oplus \in \mathbf{Op}$ whose semantics is such, that for each $n \in \mathbb{N}$, $(\{\mathbf{0}, \mathbf{1}\}^*, [\![\oplus]\!]_n)$ is a group. The operator $\oplus$ might denote exclusive or, or addition modulo some suitable power of 2.

When constructing formal expressions with $\oplus$, we use infix notation. Thus $E_1 \oplus E_2$ really means $\oplus(E_1, E_2)$.

Being a group operation is even a too strong requirement for $\oplus$. For the following rules to hold, it is sufficient when for each $n \in \mathbb{N}$ and $y \in \{\mathbf{0}, \mathbf{1}\}^*$ the mapping $x \mapsto x \; [\![\oplus]\!]_n \; y$ is a bijective function on $\{\mathbf{0}, \mathbf{1}\}^*$.

If $[\![\oplus]\!]$ is commutative, then we can swap the arguments of $\oplus$ in expressions in the statements of the following rules.

Analysis Rule X.1. *Let $E, E_1, E_2 \in \mathbf{Exp}$. If the claim $\mathsf{Uneq}(E_1, E_2)$ holds, then the claim $\mathsf{Uneq}(E_1 \oplus E, E_2 \oplus E)$ also holds.*

Proof. Follows directly from the semantics of $\oplus$. □

Analysis Rule X.2. *Let $E, E' \in \mathbf{Exp}$. If the claims $\mathsf{Uniform}(E)$ and $\mathsf{Indep}(\{E\}, \{E'\})$ hold, then the claim $\mathsf{Uniform}(E \oplus E')$ also holds.*

Proof. By the claim $\mathsf{Indep}(\{E\}, \{E'\})$, the distribution $[\![E \oplus E']\!]$ is indistinguishable from the distribution

$$\{\!| x \; [\![\oplus]\!]_n \; y \; : \; x \leftarrow [\![E]\!]_n, y \leftarrow [\![E']\!]_n |\!\}_{n \in \mathbb{N}} \; .$$

In this distribution, for each possible value of y, the value of x is uniformly distributed and therefore the value of $x \; [\![\oplus]\!]_n \; y$ is uniformly distributed as well. □

Analysis Rule X.3. *Let $E, E' \in \mathbf{Exp}$ and $X, Y \subseteq \mathbf{Exp}$. If the claims*

- $\mathsf{Uniform}(E)$
- $\mathsf{Indep}(X, Y)$
- $\mathsf{Indep}(\{E\}, X \cup Y \cup \{E'\})$

hold, then the claim $\mathsf{Indep}(X \cup \{E \oplus E'\}, Y)$ holds as well.

Proof. We are going to show that the claim $\mathsf{Indep}(\{E \oplus E'\}, X \cup Y)$ holds. The correctness of this rule follows then from Lemma 4.1.

By the claim $\mathsf{Indep}(\{E\}, X \cup Y \cup \{E'\})$, for each fixed value of the expression E' and the expressions $E'' \in X \cup Y$, the value of the expression E is still uniformly distributed. Therefore the value of the expression $E \oplus E'$ is still uniformly distributed for each fixed value of the expressions $E'' \in X \cup Y$, i.e. the value of $E \oplus E'$ does not depend on the values of the expressions $E'' \in X \cup Y$. □

6.5.5 Special Rules

For completeness of the presentation, we state some rules for other operators that we have mentioned before, namely pairing and the choice operator $\mathsf{?:}$. The correctness of these rules depends again on the properties of the semantics of these operators.

Analysis Rule P.1. *Let $E_1, E_2 \in \mathbf{Exp}$. If the claim $\mathsf{Uneq}(E_1, E_2)$ holds, then the claim $\mathsf{Uneq}((E_1, E_1'), (E_2, E_2'))$ also holds for all $E_1', E_2' \in \mathbf{Exp}$. Similar rule holds for the right component.*

Proof. Follows directly from the semantics of the pairing constructor. □

The semantics of the ternary choice operator $\mathsf{?:}$ is assumed to be the following:

$$[\![\mathsf{?:}]\!]_n(b, x, y) = \begin{cases} x, & \text{if } b \text{ denotes the value } \mathsf{true} \\ y, & \text{if } b \text{ denotes the value } \mathsf{false} \end{cases}$$

for each $b, x, y \in \mathbf{Val}_n$. When constructing formal expressions with $\mathsf{?:}$, we use infix notation. Thus $E_b \,\mathsf{?}\, E_x : E_y$ really means $\mathsf{?:}(E_b, E_x, E_y)$.

Analysis Rule C.1. *Let $X, Y \subseteq \mathbf{Exp}$ and let $E_b, E_x, E_y \in \mathbf{Exp}$. If the claims*

- $\mathsf{Indep}(X \cup \{E_b, E_x\}, Y)$
- $\mathsf{Indep}(X \cup \{E_b, E_y\}, Y)$

hold, then the claim $\mathsf{Indep}(X \cup \{E_b, E_b \,\mathsf{?}\, E_x : E_y\}, Y)$ also holds.

Proof. This is actually a special case of rule (4.29). Let $X = \{E_1, \ldots, E_k\}$ and $Y = \{F_1, \ldots, F_l\}$. We have to show that distributions

$$[\![((E_1, \ldots, E_k, E_b, E_b \,\mathsf{?}\, E_x : E_y), (F_1, \ldots, F_l))]\!] \tag{6.5}$$

and

$$\tau([\![(E_1, \ldots, E_k, E_b, E_b \,\mathsf{?}\, E_x : E_y)]\!], [\![(F_1, \ldots, F_l)]\!]) \tag{6.6}$$

are indistinguishable. Assume that they are not indistinguishable, i.e. there exists an algorithm $\mathcal{A}$ that distinguishes them. We will now construct an algorithm $\mathcal{A}_x$ that attempts to distinguish the distributions

$$[\![((E_1, \ldots, E_k, E_b, E_x), (F_1, \ldots, F_l))]\!] \tag{6.7}$$

and

$$\tau(\llbracket(E_1, \ldots, E_k, E_b, E_x)\rrbracket, \llbracket(F_1, \ldots, F_l)\rrbracket), \tag{6.8}$$

i.e. it attempts to show that the antecedent $\mathsf{Indep}(X \cup \{E_b, E_x\}, Y)$ does not hold.

Given a sample of either (6.7) or (6.8), the algorithm $\mathcal{A}_x$ first checks the value of E_b. If it is true, then $\mathcal{A}_x$ calls $\mathcal{A}$ with the same inputs and returns, whatever $\mathcal{A}$ returns. If the value of E_b is false, then $\mathcal{A}_x$ returns "failure".

Similarly, we construct an algorithm $\mathcal{A}_y$ attempting to distinguish

$$\llbracket((E_1, \ldots, E_k, E_b, E_y), (F_1, \ldots, F_l))\rrbracket$$

and

$$\tau(\llbracket(E_1, \ldots, E_k, E_b, E_y)\rrbracket, \llbracket(F_1, \ldots, F_l)\rrbracket),$$

i.e. to show that the antecedent $\mathsf{Indep}(X \cup \{E_b, E_y\}, Y)$ does not hold. Similarly to $\mathcal{A}_x$, the algorithm $\mathcal{A}_y$ first checks the value of E_b. If the value is true, it returns "failure". If the value if false, it calls the algorithm $\mathcal{A}$.

The sum of the advantages of $\mathcal{A}_x$ and $\mathcal{A}_y$ is equal to the advantage of $\mathcal{A}$. Therefore at least one of these advantages is non-negligible and at least one of the antecedents of the rule does not hold. $\square$

Our language of expressions **Exp** does not contain expressions of the form $\{E\}_{B\,?\,K\,:\,K'}$, where $E, B \in$ **Exp** and $K, K' \in$ **Keys**. We only allow to encrypt with keys, not with arbitrary "key-valued" expressions, although we could also define the computational interpretation for such kind of formal expressions. However, such an expression could be transformed to $B\,?\{E\}_K : \{E\}_{K'}$; this expression has the same computational interpretation and it is a member of **Exp**. One could say that this transformation has only restricted utility because it can lead to exponential growth of the size of the formal expression. However, this depends on the definition of "size". The size of a formal expression E should reflect the amount of memory necessary to store it; this amount obviously depends on the encoding. If we define the size of a formal expression simply as the length of the string that represents it, then the described transformation indeed changes the expression

$$\{\cdots\{\{E\}_{B_1\,?\,K_1\,:\,K'_1}\}_{B_2\,?\,K_2\,:\,K'_2}\cdots\}_{B_m\,?\,K_m\,:\,K'_m} \tag{6.9}$$

(with size proportional to m) to an expression with size proportional to 2^m. However, if the common subexpressions are only stored once, then the expression (6.9) is transformed to the expression shown on Fig. 6.3 that still has size proportional to m.

