

# Secrecy for Mobile Implementations of Security Protocols

Pablo Giambiagi

A Dissertation submitted to  
the Royal Institute of Technology  
in partial fulfillment of the requirements for  
the Degree of Licentiate of Technology

October 2001

Department of Microelectronics  
and Information Technology  
The Royal Institute of Technology

KTH Electrum 229  
SE-16440 Kista, Sweden

TRITA-IT AVH 01:05  
ISSN 1403-5286  
ISRN KTH/IT/AVH-01/05--SE

Swedish Institute  
of Computer Science

Box 1263  
SE-164 29 Kista, Sweden

SICS Tech. Report T2001:19  
ISSN 1100-3154  
ISRN SICS-T--2001/19-SE

Dissertation for the Degree of Licentiate of Technology  
presented at the Royal Institute of Technology in 2001.

## ABSTRACT

Giambiagi, P. 2001: Secrecy for Mobile Implementations of Security Protocols.  
TRITA-IT AVH 01:05, Department of Microelectronics and Information Technology,  
Stockholm.ISSN 1403-5286.

Mobile code technology offers interesting possibilities to the practitioner, but also raises strong concerns about security. One aspect of security is secrecy, the preservation of confidential information. This thesis investigates the modelling, specification and verification of secrecy in mobile applications which access and transmit confidential information through a possibly compromised medium (e.g. the Internet). These applications can be expected to communicate secret information using a security protocol, a mechanism to guarantee that the transmitted data does not reach unauthorized entities.

The central idea is therefore to relate the secrecy properties of the application to those of the protocol it implements, through the definition of a “confidential protocol implementation” relation. The argument takes an indirect form, showing that a confidential implementation transmits secret data only in the ways indicated by the protocol.

We define the implementation relation using labelled transition semantics, bisimulations and relabelling functions. To justify its technical definition, we relate this property to a notion of noninterference for nondeterministic systems derived from Cohen’s definition of Selective Independency. We also provide simple and local conditions that greatly simplify its verification, and report on our experiments on an architecture showing how the proposed formulations could be used in practice to enforce secrecy of mobile code.

*Pablo Giambiagi, Department of Microelectronics and Information Technology,  
Royal Institute of Technology, KTH Electrum 229, SE-16440 Kista, Sweden,  
E-mail: [pgiamb@it.kth.se](mailto:pgiamb@it.kth.se)*

## Acknowledgements

Several people have contributed to this thesis. My supervisor, Professor Mads Dam, has been a constant source of guidance and support, specially at those hard times when my own confidence seemed to fail me. We made the initial developments of this thesis during 1998, together with John Mullins whose uncompromising attitude towards research has taught me many good lessons. Jan Cederquist, who joined the team in 1999, helped change the direction of this work, in my opinion, for the better.

Most of the activities reported here were done within the PROMODIS project funded by the Swedish National Board for Industrial and Technical Development (NUTEK). In the context of this project, I was very fortunate to meet David Sands, and his students Andrei Sabelfeld and Johan Agat from CTH. They have influenced me in many positive ways: Among other things Johan pointed me to the work by Cohen, Dave participated at different stages of this thesis as my opponent, and Andrei managed to answer my questions in the few moments he was not writing papers at full speed.

My colleagues at SICS and at KTH/IMIT have also contributed a great deal. I certainly know that Dr. Lars-Åke Fredlund will make an excellent supervisor if he ever gives it half the energy he dedicated to comment and correct my drafts. This thesis is hopefully more entertaining, more daring and less badly written thanks to his well-intentioned criticisms. Andrés Martinelli helped also with proof-reading most parts of this thesis. My secondary advisor, Joachim Parrow, showed me the intricacies of the  $\pi$ -calculus and was always at hand with a good piece of advice. For more philosophical discussions I turned to José Luis Vivas and Babak Sadighi, who provided a good counterpoint to technicalities.

Writing a thesis soon becomes a full-time job, and the extra dedication we put in it finally affects our relations outside the office. The emotional support received from my great buddy Andrés and my beloved Elaine is simple incommensurable.

I want to thank you all.

The final stages of this work were funded under project SPC 01-4025 of the European Office of Aerospace Research and Development (EOARD).



# Contents

|   |            |
|---|------------|
| <b>Abstract</b>   | <b>ii</b>  |
| <b>Acknowledgements</b>                                       | <b>iii</b> |
| <b>1 Introduction</b>   | <b>1</b>   |
| 1.1 Background . . . . .                                      | 2          |
| 1.2 Secrecy for Mobile Code . . . . .                         | 5          |
| 1.3 Thesis Overview . . . . .                                 | 6          |
| 1.4 Contributions . . . . .                                   | 8          |
| <b>2 Related Work</b>   | <b>11</b>  |
| 2.1 Confidentiality for Computer Systems . . . . .            | 11         |
| 2.1.1 Access Control Models . . . . .                         | 11         |
| 2.1.2 Information Flow Models . . . . .                       | 12         |
| 2.2 Confidentiality for Security Protocols . . . . .          | 15         |
| 2.2.1 Formal Models . . . . .                                 | 16         |
| 2.2.2 Computational Models . . . . .                          | 17         |
| 2.3 Programming-Language Confidentiality . . . . .            | 20         |
| 2.4 Relations between Secrecy Models . . . . .                | 22         |
| 2.5 Mobile Code Security . . . . .                            | 22         |
| <b>3 A Model for Cryptographic Processes</b>                  | <b>25</b>  |
| 3.1 Security Process Algebra . . . . .                        | 26         |
| 3.1.1 The Purchasing Applet Examples . . . . .                | 32         |
| 3.1.2 The Wide-Mouthed Frog Protocol Example . . . . .        | 36         |
| 3.2 Annotations for Tracking Direct Dependencies . . . . .    | 39         |
| 3.2.1 Annotated Purchasing Applet Examples . . . . .          | 43         |
| 3.2.2 Annotated Wide-Mouthed Frog Protocol Example . . . . .  | 44         |
| 3.3 Annotations are Conservative . . . . .                    | 44         |
| <b>4 Confidential Protocol Implementation</b>                 | <b>49</b>  |
| 4.1 Secrets and Confidentiality Policies . . . . .            | 51         |
| 4.2 Towards a Notion of Confidential Implementation . . . . . | 55         |

---

|          |   |            |
|----------|---|------------|
| 4.3      | Conditional Process Relabelling . . . . .                   | 57         |
| 4.4      | Admissibility . . . . .                                     | 60         |
| 4.4.1    | Verifying Admissibility . . . . .                           | 61         |
| <b>5</b> | <b>Controlled Information Flow</b>                          | <b>67</b>  |
| 5.1      | Cohen's Selective Independency . . . . .                    | 68         |
| 5.1.1    | Separation of Variety . . . . .                             | 70         |
| 5.2      | Selective Independency for a-SecPA . . . . .                | 71         |
| 5.2.1    | History Indistinguishability . . . . .                      | 72         |
| 5.2.2    | $\Delta$ -Bisimilarity . . . . .                            | 73         |
| 5.2.3    | Selective Independency . . . . .                            | 76         |
| 5.3      | Admissibility as $\Delta$ -Bisimulations . . . . .          | 78         |
| 5.4      | Admissibility vs. Selective Independency . . . . .          | 82         |
| <b>6</b> | <b>Experimentation: An Architecture for Confidentiality</b> | <b>87</b>  |
| 6.1      | Requirements . . . . .                                      | 88         |
| 6.2      | A PCC Architecture for Confidentiality . . . . .            | 89         |
| 6.3      | Experiments . . . . .                                       | 91         |
| 6.3.1    | Modeling the Java Virtual Machine . . . . .                 | 92         |
| 6.3.2    | The Prototype . . . . .                                     | 94         |
| 6.3.3    | Some Conclusions on the Experiments . . . . .               | 98         |
| <b>7</b> | <b>Conclusions and Future Work</b>                          | <b>101</b> |
| 7.1      | Future Work . . . . .                                       | 103        |
| <b>A</b> | <b>Proofs</b>   | <b>115</b> |
| A.1      | Proofs for Chapter 4 . . . . .                              | 115        |
| A.2      | Proofs for Chapter 5 . . . . .                              | 116        |

# List of Tables

|     |   |    |
|-----|---|----|
| 3.1 | The Wide-Mouthed Frog Protocol . . . . .  | 36 |
| 3.2 | A <i>SecPA</i> implementation of the server in the Wide-Mouthed Frog<br>protocol. . . . . | 37 |
| 4.1 | Control flow for Wide-Mouthed Frog server implementation . . .                            | 63 |
| 6.1 | Partial pseudo-code for the purchasing applet . . . . .                                   | 92 |
| 6.2 | Virtual instructions and corresponding events in the model of<br>Section 3.1.1 . . . . .  | 94 |

# List of Figures

|     |  |    |
|-----|--|----|
| 3.1 | Wide-Mouthed Frog Server Implementation . . . . .                | 38 |
| 3.2 | Annotated Wide-Mouthed Frog Server Implementation . . . . .      | 45 |
| 6.1 | A Proof-Carrying Code Architecture for Confidentiality . . . . . | 90 |
| 6.2 | Proof-Carrying Code Assembler . . . . .                          | 95 |
| 6.3 | Proof-Carrying Code Checker . . . . .                            | 97 |

# Chapter 1

## Introduction

Confidentiality is that part of computer security concerned with the flow of information from secret sources to unauthorized observers, as well as the capacity of these observers to extract useful knowledge from the information.

The study of secrecy, as confidentiality is also called, touches upon many aspects of computer systems: from the construction of secure operating systems and the verification of programs, to the design of communication protocols. The first two are strongly related: The definitions given in the context of secure operating systems serve as semantical justification for the syntactical methods used to enforce secrecy in programs. However, with a few exceptions, the study of secrecy of systems and programs has been independent of the work on design and analysis of security protocols.

In recent years, the development of the communication infrastructure has resulted in an increased interest in mobile code applications, motivated by the flexibility and functionality that these applications offer. The advantages come at a price, though, as mobile programs are supposed to manipulate and communicate security-critical information. This raises natural concerns over the secrecy properties of both the security protocols chosen and their implementations.

It can therefore be argued that the task of providing secrecy for mobile code calls for a combination of the approaches to confidentiality mentioned above, i.e. for both systems security and protocol analysis. To make this clearer, consider the following two scenarios:

- A user accesses a commercial web site, browses through the goods offered, and chooses what to buy. He then downloads a program from the merchant to perform an on-line payment using his credit card. The merchant “tells” the user that the code employs a *secure e-commerce protocol*, accepted by the corresponding credit card company, and which is supposed to preserve the confidentiality of the credit card information.
- In order to cast their votes at national elections, citizens are allowed to download programs that implement the voting protocol chosen by the

state. A number of different institutions are interested in developing and providing their own implementations, among them political parties interested in helping their members with the task.

Besides illustrating the interesting functionality provided by mobile code applications, these scenarios involve programs with access to confidential information (i.e. the credit card and the ballot data) that has to be communicated using a security protocol (i.e. the e-commerce and the voting protocols). Even if it is known that each protocol has previously been verified, that the code provider (merchant, political party, etc.) is trusted, and that a cryptographic certificate prevents anyone from tampering with the code on the way to the user's computer, *how can the user be sure that the downloaded code actually preserves the confidentiality properties of the protocol?* There is, of course, the risk that the code might actually be a Trojan Horse, a program that once put in contact with confidential information will leak it to unauthorized observers. For example, the user of the on-line payment program does not want the credit card's Personal Identification Number (PIN) to be leaked to anyone but the bank that issued the card, while the voter does not want anyone else to know who he voted for.

A solution that could help the user assess the confidentiality of downloaded code requires attention to a wide range of issues, ranging from basic principles (how is confidentiality specified, modelled, and verified), implementation aspects (how are the basic principles mapped to actual code, how are confidentiality properties processed by the participating entities), to usability aspects (how can the confidentiality properties be presented to users to empower them to make informed decisions). This thesis addresses mainly the first issue, that of specifying, modelling, and verifying the confidentiality properties of mobile implementations of security protocols. In other words, it presents a notion of protocol implementation that offers flexibility to the implementor and guarantees the preservation of the confidentiality properties of the protocol. As such, this thesis stands on and combines previous work on system security and protocol analysis.

## 1.1 Background

In all approaches to confidentiality there is always a partition of users (of a computer system) or principals (participating in a protocol) made to differentiate access rights to data. When general computer systems are considered, it is traditional to assume a Multi-Level Security (MLS) scheme where each object and subject in the system is assigned a level in a security hierarchy. It is also customary to talk about only two levels: High (or Secret) and Low (or Public), as this is not conceptually different from the general case. Naturally, low-level users and programs are not authorized to access high-level data. Similarly, in protocol analysis it is common to identify trusted principals trying to

communicate secret information, and attackers (or spies) trying to get hold of it.

In spite of these similarities, there exist two markedly different ways to interpret confidentiality, or more precisely, its absence. Wittbold and Johnson called them *methodological views* in [WJ90]. In the first one, the “eavesdropping” or “wiretapping” view, there are low-level users (or programs) in a system, or attackers of a protocol, which are trying to get access to confidential information by simply interacting with the system and/or with other principals, and trying to deduce confidential information using clever mechanisms. In contrast, in the “transmission” view there are malicious processes or users with access to high-level data that attempt to transmit it to low-level users or unauthorized observers. These are usually called Trojan Horses, programs aimed at using a system as a communication channel, conveying information originating from confidential sources to low-level destinations.

Another important dimension along which we can place the different approaches to confidentiality concerns the emphasis put on the ability of low-level users or attackers to extract useful information out of their observations. As an example, we consider a simple process that inputs secret  $m$  and outputs it encrypted with a shared key  $k$ :



What information about  $m$  can a low-level user obtain out of observing  $\{m\}_k$ ? In principle, if the encryption mechanism were “perfect”, no information about  $m$  could be recovered from  $\{m\}_k$  without knowing the key  $k$ . However, we know from Information Theory that such a perfect encryption mechanism can hardly be useful<sup>1</sup>. In fact, practical cryptosystems are not information-theoretically perfect, and instead base their strength on the computational difficulty of extracting any information content out of ciphertexts like  $\{m\}_k$ . Of course, if the observer *knows* the key  $k$  and has access to the decryption mechanism, it can recover  $m$ .

In general, some approaches aim at cancelling all kind of secret information flow, while others admit flows but limit the observation power of observers. In the following we examine briefly standard techniques for ensuring confidentiality, and explain the shortcomings of these techniques. In Section 1.3 we present a solution to the problem of ensuring confidentiality of data for mobile applications.

## Confidentiality for Computer Systems

Much of the work in this area has been focused on the design and verification of trusted computing bases (TCB), that is the central components of an operating

<sup>1</sup>Informally, a perfect encryption scheme needs keys as big as the messages to encrypt.

system responsible for, among other tasks, enforcing the separation of users, files and data into security levels. Since the designer of an operating system cannot predict the functionality of every possible Trojan Horse, such a system should be built so that any Trojan Horse is rendered innocuous. This corresponds naturally to the “transmission” view.

A Trojan Horse can always exploit any unintended communication channel established by the execution of the system to transmit information to low-level users. Such a channel is usually called a covert channel. As an example consider a high-level bounded buffer. Suppose that low-level users are allowed to write to the buffer, but not to read from it. Furthermore, assume that in this implementation, whenever a low-level user tries to write to a full buffer, the system responds with an error message. This is a covert channel. A Trojan Horse with access rights over the buffer, can fill and empty it at will. To exploit this channel, a low-level user might try to write to the buffer at regular interval. Just before each access, the Trojan Horse either fills or empties the buffer, transmitting a bit of information to the low-level user at a time.

A quite popular approach to system confidentiality has therefore been to try to cancel out all flow of information from High to Low. This is the domain of properties like Noninterference [GM82, GM84a], Nondeducibility [Sut86] and p-Restrictiveness [Gra90], among many others which attempt to apply information theory to increasingly richer models of computation.

There are situations however where the noninterference approach is not appropriate and it is necessary to allow the declassification of High data by authorized entities. This is the objective of Intransitive Noninterference [Rus92, RG99]. Furthermore, it is generally accepted that covert channels are unavoidable in realistic applications. If noninterference properties cannot be verified, then an alternative would be to measure and bound the information that can flow through a covert channel. Initially, researchers considered this measure to be channel capacity, an information theoretic concept [Mil87]. However, channel capacity takes an asymptotic view, which does not characterize well the concrete amount of information that can be leaked during a limited amount of time [MK94].

At the level of programming languages there is also a long tradition of work on confidentiality. Here security has usually taken the form of independence of low-level outputs from high-level inputs. Starting from the work by Denning [Den76, DD77], through its semantical justification and encodings in the form of type systems [VSI96], to the plugging of probabilistic covert channels in multi-threaded languages [SS00], the stress has been put on restricting the flow of information, not on controlling it.

## Confidentiality for Security Protocols

Security protocols, especially if they rely on cryptography, are usually analysed assuming that the only threat comes from external attackers, and not from the legal users of the protocol. Such an assumption clearly corresponds to the

“eavesdropping” view. Moreover, since perfect encryption schemes are hardly of use in real situations, there is normally flow of secret information from the principals taking part in a protocol and any possible attacker. Instead of attempting to cancel out those flows, the idea is to estimate the amount of information that an attacker can extract from them. In the example above, if the attacker observes  $\{m\}_k$  then it neither knows the key  $k$ , nor can it learn anything about  $m$  without considerable effort.

There are basically two approaches to the problem of estimating the knowledge of the attacker. The *formal* approach abstracts cryptographic functions, instead of modelling them in detail. Usual models derive from Dolev and Yao’s perfect cryptography assumption<sup>2</sup>, which has very concrete implications to the information an attacker can extract from a ciphertext [DY83]. For example, if the attacker does not have access to the complete decryption key  $k$ , then it is able to extract absolutely no information about  $m$  by simply observing  $\{m\}_k$ . The Dolev-Yao model greatly simplifies the analysis of protocols. On the other side, in the *computational* approach attackers have complexity bounded resources and cryptographic operations are modelled in full detail (although the model relies on ad-hoc assumptions like the existence of one-way functions, i.e. a function  $f$  such that a resource-limited attacker cannot recover  $x$  from  $f(x)$ ).