6.6 Examples

In this section we show that some rather interesting/relevant things can be made with the analysis given in the previous section. Namely, we derive from our analysis

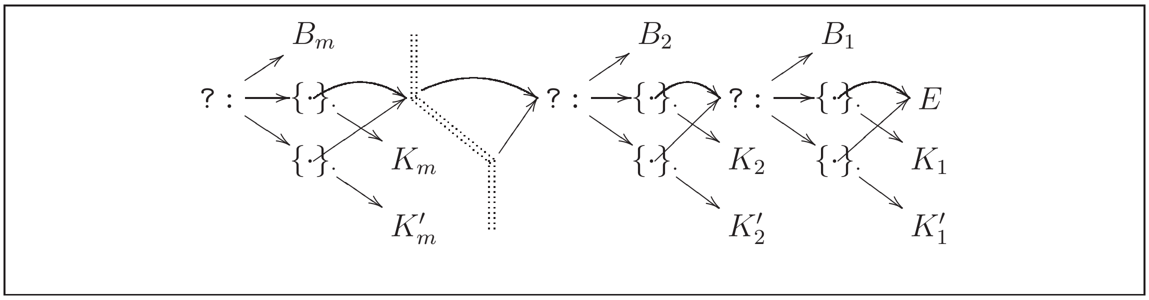


Figure 6.3: Storing the transformed expression

that certain block-cipher's modes of operation satisfy quite strong security properties. In Sec. 6.6.1 we give the specification of two modes of operation, namely Cipher Block Chaining (CBC) mode and Counter (CTR) mode *alias* XOR-mode. In Sec. 6.6.2 we give the derivation for the CBC-mode and in Sec. 6.6.3 for the CTR-mode.

6.6.1 Block-Ciphers' Modes of Operation

A block cipher is an encryption system, where the plaintext must have a certain, fixed length. Let $q(n)$ denote the length of the plaintext (and also the length of the ciphertext) for the security parameter n, we may assume $q \in \mathbf{Pol}(\mathbb{Z})$. If one wants to encrypt a bit-string x of arbitrary length (however, we may assume that $|x|$ is a multiple of $q(n)$, by suitably padding the bit-string x), then one first divides x to $q(n)$-bit blocks $x_1, \ldots, x_m$ and then combines these blocks and the encryption operation according to the chosen mode of operation.

Therefore, let $x_1 x_2 \cdots x_m$, where $x_1, \ldots, x_m \in \{\mathbf{0}, \mathbf{1}\}^{q(n)}$, be the plaintext and let K be the key.

CBC-mode. The ciphertext is $y_0 y_1 y_2 \cdots y_m$, where $y_0, \ldots, y_m \in \{\mathbf{0}, \mathbf{1}\}^{q(n)}$ are created as follows:

- y_0 picked according to the uniform distribution over $\{\mathbf{0}, \mathbf{1}\}^{q(n)}$;
- $y_i = \{x_i \oplus y_{i-1}\}_K$ for all $i \in \{1, \ldots, m\}$. Here $\oplus$ denotes bitwise exclusive or.

CTR-mode. The ciphertext is $y_0 y_1 y_2 \cdots y_m$, where $y_0, \ldots, y_m \in \{\mathbf{0}, \mathbf{1}\}^{q(n)}$ are created as follows:

- y_0 picked according to the uniform distribution over $\{\mathbf{0}, \mathbf{1}\}^{q(n)}$;
- $y_i = \{y_0 + i\}_K \oplus x_i$ for all $i \in \{1, \ldots, m\}$. Here $+$ denotes addition modulo $2^{q(n)}$ and $\oplus$ denotes bitwise exclusive or.

For both modes we show

$$\mathsf{SameDist}(\{(x_1, x_1), \ldots, (x_m, x_m), (y_0, r_0), \ldots, (y_m, r_m)\}),$$

name	claim
C^i	$\mathsf{SameDist}(\{(x_1,x_1),\ldots,(x_m,x_m),(y_0,r_0),\ldots,(y_i,r_i)\})$
D_1^i	$\mathsf{SameDist}(\{(y_i,r_i)\})$
D_2^i	$\mathsf{Indep}(\{x_1,\ldots,x_m,y_0,\ldots,y_{i-1}\},\{y_i\})$
D_3^i	$\mathsf{Indep}(\{x_1,\ldots,x_m,r_0,\ldots,r_{i-1}\},\{r_i\})$
D_4^i	$\mathsf{Uniform}(y_i)$
$\mathrm{D}_5^{i,j}$	$\mathsf{Uneq}(x_i\oplus y_{i-1},x_j\oplus y_{j-1})$
D_6^i	$\mathsf{Uniform}(x_i\oplus y_{i-1})$
$\tilde{\mathrm{D}}_6^i$	$\mathsf{Random}(x_i\oplus y_{i-1})$
$\mathrm{D}_7^{i,j}$	$\mathsf{Indep}(\{x_i\oplus y_{i-1}\},\{x_j\oplus y_{j-1}\})$
$\mathrm{D}_8^{i,j}$	$\mathsf{Indep}(\{y_{i-1}\},\{x_i,x_j\oplus y_{j-1}\})$
D_9^i	$\mathsf{Indep}(\{x_1,\ldots,x_m,y_0,\ldots,y_{i-1}\},\emptyset)$

C^i, D_1^i, D_2^i, D_3^i and D_4^i are also defined for $i=0$.

Table 6.2: Claims for showing the security of the CBC-mode

where (the interpretations of) $r_0,\ldots,r_m$ are uniformly distributed, mutually independent bit-strings of length $q(n)$. This is already a rather strong secrecy property. Abadi and Rogaway [AR00, Sec. 4.4] claim that this is sufficient for being a which-key and repetition concealing encryption.

Here the set **Inps** of inputs consists of $x_1,\ldots,x_m,y_0,r_0,\ldots,r_m$. The following claims hold about their distribution (i.e. the set $\mathcal{C}_0$ has the following elements):

S-1. $\mathsf{Uniform}(y_0)$, $\mathsf{Uniform}(r_i)$ for all $i\in\{0,\ldots,m\}$

S-2. $\mathsf{Indep}(X,Y)$ for all $X\subseteq\{y_0,r_0,\ldots,r_m\}$ and $Y\subseteq\{x_1,\ldots,x_m,y_0,r_0,\ldots,r_m\}$, such that $X\cap Y=\emptyset$.

6.6.2 Security of the CBC-Mode

For each $i\in\{1,\ldots,m\}$ and each $j\in\{1,\ldots,i-1\}$, table 6.2 gives certain names to certain claims. We thus have to derive C^m.

It is easy to derive C^0, D_1^0, D_2^0, D_3^0 and D_4^0. Indeed,

1. D_4^0 is given in S-1;
2. the rule G.5 gives $\mathsf{SameDist}(\{(x_1,x_1),\ldots,(x_m,x_m)\})$;
3. S-1 and the rule G.6 give D_1^0;
4. S-2 gives D_2^0, by taking $X=\{y_0\}$ and $Y=\{x_1,\ldots,x_m\}$;
5. S-2 also gives D_3^0, by taking $X=\{r_0\}$ and $Y=\{x_1,\ldots,x_m\}$;
6. the rule G.13, applied to $\mathsf{SameDist}(\{(x_1,x_1),\ldots,(x_m,x_m)\})$ and D_1^0, D_2^0, D_3^0, gives C^0.

The rest of the derivation proceeds in m stages, where the i-th stage (depicted in Fig. 6.4; $1 \le i \le m$) culminates with deriving C^i:

1. D_9^i is given by the rule G.4;
2. for each $j \in \{1, \dots, i-1\}$, $\mathrm{D}_8^{i,j}$ is derived from D_2^{i-1} by applying the rules G.7 and G.3;
3. for each $j \in \{1, \dots, i-1\}$, $\mathrm{D}_7^{i,j}$ is derived from D_4^{i-1} and $\mathrm{D}_8^{i,j}$ by applying the rule X.3, by taking $X = \emptyset$, $E = y_{i-1}$, $E' = x_i$ and $Y = \{x_j \oplus y_{j-1}\}$;
4. D_6^i is derived from D_4^{i-1} and D_2^{i-1} (after applying the rule G.3) by applying the rule X.2;
5. $\tilde{\mathrm{D}}_6^i$ follows from D_6^i and the rule G.11;
6. for each $j \in \{1, \dots, i-1\}$, $\mathrm{D}_5^{i,j}$ is derived from $\tilde{\mathrm{D}}_6^i$ and $\mathrm{D}_7^{i,j}$ by applying the rule G.12;
7. D_4^i follows from the rule E.2; the claims

   $$\mathsf{Uneq}(x_i \oplus y_{i-1}, x_j \oplus y_{j-1}), \text{ where } 1 \le j \le i-1$$

   must hold for applying this rule, but these are exactly the claims $\mathrm{D}_5^{i,j}$;
8. D_3^i is given by S-2, by taking $X = \{r_i\}$ and $Y = \{x_1, \dots, x_m, r_0, \dots, r_{i-1}\}$;
9. D_2^i follows from the claims D_9^i and $\mathrm{D}_5^{i,j}$ $(1 \le j < i)$ by applying the rule E.4 (the instances of NewEnc expand exactly to the claims $\mathrm{D}_5^{i,j}$);
10. D_1^i is derived from D_4^i and S-1 by applying the rule G.6;
11. C^i is derived from D_2^i, C^{i-1}, D_1^i and D_3^i by applying the rule G.13.

6.6.3 Security of the CTR-Mode

Additionally, the set **Inps** of inputs also includes $1, 2, \dots, m$ whose computational interpretations are defined to be the representations of natural numbers $1, 2, \dots, m$. The following claims additionally hold about their distribution:

S-3. $\mathsf{Constant}(i)$ for $1 \le i \le m$

S-4. $\mathsf{Uneq}(i, j)$ for $1 \le i, j \le m$ and $i \ne j$.

Also, addition is a group operation, thus the rules X.1–X.3 apply for expressions constructed with the help of the operator $+$.

For each $i \in \{1, \dots, m\}$ and each $j \in \{1, \dots, i-1\}$, table 6.3 gives certain names to certain claims. We thus have to derive C^m.

The claim C^0 is derived identically to the previous case. The rest of the derivation proceeds in m stages where the i-th stage (depicted in Fig. 6.5; $1 \le i \le m$) culminates with deriving C^i:

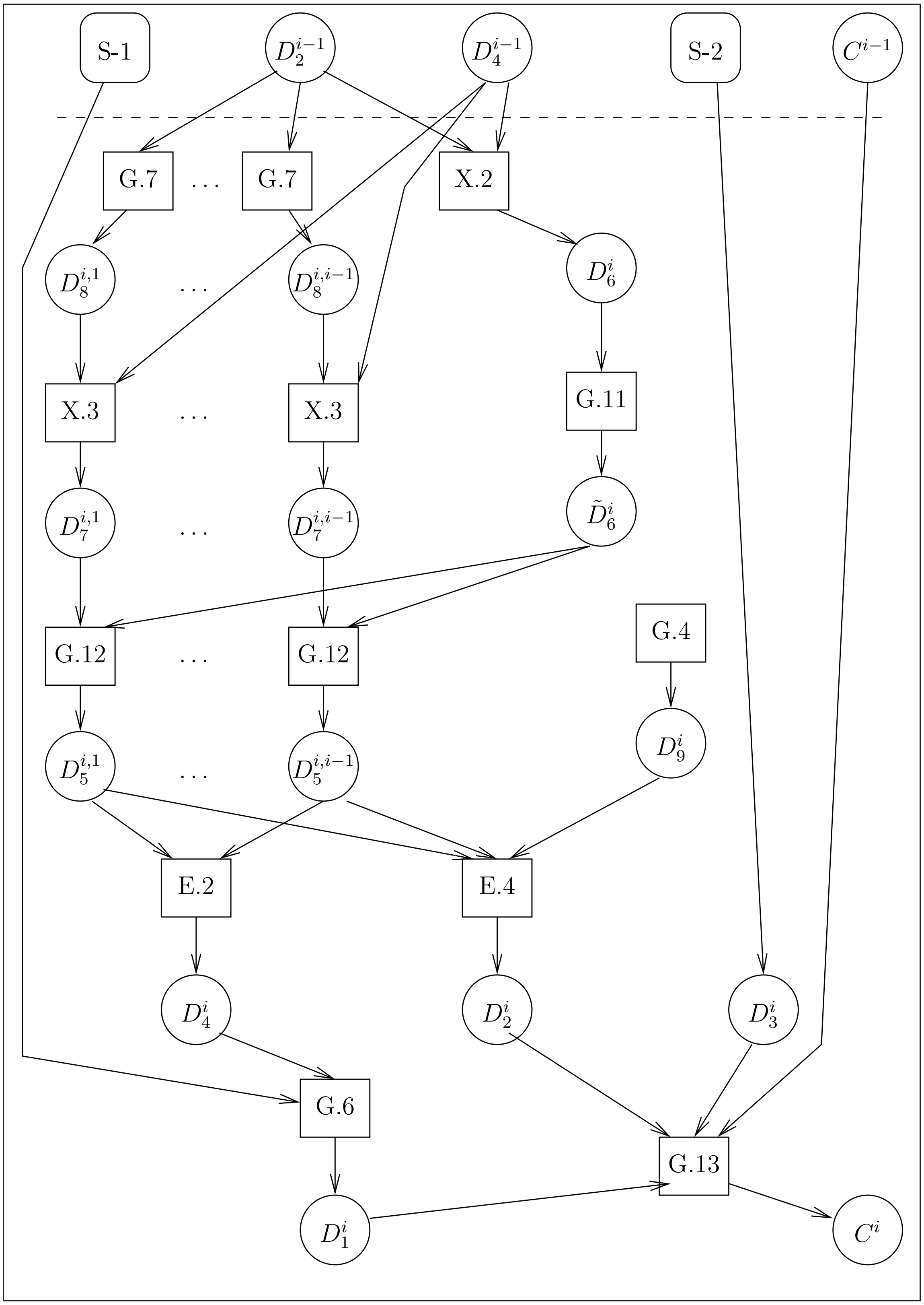


Figure 6.4: Deriving the security of CBC mode

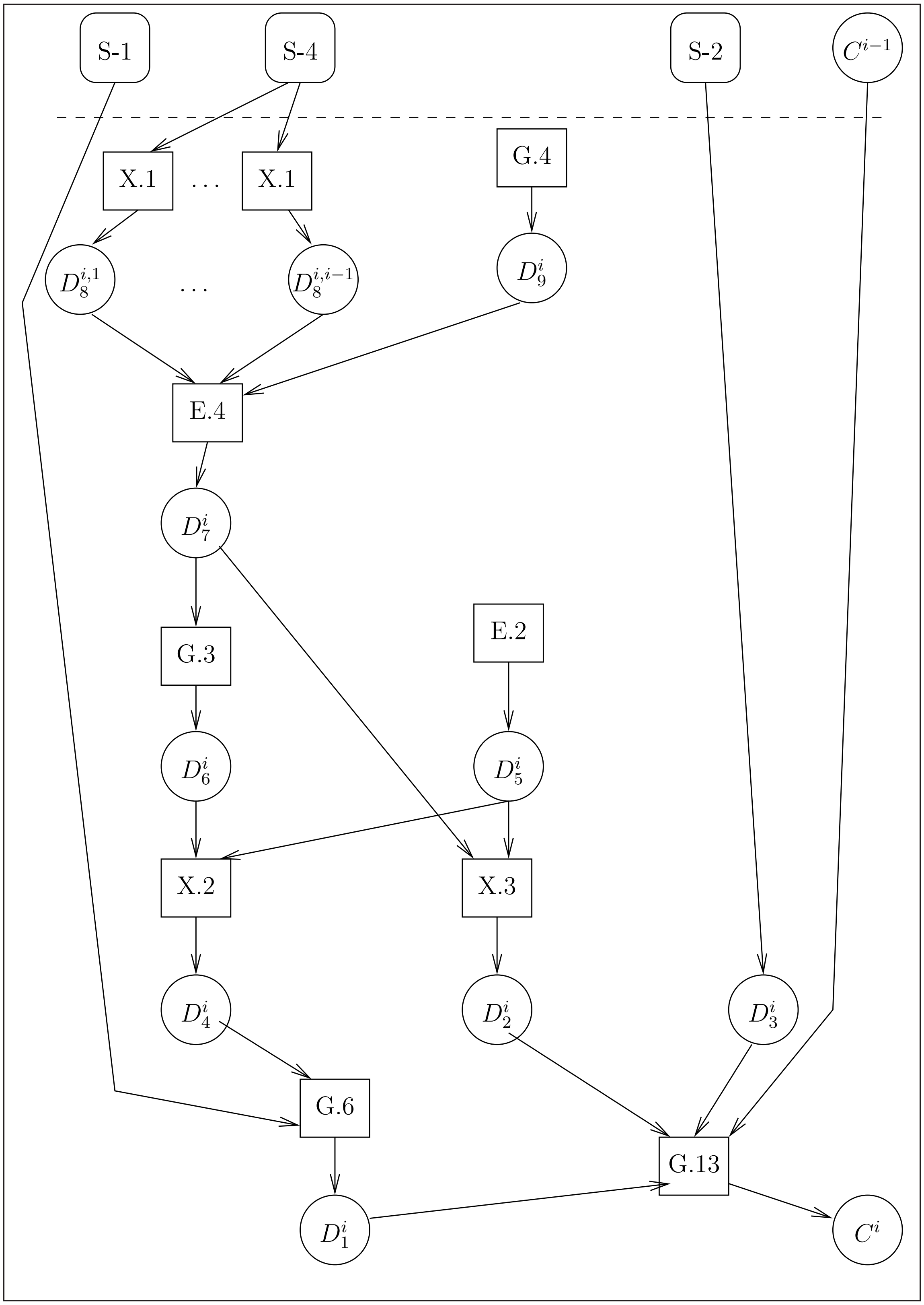


Figure 6.5: Deriving the security of CTR mode

name	claim
C^i	$\mathsf{SameDist}(\{(x_1,x_1),\ldots,(x_m,x_m),(y_0,r_0),\ldots,(y_i,r_i)\})$
D_1^i	$\mathsf{SameDist}(\{(y_i,r_i)\})$
D_2^i	$\mathsf{Indep}(\{x_1,\ldots,x_m,y_0,\ldots,y_{i-1}\},\{y_i\})$
D_3^i	$\mathsf{Indep}(\{x_1,\ldots,x_m,r_0,\ldots,r_{i-1}\},\{r_i\})$
D_4^i	$\mathsf{Uniform}(y_i)$
D_5^i	$\mathsf{Uniform}(\{y_0+i\}_K)$
D_6^i	$\mathsf{Indep}(\{\{y_0+i\}_K\},\{x_i\})$
D_7^i	$\mathsf{Indep}(\{x_1,\ldots,x_m,y_0,\ldots,y_{i-1}\},\{\{y_0+i\}_K\})$
$\mathrm{D}_8^{i,j}$	$\mathsf{Uneq}(y_0+i,y_0+j)$
D_9^i	$\mathsf{Indep}(\{x_1,\ldots,x_m,y_0,\ldots,y_{i-1}\},\emptyset)$

C^i is also defined for $i=0$.