While the *formal* approach has been quite successful in the construction of automatic and semi-automatic tools to discover buggy protocols, the *computational* approach provides more convincing results on the properties of protocols when real cryptographic operations are considered. Recent work indicates that, in certain circumstances, the perfect cryptography assumption can actually be implemented, thus justifying some of the common abstractions in the *formal* approach [AR00, AJ01].

## 1.2 Secrecy for Mobile Code

The voting and e-commerce scenarios discussed at the beginning of the introduction exemplify the kind of applications we are interested in. They are untrusted implementations of security protocols. As such, they have access to secret (high-level) data and are supposed to engage in communications through a compromised medium resulting in flows of information to eventual attackers.

Each piece of mobile code can be seen as a small system. Consequently, their secrecy properties could perhaps be modelled and verified using some of the methods mentioned in Section 1.1. However, a quick inspection of the assumptions and modelling requirements of those methods casts strong doubts on such a possibility. First, the kind of applications we consider do establish flows of information, thus excluding noninterference techniques that try to cancel them altogether. Second, there is no clearly identifiable entity in charge of declassifying information (other than the mobile code itself), so intransitive non-interference approaches seem inappropriate. Finally, for most protocols any pos-

<sup>2</sup>Not to be confused with the notion of *perfect encryption*.

sible implementation would establish communication channels from high-level sources to low-level observers that *could* encode high amounts of information in an easy to decode way. As an example, consider the e-commerce scenario. Normally a protocol would protect the user's PIN, but disregard the protection of the 1-bit datum of whether a transaction is successful or not. In such a case, a user can choose to initiate a transaction with a correct PIN to transmit a one, and with any other PIN to transmit a zero.

The example also illustrates another point: protocols are assumed to be correct from the point of view of secrecy and within the “eavesdropping” methodological view. The latter means that the verification of the protocol assumes that the communication channels established by the protocol itself are not abused. The example shows how this assumption could be violated.

Another alternative would be to apply the same model and verification procedure to the code as is applied to the protocol specification. However, this is difficult. First, protocol implementations are several orders of magnitude more complex than the protocols they implement. Consequently, it is not to be expected that one could apply protocol analysis techniques to analyze code. Second, protocols involve two or more principals, but an implementation corresponds to just one them. Moreover, protocols stipulate what happens if everything goes well, but not necessarily what to do when, for example, a challenge does not get the expected response.

### 1.3 Thesis Overview

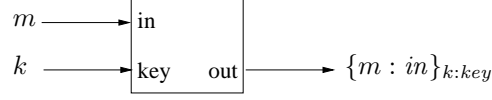
This thesis suggests a solution to the problem of modelling, specifying and verifying secrecy of mobile implementations of security protocols. The proposed solution takes the form of a confidential protocol implementation relation, linking the secrecy properties of code to those of the protocol it implements.

The implementation relation reflects the mixed nature of the problem, combining techniques and methodological views of confidentiality for both operating systems and protocols (c.f. Section 1.1). The definitions have the flavor of information flow properties, but in the context of the “eavesdropping” view (as it corresponds to the study of protocols) which means that there are stringent assumptions on the normal behavior of high-level users.

Before presenting the solution, we need to establish a model of computation that is both simple and rich enough to represent mobile implementations of security protocols. Although a labelled transition system semantics would suffice, the presentation takes a process algebraic form, hopefully making it more compact, and at the same time easier to understand and to adapt to different settings. For this purpose, Chapter 3 introduces a simple process algebra, an extension of value-passing CCS [Mil89] with cryptographic operators that is used to describe some simple but interesting examples.

The Multi-Level Security schema, so frequently used in information flow characterizations of confidentiality, is replaced in our work by a more fine-

grained labelling system. First, the channels through which secret data is input into the system are clearly identified. For the encrypter example of Section 1.1, we can identify entry channels for messages  $m$  and keys  $k$ :



As soon as a value enters the system through one of the identified channels, it is annotated with a reference to its origin (e.g.  $m : in$ , and  $k : key$ ). This annotation is used to track the evolution of secret data through computation, permitting the correct identification of outputs. In the example above, the annotated output reflects both the origin of data, and the operations used to compute its value. This is also explained in Chapter 3.

In general, protocols are given operational descriptions, determining the order and format of messages to be exchanged by the participating agents. From the point of view of an attacker that can store and replay messages, their ordering plays a small role. More important are, instead, the format of messages and the conditions under which they are emitted. It is therefore convenient to consider protocol specifications that highlight the latter, and hide unnecessary details. Since the interest is on the confidentiality properties guaranteed by the protocol, we call such an abstract view a *confidentiality policy*. This policy indicates which expressions can be constructed of secret inputs, and when it is admissible to output them. To continue with the simple example above, say that the encrypter is only allowed to output an input message  $m$  when properly encrypted by a key  $k$ . Then the policy can be written as:

$$out!\{m\}_k \leftarrow in(m) \wedge key(k)$$

Chapter 4 formalizes the notion of confidentiality policy and provides the definition of the confidential protocol implementation relation. *Admissibility*, as this relation is called, implies that the behaviors of the implementation are indistinguishable for any attacker when secret inputs are permuted and admissible outputs are abstracted away. Technically, admissibility is defined in terms of bisimulation relations, much inspired by the treatment of secrecy in the the  $\pi$ -calculus [AG98a]. This gives a nice, compact definition that is complemented with a result reducing its verification to a set of local conditions (this type of result is usually called an *unwinding theorem* in the literature).

Every time a confidentiality property is proposed, it is of great importance to determine how it compares to other, previous definitions. This issue is explored in Chapter 5. First, the notion of Selective Independency [Coh77, Coh78] is generalized from its original presentation over a functional execution model, to a nondeterministic setting. The resulting definition resembles that of Nondeducibility on Strategies [WJ90], but based on the idea of indistinguishability under permutations of input data. Then, admissibility is reduced to Selective

Independency of an appropriately altered system, thus giving a concrete characterization of the proposed confidentiality implementation relation.

All considered, the solution exhibits a number of interesting features over other alternatives considered in Section 1.2: Firstly, the efforts and results of protocol analysis are immediately reinvested. Secondly, by relating to a confidentiality policy, an implementor is not unduly constrained in the techniques used to implement the protocol. Thirdly, the existence of a simple local verification technique (unwinding theorem) and the link to Selective Independency suggests that programming-language security methods could also be applied to verify admissibility.

The remainder of the thesis is organized as follows: Chapter 2 reviews work on confidentiality at the level of complex systems, programming languages and protocol specifications. It also touches upon models of cryptographic security. Chapter 6 comments on our experiments with confidentiality of Java applets, covering several issues beyond the modeling of confidentiality. Finally, Chapter 7 presents some conclusions and discusses future work.

## 1.4 Contributions

Initial developments regarding the concept of admissible information flow was reported at the 13th IEEE Computer Security Foundations Workshop, in a paper co-authored with Mads Dam [DG00]. By means of a running example (a purchasing applet), the paper introduces a much earlier version of the admissibility property and goes on to informally discuss its motivations and practical application.

The main contributions in this thesis are:

- An extension of value-passing CCS with operators for cryptography, called *SecPA*, that allows for the encoding of the mobile code applications mentioned in this introduction.
- An annotation of processes in this algebra to help correlate changes in input with changes in output.
- A formal definition of the notion of admissible information flow (*Admissibility*) over the annotated process algebra based on a bisimulation relation between processes.
- A relation between Admissibility and a notion of confidentiality for reactive systems based on Cohen's notion of Selective Independency.
- A proof technique that suggests the possibility of using static analysis techniques to enforce Admissibility (*unwinding theorem*) even for programs that exhibit branching of control flow over secret data (provided the branching condition is admissible).

- An architecture showing how Admissibility can be used in practice to ensure confidentiality aspects of mobile code. This reports on our experiences with Java, web browsers and proof-carrying code, and addresses subtleties in the definition of an adequate user interface.

*Which of the above are my own contributions?* Most of them, although with some exceptions: The idea of expressing a confidentiality property by means of bisimulations and relabellings, as well as the architecture of Chapter 6 were developed in collaboration with Mads Dam and John Mullins, at the Royal Institute of Technology (KTH) and the Swedish Institute of Computer Science (SICS). The coding of the experiment was done in collaboration with two undergraduate students at KTH. Jan Cederquist and I wrote an extended abstract (presented at the IEEE Symposium on Logic in Computer Science 2000 [CG00]) with some initial ideas on relating the specification of the set of relabellings to the specification of the protocol. However, those ideas have been totally reworked in this thesis. Finally, a more restricted version of Theorem 4.15 appeared in a much simpler setting and without proof in [DG00].



## Chapter 2

# Related Work

The modelling, specification and verification of secrecy of mobile implementations of security protocols requires and involves models and methods from different related areas. These areas include, in first place, the study of the general confidentiality principles for computer systems. Since we are interested in programs implementing protocols, it is important to understand the particular characteristics of protocol confidentiality, as well as programming-language methods for enforcing secrecy. We also pay attention to the points of contact between these areas, since they help us understand them all better. This chapter concludes with some references to the broad range of work on ensuring wide security properties for mobile code.

### 2.1 Confidentiality for Computer Systems

In computer systems, confidentiality goes hand in hand with authorization. Information is allowed to flow only to those objects and subjects that are authorized to access it. Consequently, the first attempts at the design of a secure operating system paid particular attention to access control mechanisms.

#### 2.1.1 Access Control Models

An *access control* model is a mechanism for enforcing confidentiality. John McLean gives a very lucid description of these models in [McL94], which started with the work of Lampson, Graham and Denning at the beginning of the 70's. Essentially, access control models are states machines whose states are triples  $(S, O, M)$ . The first component,  $S$ , identifies a set of subjects (e.g. programs), while  $O$  represents a set of objects (e.g. files). The access rights that subject  $s \in S$  has over object  $o \in O$  are recorded by  $M(s, o)$ . States are changed by requests for altering  $M$  (e.g. by creating/deleting subjects/objects, and adding/deleting access rights).

The first question that arises is that of determining, for a given initial state  $q_0$ , and access right  $a$ , whether there is a run of the system starting from  $q_0$  where  $a$  is assigned to a pair  $(s, o)$  that initially did not have this access right. Harrison, Russo and Ullman (together with others working on related models) characterized the complexity of this question for different variants of their *HRU* access control model. In general, the problem is undecidable.

Initial work on access control models did not consider the problem of *Trojan Horses*. These are programs that perform operations that their users are normally not aware of, like distributing access rights to files owned by them. This problem has motivated the distinction between *Discretionary Access Control* (DAC) and *Mandatory Access Control* (MAC) policies. The essential difference being that a MAC policy restricts how users can pass rights to other users, whereas DAC does not.

Bell and La Padula produced the best known access control model for handling MAC policies [BL76]. It uses the same state machine model as above, but transitions are not allowed to modify the sets  $S$  and  $O$ . Bell and La Padula fixed the set of access rights to  $\{\text{read}, \text{write}, \text{append}, \text{execute}\}$ . They also introduced a fixed *Multi-Level System* (MLS) which comprises a lattice of security levels  $L$  and a function  $f : S \cup O \rightarrow L$  assigning levels to subjects and objects. A state of the system is *secure* if no subject can read objects above its level, nor write to an object with lower clearance. A system is then secure if every reachable state is secure. A main result, known as the *Basic Security Theorem*, links this state-based characterization of security to an equivalent transition-based description, giving an easy way to implement the model (usually through a *reference monitor* that checks all accesses at runtime).

### Covert Channels

The problem with access control models is that, although intuitive, they lack a precise semantics, in the sense that the identification of subjects and objects, and their mapping to access rights is left to the implementor. Choosing these elements is an extremely difficult task if one wants the model to prevent all possible channels from higher to lower security levels. It is normally the case that, after mapping the model's primitives to the computer system, the implementor still needs to study and estimate the capacity of the remaining channels. These channels are usually called *covert channels* and are normally divided into *storage* and *timing* channels, although the distinction between them is rather diffuse in some cases. Other classifications divide channels into noise and noiseless channels, the latter exploiting knowledge on the probabilistic distribution of the transmitted values.

### 2.1.2 Information Flow Models

Information flow theories, initially intended to explain covert channels in implementations of access control models, constitute an important step towards

giving an extensional characterization of what a confidentiality property is. The intention behind most information flow models is to guarantee that what is done by users of a higher level modifies the behavior of lower level users in no possible way.

### Noninterference

The first models addressed sequential programs and aimed at determining the flow of information from initial to end values of variables. Jones and Lipton defined what they called a surveillance set mechanism, a way of accumulating the variables upon which another one depends [JL75]. Cohen gave a semantical characterization of independency and was one of the first to relate to information theory [Coh78, Coh77]. His security condition, i.e. that a secure program must not convey variety of high-level inputs to variety of low level results, constitutes the starting point for a critical analysis of the main concepts presented in this thesis (see Chapter 5).

Feiertag, Levitt and Robinson moved on to consider deterministic systems in [FLR77], leading to the definition of Noninterference by Goguen and Meseguer, where the idea is to make sure that high level behaviour does not interfere with the observations of low level users [GM82]. In [GM84a], the same authors provided “unwinding theorems” giving sufficient conditions for noninterference in the form of invariant properties; and in [McL92], McLean suggested more direct proofs over system specifications given as trace sets.

In the nondeterministic case, the situation is far from clear. Sutherland first proposed Nondeducibility on Inputs [Sut86], which turns out not to be compositional and overlooks feedbacks, i.e. the possibility that a Trojan Horse leaks information by adapting the high level inputs it feeds to the system according to the low level outputs the system produces. The compositionality problem was solved by McCullough’s Restrictiveness [McC88, McC87, McC90]; and the feedback problem by Wittbold and Johnson’s Nondeducibility on Strategies [WJ90].

All of these are called possibilistic properties, as they do not consider both Trojan Horses that modify the probabilistic behavior of the system and low level observers able to detect changes in the associated probabilistic distributions. Most possibilistic properties are not preserved by the traditional characterization of system refinement as trace inclusion: an implementation can resolve the nondeterminism in an insecure way. Consequently, some researchers have suggested that the nondeterminism in a specification should be taken to mean underspecification, and that its implementations have to eliminate all nondeterminism of this kind [Ros95, JÖ1]. Mantel has defined a set of refinement operators that preserve several possibilistic properties and that use knowledge about their proofs over the specification [Man01] (this work also contains a thorough list of references on the refinement problem).

The possibilistic definitions of confidentiality are unable to detect the kind of covert channels that can be established by an attacker able to alter and measure variations in probabilistic behavior [WJ90]. Several probabilistic models have

been suggested: p-Restrictiveness [Gra90], Flow Model [McL90], and Applied Flow Model [Gra92]. The need to quantify over all probabilistic distributions of environment behavior makes the verification of these properties unwieldy. For that reason, Gray and Syverson proposed an epistemic logic with modalities for time, knowledge and probability to reason about probabilistic noninterference [SG95].

Another deficiency of possibilistic secrecy properties concerns their treatment of time. In most cases, time is just modelled as the causal succession of events, which lacks the details necessary to detect timing covert channels. All the same, it is expected that the ideas of probabilistic noninterference could be seamlessly carried to richer models of timed computation. In Section 2.3 we consider some research on eliminating timing channels from programs.

As the last paragraph suggested, the models used to represent computational behavior are of extreme importance in the definition of security properties. For example, initial presentations of information flow properties made use of ad-hoc models, not making clear their implications. Later approaches use labelled transition systems and process algebraic terminology.

Moskowitz and Costich recasted some possibilistic information flow properties in terms of automata, without considering branching structure and termination [MC92]. Mantel has devised a kit to express and compare different trace-based possibilistic information flow properties [Man00].

Using a labelled transition semantics, Focardi and Gorrieri were able to present different properties and compare them under a unifying model. *Security Process Algebra* (SPA) is a version of Milner's CCS [Mil89] extended with the hiding operator of CSP (" $/$ "), and where input and output actions are further divided into high-level ( $Act_H$ ) and low-level ( $Act_L$ ). Focardi and Gorrieri presented *Non Deducibility on Compositions* (NDC) as "probably the most meaningful security property in a process algebraic setting" [FG95, p. 20]. A process  $E$  satisfies *NDC* if and only if  $E / Act_H \approx_T E \setminus Act_H$ , where  $\approx_T$  is trace equivalence.

The same authors have proposed using other notions of equivalence in their definitions of confidentiality, arguing that trace equivalence (which is not deadlock sensitive) is too weak. Ryan and Schneider [RS99] have noted that "the notion of noninterference depends ultimately on our notions of process equivalence," and have studied definitions of noninterference for CSP. Abadi and Gordon, initially in the context of protocol analysis and spi-calculus [AG98b], have suggested a bisimulation-based model of secrecy [AG98a]. Lowe analyzed several shortcomings of previous definitions of information flow, and suggested a new definition over a model of CSP that discriminates more behaviours than the standard ones [Low99].

### Covert Channel Analysis

Most noninterference properties try to cancel all covert channels. However, it is widely accepted that these properties are overrestrictive and that, in practice,

covert channels are unavoidable. The alternative is to analyze and measure the covert channels present in an existing system.

In [Mil99], J. Millen gives a general account of the last two decades of covert channel analysis. He divides the subject into methods to: model, identify, measure and mitigate covert channels.

Several of the methods used to identify covert channels stemmed from the work on information flow, among them tools like: the MITRE flow analyzer, the Gypsy Flow Analyzer (part of the Gypsy Verification Environment), EHDM (which uses the SRI model, conceptually similar to Bell-La Padula's), FDM/-MLS and FDM/SRM. Most of these tools analyze formal specifications, rather than code. For example, FDM/MLS and EHDM analyze specifications written in Ina Flo and Revised Special, respectively.

Once covert channels are identified, they are measured. Some are considered innocuous because, for example, they transfer information from a user back to the same user, or because their capacity is low enough. Millen showed that for deterministic systems, noninterference implies that the system, when viewed as a channel, has zero capacity [Mil87]. He even suggested how to measure the capacity of an interfering system. Although most of the methods relate to information theory, Moskowitz and Kang have pointed out that channel capacity is not the right way to go about measuring covert channels [MK94]. A channel that transmits a bit at times  $1, 2, 4, \dots, 2^n, \dots$  has capacity zero, but can leak a considerable amount of information in the short term.

Finally, the effect of those covert channels which are considered to be harmful can be mitigated in different ways: (1) modifying the system so that these channels are eliminated, (2) delimiting their bandwidth by deliberately introducing noise and delays (e.g. "fuzzy time"), and/or (3) auditing the channel, so as to discourage their use.