Table 6.3: Claims for showing the security of the CTR-mode

1. D_9^i is given by the rule G.4;
2. for each $j \in \{1,\ldots,i-1\}$, $\mathrm{D}_8^{i,j}$ is derived from S-4 by the rule X.1;
3. D_7^i is derived from D_9^i and $\mathrm{D}_8^{i,j}$ $(1 \le j \le i-1)$ by applying the rule E.4 (the instances of NewEnc expand exactly to the claims $\mathrm{D}_8^{i,j}$);
4. D_6^i follows from D_7^i by applying the rule G.3;
5. D_5^i is given by the rule E.2;
6. D_4^i is derived from D_5^i and D_6^i by applying the rule X.2;
7. D_3^i is given by S-2;
8. D_2^i follows from D_5^i and D_7^i by applying the rule X.3, where we instantiate $X=\emptyset$, $E=\{y_0+i\}_K$, $E'=x_i$ and $Y=\{x_1,\ldots,x_m,y_0,\ldots,y_{i-1}\}$;
9. D_1^i is derived from D_4^i and S-1 by applying the rule G.6;
10. C^i is derived from D_2^i, C^{i-1}, D_1^i and D_3^i by applying the rule G.13.

6.7 Discussion

We have given a set of rules that allows us to derive new facts about the distribution of interpretations of formal messages from the facts that we already know. This set of rules does not attempt to be a "complete" set — i.e. a set that is powerful enough for making all "interesting" derivations. If necessary, one can add new rules to this set. When adding a new rule, one only has to prove the correctness of this new rule, one does not have to consider, how this rule interacts with other rules.

It seems to be possible to implement the analysis presented here. When we are trying to show that a claim $\mathsf{SameDist}(\{(E, E')\})$ holds, then we should record all the claims about the subexpressions of E and E' that we can derive. Recording the claims Constant, Random, Uniform and Uneq does not require too much memory, it is proportional to square of the size of formal expressions. The claims Indep could be recorded similarly to Chapter 5, using boolean functions (represented as BDDs) to represent them. We will cover the claims SameDist later; for now note that there is no rule, where some claim SameDist is one of the antecedents and the consequent is not some claim SameDist.

An analysis step would consist of two parts:

1. Trying to derive new Constant-, Uniform-, Random- and Uneq-claims. The analysis could go through all possible such claims (or we could also try to devise a mechanism that only considers claims for which the antecedents have changed) and verify, whether it can be derived by any of the rules G.10, G.11, G.12, E.1, E.2, E.3, X.1, X.2, P.1.

2. Trying to derive new Indep-claims. This has to happen similarly to the transfer functions presented in Sec. 5.3. Given the boolean function that records all the claims that we have already derived, we have to construct a new boolean function that records all claims that we can derive in one step (using the rules G.2, G.4, G.7, G.9, E.4, X.3, C.1).

We also have to record some SameDist-claims about the subexpressions, but not all of them. First, we only need claims in the form $\mathsf{SameDist}(E_1, E_2)$, i.e. the set of pairs of expressions that is the argument of SameDist only has to contain a single pair. Second, as the only non-trivial way to derive the claim $\mathsf{SameDist}(o(E_1, \dots, E_k), o(E'_1, \dots, E'_k))$ is by rule G.8, we only need to possibly record the following claims $\mathsf{SameDist}(E_1, E_2)$:

- $\mathsf{SameDist}(E, E')$, where E and E' are these expressions, the indistinguishability of whose interpretations we are trying to show.

- If we need to record $\mathsf{SameDist}(o(E_1, \dots, E_k), o(E'_1, \dots, E'_k))$, then we also need to record $\mathsf{SameDist}(E_i, E'_i)$ for all $i \in \{1, \dots, k\}$.

Having thus modified the claims SameDist, we also have to modify some analysis rules to make use of them. Namely, the rules G.8 and G.13 should be replaced by the following rule:

Let $E = o(E_1, \dots, E_k)$ and $E' = o(E'_1, \dots, E'_k)$. If the following claims hold:

$$\mathsf{SameDist}(E_1, E'_1), \dots, \mathsf{SameDist}(E_k, E'_k),$$
$$\begin{bmatrix} \mathsf{Indep}(\{E_1\}, \{E_2, \dots, E_k, E'_1, \dots, E'_k\}), \\ \mathsf{Indep}(\{E_2\}, \{E_3, \dots, E_k, E'_1, \dots, E'_k\}), \\ \cdots\cdots\cdots\cdots\cdots\cdots \\ \mathsf{Indep}(\{E'_{k-1}\}, \{E'_k\}) \end{bmatrix}$$

then the claim $\mathsf{SameDist}(E, E')$ also holds.

The proof of this rule is similar to the proof of rule G.13. Recalling the explanation of rule (4.18), we see that we require here that the expressions $E_1, \ldots, E_k$ and $E'_1, \ldots, E'_k$ are all independent of each other.

Chapter 7

Related Work

In this chapter we overview some related work. Sec. 7.1 and Sec. 7.2 are devoted to earlier works about secure information flow, that have had relevance to the definitions given in this thesis. Sec. 7.1 surveys results about secure information flow in general, while Sec. 7.2 pays attention to aspects of handling probabilism in the programming language. In Sec. 7.3 we give an overview of the earlier results bringing two aspects of cryptography closer to each other. Finally, Sec. 7.4 reviews some work in the area of protocol analysis, attempting to not do unfounded (by security definitions) abstractions of cryptographic primitives.

7.1 Secure Information Flow

The use of program analysis to determine information flows was pioneered by Denning [Den76, DD77]. She instrumented the semantics of programs with annotations that expressed, the values of which program variables flow into which other program variables. The definition of secure information flow was given in terms of these instrumentations [DD77] — a program was defined secure if there were no flows from one variable to another, unless the security level of the first variable was lower than the security level of the second.

Denning and Denning [DD77] also gave a verification procedure to check for this condition. The programmer had to specify the security level of each variable. The procedure checked whether at each assignment $x := o(x_1, \ldots, x_k)$ the security level of x was at least as high as the least upper bound of the security levels of $x_1, \ldots, x_k$ and all variables b that are the guards of *if*-statements and loops containing that assignment.

Definitions based on instrumentations should be avoided, as it is not clear, how these instrumentations relate to the actual semantics of the program. Of course, it is quite easy to move from these instrumentations to the information-theoretic independence of of high-security inputs and low-security outputs.

Volpano et al. [VSI96] have defined a program secure if there exists a simulation of the program that operates only on public variables and delivers the same public outputs for given public inputs as the original program. They also gave a type system for certifying programs for secure information flow. This existence of a low-security simulation is a certain kind of non-interference [GM82] property. In general, one part of a system does not interfere with another part, if the second part's view of the entire system does not depend on the first part's actions. In [VSI96] one requires that the variables of higher security levels do not interfere with the variables in lower security levels.

Later, Volpano and Smith [VS00, Vol00] have attempted to weaken the security definition a little to be able to accommodate one-way functions in the programming language and in the analysis. In [VS00] they assume that there is a certain high security variable h and an unary operator *match*, such that $match(x)$ returns either true or false, depending whether the value of the variable x is equal to the value of the variable h or not. The security definition is such, that if x is a low security variable, then the result of $match(x)$ still can be considered to be a low security value. In [Vol00] a secure program is allowed to treat as a low security value the result of comparing the value of an one-way function applied to a low security variable with the result of applying this one-way function to a certain high-security variable h.

The security definition in [VS00, Vol00] is such, that a secure program cannot copy the value of h to a low security variable. Note, however, that the definition does not say anything about letting a low security variable contain *partial information* about h. Also, the allowed use (without having to consider the result of the operation to be a high security value) of operations violating the non-interference property (i.e. *match*-operations or one-way functions) is rather limited.

Leino and Joshi [LJ98] have given the definition of secure information flow for programs directly in terms of programs' input and output states. A program is defined secure if its denotational semantics is a function from input states to output states, such that the public part of the output state does not depend on the private part of the input states. The meaning of "does not depend" is left unspecified. It is meant to be initiated according to the desired security properties.

7.2 Probabilistic Noninterference

A system may have probabilistic behaviour, i.e. if the system is in a certain state then its next state is not uniquely determined, but is picked according to a certain probability distribution from a set of states. An example of such system is a multithreaded program, where the scheduler picks the next thread to make a step from the set of all existing threads.

Such a system may be modelled by a *non-deterministic* semantics, where the next state of a system is not uniquely determined, but is non-deterministically chosen from a set of states. An analysis for multithreaded programs with such non-deterministic semantics has been given by Smith and Volpano [SV98].

However, such "possibilistic" semantics cannot take into account that different states may have different probabilities of being the next state. These different probabilities may be used to leak information. For example, let b be a high security boolean variable and consider the expression $\mathsf{FlipCoin}\,?\,b : \mathsf{FlipCoin}$, where FlipCoin returns either true or false with probability $\frac{1}{2}$. No matter what the value of b is, the value of this expression can be both true or false, so possibilistically the value of b does not affect the value of the expression. However, the probability distribution of the value of that expression depends on the value of b.