### Intransitive Noninterference

Looking back at the two scenarios presented in the introduction, it may seem that information flow properties are not appropriate to handle downgrading. In an MLS system, encrypting a high-level value (thus reclassifying its as low-level) constitutes a clear violation of information flow. Intransitive noninterference [Rus92, RG99], where selected subjects are given the ability to downgrade information, is not satisfactory as it does not control *what* is downgraded in this way. *Admissibility*, described in this thesis, proposes a solution to this problem. In connection with this, Ryan and Schneider have recently presented a generalization of noninterference in the context of CSP to handle downgrading [RS99].

## 2.2 Confidentiality for Security Protocols

The particular features of security protocols, in particular their reliance on cryptographic operations to achieve secrecy, integrity and authentication, have

kept them apart from the theories of confidentiality for computer systems. It is convenient to take a look at the diverse methods used to analyse these protocols.

In general, the methodological approach corresponds to the “eavesdropping” view, as described in the introduction. The idea is to model the capacity of the adversary to extract useful information by tampering with the public communication channels employed by the protocol participants.

Research in this area can be divided into two main groups. Section 2.4 lists some attempts trying to relate them.

### 2.2.1 Formal Models

These approaches find their common denominator in the work of Dolev and Yao [DY83], which makes two fundamental assumptions:

1. Nondeterministic adversary: An adversary may attempt any possible attack, it is in control of the communication network, and it is able to spy, kill and fake messages. A protocol is considered secure if no possible interleaving of actions results in a security breach.
2. Perfect cryptography: Cryptographic operations are seen as functions on a space of symbolic (formal) expressions. Their security properties are also modelled formally. For example, an attacker that did not have access to a key cannot obtain any information from a ciphertext.

Formal approaches include specialized logics, special-purpose tools for cryptographic protocol analysis, model-checking and theorem proving. They all aim at designing some high level language where security systems can be both expressed and analyzed formally.

Most of the achievements of these methods reside in finding deficiencies in protocols, even automatically. They treat systems of increasing complexity and provide high-level reasoning. They have also permitted the extraction of general principles for the construction of new protocols. On the down side, they cannot give a totally convincing argument for protocol correctness. Protocols can be verified using these approaches, but still contain security holes (which lie outside the assumptions in the Dolev-Yao model).

Millen’s *Interrogator* and Meadows’s *NRL Protocol Analyzer (NPA)* were among the first tools to exploit these ideas. They would describe the protocol as a set of Horn clauses and then search backwards for contradiction starting from possible errors.

Kemmerer used Ina Jo (see previous section) to model cryptographic protocols in a conventional specification language, thus establishing an early link to the work on security of computer systems.

More recent uses of model checkers include: CSP/FDR [Low96] and Mur $\phi$  ([MMS97]). In principle, model checkers can only be used to detect confidentiality breaches in protocols, not to prove them correct. The reason being the

need to consider, in principle, an unbounded number of participants and protocol runs, as well as infinite data and message spaces. In order to improve on the generality of the results obtainable with model checking tools, current research is devoted at finding appropriate lower bounds for all the aspects mentioned above. The work on data independence by Roscoe and Lazic [Ros98] constitutes a good example of this.

Paulson's Inductive Method [Pau98a] relies on theorem proving to handle an unbounded number of participants and an unlimited message space. Confidentiality, like in the previous approaches, is described as a safety property, usually stating that the intruder never learns a specific value. Paulson's method consists in proving that this property is satisfied by all reachable states. The work on Strand Spaces [FHG98] is closely related to that of Paulson's: A strand is a sequence of events representing either the execution of a legitimate agent or else of the attacker. A strand space is a collection of strands together with a graph structure reflecting causality between events in the strands. Proofs of confidentiality in the framework of strand spaces rely on some sort of inductive principle over the graph structure. Millen and Ruess [MR00] have recently improved this approach, by identifying and proving protocol-independent results. Using their techniques, the proofs of secrecy of some protocols become simple enough so as to be carried out manually. The strand space model is also used by the Athena tool [Son99].

Focardi et al. have used *NDC* to express authentication [DFG99], non-repudiation [FM99a], and secrecy [FGM00], even in the presence of cryptographic operators. Their tool, CoSeC [FG97], is based on *CryptoSPA* (an extension of *SPA* with cryptographic operators and deduction rules à la Dolev-Yao), and can be used to verify protocols using *NDC* based properties.

Abadi and Gordon's work on the spi-calculus [AG98b] uses a more stringent notion of secrecy, related to the work on information flow models. Suppose that the protocol is represented as the parameterized process  $P(x)$ . Then, secrecy of  $x$  means that  $P(M)$  and  $P(N)$  are equivalent, for all possible values  $M$  and  $N$  for  $x$ . Equivalence is testing bisimulation, modified to handle cryptographic operations.

The work on *SPA* and spi calculus, has shown a connection between confidentiality for protocols, and confidentiality for secure systems. Both employ a stronger notion of secrecy than that in previous approaches. Contrary to *SPA*, there is no need to represent the intruder explicitly in the spi calculus.

### 2.2.2 Computational Models

There are other ways of modelling the adversary of a security protocol. Shannon applied information theory to analyze the case of an adversary with unlimited computational resources. For this reason, this model is usually called *unconditional security*. In the case of encryption, where unconditional security is called *perfect secrecy*, there are two main results: First, that for symmetric-key encryption the key must have as many bits as the entropy of all messages to transmit.

An example of such a system is the *one-time pad*. Second, that public-key encryption **cannot** be unconditionally secure: a computationally-unbounded adversary can encrypt one plaintext after the other with the public key, till it finds the one that produces the right ciphertext.

### Complexity-Theoretic Security

Modern security drops the assumption of an all-powerful adversary, and moves to consider whether there is a feasible attack on a system, instead of whether there is a possible one. Most of the approaches belonging to this group consider adversaries which have complexity bounded resources. These approaches also present the following common basic structure:

1. A definition of the computational power of observation of any adversary, and
2. a notion of what is distinguishable and what is not distinguishable by any such adversary

The complexity of an attacker is usually measured relative to a security parameter  $k$ , fixed at the time the cryptographic system is set up. For example,  $k$  is commonly linked to the size of keys in an encryption scheme. It is usual, though not essential, to model adversaries by probabilistic polynomial-time Turing Machines. It is possible to use other conventions, as long as the notions of efficient and feasible computation are sufficiently robust and rich.

For each value of the security parameter  $k$ , we can consider the capacity of an adversary (computing on time polynomial over  $k$ ) to distinguish two probability distributions. A set of probability distributions (one for each value of the security parameter) is called an *ensemble*.

Given  $X = \{X_n\}$  and  $Y = \{Y_n\}$ , two probability ensembles, such that  $X_n, Y_n$  range over strings of length  $n$ , the advantage of adversary  $A$  is defined as

$$d_A(n) \triangleq |Pr(A(X_n) = 1) - Pr(A(Y_n) = 1)|$$

The advantage indicates the ability of  $A$  to distinguish the two ensembles. If  $d_A(n)$  is *negligibly* small for each value of  $n$  and for every possible computationally bounded adversary  $A$ , then  $X$  and  $Y$  are computationally indistinguishable (A function  $\nu$  is *negligible* if for every constant  $c \geq 0$  there exists an integer  $k_c$  such that  $\nu(k) < k^{-c}$ ,  $\forall k \geq k_c$ ).

In complexity-theoretic security, cryptographic methods are designed taking the weakest assumptions on the adversary. Their effectiveness is assessed using asymptotic analysis (running time as a function of the security parameter  $k$ ) and worst-case analysis. The results in this area establish the plausibility or not of attacks.

At first sight, these analysis seem to be merely qualitative, and the evaluation of its results requires special care. However, this needs not be the case,

as the following section shows. Complexity-theoretic models provide a clear set of mathematical principles against which more practical approaches can be compared [MvOV97, p. 43].

### Provable Security

A cryptographic method is said to be *provably secure* if the difficulty of defeating it can be shown to be essentially that of solving a well-known and supposedly difficult problem (typically a number-theoretic problem such as integer factorization or the computation of discrete logarithms). This is a reductionist approach [GM84b] which presents similarities with the reduction methods used Complexity Theory. Most proofs provide a precise construction of the breaking adversary for the simpler primitives, starting from an adversary that can break the protocol. Once instantiated for any desired value of the security parameter, a concrete estimate of the difficulty of breaking the protocol can be obtained relative to the difficulty of breaking its primitive components.

The problem with Provable Security is that the functions that are taken as primitive tend to be as hard to implement as the functions that are derived from them. The protocols that result are usually considered inefficient. For example, it is known that pseudo random functions can be constructed using more primitive one-way functions like RSA, but in practice this is hardly done this way.

### Practical security: the Random-Oracle Model

One of the objectives of modern cryptography is to develop a basic toolset of cryptographic primitives which would allow the development of more complex and elaborate algorithms. At the center of all of them lies the notion of a one-way function (OWF): A function that is “easy” to compute but infeasible to invert. OWFs provide the basis for good encryption techniques. OWFs with trapdoors provide the foundations for public-key cryptography.

However, OWFs are perhaps too low-level to construct effective high-level algorithms. In practice, several very efficient cryptographic algorithms in use today have been designed starting from the assumption that certain block ciphers are secure (while, in most cases, this claim has never been proved). What is then a good theoretical model of the security of a block cipher?

The Random-Oracle Model [BR93] combines complexity-theoretic and provable security for the analysis of practical protocols, including those using block ciphers. The idea is to consider an idealized version of the protocol where the intervening parties have access to a common random function. Of course, the idealized protocol is far from realizable, as random functions are far too big to store. Using provable security arguments, it is then possible to estimate the security of the protocol w.r.t. the security of the implementation of a pseudorandom function. By complexity-theoretic methods, it is then possible to approximate the maximum advantage a feasible (i.e. computationally bounded)

adversary can achieve over the pseudorandom function. Putting both results together, an estimation of the security of the real protocol is obtained.

In the case of block ciphers, these are a fixed-key permutation functions, which can be modelled as families of finite pseudo-random functions  $\{F_K\}$ , indexed by small keys. It is known that these functions can be obtained from pseudo-random number generators, which, in turn, have been proved to exist if there exist OWFs. If  $K$  is shared between principals  $A$  and  $B$ , and we use  $F_K$  in place of a random function  $f$  in some scheme designed in the Random-Oracle Model, then the resulting scheme is still secure as long as the adversary is computationally bounded.

### 2.3 Programming-Language Confidentiality

Access control models are usually implemented by means of a reference monitor whose function is to check, at runtime, that every access request complies to a security policy. In practice, however, access control models are incapable of eradicating all covert channels. Dorothy and Peter Denning were the among the first to suggest program analysis techniques to eliminate insecure information flows [DD77]. Their analysis uses a MLS lattice [Den76], and makes sure that the security level of each variable always dominates that of the data that is assigned to it. At the time it was published, the analysis lacked a formal proof of soundness.

Andrews and Reitman proposed a data flow analysis that superimposes a set of variables and their updates onto the program to analyze [AR80]. For each variable  $x$  in the original program, the analysis uses a variable  $\underline{x}$  to record its security class. Furthermore, two extra variables keep track of indirect dependencies *within* and *between* statements in sequential programs, as well as dependencies through process synchronization in parallel programs. An axiomatic semantics of the resulting program is used to state confidentiality, and a simplified deductive logic (concerning only class variables) is proposed to verify them. Neither in this case have the authors provided a proof of soundness relating the logic to the semantics.

Cohen studied dependencies in a functional execution model from an information theory perspective [Coh77]. From this semantical characterization of security, he derived a set of proof rules to verify the absence of information paths in simple sequential programs [Coh78].

Mizuno and Schmidt's verification method takes the form of an abstract interpretation defined over the denotational semantics of a procedural language. It consists of a compile-time analysis, that addresses intraprocedural flows, and a link-time analysis, that considers interprocedural flows. The abstract interpretation is proved correct with respect to an *instrumented semantics* where every value is annotated with its security class.

Volpano, Smith and Irvine were arguably the first to give a proof of soundness of Denning's analysis using a standard (i.e. not instrumented) semantics [VSI96].

This analysis takes the form of a type system reflecting the security levels of variables in simple imperative programs. In later instances, Volpano and Smith have extended their analysis to cope with thread synchronization [SV98], probabilistic schedulers [VS98a] and limited downgrading [VS00, Vol00].

Since then, several researchers have designed type systems for confidentiality. In the *SLam calculus*, Heintze and Riecke define and prove soundness of a type system for a kind of  $\lambda$ -calculus that expresses both access control and information flow [HR98]. Myers and Liskov proposed a quite fine-grained labelling system that records the owners of the data contributing to each value, as well as the readers these owners are willing to give access to. Although this takes the form of combined static and dynamic type systems, these authors have not provided formal proofs of soundness of these systems [ML98, Mye99]. Abadi also proposed a type system to guarantee secrecy of security protocols written in the spi-calculus [Aba97].

With a few exceptions, notably Cohen's work, most initial approaches to programming-language confidentiality had a syntactic nature, mainly taking the form of control flow analyses and type systems. This is now changing. For example, Leino and Joshi suggested a semantical characterization of secure programs which is general enough to be applied over different semantic formalisms [LJ00]. In their examples, they use relational semantics and a Hoare-logic to simplify the verification task. As a limitation, their equational characterization assumes a functional view of programs, where high-level data cannot be modified by the environment during execution.

Sabelfeld and Sands' denotational semantics-based characterization of non-interference can express both Cohen's and Joshi and Leino's definitions [SS99]. These ideas were later carried over to unlabelled probabilistic transition systems. Using these systems as semantical basis, they defined scheduler-independent confidentiality properties for a multithreaded programming language [SS00]. This language is extended with synchronization primitives in [Sab01].

Agat enriched the operational semantics of a simple imperative programming language to reflect the timing behavior of programs and thus study their timing covert channels. His approach is exceptional in that it not only detects this kind of covert channels, but also indicates how to modify a program to achieve timed noninterference [Aga00].

So far, all work cited refer to models that express and verify confidentiality properties of general programs. For programs implementing security protocols, there is not much previous work that the author knows of besides that of El Kadhi. He devised an abstract interpretation analysis of protocol implementations [Kad01, BK01] that estimates the attacker's knowledge using Bolignano's instantiation of the Dolev-Yao assumptions [Bol97]. El Kadhi's analysis does relate the properties of the code to the properties of the protocol. Instead, confidentiality properties are verified analyzing only the code.

## 2.4 Relations between Secrecy Models

The effort put into using standard languages together with their semantical models to recast and compare previous approaches to confidentiality is beginning to pay off. Not only have they proved useful to compare different pieces of work within each of the research areas described in the previous three sections, but also are they hinting towards connections between these areas.

Lincoln et al. [LMMS99] have given a process algebraic model of *Provable Security*. In this model, the security of a process (that represents a protocol implementation) is related to that of an idealized version of the protocol using complexity-theoretic arguments. Pfitzmann, Schunter and Waidner [PSW00] have studied these ideas (that they identify as the *simulatability* approach) and their relation to formal models (see Section 2.2.1). Similarly, Abadi and Rogaway have investigated the relation between formal and computational models [AR00]. Their results give a semantical justification of some of the Dolev-Yao assumptions, in particular those concerning the abstraction of cryptographic operations and of values (e.g. nonces and keys). A good by-product is the discovery of extra restrictions on the formal models that help clarify their meaning. Abadi and Jürjens have extended these results to whole processes, although the modelling language used is not standard [AJ01].

While the previous references illustrate a currently growing interest to combine and relate the different models of confidentiality for security protocols, they also point towards a convergence with programming-language techniques. This is so because some of the formalisms used to model protocols can also be used to give semantics to programming languages.

Finally, there is a strong connection between programming-language confidentiality and information flow models. In general, the language-based methods try to realize in practice concepts and models derived from information flow theories. As we have seen, the connection between theory and practice has not always been formally pursued. However, recent work in the semantical foundations of program security has facilitated the study of its relation to information flow models [MS01].

## 2.5 Mobile Code Security

In the context of mobile code, new general security issues arise in connection to the risk of exposing local systems to untrusted foreign code [Che98, VS98b, Dea97]. The success of *Java* has stirred interest in formalizations of its Bytecode Verifier, a part of the Java Virtual Machine which uses static (data flow) analysis techniques to enforce type safety [FM99b, Qia99, SA98, Gol97, BS98, JMT97]. How to extend the verification of JVM and other mobile code to these domains is a matter of current research. Some techniques point at developing rich type systems at the level of assembly code that can be used to easily verify that the code abides to certain resource usage policies [SMH00, MWCG98]. Others

---

propose combinations of runtime checks and static type systems to enforce safety properties on the mobile code [Koz99]. The idea of Proof-Carrying Code is, instead, to place the verification burden on the code producer, and in principle can handle more than just safety properties [Nec97, NL98b, NL98a]. In fact, these are only a few samples of a research area currently experiencing furious activity.



## Chapter 3

# A Model for Cryptographic Processes

A good model simplifies the analysis, but it never abstracts away crucial features from the object of study. Good models also provide generality, with the hope that results should carry over to other, slightly different settings.

In this chapter we model, at a quite high level of abstraction, the kind of applications we are interested in. They consist of down-loadable programs that implement cryptographic protocols (thus they should have access to various cryptographic primitives), and that probably interact with the host where they execute by invoking various local procedures.

To model the execution of our applications, we use labelled transition systems; to represent them compactly, a simple process algebra. There are several advantages in this approach: Labelled transition systems constitute a common semantical basis for describing an operational semantics and have thus been applied to a number of different programming languages and paradigms. They can model concurrent processes together with diverse synchronization mechanisms, all at a convenient level of abstraction. Moreover, the simplicity of the model helps identify precisely what can be observed of a system, a quality that has been appreciated in previous formal models for security. Finally, labelled transition systems can be extended quite easily in various directions, to include more information about the execution of a system, like probabilistic distributions and time. Regarding the use of a process algebra, this is not new either: several research efforts have taken that direction before (e.g. [FG95, AG98b, RS99]).

In the first section of this chapter, we present a simple process algebra, an extension of Milner's Calculus of Communicating Systems (CCS) with cryptographic operations, much inspired by work on the spi-calculus [AG98b] and CryptoSPA [FGM00]. Two peculiarities set our approach apart, though. First, we introduce a prefix operator for invoking local functions so that we avoid modelling them in the algebra. Second, not only is the evaluation of data kept

at a rather symbolic level, but also the structure of values is enriched to reflect the history of operations performed to produce them. The latter is specially important to be able to track and identify correct leaks of information.

After illustrating our process algebraic model with the aid of some examples, we extend it with annotations, where secret values are identified and tracked. This gives further precision to determine (see the next chapter) whether a process respects a confidentiality property. The chapter concludes by relating the annotated and non-annotated algebras.

### 3.1 Security Process Algebra

Starting in this chapter and onto the next one, the concept of secret plays a central role in our modelling decisions. In general, a secret is a piece of information. Shannon defined information as “that which reduces *uncertainty*.” The generality of this definition indeed suggests that a secret can take many forms. In this work, we pay special attention to secret data, where data can be understood as “material units of information about a portion of the real world that can be processed by explicit procedures and maintains its characteristics during repeated use” (Krippendorff, Principia Cybernetica Project).

A computer system operates on data values. We are interested in systems that input secret data, e.g. a credit card number, and then modify their output and behavior according to its value. How can we identify a secret value as it is processed by a system? At the moment of input, it seems easy to identify secret data. However, in most other occasions this is not that obvious. As an example, suppose that variable  $x$  is known to contain a secret even number. Then, the value that results from evaluating  $x \bmod 2$  does not contain any information, as knowing it does not reduce any uncertainty. Although  $x$  represents a secret value,  $x \bmod 2$  does not. Most approaches to confidentiality would start by assigning a high security level to  $x$  and then just do the same for  $x \bmod 2$ , which is a safe loss of precision (cf. [DD77, VSI96, SS01]). On the other side, it is clear that knowing how a value was constructed from input values (in the example, by computing a remainder upon division by 2) represents an important step towards identifying secret data. The values we consider in our process algebra reflect therefore their construction.

**Expressions and Values** Let  $x$  range over a set of variables, and  $k$  over a set of constants, like integers and booleans. The set of expressions is then given by the following grammar.

$$(Expr) \quad e ::= x \mid k \mid (e_1, \dots, e_n) \mid \{e_1\}_{e_2} \mid \{|e_1|\}_{e_2} \mid \pi_i(e)$$

If  $n = 0$  in  $(e_1, \dots, e_n)$  the resulting expression is a constant usually called *unit*. The expression  $\{e_1\}_{e_2}$  indicates the encryption of  $e_1$  with key  $e_2$ , while  $\{|e_1|\}_{e_2}$  indicates decryption. Finally,  $\pi_i(e)$  indicates the projection of the  $i$ -th coordinate of  $e$ . The choice of operators is deliberately limited, for they

represent primitive operations in our language for processes. These limitations will be dropped later, by means of the use of local functions.

A value, ranged over by  $v$ , is a ground expression (i.e. an expression without variables). We call  $Val$  the set of values obtained by choosing a certain fixed set of constants.

Boolean expressions, ranged over by  $b$ , are used to describe branching conditions:

$$(BoolExpr) \quad b ::= x \mid \mathbf{true} \mid \mathbf{false} \mid e_1 \leq e_2 \mid e_1 = e_2 \mid \neg b \mid b_1 \vee b_2$$

We can now proceed to describe the syntax and semantics of our process algebra, which we call *SecPA* (to avoid confusion with Focardi and Gorrieri's SPA), as an extension of Milner's value-passing CCS. Of course, we replace Milner's set of values  $\mathcal{V}$  with our  $Val$ .

**Syntax** Given a set  $Ch$  of channel names, ranged over by  $c$ , *SecPA* process terms are given by the grammar:

|            |                                    |                                       |
|------------|------------------------------------|---------------------------------------|
| $P, Q ::=$ | $0$                                | nil                                   |
|            | $\tau.P$                           | internal action                       |
|            | $c!e.P$                            | output $e$ on channel $c$             |
|            | $c?x.P$                            | input from channel $c$                |
|            | $x := c(e).P$                      | call to local function $c$            |
|            | $\sum_{i \in I} P_i$               | a summation, $I$ an indexing set      |
|            | $P \mid Q$                         | parallel composition                  |
|            | $P \setminus L$                    | restriction, $L \subseteq Ch$         |
|            | $P / L$                            | hiding, $L \subseteq Ch$              |
|            | if $b$ then $P$                    | conditional, $b$ a boolean expression |
|            | let $(x_1, \dots, x_n) = e$ in $P$ | tuple matching                        |
|            | case $e_1$ of $\{x\}_{e_2}$ in $P$ | shared-key decryption                 |
|            | $C(e_1, \dots, e_n)$               | constant of arity $n$                 |

For each constant  $C$  with arity  $n$  there is a defining equation

$$C(x_1, \dots, x_n) \triangleq P$$

where the right-hand side  $P$  may contain no free value variables except those listed to the left, i.e.  $x_1, \dots, x_n$  (all distinct variables).

We have drawn inspiration from SPA [FG95] to include the hiding operator from CSP, and from the *spi-calculus* in the case of the **let** and **case** constructs. A **let** process is used to deconstruct a tuple, while a **case** is used to decrypt an encrypted value.

**Local Function Calls** *SecPA* contains an operator not present in either SPA nor the *spi-calculus*: the local function call prefix. It represents a simple search-and-load operation. For example, we can write  $k := \text{key}(p)$ , where  $k$  is the key associated to principal  $p$ , to model the fetching of keys from a local key-store. In a way, this operator represents an atomic combination of an input and an output prefix. Local function calls are meant to abstract away services provided by the trusted computing base (TCB). This is particularly important for analysis, since the TCB would normally consist of a quite considerable amount of code, most in the form of library methods whose properties can be studied independently of code downloaded from untrusted sources.

**Decryption** *SecPA* represents the operation of decrypting a ciphertext by means of the **case** construct. This is the way decryption is modelled in the *spi-calculus*. The *spi-calculus* is a version of the  $\pi$ -calculus extended with a richer name structure (which includes domains for keys, plaintext and ciphertext) and a special syntactic construction to handle decryption. The authors have argued that the syntactical approach is simpler and more effective than a direct encoding (of keys and encrypted data) using only the primitives in the  $\pi$ -calculus [AG98b, Appendix A]. However, this issue is still in discussion.

For the processes we are interested in modelling there is no need for the power of the restriction operator. At the same time, we have found it convenient to keep channel names separated from data. For the sake of simplicity, we discarded using the  $\pi$ -calculus or any other calculus of mobility. In view of which, we did not have to choose whether to encode encrypted data or represent it syntactically. We faced another choice instead: whether to use the **case** construct, or to use the  $\{ \_ \}_\_$  expression. That is, since our expressions already contained an operator for decryption, why then introduce a syntactic construct for decryption?

The reason is mainly aesthetic. Any decent encryption scheme should introduce enough redundancy in its ciphertexts so that attempting a decryption with a wrong key results in some kind of error notification. If we wanted to represent decryption with an expression, we would have been forced to encode the redundancy, or at least the error detection mechanism. That would have unnecessarily obfuscated our examples.

Finally, it should be said that we could have modelled decryption as a local function call, e.g. as  $x := \text{dec}(e_1, e_2)$ . This would have made decryption observable (unless we hid channel *dec*), which in some contexts might even be desirable (for example, if the timing behavior of the cryptographic function can be exploited to establish a covert channel [Koc96]). However, this would suffer from the same problems as the approach based on decryption expressions.

**Public-Key Encryption** In [AG98b, Chapter 7], Abadi and Gordon show how to add hashing functions, public-key encryption and digital signatures to their calculus. The same can be done here. In the case of public-key encryption,

we need to assume the existence of a function over the domain of keys that given a public key  $x$  returns its private counter-part  $x^{-1}$ . We can then add to the process definition

$$P, Q ::= \dots \\ [(e_1, e_2) \vdash_{dec} x]P \quad \text{public-key decryption}$$

this time inspired by the notation in CryptoSPA [FGM00] (in the *spi-calculus*, the suggested notation is “case  $e_1$  of  $\{|x|\}_{e_2^-}$  in  $P$ ” which is confusing because  $e_2^-$  represents the private key used to encrypt  $x$ , while what is needed for decryption is  $e_2^+$ , the public key). Other notations are possible. We could have use, for example, “case  $x$  of  $\{|e_1|\}_{e_2}$  in  $P$ ” or “let  $x = \{|e_1|\}_{e_2}$  in  $P$ ”, but this is just a matter of taste.  $\square$

**Transitional Semantics** The semantics of *SecPA* is given in terms of labelled transition systems in the standard way: states are process terms, and the transition relation is defined as the smallest relation satisfying the rules below.

$$\begin{array}{c} \text{IN} \frac{}{c?x.P \xrightarrow{c?v} P[v/x]} \qquad \text{FUN} \frac{}{x := c(v_1).P \xrightarrow{v_2 := c(v_1)} P[v_2/x]} \\[10pt] \text{OUT} \frac{}{c!v.P \xrightarrow{c!v} P} \qquad \text{TAU} \frac{}{\tau.P \xrightarrow{\tau} P} \qquad \text{COM}_1 \frac{P \xrightarrow{\alpha} P'}{P | Q \xrightarrow{\alpha} P' | Q} \\[10pt] \text{COM}_2 \frac{Q \xrightarrow{\alpha} Q'}{P | Q \xrightarrow{\alpha} P | Q'} \qquad \text{COM}_3 \frac{P \xrightarrow{c!v} P' \quad Q \xrightarrow{c?v} Q'}{P | Q \xrightarrow{\tau} P' | Q'} \\[10pt] \text{REST} \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L} (subj(\alpha) \notin L) \qquad \text{COND} \frac{P \xrightarrow{\alpha} P'}{\text{if } b \text{ then } P \xrightarrow{\alpha} P'} (b = \mathbf{true}) \\[10pt] \text{HID}_1 \frac{P \xrightarrow{\alpha} P'}{P/L \xrightarrow{\alpha} P'/L} (subj(\alpha) \notin L) \qquad \text{HID}_2 \frac{P \xrightarrow{\alpha} P'}{P/L \xrightarrow{\tau} P'/L} (subj(\alpha) \in L) \\[10pt] \text{LET} \frac{P[\pi_1(v)/x_1, \dots, \pi_n(v)/x_n] \xrightarrow{\alpha} P'}{\text{let } (x_1, \dots, x_n) = v \text{ in } P \xrightarrow{\alpha} P'} (\exists v_1, \dots, v_n : v = (v_1, \dots, v_n)) \\[10pt] \text{CASE} \frac{P[\{|v_1|\}_{v_2}/x] \xrightarrow{\alpha} P'}{\text{case } v_1 \text{ of } \{x\}_{v_2} \text{ in } P \xrightarrow{\alpha} P'} (\exists v : v_1 = \{v\}_{v_2}) \qquad \text{SUM}_j \frac{P_j \xrightarrow{\alpha} P'_j}{\sum_i P_i \xrightarrow{\alpha} P'_j} \end{array}$$

$$\text{DEF} \frac{P[\tilde{e}/\tilde{x}] \xrightarrow{\alpha} P'}{C(e_1, \dots, e_n) \xrightarrow{\alpha} P'} (C(x_1, \dots, x_n) \triangleq P)$$

As usual, the subject of a non-silent action  $\alpha$ , written  $\text{subj}(\alpha)$ , is the channel actually used in the communication. The equality,  $=$ , is semantical and thus presupposes an evaluation of the terms under comparison. How this term evaluation is done is not important here, although it is expected to satisfy properties like  $\pi_1(v_1, v_2) = v_1$  and more or less precisely reflect the concrete cryptographic functions in use. For example, in the case of RSA, it is expected to verify  $\{|\{v_1\}_{v_2^{-1}}|\}_{v_2} = v_1$ .

For completeness, we give the rule for public-key decryption:

$$\text{PK-DEC} \frac{P[\{|\{v_1\}_{v_2}/x|\} \xrightarrow{\alpha} P']}{[(v_1, v_2) \vdash_{\text{dec}} x]P \xrightarrow{\alpha} P'} (\exists v : v_1 = \{v\}_{v_2^{-1}})$$

The choice of an appropriate model is a delicate one. Besides handling cryptographic primitives, local function calls and values, it must permit the encoding of our applications of interest.

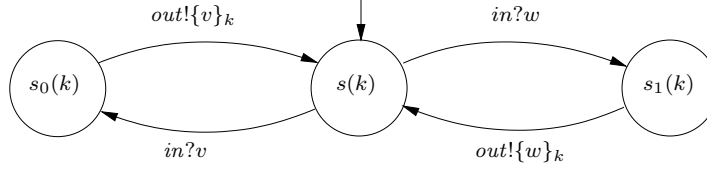
As a little warm-up, consider a very simple protocol: in each run of the protocol, a value  $x$  is to be read from channel *in*, and then transmitted, encrypted with secret key  $k$ , along channel *out*. It is understood that the confidentiality of  $x$  is to be preserved even when the protocol admits that the output might depend on the input. This is perfectly ok. For example, under the assumption of perfect cryptography, an observer of channel *out* cannot discover which value was actually input, provided that it is not able to decrypt  $\{x\}_k$ .

The formalization of this requirement will be discussed in the next chapter. For the moment, we take a look at three different possible implementations of our simple protocol. We are only interested in showing how these examples can be encoded in *SecPA*, and what their semantics are. The analysis of each example will be done somewhat later.

**Example 3.1 (Encrypter)** *Our first implementation follows the protocol to the letter. As expected, it reads a value from input channel in, and transmits it encrypted with secret key k along channel out, before restarting. Taking the encryption key as a parameter, we can write this as a SecPA process:*

$$s(k) \triangleq \text{in}?x. \text{out}!\{x\}_k. s(k)$$

*The labelled transition system induced by its semantics is schematically depicted by the following figure:*

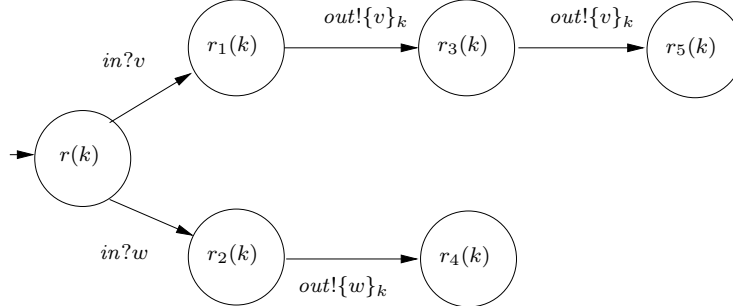


where the arrow head indicates initial states (one for each value of  $k$ ), and where we have decided to represent the input of only two different values,  $v$  and  $w$ .

**Example 3.2 (Bad Encrypters)** Forgetting for a moment that we are dealing with a very trivial protocol, suppose that two code providers offer implementations  $r(k)$ , resp.  $t(k)$ . Each provider promises that their code correctly implements our naive protocol. For our part, we are not really interested in whether their codes fully implement the protocol. We want instead to know that the confidentiality properties of the protocol are at least preserved. Take a look at the two pieces of code now:

Bad Encrypter #1:  $r(k) \triangleq in?x. out!\{x\}_k. \text{if } x = v \text{ then } out!\{x\}_k. 0$

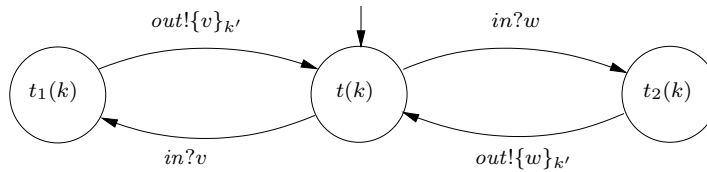
The associated labelled transition system is again depicted schematically, by considering only two distinct values:



This process encrypts its input before transmitting it, but an observer can learn whether the input was  $v$  by counting the number of output messages.

Bad Encrypter #2:  $t(k) \triangleq in?x. out!\{x\}_{k'}. t(k)$

with associated labelled transition system:



*This process changes to a different encryption key  $k'$ , which is another simple way of defeating confidentiality (specially if the key  $k'$  is known to the observer of channel out).*

These examples, though very simple, should have given an idea of how we plan to use *SecPA*, and at the same time, which kind of problems we try to avoid with our techniques.

The following two sections illustrate the use of *SecPA* with more elaborate and somewhat more realistic examples, and with more complex protocols.

### 3.1.1 The Purchasing Applet Examples

We recall one of the scenarios from the introduction (see p. 1), a piece of code (applet) that is downloaded to make a payment through the Internet. The protocol we want this applet to implement is a variant of the 1-Key-Protocol (1KP), an electronic payment protocol [BGH<sup>+</sup>95].

The 1KP protocol involves three players: A Customer, a Merchant and an Acquirer. The Customer possesses a credit card and a PIN, with which it places an order to the Merchant. The Acquirer is a front-end to the existing credit card clearing/authorization network, that receives payments records from merchants and responds by either accepting or rejecting the request.

The purchasing applet in our scenario implements the Customer's side of the 1KP protocol, which we have lightly simplified in order to keep the presentation at a reasonable level of detail.

According to the protocol, the applet should request from the customer, among other information:

1. the name *acq* of the Acquirer,
2. ordering information *order* (item, price agreed, delivery, date and time, etc.), and
3. accounting information *acc* (account number, expiry date, PIN code), intended for *Acquirer*.

Then the applet should consult the TCB for the public key  $K$  of the Acquirer and send the order together with a purchase slip to the Merchant:

$$(acq, order, \{order, acc\}_K)$$

The Merchant passes the slip  $\{order, acc\}_K$  to the Acquirer who can then validate the purchase. The Acquirer returns a notification to the Merchant indicating whether the purchase was cleared. If cleared, the Merchant passes the goods together with an invoice to the Customer.

The remainder of this section is dedicated to illustrate various possible implementations of the purchasing applet and their encoding in *SecPA*. In doing

so, we deliberately adopt a very lax notion of implementation. Since protocol descriptions leave usually plenty of room for the implementor to fill in (an important feature), in a vast medium like the Internet we cannot simply pretend that the implementor abides to the limitations of any refinement based methodology. Moreover, it is well-known that most confidentiality properties are not preserved by refinement; which means that even if the implementor had derived the code from an appropriately formalized specification of the protocol, that alone would not suffice to guarantee confidentiality. In Chapter 4 we address the issue of how to specify the confidentiality requirements behind an informal protocol description and what it means that the protocol fulfills those requirements.