Volpano and Smith [VS98] have given a security definition and a corresponding type system for programs with probabilistic semantics. The semantics is a sequence of probability distributions over states; going from one element to the next in the sequence corresponds to making one program step. The program states also contain these parts of the threads that still have to be executed. The type system is such, that if a program P is well typed, then the executions of this program on two input states s and s' that agree on low security variables proceed in lock-step. This means, that the distribution over states d_n, that is the n-th element in the sequence starting with $\eta^{\mathcal{D}}(s)$, and the distribution over states d'_n, that is the n-th element in the sequence starting with $\eta^{\mathcal{D}}(s')$, agree on low security variables — if one projects the high security variables out from the d_n and d'_n, then these two distributions are equal.

This definition by Volpano and Smith (which was indeed quite clumsy, with program states also containing the part of the program that is still left to execute, and with the equality of program states requiring the syntactic equality of these parts of the program) has been refined by Sabelfeld and Sands [SS00], making it an instance of the *P-restrictiveness* property of Gray [Gra90]. It defined a system secure, if its outputs, observable to low security user, do not depend on the behaviour of high security users. Sabelfeld and Sands still demand that the executions of a secure program on two input states that agree in low security variables proceed in lock step, but in this case, the probabilism is not in the program states, but in the transitions between the states.

The security definition in this thesis, Def. 3.2 is an instance of the definition of Leino and Joshi [LJ98] and it can be seen as the computational analogue to the (information-theoretical) description of security by Sabelfeld and Sands [SS99, Sec. 5.2].

7.3 Two Aspects of Cryptography

As we already have mentioned in the introduction of this thesis, there are two different approaches to arguing about the properties of cryptographic operations — formal and computational. In this thesis, the computational approach corresponds to the semantics of programs given in Chapter 3 and to the interpretation of formal expressions given in Sec. 6.2. The formal approach corresponds to the analyses in Chapter 4 and Sec. 6.5. We have shown how the results in the formal framework translate to results in the computational framework. We are aware of two papers that have aimed to give similar results.

for $i = 1$ to $n()$ **do**	**for** $i = 1$ to $n()$ **do**
spend time...	spend time...
end for	**end for**
output 1	**output** h

Figure 7.1: Two equivalent programs for [AJ01]

Abadi and Rogaway [AR00] define a computational interpretation of formal expressions and give an equivalence relation over the set of formal expressions, similar to the one defined in Chapter 6. They show that if two formal expressions are equivalent then their computational interpretations are indistinguishable. Recently, Micciancio and Warinschi [MW02] have shown that for certain stronger cryptographic properties of the encryption operation, the opposite (indistinguishability implies equivalence) also holds.

Abadi and Jürjens [AJ01] generalise these results from formal expressions to an entire process algebra. They consider (formal) systems consisting of programs and channels between them, where at each execution step, each program reads the values on its inputs, does the computation, and places the results on its outputs. For a system, they define two execution traces — a formal trace and a computational trace. A trace is a sequence (indexed by execution steps) of tuples (indexed by channels) of values in the channels. In formal trace, these values are formal expressions; in computational trace, they are bit-strings (and the computational trace is actually a family of probability distributions over such sequences of tuples of bit-strings).

Some of the channels are considered to be public, others are private. Two formal traces are considered to be equivalent if the messages appearing at the same public channel at the same execution step are equivalent for someone that knows all keys that appear on public channels. Two computational traces are considered to be equivalent if all their equal-length prefixes are indistinguishable. Abadi and Jürjens show that the equivalence of formal traces of two systems implies the equivalence of their computational traces.

The language for specifying programs in [AJ01] does not contain a looping construct. However, as the programs are executed over and over (each execution step of the system corresponds to executing the programs), the language is really Turing-complete. The difference between [AJ01] and the work presented in this thesis in Chapters 3 and 4 is in the security definition. The security definition in [AJ01] (the definition of equivalence for computational traces) does not require the analysis (the equivalence of formal traces) to abstract least fixed points.

One can argue that a security definition that considers only prefixes (of constant length in the security parameter) of computational traces is less intuitive than Def. 3.2. Indeed, assume that h is a high security variable and $n()$ is a nullary operation that returns the security parameter n. Consider two programs (in sense of this thesis) in Fig. 7.1. These programs have the following computational traces:

- The trace of the left program is (restricted to the output channel)

$$\{\langle \underbrace{\epsilon, \ldots, \epsilon}_{n}, 1, \epsilon, \epsilon, \ldots \rangle\}_{n\in\mathbb{N}} \ . \tag{7.1}$$

- The trace of the right program is (restricted to the output channel)

$$\{\langle \underbrace{\epsilon, \ldots, \epsilon}_{n}, h, \epsilon, \epsilon, \ldots \rangle\}_{n\in\mathbb{N}} \ . \tag{7.2}$$

At the m-th position of the sequence stands the value on the output channel at the m-th time moment. ϵ means that no value is outputted to the channel at this time.

Let $m \in \mathbb{N}$ be fixed and consider the prefixes of traces (7.1) and (7.2) of length m. They are

$$\begin{array}{llcl}
 & \overbrace{}^{m} & & \overbrace{}^{m} \\
n = 0 & \langle 1 \quad \epsilon \quad \ldots \quad \epsilon \rangle & & \langle h \quad \epsilon \quad \ldots \quad \epsilon \rangle \\
n = 1 & \langle \epsilon \quad 1 \quad \ldots \quad \epsilon \rangle & & \langle \epsilon \quad h \quad \ldots \quad \epsilon \rangle \\
\ldots\ldots\ldots & \ldots\ldots\ldots & & \ldots\ldots\ldots \\
n = m-1 & \langle \epsilon \quad \epsilon \quad \ldots \quad 1 \rangle & \text{and} & \langle \epsilon \quad \epsilon \quad \ldots \quad h \rangle \ . \\
n = m & \langle \epsilon \quad \epsilon \quad \ldots \quad \epsilon \rangle & & \langle \epsilon \quad \epsilon \quad \ldots \quad \epsilon \rangle \\
n = m+1 & \langle \epsilon \quad \epsilon \quad \ldots \quad \epsilon \rangle & & \langle \epsilon \quad \epsilon \quad \ldots \quad \epsilon \rangle \\
\ldots\ldots\ldots & \ldots\ldots\ldots & & \ldots\ldots\ldots
\end{array}$$

Asymptotically, when n tends to infinity, the prefixes of both (7.1) and (7.2) are sequences (of length m) of ϵ-s only. Hence the prefixes of traces (7.1) and (7.2) are asymptotically equal and therefore indistinguishable.

We have shown that according to the security definition of [AJ01], the two programs in Fig. 7.1 are equivalent. They are not equivalent according to Def. 3.2.

Obviously, the security definition of [AJ01] could be extended to consider polynomial-length (in n) prefixes of computational traces. Their proofs of equivalence, however, work only for constant-length prefixes.

7.4 Faithfully Handling Cryptographic Primitives

There are also other results attempting to precisely formalise and analyse cryptographic protocols, taking into account that the security properties satisfied by cryptographic primitives give only complexity-theoretical, not absolute guarantees. One may either try to directly operate with these complexity-theoretical security properties, or to abstract away from these properties and prove the abstraction correct (like we have done in this thesis).

This approach has extensively been used for *secure multiparty function evaluation* [CDG87, MR91], where a number of participants each input one argument of the function, and in return they all get the value of the function on these arguments, but they get no further information about the arguments inputted by other participants. A protocol realising the computation of that function to be secure, if everything that a subset of participants can see during an execution of the protocol could be computed from the inputs of these participants and from the value of the computed function. The abstraction of a protocol for secure multiparty function evaluation is a black box that privately recieves each participant's input and returns the value of the function to everyone.

Mitchell et al. [LMMS98, LMMS99, Mit01] have defined a non-standard semantics for a process algebra — the π-calculus. The semantics is computational — the messages are bit-strings and nondeterminism in standard semantics is replaced by a probabilistic choice between the possibilities (the semantics fixes the probabilities). In standard semantics, the nondeterminism arises from the parallel composition of processes, where the next process to make a step is chosen nondeterministically.

In these papers, some protocols have been specified in the π-calculus and shown to be correct with respect to this computational semantics. There is no abstraction, though. The proofs of correctness directly work with the computational semantics of protocols. As such, their derivation probably cannot be automated.

Pfitzmann et al. [Pfi96, PSW00, PW00, PW01, Bac02] have devised a generic framework for defining and proving, when a concrete system (for example, a set of protocols) satisfies an abstract specification. In this framework, one considers the interaction of the system with a honest user and an adversary. A concrete system is defined to be *at least as secure as* an abstract system, if for each honest user and each adversary, interacting with the user and the concrete system, there exists an adversary, interacting with the same user and the abstract system, such that the views of the honest user in these two interactions are indistinguishable from each other. This definition is called *simulatability*.

Pfitzmann et al. had the aim, that the specification of the abstract system should be simpler than the specification of the concrete system. Ideally, the abstract system is completely deterministic and non-cryptographic, such that formal methods could be used to argue about its behaviour.

Chapter 8

Conclusions and Future Work

In this thesis we have given a definition for computationally (as opposed to information-theoretically) secure information flow and an analysis to check programs for such kind of secureness of information flow. We have shown that programs can be efficiently analysed for computationally secure information flow, if the encryption operation is repetition and which-key concealing. Hopefully, we have also shown that it is realistic to automatically analyse systems containing encryption, where the encryption operation is just a pseudorandom permutation/function.