We assume that the applet is given the Acquirer's name through channel *acq*, the Customer's account info through channel *acc*, and the details of a particular buying order through channel *order*. We interpret  $\{ \_ \}_k$  as public-key encryption, and use the local function *PubKey()* to recover the public key associated to a given principal.

**Example 3.3 (Straightforward implementation)** *A simple and direct implementation of the Purchasing Applet. It waits for the user to provide acquirer's name, order and account information, then it recovers the acquirer's public key, and finally sends everything to the merchant with the expected format.*

$$PA_1 \triangleq acq?x. order?y. acc?z. k := pubKey(x). merchant!(x, y, \{(y, z)\}_k). PA_1$$

**Example 3.4 (Key validation)** *Since the Acquirer's id and public key are not confidential, we do not want to restrict what an implementation does with them, as long as they are not used to encode secret information. As an example, the applet might communicate the id and the public key of the Acquirer to the Merchant, and only proceed if the merchant validates the key as effectively belonging to the Acquirer. In such a way, the Merchant avoids starting to process an order that later will be rejected by the Acquirer.*

$$\begin{aligned} PA_2 &\triangleq acq?x. order?y. acc?z. k := pubKey(x). \\ &\quad merchant!(x, k). merchant?ok. \\ &\quad (PA_2 + \text{if } ok = \mathbf{true} \text{ then } merchant!(x, y, \{(y, z)\}_k). PA_2) \end{aligned}$$

**Example 3.5 (Wallet)** *A slight variation over Example 3.3 with two parallel subprocesses, one in charge of obtaining the Acquirer's data and Customer's*

account number, the other in charge of processing orders.

$$\begin{aligned}
PA_3 &\triangleq (Wallet \mid Buyer) \setminus \{c\} \\
Wallet &\triangleq acq?x. acc?z. k := pubKey(x). Wallet(x, z, k) \\
Wallet(x_0, z_0, k_0) &\triangleq (acq?x. acc?z. k := pubKey(x). Wallet(x, z, k)) \\
&\quad \mid (c!(x_0, z_0, k_0). Wallet(x_0, z_0, k_0)) \\
Buyer &\triangleq order?y. c?u. \\
&\quad \text{let } (x, z, k) = u \text{ in } merchant!(x, y, \{(y, z)\}_k). Buyer
\end{aligned}$$

Notice how channel  $c$ , which is used for internal communication between *Wallet* and *Buyer*, is protected by restriction.

**Example 3.6 (Wallet2)** A more realistic implementation of a wallet-based applet would take the wallet as a service provided by the trusted computing base.

$$PA_4 \triangleq order?y. u := wallet(). \text{let } (x, z, k) = u \text{ in } merchant!(x, y, \{(y, z)\}_k). PA_4$$

The resulting code is, obviously, very similar to that of *Buyer* in Example 3.5. However, notice that here there is no counterpart to the restriction of channel *wallet*.

The implementations of the purchasing applet in the examples so far seem well-intentioned. That need not be always the case. A malicious code provider could resort to a number of tricks to get hold of the Customer's account number (PIN). If the whole or part of the PIN can be deduced by an observer without access to the Acquirer's private key, then we say that there is an insecure information flow (or leak). There are several classifications of leaks, which are more or less comprehensive, with more or less intersection between the different classes. In [SS00], insecure flows are classified into *direct*, *indirect*, *termination behavior*, *probabilistic*, and *externally and internally observable timing* flows.

**Example 3.7 (Direct Leak)** A very simple way for a malicious applet to leak the user's account info would be to encode it as part of the order.

$$MA_1 \triangleq acq?x. order?y. acc?z. k := pubKey(x). merchant!(x, z, \{(y, z)\}_k). MA_1$$

Notice that the account number,  $z$ , appears in the second component of the triple sent to the merchant. The order is expected in this place.

How could this be an implementation of the purchasing applet? As it was indicated above, the protocol implementor is free to decide on the format to encode the order before it is sent to the Merchant. A format always represents certain amount of redundancy which can, in some cases, be manipulated to encode at least part of the account number.

**Example 3.8 (Indirect leak)** *In this case, the implementation simply performs an extra, visible operation depending on the Customer's account number.*

$$\begin{aligned} MA_2 &\triangleq \text{acq}?x.\text{order}?y.\text{acc}?z.k := \text{pubKey}(\text{MerchantId}). \\ &\quad \text{merchant}!(x, y, \{(y, z)\}_k). (MA_2 + \text{if } \text{acc} = v \text{ then } \text{out}!1.MA_2) \end{aligned}$$

**Example 3.9 (Termination behavior)** *This applet might look buggy to an innocent user: it seems to crash sometimes.*

$$\begin{aligned} MA_3 &\triangleq \text{acq}?x.\text{order}?y.\text{acc}?z. \\ &\quad k := \text{pubKey}(\text{MerchantId}). \text{merchant}!(x, y, \{(y, z)\}_k). \\ &\quad (\text{if } \text{acc} \neq v \text{ then } MA_3) + (\text{if } \text{acc} = v \text{ then } 0) \end{aligned}$$

Indeed, the example should be more detailed to demonstrate how the account number could be leaked bit by bit in several attempts at using the applet. What is needed is the ability to test a different bit of the account number on each run of the applet.

**Example 3.10 (Externally observable timing leak)** *This is similar to Example 3.8, but the observer must measure the time between observable actions to learn something about the Customer's account number.*

$$\begin{aligned} MA_4 &\triangleq \text{acq}?x.\text{order}?y.\text{acc}?z.k := \text{pubKey}(\text{MerchantId}). \\ &\quad (\text{if } \text{acc} \neq v \text{ then } \text{merchant}!(x, y, \{(y, z)\}_k). MA_4) \\ &\quad + (\text{if } \text{acc} = v \text{ then } \tau. \text{merchant}!(x, y, \{(y, z)\}_k). MA_4) \end{aligned}$$

Internally observable timing flow is a kind of probabilistic leak. A richer language and semantical model is needed if we want to express and study probabilistic flows. This is a constant in the modelling and treatment of confidentiality. The stronger the power of observation of an attacker, the finer the model that is needed to prevent useful leaks.

The classification of insecure flows does not pretend to be complete. However, since it appeared in a context of non-interference, it is not concerned with situations where some dependency is allowed between secret and non secret data (as is the case with our purchasing applet, which establishes a dependency between the account number  $\text{acc}$  and the purchase slip  $\{\text{order}, \text{acc}\}_K$ ). The following example shows that special attention has to be paid to identifying dependencies if we are to allow some of them:

**Example 3.11 (Wrong encryption key)** *By using a key different than the acquirer's public-key, the merchant might be able to decrypt more than what it is allowed to.*

$$\begin{aligned} MA_5 &\triangleq \text{acq}?x.\text{order}?y.\text{acc}?z.k := \text{pubKey}(\text{MerchantId}). \\ &\quad \text{merchant}!(x, y, \{(y, z)\}_k). MA_5 \end{aligned}$$

### 3.1.2 The Wide-Mouthed Frog Protocol Example

While the previous section has dealt with a few different implementations of the purchasing applet, 1KP is in fact a very simple protocol. Although still simple in general terms, the Wide-Mouthed Frog protocol presents further challenges including: nonce generation, (shared-key) decryption, a longer round of messages, and challenge-response authentication. In this section, we show how a reasonable implementation of the WMF protocol can be encoded in *SecPA*.

**The Protocol** The Wide-Mouthed Frog Protocol allows two principals,  $A$  and  $B$ , establish a session key  $K_{AB}$  in order to communicate securely. Principal  $A$  (resp.  $B$ ) shares key  $K_{AS}$  (resp.  $K_{BS}$ ) with server  $S$ . The original version of this protocol [BAN89] uses timestamps as protection against replay attacks. The version considered here (Table 3.1) is taken from [AG98b], where timestamps are replaced with nonce handshakes. Note that server  $S$  and principal  $B$  generate

|           |                     |  |          |
|-----------|---------------------|--|----------|
| Message 1 | $A \rightarrow S :$ | $A$                                    | on $c_S$ |
| Message 2 | $S \rightarrow A :$ | $N_S$                                  | on $c_A$ |
| Message 3 | $A \rightarrow S :$ | $A, \{A, A, B, K_{AB}, N_S\}_{K_{AS}}$ | on $c_S$ |
| Message 4 | $S \rightarrow B :$ | $*$                                    | on $c_B$ |
| Message 5 | $B \rightarrow S :$ | $N_B$                                  | on $c_S$ |
| Message 6 | $S \rightarrow B :$ | $\{S, A, B, K_{AB}, N_B\}_{K_{BS}}$    | on $c_B$ |
| Message 7 | $A \rightarrow B :$ | $A, \{M\}_{K_{AB}}$                    | on $c_B$ |

Table 3.1: The Wide-Mouthed Frog Protocol

nonces  $N_S$  and  $N_B$ , respectively and once for each session, while principal  $A$  generates the session key  $K_{AB}$ . Channel names have been chosen to reflect the intended recipients of the messages they carry.

The above presentation of the protocol follows a descriptive style that is both compact and informal (as well as a popular one in the literature). Languages like the *spi-calculus* have been proposed to formalize and analyze cryptographic protocols. The situation with *SecPA* is similar, but while the *spi-calculus* aims at analyzing the protocol as a whole, for properties like authentication and secrecy, we are (initially) only interested in modelling realistic implementations of the individual agents participating in the protocol.

**A Server Implementation** Consider the server in the Wide-Mouthed Frog Protocol: to implement it, one needs to take into account the server's availability under multiple requests from different principals, the generation of channels to attend each request, the generation of nonces, a special treatment for flawed messages, decryption, timeouts, etc.

A question appears naturally:

How can one be confident that an implementation does not introduce insecure flows that a careful design of the protocol specification has already eliminated?

This example illustrates, through this and the coming chapters, our approach to an answer. For the moment, we need an implementation.

We propose a simple implementation of the server that handles a request at a time. To model timeouts and inputs of the wrong type we let the server abort the processing of a request and start processing the next one, which we encode using the choice operator (+). Expressions like  $(S + \text{let } \dots)$  could of course be simplified by modifying the semantics of the let operator, but we have opted to keep the semantics of our language at a reasonable level of complexity for the sake of presentation.

Table 3.2 lists the code of the resulting *SecPA* process. The atypical inden-

|                |  |      |
|----------------|--|------|
| $S \triangleq$ | $c_S?x_A.$   | (1)  |
|                | $N_S := \text{nonce}().$   | (2)  |
|                | $\text{out}!(x_A, N_S).$   | (3)  |
| $(S +$         | $c_S?x.$   | (4)  |
| $(S +$         | $k_A := \text{key}(x_A).$  | (5)  |
| $(S +$         | $\text{let } (x'_A, x_{\text{cipher}}) = x \text{ in}$                               | (6)  |
|                | $\text{if } x'_A = x_A \text{ then}$   | (7)  |
|                | $\text{case } x_{\text{cipher}} \text{ of } \{y\}_{k_A} \text{ in}$                  | (8)  |
|                | $\text{let } (y_A, z_A, x_B, x_{\text{key}}, x_{\text{nonce}}) = y \text{ in}$       | (9)  |
|                | $\text{if } (y_A = x_A \wedge z_A = x_A \wedge x_{\text{nonce}} = N_S) \text{ then}$ | (10) |
|                | $\text{out}!(x_B, *).$   | (11) |
| $(S +$         | $c_S?y_{\text{nonce}}.$  | (12) |
| $(S +$         | $k_B := \text{key}(x_B).$  | (13) |
|                | $\text{out}!(x_B, \{s', x_A, x_B, x_{\text{key}}, y_{\text{nonce}}\}_{k_B}. S))))$   | (14) |

Table 3.2: A *SecPA* implementation of the server in the Wide-Mouthed Frog protocol.

tation is meant to help the reader follow the main flow of control, regardless of the possible session restarts embodied by the  $S$  summands. The nonce  $N_S$  is generated by invoking local function  $\text{nonce}()$  (line 2). All output messages go through channel  $\text{out}$ , where the first component of the message indicates its destination (lines 3, 11 and 14). Like in the Purchasing Applet example (Section 3.1.1), we assume the existence of a local function  $\text{key}(e)$  for recovering

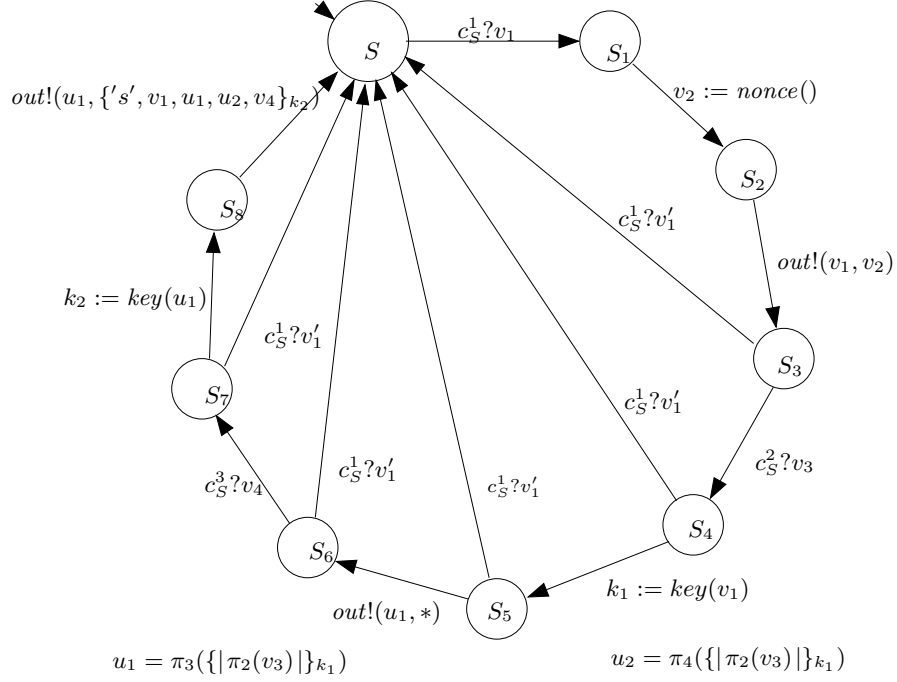


Figure 3.1: Wide-Mouthed Frog Server Implementation

from a local store the shared key associated with a given principal  $e$  (lines 5 and 13). If the local store contains no shared key for  $e$ , the input statement is not executed and the only enabled action aborts the session and restarts the server.

Figure 3.1 reflects, in a schematic way, the transitions that process  $S$  can perform. Each input transition summarizes all possible inputs by assigning them a name in the picture (eg.  $v_1$ ,  $v_2$ , etc.). The arrows into state  $S$  represent restarts of the protocol, renewing the assignment of values to the names above. All transitions are unconditionally enabled, except  $S_5 \xrightarrow{\text{out}!(u_1, *)} S_6$ . This transition is only enabled if there exist values  $x$  and  $y$  such that  $v_3 = (v_1, \{v_1, v_1, x, y, v_2\}_{k_1})$ .

According to the definition given in the figure,  $u_1$  gets the value of  $x$  while  $u_2$  that of  $y$ . Value  $u_1$  corresponds to the identity of principal  $B$  in the protocol description, and  $u_2$  corresponds to the session key  $K_{AB}$ .

**Confidentiality and the WMF protocol** The Wide-Mouthed Frog protocol was designed to let two principals securely exchange a new session key, using a server to authenticate themselves. The authentication mechanism relies on the confidentiality of the keys that each principal shares with the server, as well as the unguessability of nonces. In this thesis, we consider only the first

item, i.e. confidentiality, and note that the quality of nonces has to be verified using some orthogonal techniques.

The confidentiality of the session key, namely  $x_{key}$ , is guaranteed by the protocol itself. Although the server is allowed to learn the exact value of  $x_{key}$ , it can only downgrade information about the shared key in certain admissible ways. These indicate, among others, how the session key should be decrypted and encrypted using session keys  $k_A$  and  $k_B$ .

We can consider a cryptographic protocol as setting a limit to the amount of information that can be downgraded about each of its local inputs. Characterizing them is a complex task that goes beyond a listing of all outputs that are admissible. To take an example from the Wide-Mouthed Frog protocol, notice that an observer might be able to learn from any implementation whether  $x$  is a pair (cf. line 6 in Table 3.2) and whether its second component is encrypted by the key shared between  $A$  and the server (line 8).

## 3.2 Annotations for Tracking Direct Dependencies

Each of the examples presented so far contain information flows and dependencies that the protocol they implement has deemed admissible. This section deals with the problem of identifying those dependencies with enough precision so that a program that exhibits them is not rejected as insecure, and the insecure flows of a program do not get confused with secure ones.

In *SecPA*, values have enough structure so that for each of them we know how it has been constructed from initial values. It certainly helps, but it is not sufficient. In Example 3.7, the value transmitted to the Merchant and which exhibits an inadmissible dependency on the account number, has the same structure as the corresponding value in  $PA_1$ , which is ok. We naturally want to tell one from the other.

The key observation is that the structure of values in *SecPA* has not enough information. The solution is to annotate values with their origin, i.e. the name of the channel through which they have entered the system. We start from *SecPA* and extend its semantics to adequately maintain the annotations through execution. What we obtain is just the semantical counterpart of what in practice would be achieved by means of a data flow analysis.

While reading the rest of this chapter, bear in mind that it is just meant to build up some terminology and machinery with which to express the main results in the following chapter.

**Annotated Expressions and Annotated Values** We revise the development of *SecPA* in the previous sections. The first step is to introduce annotations into the value structure. An annotated value  $w_1 : c()$  indicates that  $w$  (probably

also annotated) was input through channel  $c$ . Similarly,  $w_1 : f(w_2)$  shows that  $w_1$  was input as a result of local function call  $f(w_2)$ .

$$\begin{aligned} (a\text{-Expr}) \quad \epsilon &::= x \mid k \mid (\epsilon_1, \dots, \epsilon_n) \mid \{\epsilon_1\}_{\epsilon_2} \mid \{|\epsilon_1|\}_{\epsilon_2} \mid \pi_i(\epsilon) \mid \epsilon_1 : c(\epsilon_2) \\ (a\text{-Val}) \quad w &::= k \mid (w_1, \dots, w_n) \mid \{w_1\}_{w_2} \mid \{|\epsilon_1|\}_{w_2} \mid \pi_i(w) \mid w_1 : c(w_2) \end{aligned}$$

where  $c \in Ch$  is a channel name and  $n \geq 0$ . The expression  $\epsilon : c()$  will be noted  $\epsilon : c$ . Analogously,  $w : c()$  will be written  $w : c$ .