Let us present here some thoughts about the achieved results and also about the results that have not (yet) materialised.

8.1 Using the Program Structure

The analysis in Chapter 4 abstracted the denotational semantics of the program. The analysis did not have to consider *how* the program computes; it only kept track of the abstractions of distributions over program states, i.e. *what* had been computed. This is really quite surprising, as the paper by Abadi and Rogaway [AR00], which in some sense served as the starting point for the research in thesis, and which had the same security requirements on the encryption operation as we did in Chapter 4, made use of information that is equivalent to *how* a program computes. Namely, they required that the formal expressions over which the equivalence relation $\cong$ was defined (similar to the relation $\cong$ in Sec. 6.3 of this thesis), did not contain encryption cycles. The structure of formal expressions is the analogue to *how* the program computes, and restricting this structure is analogous to restricting, how the computation of the program may proceed.

The paper by Abadi and Jürjens [AJ01] still contained the same requirement about the absence of encryption cycles. In our analysis we introduced the encrypting black boxes $[k]_{\mathcal{E}}$ and their independence from other variables seems to also carry information about potential encryption cycles. For example, if k_1 and k_2 are keys, $l_1 := \mathcal{E}nc(k_1, k_2)$ and $l_2 := \mathcal{E}nc(k_2, k_1)$, then it is impossible to derive from the analysis in Chapter 4, that $\{[k_1]_{\mathcal{E}}\}$ is independent of $\{l_1, l_2\}$.

Of course, when the encryption operation is only a pseudorandom permutation, then we no longer can ignore, how the program computes or what the structure of formal expressions is. We need to keep track, which plaintexts are equal and which are not. So we have again introduced the requirement about the absence of encryption cycles. Maybe we could somehow do without this requirement, if we managed to introduce the encrypting black boxes again. But as long as we cannot handle loops, checking for the encryption cycles does not seem to be a big burden.

Another reflection of the structure of formal expressions in the analysis presented in Chapter 6 is the predicate $\mathsf{NewEnc}(K, E, E')$. This predicate makes use of the structures of both E and E'. Considering its use, there seems to be no way to remove it; if in future we manage to extend the analysis presented in Chapter 6 to a full programming language, then we still have to keep track of the equality of encrypted bit-strings and still have to have an analogue to NewEnc.

8.2 Future Work

The are several directions in which the work presented here could be enhanced. Some of these enhancements seem to be rather simple to do, but most of them require significant new ideas.

8.2.1 Other Cryptographic Primitives

In our thesis we only considered passive adversaries. Therefore it made no sense for us to consider digital signatures or message authentication codes (or most other cryptographic primitives) as parts of our programming language, because these primitives are meant to repeal active attacks. A natural question would be to ask, are there any other cryptographic primitives that are aimed to preserve the confidentiality, and how would we analyse these primitives.

There are some such primitives. One of them is the public-key encryption. Integrating it to the programming language and to the analysis seems to be rather easy. We would need two extra operators in the programming language — one for generating the public/private key pair (this pair is also used as the private key when decrypting) and another for separating the public key from the key pair. In the analysis, the public key would behave rather similarly to the encrypting black boxes we had in the analysis. Namely, one can encrypt with a public key, but not decrypt.

Another cryptographic primitive that is used to preserve confidentiality are one-way functions. One-way functions have been treated before in the context of secure information flow [VS00, Vol00], but this treatment has been rather lacking. The main issue here is the security definition, there seems to be no good way for giving it. One-way functions do not provide semantic security, as they do not have any secret parts. However, in the definition of secure information flow, anything less than semantic security does not seem to be right.

We could sidestep these issues by assuming the one-way function to be a *random oracle* [BR93b]. A random oracle is basically the oracle $\mathcal{PRF}$ in Fig. 2.4. It provides semantic security. We can analyse it in the same way as we analysed pseudorandom permutations in Chapter 6. A one-way function would be like an encryption with some fixed key K; this key K itself would not be available. But making such assumptions about the one-way functions seems to dodge the real issues with them.

8.2.2 Approximating Fixed Points

Both the concrete and abstract semantics of a program were defined through a least/greatest fixed-point operation (at least if the program contained loops). Our main difficulties in proving the abstract semantics correct stemmed from relating these fixed points. If we had had no specifications through fixed points (i.e. if we had had no loops in our programming language), then we could just have given a correctness proof for each of the rules in Figures 4.2, 4.3 and 4.5 separately and the correctness of the entire analysis would have directly followed from the correctness of each of these rules.

If we are going to extend our analysis (to more primitives and/or to more powerful adversaries), then one possibility to prove the extended analysis correct might involve extending also the configurations (Sec. 4.4.3) respectively. However, a more pleasant result would abstract away from these configurations somehow. We would have to find these parts of the correctness proof of the analysis that are not too intimately related with the structure of the analysis. Unfortunately, it seems to us that there are not many such parts. The configurations are pretty much designed for the analysis at hand; the structure of the configurations and the ways of changing them make up the bulk of the correctness proof.

8.2.3 Active Adversaries

The main future research direction is obviously going to be the handling of more powerful adversaries. An active adversary is not limited to the listening of the outputs of the system under consideration, it can also influence its inputs.

As long as we do not have theorems for the approximation of fixed points, obtaining the results for the general case (for example, relating the computational semantics of π-calculus given in [LMMS99] with some traditional semantics) will be hard. However, we could try to relate the computational and non-computational semantices of some simpler languages/formalisms, namely those that do not contain a looping construct. There are simple, intuitive formalisms (without a looping construct) for expressing cryptographic protocols, for example strand spaces [THG99] or cord spaces [DMP01]. Some results already exist in this area — Guttman et al. [GTZ01] have related the traditional semantics of strand spaces (where the messages are formal expressions) with a semantics, where the messages are bit-strings. For a special class of protocols they have shown that if some attack does not succeed in the formal semantics, then it has at most a very small probability of success in the semantics where the messages are bit-strings. Therefore a proof of security for a protocol in the formal semantics translates to a proof of security for that protocol in the semantics based on bit-strings. The security guarantee is stronger than computational — it is statistical[1], comparable to (although not the same as) the security guarantees of the one-time pad. Obviously, such a strong security guarantee comes with a correspondingly high price, here the price is measured in bits of the necessary length of shared secrets.

It seems to us that obtaining an analogue (for the computational case) of the results in [GTZ01] is not very hard. In a certain sense, this would amount to generalising the results by Abadi and Jürjens [AJ01] to include active adversaries.

[1]This is the reason, why we avoided calling the semantics, operating on bit-strings, "computational".

Bibliography

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A Core Calculus of Dependency. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, January 1999. ACM Press.

[AES01] Advanced Encryption Standard. Federal Information Processing Standards Publication 197 (FIPS PUB 197), 26 November 2001.

[AG99] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, January 1999.

[AJ01] Martín Abadi and Jan Jürjens. Formal Eavesdropping and Its Computational Interpretation. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, volume 2215 of *LNCS*, pages 82–94, Sendai, Japan, September 2001. Springer-Verlag.

[AR00] Martín Abadi and Phillip Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *International Conference IFIP TCS 2000*, volume 1872 of *LNCS*, pages 3–22, Sendai, Japan, August 2000. Springer-Verlag.

[Bac02] Michael Backes. *Cryptographically Sound Analysis of Security Protocols*. PhD thesis, Universität des Saarlandes, 2002.

[BAN90] Michael Burrows, Martín Abadi, and Roger M. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-Privacy in Public-Key Encryption. In Colin Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International*

Conference on the Theory and Application of Cryptology and Information Security, volume 2248 of *LNCS*, pages 566–582, Gold Coast, Australia, December 2001. Springer-Verlag.

[BDJR97] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A Concrete Security Treatment of Symmetric Encryption. In *38th Annual Symposium on Foundations of Computer Science*, pages 394–403, Miami Beach, Florida, October 1997. IEEE Computer Society Press.

[BKR94] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The Security of Cipher Block Chaining. In Yvo Desmedt, editor, *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference*, volume 839 of *LNCS*, pages 341–358, Santa Barbara, California, August 1994. Springer-Verlag.

[BR93a] Mihir Bellare and Phillip Rogaway. Entity Authentication and Key Distribution. In Douglas R. Stintson, editor, *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference*, volume 773 of *LNCS*, pages 232–249, Santa Barbara, California, August 1993. Springer-Verlag.

[BR93b] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, November 1993. ACM Press.

[CDG87] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty Computations Ensuring Privacy of Each Party's Input and Correctness of the Result. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques*, volume 293 of *LNCS*, pages 87–119, Santa Barbara, California, August 1987. Springer-Verlag.

[Cou00] Patrick Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 2000.

[CSF00] *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, Cambridge, England, July 2000. IEEE Computer Society Press.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[Den76] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[DES99] Data Encryption Standard. Federal Information Processing Standards Publication 46-3 (FIPS PUB 46-3), 25 October 1999.

[DMP01] Nancy Durgin, John Mitchell, and Dusko Pavlovic. A compositional logic for protocol correctness. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, Cape Breton, Nova Scotia, June 2001. IEEE Computer Society Press.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.

[GM82] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, April 1982. IEEE Computer Society Press.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic Encryption. *Journal of Computer and System Sciences*, 28(2):270–299, April 1984.