**Syntax** The syntax of process terms in *a-SecPA* (annotated Security Process Algebra) is like that of *SecPA*, extended with shielded processes:

$$\begin{aligned} P, Q &::= \dots \\ D \triangleright P &\quad (\text{shielded process over } D \subseteq Ch, P \text{ unshielded}) \end{aligned}$$

In *a-SecPA* we substitute annotated expressions  $\epsilon$  and annotated boolean expressions  $\theta$  for resp. expressions and boolean expressions.

For reasons that are discussed in the following chapter, there is no need to register the origin of every value, but only of those that are input through a restricted set of channels  $D$ . We call  $\triangleright$  the *shield* operator. The purpose of shielding a process under  $D$  is just to indicate which values to annotate.

Note that since every expression is indeed an annotated expression, every *SecPA* process is also an (unshielded) *a-SecPA* process.

**Semantics** The semantics of *a-SecPA* takes care of expression annotations to keep track of their origins. We use  $\rightarrow_a$  to identify the new transitions. Since most of the rules are directly obtained from the semantics of *SecPA*, we show here only those that require some extra care, together with the rules for the new shielded process construct.

The most important transition rules for a shielded process concern input. A process  $D \triangleright P$  can only input values that have no annotations. However, it annotates every value input from a channel in  $D$  with its origin. This annotation is propagated by the transition relation  $\rightarrow_a$  within the shielded process.

$$\begin{aligned} \text{A-SHLD}_1 & \frac{P \xrightarrow{c?(v:c)}_a Q}{D \triangleright P \xrightarrow{c?v}_a D \triangleright Q} (c \in D) \\ \text{A-SHLD}_2 & \frac{P \xrightarrow{(v:c(w)) := c(w)}_a Q}{D \triangleright P \xrightarrow{v := c(w)}_a D \triangleright Q} (c \in D) \\ \text{A-SHLD}_3 & \frac{P \xrightarrow{\alpha}_a Q}{D \triangleright P \xrightarrow{\alpha}_a D \triangleright Q} (c \notin D \text{ and } (\alpha = c?w \text{ or } \alpha = w_1 := c(w_2))) \end{aligned}$$

The shield is permeable from the inside out, so that any output carries with it all accumulated annotations.

$$\text{A-SHLD}_4 \frac{P \xrightarrow{a} Q}{D \triangleright P \xrightarrow{a} D \triangleright Q} (\alpha = \tau \text{ or } \alpha \text{ is an output action})$$

For the LET, CASE and COND rules, only the side conditions need to be changed to erase all annotations before evaluation. The following function erases the annotations from an annotated expression  $\epsilon$  yielding a simple expression  $e$ .

$$\begin{aligned} \llbracket x \rrbracket &= x \\ \llbracket k \rrbracket &= k \\ \llbracket (\epsilon_1, \dots, \epsilon_n) \rrbracket &= (\llbracket \epsilon_1 \rrbracket, \dots, \llbracket \epsilon_n \rrbracket) \\ \llbracket \{\epsilon_1\}_{\epsilon_2} \rrbracket &= \{\llbracket \epsilon_1 \rrbracket\}_{\llbracket \epsilon_2 \rrbracket} \\ \llbracket \{|\epsilon_1|\}_{\epsilon_2} \rrbracket &= \{|\llbracket \epsilon_1 \rrbracket|\}_{\llbracket \epsilon_2 \rrbracket} \\ \llbracket \pi_i(\epsilon) \rrbracket &= \pi_i(\llbracket \epsilon \rrbracket) \\ \llbracket \epsilon_1 : c(\epsilon_2) \rrbracket &= \llbracket \epsilon_1 \rrbracket \end{aligned}$$

After extending this definition over annotated boolean expressions in the obvious way, we can now write all the updated rules:

$$\begin{aligned} \text{A-LET} & \frac{P[\pi_1(w)/x_1, \dots, \pi_n(w)/x_n] \xrightarrow{a} P'}{\text{let } (x_1, \dots, x_n) = w \text{ in } P \xrightarrow{a} P'} (\exists v_1, \dots, v_n : \llbracket w \rrbracket = (v_1, \dots, v_n)) \\ \text{A-CASE} & \frac{P[\{|\llbracket w_1 \rrbracket|\}_{w_2}/x] \xrightarrow{a} P'}{\text{case } w_1 \text{ of } \{x\}_{w_2} \text{ in } P \xrightarrow{a} P'} (\exists v : \llbracket w_1 \rrbracket = \{v\}_{\llbracket w_2 \rrbracket}) \\ \text{A-COND} & \frac{P \xrightarrow{a} P'}{\text{if } \theta \text{ then } P \xrightarrow{a} P'} (\llbracket \theta \rrbracket = \mathbf{true}) \end{aligned}$$

Let us take a look at the “encrypter” examples (3.1 and 3.2). In each case, secret values are input through channel *in*. The intention is that those values can only leave the process through channel *out* and encrypted with the right key  $k$ .

Consider first process  $s(k)$  where we take care of dependencies for values input through channel *in*. This is done by shielding the process under  $\{in\}$ . Here is a possible partial execution of  $s(k)$ :

$$\{in\} \triangleright s(k) \xrightarrow{in?v}_a \{in\} \triangleright out!\{v: in\}_k.s(k) \xrightarrow{out!\{v: in\}_k}_a \{in\} \triangleright s(k)$$

We have thus managed to track the evolution of values input through channel *in*, but we should also take care of the key. There is no information in the last transition linking the key used in the expression  $\{v: in\}_k$  with the parameter of the process. To do so, we add a local function call that fetches the actual parameter, like in  $S \triangleq k := key().s(k)$ , and shield the resulting process under  $D \triangleq \{in, key\}$ :

$$\begin{aligned} D \triangleright S & \xrightarrow{k := key()}_a D \triangleright s(k: key) \\ & \xrightarrow{in?v}_a D \triangleright out!\{v: in\}_{k:key}.s(k: key) \\ & \xrightarrow{out!\{v: in\}_{k:key}}_a D \triangleright s(k: key) \end{aligned}$$

We apply the same ideas to the *bad* encrypters in Example 3.2 similarly modified to recover the encryption key in first place (i.e.,  $R \triangleq k := key().r(k)$  and  $T \triangleq k := key().t(k)$ ). We start with *T*:

$$\begin{aligned} D \triangleright T & \xrightarrow{k := key()}_a D \triangleright t(k: key) \\ & \xrightarrow{in?u}_a D \triangleright out!\{u: in\}_{k'}.t(k: key) \\ & \xrightarrow{out!\{u: in\}_{k'}}_a D \triangleright t(k: key) \end{aligned}$$

Now, the annotations in the last transition tell us that the output does not have the form we expect from the description of the protocol (see p. 3.1). In other words, *T* might not preserve the confidentiality properties of the protocol.

However, the situation is not that clear with *R*:

$$\begin{aligned} D \triangleright R & \xrightarrow{k := key()}_a D \triangleright r(k: key) \\ & \xrightarrow{in?u}_a D \triangleright out!\{u: in\}_{k:key} . \\ & \quad \text{if } (u: in) = v \text{ then } out!\{v: in\}_{k:key}.0 \\ & \xrightarrow{out!\{u: in\}_{k:key}}_a D \triangleright \text{if } (u: in) = v \text{ then } out!\{v: in\}_{k:key}.0 \end{aligned}$$

Since  $\llbracket u: in \rrbracket = u$ , if  $u \neq v$  then there are no more enabled transitions. However, if  $u = v$  then the system can perform:

$$D \triangleright \text{if } (u: in) = v \text{ then } out!\{v: in\}_{k:key}.0 \xrightarrow{out!\{u: in\}_{k:key}}_a D \triangleright 0$$

In both situations, the annotations in the outputs do not allow us to rule out process  $R$  as a bad encrypter. The reason is that annotations take only care of direct dependencies, but process  $R$  establishes an indirect dependency between its input and the quantity of outputs. Treating this kind of dependency requires extra machinery and is the subject of Chapter 4.

### 3.2.1 Annotated Purchasing Applet Examples

All of the Purchasing Applet examples represent different implementations of a very simple protocol. Our intention is to find a way to test whether any of those implementations preserves the confidentiality of the Customer's account information. In all examples, the account number is obtained from channel  $acc$ . We therefore shield some of the implementations of the purchasing applet under  $D = \{acc\}$ .

**Example 3.12 (Straightforward implementation)** *We consider  $PA_1$ , as in example 3.3, shielded under  $D$ . The following is a possible trace of a single loop iteration of this applet.*

$$\begin{aligned} D \triangleright PA_1 & \\ \xrightarrow{acc?v_1}_a & D \triangleright \left( \begin{array}{l} order?y. acc?z. k := pubKey(v_1). \\ merchant!(v_1, y, \{y, z\}_k). PA_1 \end{array} \right) \\ \xrightarrow{order?v_2}_a & D \triangleright \left( \begin{array}{l} acc?z. k := pubKey(v_1). \\ merchant!(v_1, v_2, \{v_2, z\}_k). PA_1 \end{array} \right) \\ \xrightarrow{acc?v_3}_a & D \triangleright \left( \begin{array}{l} k := pubKey(v_1). \\ merchant!(v_1, v_2, \{v_2, v_3: acc\}_k). PA_1 \end{array} \right) \\ \xrightarrow{k_1 := pubKey(v_1)}_a & D \triangleright (merchant!(v_1, v_2, \{v_2, v_3: acc\}_{k_1})PA_1) \\ \xrightarrow{merchant!(v_1, v_2, \{v_2, v_3: acc\}_{k_1})}_a & D \triangleright PA_1 \end{aligned}$$

Examples 3.4 and 3.5 are similar in the sense that the annotations in the only output transition coincide with the annotations in the previous example.

**Example 3.13** Example 3.7 illustrated a direct leak of the user account information. This is also evident in the annotations of the shielded process:

$$D \triangleright MA_1 \xrightarrow{*_a} \frac{\text{merchant!}(v_1, v_3: \text{acc}, \{v_2, v_3: \text{acc}\}_{k_1})}{\rightarrow_a} D \triangleright MA_1$$

**Example 3.14** Example 3.11 showed another way of leaking the user account information, this time by using a different public key for encryption. However, in this case, this is not obvious from a simple observation of the annotated actions of the shielded process:

$$D \triangleright MA_5 \xrightarrow{*_a} \frac{\text{merchant!}(v_1, v_2, \{v_2, v_3: \text{acc}\}_{k_1})}{\rightarrow_a} D \triangleright MA_5$$

We could fix this problem using the shield  $D' \triangleq \{\text{acc}, \text{acq}, \text{pubKey}\}$  to obtain:

$$D' \triangleright MA_5 \xrightarrow{*_a} \frac{\text{merchant!}(v_1 : \text{acq}, v_2, \{v_2, v_3: \text{acc}\}_{k_1: \text{pubKey}(\text{MerchantId})})}{\rightarrow_a} D' \triangleright MA_5$$

Now it is evident that the output is not the desired one. With  $D'$  as the shield, an output exhibiting dependencies of the account number should have the form:

$$\text{merchant!}(v_1 : \text{acq}, v_2, \{v_2, v_3: \text{acc}\}_{k_1: \text{pubKey}(v_1: \text{acq})})$$

However, it is not really necessary to use a bigger set than  $D$  as the shield. The same mechanisms used to detect indirect dependencies can be used to detect the insecure flow in Example 3.11. Those mechanisms include their own tracking of data, introducing some redundancy that can be exploited to keep the smaller shield (see Chapter 4).

### 3.2.2 Annotated Wide-Mouthed Frog Protocol Example

As discussed before, to guarantee the admissible treatment of shared and session keys in the Wide-Mouthed Frog protocol, it suffices to control the uses of the shared keys. This is achieved by shielding  $S$  under channel  $key$ . Figure 3.2 represents the executions of  $D \triangleright S$  where  $D = \{key\}$ .

## 3.3 Annotations are Conservative

The introduction of  $a\text{-SecPA}$  would not be complete if we did not state how its semantics compares against the semantics of  $\text{SecPA}$ . The main result of this section states that, in passing from  $\text{SecPA}$  to  $a\text{-SecPA}$ , we do not lose behaviors. This can be interpreted as giving more evidence to support what was hinted by the examples of Section 3.2, i.e. that we can analyse the properties of a  $\text{SecPA}$  process by first converting it into a  $a\text{-SecPA}$  process.

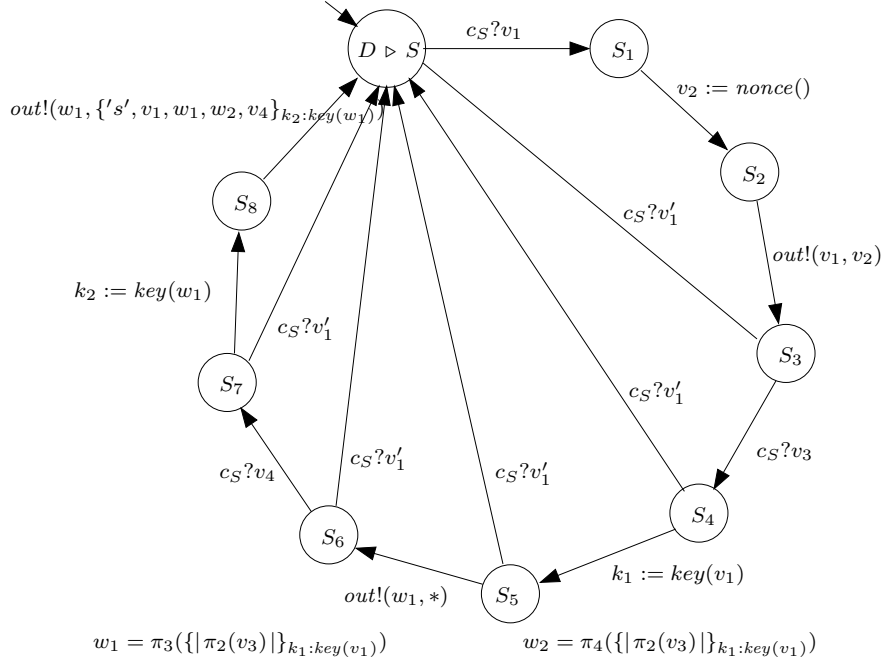


Figure 3.2: Annotated Wide-Mouthed Frog Server Implementation

We start by relating process terms in  $a\text{-SecPA}$  to process terms in  $\text{SecPA}$  using the function  $\text{erase} : a\text{-SecPA} \rightarrow \text{SecPA}$  that eliminates the annotations from the expressions contained in an  $a\text{-SecPA}$  process.

The definition is given over the structure of process terms:

|  |  |
|--|--|
| $\text{erase}(0)$  | $= 0$  |
| $\text{erase}(\tau.P)$   | $= \tau.\text{erase}(P)$   |
| $\text{erase}(c!\epsilon.P)$   | $= c!\llbracket \epsilon \rrbracket.\text{erase}(P)$   |
| $\text{erase}(c?x.P)$  | $= c?x.\text{erase}(P)$  |
| $\text{erase}(x := c(\epsilon).P)$                                     | $= x := c(\llbracket \epsilon \rrbracket).\text{erase}(P)$                                     |
| $\text{erase}(\sum_{i \in I} P_i)$                                     | $= \sum_{i \in I} \text{erase}(P_i)$   |
| $\text{erase}(P   Q)$  | $= \text{erase}(P)   \text{erase}(Q)$  |
| $\text{erase}(P \setminus L)$  | $= \text{erase}(P) \setminus L$  |
| $\text{erase}(P / L)$  | $= \text{erase}(P) / L$  |
| $\text{erase}(\text{if } \theta \text{ then } P)$                      | $= \text{if } \llbracket \theta \rrbracket \text{ then } \text{erase}(P)$                      |
| $\text{erase}(\text{let } (x_1, \dots, x_n) = \epsilon \text{ in } P)$ | $= \text{let } (x_1, \dots, x_n) = \llbracket \epsilon \rrbracket \text{ in } \text{erase}(P)$ |

$$\begin{aligned}
\text{erase}(\text{case } \epsilon_1 \text{ of } \{x\}_{\epsilon_2} \text{ in } P) &= \text{case } \llbracket \epsilon_1 \rrbracket \text{ of } \{x\}_{\llbracket \epsilon_2 \rrbracket} \text{ in } \text{erase}(P) \\
\text{erase}(C(\epsilon_1, \dots, \epsilon_n)) &= C(\llbracket \epsilon_1 \rrbracket, \dots, \llbracket \epsilon_n \rrbracket) \\
\text{erase}(D \triangleright P) &= \text{erase}(P)
\end{aligned}$$

It is easy to verify how  $\llbracket \cdot \rrbracket$  and  $\text{erase}(\cdot)$  commute with variable substitutions:

**Lemma 3.15** *Given annotated expressions  $\epsilon_i$ , annotated boolean expression  $\theta$  and  $a\text{-SecPA}$  process  $P$ , we have:*

1.  $\llbracket \epsilon_1[\epsilon_2/x] \rrbracket = \llbracket \epsilon_1 \rrbracket[\llbracket \epsilon_2 \rrbracket/x]$
2.  $\llbracket \theta[\epsilon/x] \rrbracket = \llbracket \theta \rrbracket[\llbracket \epsilon \rrbracket/x]$
3.  $\text{erase}(P[\epsilon/x]) = \text{erase}(P)[\llbracket \epsilon \rrbracket/x]$

A bit to the side, we extend the definition of  $\llbracket \cdot \rrbracket$  over  $a\text{-Act}$ . It helps keep the notation simple:

$$\begin{aligned}
\llbracket \tau \rrbracket &= \tau \\
\llbracket c!w \rrbracket &= c!\llbracket w \rrbracket \\
\llbracket c?w \rrbracket &= c?\llbracket w \rrbracket \\
\llbracket w_1 := c(w_2) \rrbracket &= \llbracket w_1 \rrbracket := c(\llbracket w_2 \rrbracket)
\end{aligned}$$

The following two results show that  $\text{erase}(\cdot)$  and  $\llbracket \cdot \rrbracket$  define a morphism from the semantics of  $a\text{-SecPA}$  into the semantics of  $\text{SecPA}$ . In other words, every trace of an  $a\text{-SecPA}$  process can be purged of annotations to obtain a trace in  $\text{SecPA}$ .