[Gol95] Oded Goldreich. *Foundations of Cryptography (Fragments of a Book)*. Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, 23 February 1995. available with updates at `http://www.wisdom.weizmann.ac.il/~oded/frag.html`.

[Gol99] Dieter Gollmann. *Computer Security*. Wiley, 1999.

[Gra90] James W. Gray III. Probabilistic Noninterference. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, California, May 1990. IEEE Computer Society Press.

[GTZ01] Joshua D. Guttman, F. Javier Thayer, and Lenore D. Zuck. The faithfulness of abstract protocol analysis: message authentication. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 186–195, Philadelphia, PA, November 2001. ACM Press.

[Lan92] William Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, 1992.

[Lau01] Peeter Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In Sands [San01], pages 77–91.

[LJ98] K. Rustan M. Leino and Rajeev Joshi. A Semantic Approach to Secure Information Flow. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC '98*, volume 1422 of *LNCS*, pages 254–271, Marstrand, Sweden, June 1998. Springer-Verlag.

[LM90] Xuejia Lai and James L. Massey. A Proposal for a New Block Encryption Standard. In Ivan Damgård, editor, *Advances in Cryptology - EUROCRYPT '90, Workshop on the Theory and Application of Cryptographic Techniques*, volume 473 of *LNCS*, pages 389–404, Århus, Denmark, May 1990. Springer-Verlag.

[LMMS98] Patrick Lincoln, John C. Mitchell, Mark Mitchell, and Andre Scedrov. A Probabilistic Poly-Time Framework for Protocol Analysis. In *CCS '98, Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 112–121, San Francisco, California, November 1998. ACM Press.

[LMMS99] Patrick Lincoln, John C. Mitchell, Mark Mitchell, and Andre Scedrov. Probabilistic Polynomial-Time Equivalence and Security Analysis. In Jeanette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *LNCS*, pages 776–793, Toulouse, France, September 1999. Springer-Verlag.

[LR85] Michael Luby and Charles Rackoff. How to Construct Pseudo-Random Permutations from Pseudo-Random Functions (Abstract). In Hugh C. Williams, editor, *Advances in Cryptology - CRYPTO '85*, volume 218 of *LNCS*, page 447, Santa Barbara, California, August 1985. Springer-Verlag.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[Mar99] Florian Martin. *Generating Program Analyzers*. PhD thesis, Universität des Saarlandes, June 1999.

[Mit01] John C. Mitchell. Probabilistic Polynomial-Time Process Calculus and Security Protocol Analysis. In Sands [San01], pages 23–29.

[MR91] Silvio Micali and Phillip Rogaway. Secure Computation (Abstract). In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference*, volume 576 of *LNCS*, pages 392–404, Santa Barbara, California, August 1991. Springer-Verlag.

[MRS86] Silvio Micali, Charles Rackoff, and Bob Sloan. The Notion of Security for Probabilistic Cryptosystems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86*, volume 263 of *LNCS*, pages 381–392, Santa Barbara, California, August 1986. Springer-Verlag.

[MW02] Daniele Micciancio and Bogdan Warinschi. Completeness theorems for the Abadi-Rogaway logic of encrypted expressions. In *Workshop on Issues in the Theory of Security - WITS 2002*, Portland, Oregon, January 2002.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[Pfi96] Birgit Pfitzmann. *Digital Signature Schemes: General Framework and Fail-Stop Signatures*, volume 1100 of *LNCS*. Springer-Verlag, 1996.

[PSW00] Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Cryptographic Security of Reactive Systems. In Steve Schneider and Peter Ryan, editors, *Workshop on Secure Architectures and Information Flow*, volume 32 of *Electronic Notes in Theoretical Computer Science*, Royal Holloway, University of London, 2000. Elsevier Science.

[PW00] Birgit Pfitzmann and Michael Waidner. Composition and integrity preservation of secure reactive systems. In *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 245–254, Athens, Greece, November 2000. ACM Press.

[PW01] Birgit Pfitzmann and Michael Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In *2001 IEEE Symposium on Security and Privacy*, pages 184–200, Oakland, California, May 2001. IEEE Computer Society Press.

[San01] David Sands, editor. *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001*, volume 2028 of *LNCS*, Genova, Italy, April 2001. Springer-Verlag.

[Sch96] Bruce Schneier. *Applied Cryptography; Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 1996.

[Sch00] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[SS99] Andrei Sabelfeld and David Sands. A Per Model of Secure Information Flow in Sequential Programs. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99*, volume 1576 of *LNCS*, pages 40–58, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[SS00] Andrei Sabelfeld and David Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00)* [CSF00], pages 200–214.

[SV98] Geoffrey Smith and Dennis M. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998. ACM Press.

[THG99] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand Spaces: Proving Security Protocols Correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.

[Vol99] Dennis Volpano. Safety Versus Secrecy. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis, 6th International Symposium, SAS '99*, volume 1694 of *LNCS*, pages 303–311, Venice, Italy, September 1999. Springer-Verlag.

[Vol00] Dennis M. Volpano. Secure Introduction of One-way Functions. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00)* [CSF00], pages 246–254.

[VS98] Dennis M. Volpano and Geoffrey Smith. Probabilistic Noninterference in a Concurrent Language. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 34–43, Rockport, Massachusetts, June 1998. IEEE Computer Society.

[VS00] Dennis M. Volpano and Geoffrey Smith. Verifying Secrets and Relative Secrecy. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 268–276, Boston, Massachusetts, January 2000. ACM Press.

[VSI96] Dennis M. Volpano, Geoffrey Smith, and Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

[WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer-Verlag, 1992.

[Yao82] Andrew C. Yao. Theory and applications of trapdoor functions (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 80–91, Chicago, Illinois, November 1982. IEEE Computer Society Press.

Index of Notation

Nonalphabetic

Symbol	Description	page
$\{\!\vert E : C \vert\!\}$	Distribution of E under the conditions C	12
$[\![o]\!]$	The semantics of the arithmetic operator o	30
$[\![G, C]\!]$	The interpretations of the flowchart G together with the configuration C	74
$[\![E]\!]$	The interpretation of the formal expression E	144
$\sqsubseteq$	Finer-grainedness of configurations	96
$\sqsubseteq$	Subexpression	144
$E \vdash E'$	The formal expression E' is obtainable from the formal expression E	143
$[\![\cdot]\!] \models C$	The claim C holds for the interpretation of expressions $[\![\cdot]\!]$	155

Alphabetic

Latin

Symbol	Description	page
$\mathsf{2Chart}_{\mathsf{P};X,Y}$	The union of $\mathsf{Chart}_{\mathsf{P};X}$ and $\mathsf{Chart}_{\mathsf{P};Y}$	73
$\mathbb{A}^{(\mathbf{Var})}[\![\mathsf{P}]\!]$	The abstract semantics of the program P	47
$\mathbf{Adv}_{\mathcal{A}}^{D,D'}$	The advantage of the algorithm $\mathcal{A}$ over a coin-flip in distinguishing D and D'	15
$\mathbb{B}$	The set of booleans	8
B^{key}	A component of the abstract values for data flow analysis	128
$\mathsf{BoxNull}(C)$	A component of the configuration C	77
B^{sec}	A component of the abstract values for data flow analysis	128
$\mathsf{bb\_use}$	The function mapping black-box-carrying input edges of flowcharts to the place where this black box is used	68
$\mathsf{Chart}_{\mathsf{P};X}$	The flowchart of program P with outputs $X \subseteq \widetilde{\mathbf{Var}}$	67
$\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$	The concrete semantics of the program P	31
$\mathbf{Conf}^{\mathrm{L}}_{\mathsf{P};X,Y}$	The set of configurations corresponding to generating the outputs X and Y of P together	81
$\mathbf{Conf}^{\mathrm{R}}_{\mathsf{P};X,Y}$	The set of configurations corresponding to generating the outputs X and Y of P separately	82
$\mathsf{Constant}$	Elements of the language of claims $\mathbf{PC}$	154
$\mathsf{constEdge}(C)$	A component of the configuration C	88
conv	A family of mappings from formal expressions to their values	145

Symbol	Description	page
$\mathbb{C}_{\mathrm{term}}[\![\mathsf{P}]\!]$	The concrete semantics of the program P for input distributions, where P runs in expected polynomial time	37
$\mathcal{D}(X)$	The set of probability distributions over the set X	12
$\mathrm{DF}_{\mathsf{G}}(\mathcal{X})$	The output of data flow analysis on the control flow graph G for the input $\mathcal{X}$	125
$\mathbf{Distr}$	The set of families of probability distributions over the sets $\mathbf{State}_n$	29
$\mathbf{Distr}\bot$	The set of families of probability distributions over the sets $\mathbf{State}_{n\bot}$	29
$\mathbf{Distr}_{\mathsf{pol}}$	The set of polynomial-time constructible families of probability distributions over the sets $\mathbf{State}_n$	29
$\mathbf{Distr}\bot_{\mathsf{pol}}$	The set of polynomial-time constructible families of probability distributions over the sets $\mathbf{State}_{n\bot}$	29
$\mathcal{D}^{\mathbb{N}}(X)$	The set of families of probability distributions over the sets X_n	13
E	The function mapping each key to the set of keys that are equal to it	138
$\mathsf{edgeNull}(C)$	A component of the configuration C	79
$\mathsf{edgeParts}(C)$	A component of the configuration C	77
$\mathcal{E}nc$	The encryption operator	28
$\mathsf{EncNull}(C)$	A component of the configuration C	78
$\mathbf{Exp}$	The set of formal expressions	143
$\mathcal{F}(\mathbf{Var})$	The set of pairs of subsets of $\widetilde{\mathbf{Var}}$	44
$\hat{\mathcal{F}}_K(\mathbf{Var})$	The set of pairs of subsets of $\widetilde{\mathbf{Var}}$ that interest the data flow analysis for secure information flow	128
$\mathbf{f}_{X;i}$	The function mapping known variables after A_i to known variables before A_i for an assignment A_i	83
$\mathbf{f}_{X;i}^{\mathsf{false}}$	The function mapping known variables after A_i to known variables before A_i for the false-branch of the vectorised choice A_i	83
$\mathbf{f}_{X;i}^{\mathsf{true}}$	The function mapping known variables after A_i to known variables before A_i for the true-branch of the vectorised choice A_i	83
$\mathsf{fictitious}(C)$	A component of the configuration C	88
$\mathsf{fictUse}(C)$	A component of the configuration C	88
$\mathsf{FlipCoin}$	The coin flipping operator	119
G_{P}	The control flow graph of the program P	122
$\mathcal{G}en$	The key generation operator	28
$\mathcal{G}en$	The label of these nodes of the flowchart that create encrypting black boxes	67