**Lemma 3.16** *Let  $P$  be a process term in  $a\text{-SecPA}$ , and  $\alpha$  an action in  $a\text{-Act}$ . Then  $\text{erase}(P)$  is a process term in  $\text{SecPA}$  and  $\llbracket \alpha \rrbracket$  is an action in  $\text{Act}$ . Moreover, for all  $\text{SecPA}$  process  $Q$ ,  $\text{erase}(Q) = Q$ .*

**Proposition 3.17** *Let  $P$  and  $Q$  be process terms in  $a\text{-SecPA}$  and  $\alpha$  in  $a\text{-Act}$  such that  $P \xrightarrow{\alpha}_a Q$ . Then  $\text{erase}(P) \xrightarrow{\llbracket \alpha \rrbracket} \text{erase}(Q)$  in  $\text{SecPA}$ .*

The following proposition establishes a corresponding dual relation, when considering processes in  $a\text{-SecPA}$  that do not contain shields.

**Proposition 3.18** *Let  $P$  be a process term of  $a\text{-SecPA}$  without shields. For all  $Q'$  in  $\text{SecPA}$ ,  $\beta \in \text{Act}$  such that  $\text{erase}(P) \xrightarrow{\beta} Q'$ ,*

- *if  $\beta = c!v$  or  $\beta = \tau$  then there are  $\alpha \in a\text{-Act}$  and  $Q$  in  $a\text{-SecPA}$  s.t.*  
 $P \xrightarrow{\alpha}_a Q$ ,  $Q' = \text{erase}(Q)$  and  $\llbracket \alpha \rrbracket = \beta$ .

- if  $\beta = c?v$  then for all  $w \in a\text{-Val}$  s.t.  $\llbracket w \rrbracket = v$ , there is  $Q$  in  $a\text{-SecPA}$  s.t.  $P \xrightarrow{c?w}_a Q$  and  $Q' = \text{erase}(Q)$ .
- if  $\beta = (v_1 := c(v_2))$  then for all  $w_1 \in a\text{-Val}$  s.t.  $\llbracket w_1 \rrbracket = v_1$ , there are  $Q$  in  $a\text{-SecPA}$  and  $w_2 \in a\text{-Val}$  s.t.  $P \xrightarrow{w_1 := c(w_2)}_a Q$ ,  $\llbracket w_2 \rrbracket = v_2$  and  $Q' = \text{erase}(Q)$ .

**Corollary 3.19** *Given  $P$  a process in  $a\text{-SecPA}$  without shields,  $Q'$  in  $\text{SecPA}$  and  $\beta \in \text{Act}$  satisfying  $\text{erase}(P) \xrightarrow{\beta} Q'$ , there are  $\alpha \in a\text{-Act}$  and  $Q$  in  $a\text{-SecPA}$  s.t.  $P \xrightarrow{\alpha}_a Q$ ,  $Q' = \text{erase}(Q)$  and  $\llbracket \alpha \rrbracket = \beta$ .*

Using the second part of Lemma 3.16, the following corollary shows that a process in  $\text{SecPA}$  does not lose traces when considered as a process in  $a\text{-SecPA}$ .

**Corollary 3.20** *Let  $P$  be a process term in  $\text{SecPA}$ . If  $P \xrightarrow{\beta_1} P_1 \xrightarrow{\beta_2} \dots \xrightarrow{\beta_n} P_n$  then there are  $\alpha_1, \dots, \alpha_n \in a\text{-Act}$ , and  $\bar{P}_1, \dots, \bar{P}_n$  in  $a\text{-SecPA}$  such that  $P \xrightarrow{\alpha_1}_a \bar{P}_1 \xrightarrow{\alpha_2}_a \dots \xrightarrow{\alpha_n}_a \bar{P}_n$  where  $\llbracket \alpha_i \rrbracket = \beta_i$  and  $\text{erase}(\bar{P}_i) = P_i$ , for  $1 \leq i \leq n$ .*

In all our examples, we have been interested in the shielded process  $D \triangleright P$  where, initially,  $P$  is a process in  $\text{SecPA}$ . In those cases, we can be very precise when comparing traces of  $P$  in  $\text{SecPA}$  with traces of  $D \triangleright P$  in  $a\text{-SecPA}$ .

**Corollary 3.21** *Let  $P$  be a process term in  $\text{SecPA}$  and  $D$  a set of channels. If  $P \xrightarrow{\beta_1} P_1 \xrightarrow{\beta_2} \dots \xrightarrow{\beta_n} P_n$  then there are  $\alpha_1, \dots, \alpha_n$ , and  $\bar{P}_1, \dots, \bar{P}_n$  such that  $D \triangleright P \xrightarrow{\alpha_1}_a \bar{P}_1 \xrightarrow{\alpha_2}_a \dots \xrightarrow{\alpha_n}_a \bar{P}_n$ , and for all  $1 \leq i \leq n$ :*

- $\text{erase}(\bar{P}_i) = P_i$
- if  $\beta_i = \tau$  then  $\alpha_i = \beta_i$
- if  $\beta_i = c!v$ , then  $\alpha_i = c!w$ ,  $\llbracket w \rrbracket = v$
- if  $\beta_i = c?v$ , then  $\alpha_i = \beta_i$
- if  $\beta_i = v_1 := c(v_2)$ , then  $\alpha_i = v_1 := c(w_2)$ ,  $\llbracket w_2 \rrbracket = v_2$
- for every subexpression  $w' : d(w'')$  of  $\alpha_i$ ,  $w'$  is a value and  $d \in D$



## Chapter 4

# Confidential Protocol Implementation

The formal languages of the previous chapter allowed us to express and discuss quite a few implementations of different (simple) cryptographic protocols. From the discussions, we can draw two main conclusions: First, that practice demands a very flexible notion of *implementation*. Second, since any notion of implementation is intimately related to the concept of specification, it is fundamental to understand how a protocol represents such a specification.

Being more concrete, we want to define “confidential protocol implementation”, which means that we have to extract from a protocol some sort of confidentiality policy, a high-level overall plan covering the amount and quality of information that each observer of the protocol is allowed to infer. Given that it is probably very difficult to capture such a loose mixture of information and knowledge restrictions, we take an indirect approach. The idea is to understand a protocol as defining the dependencies that any confidential implementation may *at most* establish between secret data and observable behavior. This set of dependencies constitutes the confidentiality policy associated to the protocol. Moreover, we assume that the protocol has been verified separately, to prove that its set of dependencies actually implies the desired confidentiality properties.

A policy should specify very precisely the dependencies that are considered admissible, without otherwise placing excessive requirements on the implementation relation. If the policy were smaller (i.e. admitted less dependencies), then most realistic and safe implementations of the protocol would be rejected. If it were bigger (i.e. admitted more dependencies than the protocol), programs with insecure flows may be considered to implement the protocol.

Although this might sound simple enough, it is definitely not so. Again, the concept of secret plays a central role. For the Purchasing Applet examples, if we chose to consider the account number as the only secret, we would open up

the door to implementations that leak the key, which in turn lets an observer get to the account number. Fortunately, we can rely on the analysis of the protocol. Any such analysis must necessarily identify the initial knowledge of the attacker. Whatever is not known by the attacker, might be safely assumed to be a secret. In principle the protocol analysis indicates what the secrets are.

We can now give the intuition behind our notion of confidential protocol implementation. A program confidentially implements a protocol if its own set of secret dependencies is bounded by the confidentiality policy associated to the protocol. It all then reduces to the ability to collect the dependencies established by a program. But, how does one go about determining the set of dependencies? Naturally, the usual noninterference models, where dependencies on secrets (high-level data) are prohibited, are of little help. The language of Chapter 3 lets us express our implementations, where it is possible to determine direct dependencies on secrets with the aid of the annotations in *a-SecPA*. However, as it was noticed in the cases of encrypter *R* (p. 42) and Example 3.14 (with  $D = \{acc\}$ ), this does not suffice. Indeed, the annotations in *a-SecPA* were never meant to cope with all dependencies. They are just a piece of a mechanism to track *strong dependencies* [Coh78] between secret inputs and behaviors.

Cohen defined strong dependencies in the context of state transformers as a relation between initial and end values of program variables. Obviously, the approach does not apply directly to our *SecPA* processes, which are indeed reactive systems where the concept of variable is more diffuse. Instead, our intention is to capture dependencies by correlating changes in system behavior with changes in input. For the purchasing applet example, changes in the confidential parameter, say  $z$ , received as part of a message

$$acc?z \tag{4.1}$$

should give rise to “proportional” changes in applet output messages

$$merchant!(x, y, \{(y, z)\}_k) \tag{4.2}$$

but affect applet behavior in no other way. To make this idea precise the following ingredients are needed:

1. a notion of behavior equivalence, and
2. a mechanism to model changes in values and their effects, as changes in actions.

Several equivalences have been considered in the information flow theory literature, including trace equivalence [GM82, GM84a], failures equivalence [RG99], and bisimulation equivalence [FG95]. The work reported here is to a large extent independent of the specific choice of equivalence, so it suffices for now to just assume the existence of *some* equivalence relation on states,  $\sim$ , reflecting the idea of behavior identity, or indistinguishability by an external observer.

To model action changes we use the notion of a *relabelling* as a mapping permuting actions. Relabellings are used to enforce the correlation between received and transmitted values required for confidentiality. This justifies our presentation of *a-SecPA*, as the definition of a relabelling is made possible by the extra information included in annotated actions and values.

This chapter proceeds as follows: First, we make precise the notion of confidentiality policy (an intensional specification of a secrecy specification) and produce policies for our examples from Chapter 3. Then, we use the encrypter examples to motivate our use of relabelling functions to detect dependencies outside the policy. In third place, we show how to associate a set of relabelling functions to a policy, which lets us define the desired implementation relation, *Admissibility*. We conclude by applying the resulting definition to some examples which hint on an efficient method to verify it. The following chapter studies the relationship between admissibility and Cohen's work on confidentiality.

## 4.1 Secrets and Confidentiality Policies

A secret is a piece of data, a value. As such, it must either be present within the system before execution starts, or be input as the system executes.

In our model, secrets enter a system only by being input (if, like in the encrypter examples, a certain secret value resides within the system before it starts executing, we introduce an initial input to avoid having to treat this case separately). A credit card number provided by the user enters the system through an appropriately labelled entry channel. In other words, we characterize data by the channels used to input it. If the channel is considered to be a secret channel, then the input value is taken to be a secret. This, of course, requires the channel to be a trusted one. In other words, that the attacker cannot observe and/or meddle with the data carried by it. For example, the credit card number mentioned above is assumed to be provided by the local trusted computing base (TCB) through a channel that is not observable (even indirectly) by the attacker.

Besides secrets, we also consider critical inputs, values that are used to identify admissible dependencies. For example, in the case of the purchasing applets, the key used to encrypt the user's credit number is a critical value, for only if the right key is used the dependency is admissible.

Let  $\mathcal{E}$  indicate the set of channels through which secrets can enter the system; and  $\mathcal{C}$  those channels whose inputs are not secrets themselves ( $\mathcal{E} \cap \mathcal{C} = \emptyset$ ) but would be used to identify admissible outputs carrying partial secret information.

If  $P \xrightarrow{c(v_1)?v_2} Q$  and  $c \in \mathcal{E}$ , then  $v_2$  is a secret whose evolution throughout  $Q$ 's behavior we should track in order to identify the outputs that depend upon it. Direct dependencies will be tracked using the annotations of Chapter 3, while indirect dependencies will be controlled by applying the techniques described in the following sections.

Having identified secrets and critical inputs, we are now in a position to de-

fine confidentiality policies that describe what information (dependent on those secrets) can be downgraded, on which channel and when. This will constitute a high-level description of the property to achieve by means of relabelling sets and process equivalences.

**Definition 4.1 (Confidentiality Policy)** *A confidentiality policy is a triple  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$ . The sets  $\mathcal{E}$  and  $\mathcal{C}$  are as described above and  $\mathcal{A}$  is an indexed set of clauses of the form*

$$j : \quad c!e \leftarrow c_1(e_1)(x_1) \wedge \dots \wedge c_n(e_n)(x_n) \wedge b$$

where  $j \in \mathcal{J}$  is an index identifying the clause,  $e \in \text{Expr}$ ,  $b \in \text{BoolExpr}$ , and for all  $1 \leq i \leq n$ ,  $c_i \in \mathcal{E} \cup \mathcal{C}$ ,  $e_i \in \text{Expr}$  and variables in  $e_i$  do not belong to  $\{x_i, \dots, x_n\}$ .

The restriction on the variables appearing in each  $e_i$  lets us annotate those that correspond to secret inputs. To make this explicit, we define a function  $\text{Ann}: \text{Expr} \rightarrow a\text{-Expr}$  for each clause in  $\mathcal{A}$

$$\begin{aligned} \text{Ann}(x) &= \begin{cases} x_i : c_i(\text{Ann}(e_i)) & \text{if } x = x_i \text{ and } c_i \in \mathcal{E} \\ x & \text{otherwise} \end{cases} \\ \text{Ann}(k) &= k \\ \text{Ann}(e_1, \dots, e_n) &= (\text{Ann}(e_1), \dots, \text{Ann}(e_n)) \\ \text{Ann}(\{e_1\}_{e_2}) &= \{\text{Ann}(e_1)\}_{\text{Ann}(e_2)} \\ \text{Ann}(p_i(e)) &= p_i(\text{Ann}(e)) \end{aligned}$$

Then for each  $e_i$  appearing in a clause of  $\mathcal{A}$ , we have that  $\text{Ann}(e_i)$  is an expression where all variables appearing in it are annotated.

Intuitively, a clause in the policy declares an output matching  $c!e$  (similarly, a function call of the form  $v := c(e)$ ) to be admissible if the conditions to the right of the arrow are satisfied. Conjuncts of the form  $c_i(e_i)(x_i)$  are satisfied if variable  $x_i$  matches the last input from channel  $c_i(e_i)$ . As before, conjuncts of the form  $c()(x)$  will be written  $c(x)$ . Finally, the boolean expression  $b$  represents an extra condition by establishing a relation between the values input through the different channels.

More formally, let a context  $s$  be a partial mapping assigning values to channels. We call  $\text{Context} \triangleq [Ch \rightarrow a\text{-Val} \rightarrow Val]$  the set of all possible contexts. The square brackets indicate that a *Context* is a partial function; intuitively, it records the context in which an output takes place, by listing the last input received over each channel. The second parameter, an annotated action, is used to cope with return values from function calls. Local function names are considered to be channels. Note that we only need the value of a context over channels in  $\mathcal{E} \cup \mathcal{C}$ , but including all other channels causes no harm and simplifies some definitions.

Given a context, we decide whether a particular output taking place in that context is admitted by the policy.

**Definition 4.2 (Admissible Output)** Let  $\alpha$  be an annotated action of the form  $c!w$  or  $w' := c(w)$ . A confidentiality policy  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  admits annotated action  $\alpha$  in context  $s$  iff either

- no channel annotation in  $w$  belongs to  $\mathcal{E}$  (that is, the output does not depend directly on any secret entry), or
- there is a clause  $j : cle \leftarrow c_1(e_1)(x_1) \wedge \dots \wedge c_n(e_n)(x_n) \wedge b$  in  $\mathcal{A}$  and a substitution  $\sigma$  assigning values to variables, such that:  $Ann(e)\sigma = w$ ,  $s(c_i(Ann(e_i)\sigma)) = x_i\sigma$ , for every  $1 \leq i \leq n$ , and  $b\sigma = \text{true}$ .

In this situation, we write  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash_j \alpha \text{ ok}$ . By dropping the index, like in  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \alpha \text{ ok}$ , we indicate the existence of an appropriate index  $j$  such that the assertions above hold.

Notice that we have defined this property over annotated actions. If we had limited ourselves to actions without annotations, first there would be no information to determine whether an output depends on secrets. Furthermore, even if we had just added a single bit annotation to identify values depending on secrets, there would be a high risk of confusing an inadmissible output with an admissible one (cf. Example 3.7). We will return to this issue after the introduction of relabelling functions.

The notation for confidentiality policies leaves implicit a very important feature: While a given policy indicates which outputs and local function calls are considered admissible, it also restricts the information an observer can learn from any of them. If  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash c!w \text{ ok}$  holds because of clause  $j : cle \leftarrow c_1(e_1)(x_1) \wedge \dots \wedge c_n(e_n)(x_n) \wedge b$  in  $\mathcal{A}$ , an observation of  $c!\llbracket w \rrbracket$  can only convey to an observer the knowledge of the occurrence of all needed inputs plus the existence of a substitution  $\sigma$  satisfying condition  $b$ . This strong restriction on information flow will be enforced by deriving an appropriate set of relabellings for each confidentiality policy.

**Lemma 4.3** Let  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \alpha \text{ ok}$  where  $\alpha = o!w$  or  $\alpha = (v := o(w))$ . If  $v_0 : c(w_0)$  is a subterm of  $w$  and  $c \in \mathcal{E}$ , then  $s(c(w_0)) = v_0$ .

*Proof.* See Appendix A.

We conclude this section by providing confidentiality policies for the three main examples introduced in Chapter 3.

**Example 4.4 (Confidentiality Policy for the Encrypter)**

Clearly, the secret to protect is input through channel  $in$ , but we should protect also the access to the shared keys stored at the encrypter process. The latter corresponds to channel key, and therefore  $\mathcal{E} = \{in, key\}$ . There are no critical outputs needed to characterize admissible outputs:  $\mathcal{C} = \emptyset$ . Finally, the set  $\mathcal{A}$  contains the single clause:

$$0 : \quad out!\{x\}_k \leftarrow in(x) \wedge key(k)$$

which states that the only output allowed to depend on secrets input through  $\mathcal{E}$  is obtained by encrypting  $i$ 's input with  $key$ 's input.