Symbol	Description	page
$\mathcal{G}en_{\mathrm{val}}$	The label of these nodes of the flowchart that create new encryption keys	67
$\mathit{gfp}\ f$	The greatest fixed point of the function f	10
$\mathit{gfp}^y\ f$	The greatest fixed point of f that is less than or equal to y	10
$\mathbf{h}(X)$	The height of the partially ordered set X	9
$\mathsf{IfNewKeys}(C)$	A component of the configuration C	79
Indep	Elements of the language of claims **PC**	154
indeps	The projection onto the second component of $\mathcal{PF}(\mathbf{Var})$	45
$\mathbf{Init}_n$	The set of mappings from atomic expressions to their possible values, for the security parameter n	144
init	A family of mappings from atomic expressions to their values	144
$\mathsf{InpKeys}(C)$	A component of the configuration C	77
$\mathsf{InpParts}(C)$	A component of the configuration C	75
$\mathbf{Inps}$	The set of formal inputs to expressions	152
$\mathbf{Keys}$	The set of formal keys	143
keys	The projection onto the first component of $\mathcal{PF}(\mathbf{Var})$	45
$\mathbf{KeyVal}_n$	The set of possible values for formal keys for security parameter n	144
$\mathsf{KnownInps}(C)$	A component of the configuration C	81
KV_i^X	The set of sets of possibly known variables after A_i	80
$\mathit{lfp}\ f$	The least fixed point of the function f	10
$\mathit{lfp}^y\ f$	The least fixed point of f that is greater than or equal to y	10
$\mathsf{matters}(C)$	The set of known non-fictitious inputs of the flowchart with the configuration C	95
$\mathit{merge}(N, Z)$	The operation of merging together the control flow of branches, where N is the variable carrying control-flow information and Z is the set of variables that have been potetntially changed at the branches	51
$\mathit{Mk}\mathcal{E}$	The label of these nodes of the flowchart that wrap values into encrypting black boxes	67
$\mathit{Mk}\mathcal{E}\mathrm{Set}_i$	The set of nodes of the flowchart that are labeled with $\mathit{Mk}\mathcal{E}$ and correspond to the assignment A_i	75
$\mathbb{N}$	The set of non-negative integers	8
NewEnc	Synonyms for subsets of the language of claims **PC**	159
$\mathbf{Op}$	The set of arithmetic operators in a program	28
$\mathsf{OpParts}_i(C)$	A component of the configuration C	75

Symbol	Description	page
$\mathsf{origins}$	Function mapping an input edge e of $\mathsf{Chart}_{\mathsf{P};X}$ to the set of input edges of $\mathsf{Chart}_{\mathsf{A};\mathsf{P};X}$ that define the value on e	73
$\mathcal{P}(X)$	The set of all subsets of the set X	9
$\mathbf{Parts}(X)$	The set of all partitions of the set X	9
$\mathbf{PC}$	The language of claims about interpretations of formal expressions	154
$\mathcal{PF}(\mathbf{Var})$	The domain of the static program analysis	45
$\mathcal{P}\hat{\mathcal{F}}(\mathbf{Var})$	The domain of the data flow analysis for secure information flow	127
$\mathcal{P}_{\mathrm{L}}(X)$	The set of all downwards closed subsets of the partially ordered set X	9
$\mathsf{Plaintext}$	The set of bit-strings that can be encrypted	20
$\mathbf{Pol}(\mathbb{Z})$	The set of polynomials with integer coefficients	8
$\mathcal{P}_{\mathrm{U}}(X)$	The set of all upwards closed subsets of the partially ordered set X	9
$\mathcal{R}(X)$	Partially ordered set X with reversed order	8
Random	Elements of the language of claims $\mathbf{PC}$	154
$\mathcal{S}(X)$	The set of all permutations over the set X	21
$\mathsf{SameDist}$	Elements of the language of claims $\mathbf{PC}$	154
$\mathsf{SameIf}(C)$	A component of the configuration C	88
$\mathbf{ST}_n$	The set of pairs $\langle$program state, running time$\rangle$ for the security parameter n	30
state	Projection onto the first, "state" component of $\mathbf{ST}_n$	30
$\mathbf{State}_n$	The set of program state for the security parameter n	29
$\mathbf{StrOut}_n^{X;Y}$	The set of outputs of the flowchart $\mathsf{2Chart}_{\mathsf{P};X,Y}$	57
$\mathcal{SV}_K(\mathbf{Var})$	The set of such subsets of $\widetilde{\mathbf{Var}}$, whose independence of other subsets of $\widetilde{\mathbf{Var}}$ interests the data flow analysis for secure information flow	128
$\mathbf{T}$	The set of possible lengths of computations. Equals $\mathbb{N}$	30
$\mathbf{TerD}[\mathsf{P}]$	The set of all such initial probability distributions, for which P runs in expected polynomial time	36
time	Projection onto the second, "time" component of $\mathbf{ST}_n$	30
$\mathbf{Trans}_n$	The type of the n-th component of the semantics $\mathbb{C}_{\mathrm{len}}[\![\mathsf{P}]\!]$	30
$\mathcal{U}(X)$	The uniform probability distribution over the set X	21
Uneq	Elements of the language of claims $\mathbf{PC}$	154
$\mathsf{Uniform}$	Elements of the language of claims $\mathbf{PC}$	154
$\mathbf{Val}_n$	The set of values of variables for the security parameter n	29
$\widetilde{\mathbf{Val}}_n$	The set of values of variables and probabilistic functions over values of variables for the security parameter n	44

Symbol	Description	page
$\mathbf{Var}$	The set of program variables	28
$\widetilde{\mathbf{Var}}$	The set of variables x and pseudovariables $[x]_{\mathcal{E}}$ of the program	44
$\mathbf{Var}_{\mathrm{all}}$	The set of variables of *if* b *then* P_1 *else* P_2 after unrolling	59
$\mathbf{Var}_{\mathrm{asgn}}$	The set of variables that have been assigned to in a branch or in the loop body	51
$\mathbf{Var}_{\mathrm{asgn}}^{\mathsf{false}}$	The set of variables in $\mathbf{Var}_{\mathrm{asgn}}$, having been renamed for the false-branch of the *if*-statement	59
$\mathbf{Var}_{\mathrm{asgn}}^{\mathsf{true}}$	The set of variables in $\mathbf{Var}_{\mathrm{asgn}}$, having been renamed for the true-branch of the *if*-statement	59
$\mathbf{Var}_{\mathrm{P}}$	The set of public variables of the program	36
$\mathbf{Var}_{\mathrm{S}}$	The set of private variables of the program	36
VC_{P}	The set of all $i \in \mathbb{N}$, such that the i-th element in the sequence of assignments and vectorised choices P is a vectorised choice	84
$\mathbb{Z}$	The set of integers	8

Greek

Symbol	Description	page
$\beta^{\mathbb{I}}_{\mathbf{Var}}$	The function abstracting distributions over program states by pairs of mutually independent sets of variables	44
$\beta^{\mathbb{K}}_{\mathbf{Var}}$	The function abstracting distributions over program states by variables that are distributed like keys	45
$\beta^{\mathbb{KI}}_{\mathbf{Var}}$	The abstraction function applying $\beta^{\mathbb{K}}_{\mathbf{Var}}$ and $\beta^{\mathbb{I}}_{\mathbf{Var}}$ in parallel	45
$\eta^{\mathcal{D}}$	The natural injection from X to $\mathcal{D}(X)$	12
λ_I	The labelling of the inputs of the flowchart with the elements of $\widetilde{\mathbf{Var}}$	67
λ_N	The labelling of the nodes of the flowchart with the operators	67
λ_O	The labelling of the outputs of the flowchart with the elements of $\widetilde{\mathbf{Var}}$	67
$\prod_{n\in\mathbb{N}} X_n$	Cartesian product of the sets X_n	8
$\overrightarrow{\rho}(v)$	The sequence of input edges of the node v	67
$\overleftarrow{\rho}(v)$	The output edge of the node v	67
τ	Function, injectively mapping pairs of bit-strings to bit-strings	144