**Example 4.5 (Confidentiality Policy for the Purchasing Applets)**

We are mainly interested in protecting the confidentiality of the user account information and then  $\mathcal{E} = \{acc\}$ . However, we have noticed before that, to protect the account number, we also need to make sure that the right public-key is used to encrypt the sensitive outputs of the applet. Therefore,  $\mathcal{C} = \{acq, pubKey\}$ . The set  $\mathcal{A}$  contains the single clause:

$$0 : \quad merchant!(x, y, \{(y, z)\}_k) \leftarrow acq(x) \wedge acc(z) \wedge pubKey(x)(k)$$

Notice how variable  $y$  is not bound to the right of the clause, reflecting the fact that we do not put any requirement on the format of the order beyond the restriction that it should not be used to encode secret information.

**Example 4.6 (Confidentiality Policy for the WMF protocol)**

As noted in Section 3.1.2, the confidentiality of the session key  $x_{key}$  is guaranteed by the protocol. This is true provided that (among other things) the confidentiality of the keys shared by the server with the participating principals is preserved. That is, we should keep a tight control over accesses to the key store, and therefore  $\mathcal{E} = \{key\}$ .

The server is allowed to establish a dependency between the value of the session key and its own behavior. However, this can only occur under some restricted circumstances. For example, the server should have previously received a 5-tuple (with the nonce  $N_S$  as its fifth coordinate) encrypted under  $x_A$ 's shared key. This implies that the channel associated to the nonce generation function is to be considered critical:  $\mathcal{C} = \{nonce\}$ .

Of the three outputs that the server is expected to perform according to the specification of the Wide-Mouthed Frog protocol (Table 3.1), only those corresponding to messages 4 and 6 are allowed to leak partial information about inputs from  $\mathcal{E}$ . In both cases, the identity of the receiver (i.e.  $B$ ) is obtained from Message 3 using key  $K_{AS}$ . Message 6 also requires key  $K_{BS}$ . However these are not the only outputs that the server can only construct if it knows the identity of  $B$ . Clearly, we should also consider the function call used to determine  $B$ 's shared key (Table 3.2, row 13). The corresponding clauses in the confidentiality policy are:

$$\begin{aligned} 0 : \quad out!(\pi_3(\{|x|\}_{k_A}), *) &\leftarrow nonce(N_S) \wedge key(x_A)(k_A) \\ &\quad \wedge \{|x|\}_{k_A} = (x_A, x_A, x_B, x_{key}, N_S) \\ 1 : \quad key!\pi_3(\{|x|\}_{k_A}) &\leftarrow nonce(N_S) \wedge key(x_A)(k_A) \\ &\quad \wedge \{|x|\}_{k_A} = (x_A, x_A, x_B, x_{key}, N_S) \\ 2 : \quad out!(\pi_3(\{|x|\}_{k_A}), \{s', x_A, \pi_3(\{|x|\}_{k_A}), \pi_4(\{|x|\}_{k_A}), y_{nonce}\}_{k_B}) &\leftarrow \\ &\quad nonce(N_S) \wedge key(x_A)(k_A) \wedge key(\pi_3(\{|x|\}_{k_A}))(k_B) \\ &\quad \wedge \{|x|\}_{k_A} = (x_A, x_A, x_B, x_{key}, N_S) \end{aligned}$$

## 4.2 Towards a Notion of Confidential Implementation

We illustrate the approach in the context of the *Encrypter* systems (Examples 3.1 and 3.2). For the time being, we will prioritize simplicity over formality in order to give a flavor of the approach and its difficulties.

**Process Relabelling** In *CCS*, a process  $P$  can be relabelled under a relabelling function  $f$  rendering a new process  $P[f]$  whose semantics is given by the following rule:

$$\frac{P \xrightarrow{\alpha} Q}{P[f] \xrightarrow{f(\alpha)} Q[f]}$$

where  $f$  is required to satisfy  $f(\tau) = \tau$  and  $f(\bar{\alpha}) = \overline{f(\alpha)}$  for every action  $\alpha$ .

We extend this definition to *SecPA*, but we treat value-passing actions slightly differently from the way they are treated in value-passing *CCS*: A relabelling function  $f: Act \rightarrow Act$  is requested to satisfy:

- Channels should be respected: For every channel  $c$  and every value  $v$ , there is a value  $v'$  such that  $f(c?v) = c?v'$  and  $f(c!v) = c!v'$ ,

As an example in the context of the *Encrypter* systems (Ex. 3.1 and 3.2), consider any relabelling  $f: Act \rightarrow Act$  satisfying

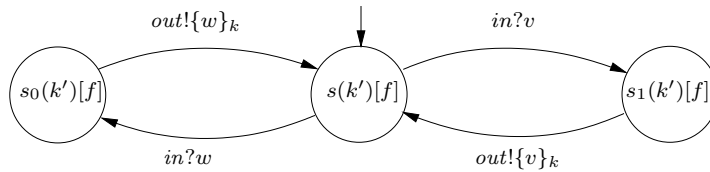
$$\begin{aligned} f(in?v) &= in?w \\ f(in?w) &= in?v \\ f(out!\{v\}_k) &= out!\{w\}_{k'} \\ f(out!\{w\}_k) &= out!\{v\}_{k'} \\ f(\alpha) &= \alpha, \text{ for all other actions.} \end{aligned}$$

where  $k$  and  $k'$  are some fixed key values.

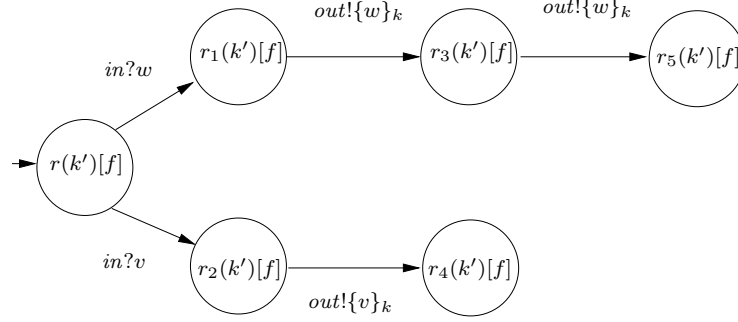
Notice how  $f$  permutes all inputs, but only the outputs that any good implementation of an *Encrypter* is expected to perform.

### Example 4.7 (Relabelled systems)

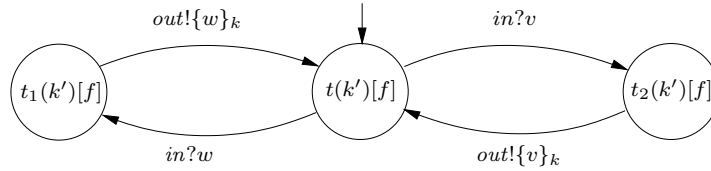
**Encrypter:** For process  $s(k')$  of Example 3.1, we get the following transition system for  $s(k')[f]$



Bad Encrypter #1: For  $r(k')$  of Example 3.2, the relabelled transition system  $r(k')[f]$  is



Bad Encrypter #2: When  $t(k')$  is defined as in Example 3.2, the relabelled transition system  $t(k')[f]$  is



**An approach to confidentiality** Each *Encrypter* system can be compared against its corresponding relabelled version using some behavior equivalence. If we use strong equivalence, we obtain:

$$\begin{array}{ll}
 \text{Encrypter:} & s(k')[f] \sim s(k) \\
 \text{Bad Encrypter \#1:} & r(k')[f] \not\sim r(k) \\
 \text{Bad Encrypter \#2:} & t(k')[f] \not\sim t(k)
 \end{array}$$

We can see that relabelling  $f$ , which intuitively permutes all inputs and only admissible outputs, lets us detect the presence of inadmissible information flows in both *Bad Encrypter* examples.

The proposal is therefore to cast confidentiality as invariance, up to state equivalence, under a set  $\mathcal{F}$  of relabellings.

**The difficulties** Realizing this approach presents both conceptual and technical difficulties. The former include the identification of what information is actually protected (i.e., what the secrets are, see Section 4.1) and what is achieved by using a given relabelling set  $\mathcal{F}$  (i.e., what confidentiality property it represents, see Chapter 5).

From a technical point of view, our preliminary definition of a relabelling function needs to be extended to cope with more elaborate systems and confidentiality requirements (like those posed by the Wide-Mouthed Frog protocol in Section 3.1.2). The fundamental observation is that, if a relabelling permutes an output value, then the way this value is permuted depends on the permutation of the input values used to construct it. We work with *a-SecPA* since, in this calculus, the information needed to define such a relabelling is stored in the value annotations themselves. The results by the end of Chapter 3, linking the semantics of *a-SecPA* to that of *SecPA*, justify this decision. Defining the set of relabelling functions  $\mathcal{F}$  so that it represents a given confidentiality policy is also a delicate matter.

The rest of this chapter is dedicated to solving the technical difficulties listed above, defining our notion of confidential protocol implementation and illustrating with examples.

### 4.3 Conditional Process Relabelling

We aim at modelling a confidentiality policy as a property of processes with an associated verification technique. As a first step, we consider general relabelling functions whose definition depends on the history of inputs. We also consider the application of these relabellings to processes in *a-SecPA*.

In the previous section, we applied a relabelling to all three encrypter examples, written in *SecPA*. The intention was to make a point about the use of relabellings to restrict admissible information flow, and therefore the machinery was kept to a necessary minimum. Providing a general enough definition poses three main requirements over a relabelling. Firstly, when a relabelling permutes an input to another value, it should keep that permutation and apply it every time that input value forms part of an output. Secondly, a relabelling should not map an admissible output into an inadmissible one. Finally, a relabelling function might depend on what inputs have been performed in the past, and relations between them (just like in the clauses of the confidentiality policy).

The first requirement is met by resorting to *a-SecPA*. In this language, every output value is annotated with the inputs it depends on. It is then easy to make sure that a relabelling permutes values in a consistent way, by associating it to a single secret permuter:

**Definition 4.8 (Secret Permuter)** *Given a set  $\mathcal{E}$  of secret input channels, a*

secret permuter is a function  $g: a\text{-Val} \rightarrow a\text{-Val}$  which satisfies  $g^{-1} = g$  and

$$\begin{aligned} g(k) &= k \\ g(w_1, \dots, w_n) &= (g(w_1), \dots, g(w_n)) \\ g(\{w_1\}_{w_2}) &= \{g(w_1)\}_{g(w_2)} \\ g(p_i(w)) &= p_i(g(w)) \\ g(w_1 : c(w_2)) &= \begin{cases} v : c(g(w_2)) & \text{for some value } v, \text{ if } c \in \mathcal{E} \text{ and } w_1 \text{ is a value} \\ g(w_1) : c(g(w_2)) & \text{otherwise} \end{cases} \end{aligned}$$

A secret permuter  $g$  preserves the annotations in an annotated value expression. It only permutes values that do not contain annotations and have been input through channels in  $\mathcal{E}$ . Observe that the definition implies that for every value  $v$  and every parameterized channel  $c(w)$ , there is a value  $v'$  such that  $g(v : c(w)) = v' : c(g(w))$ . Moreover, when  $w = ()$ ,  $g(v : c) = v' : c$ . This property will prove important in the permutation of inputs of a shielded process (see Section 3.2).

Our second requirement for the definition of relabelling concerns the preservation of admissible outputs (see Definition 4.2).

**Definition 4.9 (Secret Permuter that Preserves Admissible Outputs)**

Given a confidentiality policy  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  and a secret permuter  $g$ , then  $g$  is said to preserve admissible outputs if

$$(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash_j c!w \text{ ok} \Leftrightarrow (\mathcal{E}, \mathcal{C}, \mathcal{A}), \bar{g}(s) \vdash_j c!g(w) \text{ ok}$$

and

$$(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash_j w_1 := c(w_2) \text{ ok} \Leftrightarrow (\mathcal{E}, \mathcal{C}, \mathcal{A}), \bar{g}(s) \vdash_j w_1 := c(g(w_2)) \text{ ok}$$

The permuted context  $\bar{g}(s)$  is defined by

$$\bar{g}(s)(c(w)) = \llbracket g(s(c(w'))) : c(w') \rrbracket$$

where  $w' = g(w)$ .

The third requirement placed upon a general definition of relabelling function is that this function should depend on the evolution of the system to be relabelled. We abstract that evolution by means of a *labelled transition system*  $R = (\text{Context}, a\text{-Act}, \rightarrow, s_0)$  where:

- each state is a context,
- the initial state,  $s_0$ , is the empty map in *Context*, and

- the transition relation  $\rightarrow \subseteq \text{Context} \times a\text{-Act} \times \text{Context}$  is given by the rules:

$$\begin{array}{ll} s \xrightarrow{\tau} s & s \xrightarrow{c?w} s[c() \mapsto \llbracket w \rrbracket] \\ s \xrightarrow{c!w} s & s \xrightarrow{w_1 := c(w_2)} s[c(w_2) \mapsto \llbracket w_1 \rrbracket] \end{array}$$

The notation  $s[c(w) \mapsto v]$  indicates the standard redefinition of mapping  $s$  over  $c(w)$  so that

$$s[c(w) \mapsto v](c'(w')) = \begin{cases} v & c = c', w = w' \\ s(c'(w')) & \text{otherwise} \end{cases}.$$

The idea is to define a relabelling to consist of a pair  $(R, f)$  where  $R$  is the labelled transition system defined above and  $f: \text{Context} \times a\text{-Act} \rightarrow a\text{-Act}$  permutes actions depending on the current state (i.e., context) of  $R$ . Since  $R$  is fixed, we should identify a relabelling by simply showing its  $f$  function.

We can now combine the two solutions by limiting our interest to relabellings defined in terms of secret permuters and a confidentiality policy:

**Definition 4.10 (Conditional Relabelling)** *Let  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  be a confidentiality property and  $g$  a secret permuter that preserves admissible outputs. We define a conditional relabelling  $f_g: \text{Context} \times a\text{-Act} \rightarrow a\text{-Act}$ :*

$$\begin{aligned} f_g(s, \tau) &= \tau \\ f_g(s, c?w) &= c?w' \text{ where } w' : c = g(w : c) \\ f_g(s, w_1 := c(w_2)) &= \begin{cases} w'_1 := c(g(w_2)) & \text{if } (\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash w_1 := c(w_2) \text{ ok} \\ w'_1 := c(w_2) & \text{otherwise} \end{cases} \\ &\quad \text{where } w'_1 : c(g(w_2)) = g(w_1 : c(w_2)) \\ f_g(s, c!w) &= \begin{cases} c!g(w) & \text{if } (\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash c!w \text{ ok} \\ c!w & \text{otherwise} \end{cases} \end{aligned}$$

We conclude the section by adding a conditional relabelling operator to  $a\text{-SecPA}$ . The semantics of this operator will use  $R$  to keep track of the inputs performed by the process. It will then apply  $f$  to relabel the current input.

**Proposition 4.11** *Let  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  be a confidentiality policy and  $g$  a secret permuter that preserves admissible outputs. Then, for all action  $\alpha \in a\text{-SecPA}$  and all context  $s$ ,*

$$f_g(\bar{g}(s), f_g(s, \alpha)) = \alpha$$

*Proof.* The proof is done by a simple case analysis over the definition of  $f_g$ , using that  $g^{-1} = g$ .  $\square$

**Lemma 4.12** *If  $s \xrightarrow{\alpha} s'$  then  $\bar{g}(s) \xrightarrow{f_g(s, \alpha)} \bar{g}(s')$*

*Proof.* See Appendix A.

**Definition 4.13 (Conditionally relabelled process)** *Given an a-SecPA process  $P_0$  and a conditional relabelling  $f$ ,  $P_0[s_0 \vdash f]$  is a conditionally relabelled process whose semantics is given by the following rule:*

$$\frac{P \xrightarrow{\alpha}_a P' \quad s \xrightarrow{\alpha} s'}{P[s \vdash f] \xrightarrow{f(s, \alpha)}_a P'[s' \vdash f]}$$

## 4.4 Admissibility

In the previous section, we introduced all the machinery needed to formalize the ideas sketched in the introduction to this chapter. Here we define when a SecPA process satisfies a confidentiality policy.

**Definition 4.14 (Admissibility)** *A SecPA process  $P$  is called admissible w.r.t. confidentiality policy  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  if*

$$\mathcal{E} \triangleright P \sim (\mathcal{E} \triangleright P)[s_0 \vdash f_g]$$

*for every possible conditional relabelling  $f_g$ .*

Consider our straightforward implementation of a purchasing applet (Example 3.3). To ease our manipulation of the reachable states of process  $\mathcal{E} \triangleright PA_1$ , where  $\mathcal{E} = \{acc\}$ , we adopt the following abbreviations:

$$\begin{aligned} p_0 &\triangleq \mathcal{E} \triangleright PA_1 \\ p_1(x) &\triangleq \mathcal{E} \triangleright \left( \begin{array}{l} order?y. acc?z. k := pubKey(x). \\ merchant!(x, y, \{y, z\}_k). PA_1 \end{array} \right) \\ p_2(x, y) &\triangleq \mathcal{E} \triangleright \left( \begin{array}{l} acc?z. k := pubKey(x). \\ merchant!(x, y, \{y, z\}_k). PA_1 \end{array} \right) \\ p_3(x, y, z) &\triangleq \mathcal{E} \triangleright \left( \begin{array}{l} k := pubKey(x). \\ merchant!(x, y, \{y, z: acc\}_k). PA_1 \end{array} \right) \\ p_4(x, y, z, k) &\triangleq \mathcal{E} \triangleright (merchant!(x, y, \{y, z: acc\}_k). PA_1) \end{aligned}$$

From Example 3.12, we know that

$$\begin{aligned} p_0 &\xrightarrow{acq?v_1}_a p_1(v_1) \xrightarrow{order?v_2}_a p_2(v_1, v_2) \xrightarrow{acc?v_3}_a p_3(v_1, v_2, v_3) \\ &\xrightarrow{k_1 := pubKey(v_1)}_a p_4(v_1, v_2, v_3, k_1) \xrightarrow{merchant!(v_1, v_2, \{v_2, v_3: acc\}_{k_1})}_a p_0 \end{aligned}$$

Now, we show that  $PA_1$  is an admissible process w.r.t. to the confidentiality policy as given in Example 4.5. We need to establish that, for every possible secret permuter  $g$ ,

$$p_0 \sim p_0[s_0 \vdash f_g]$$

Consider therefore, for each secret permuter  $g$ , the relation

$$\begin{aligned} R_g \triangleq & \{ (p_0, p_0[s_0 \vdash f_g]), (p_1(v_1), p_1(v_1)[s_1 \vdash f_g]), \\ & (p_2(v_1, v_2), p_2(v_1, v_2)[s_2 \vdash f_g]), \\ & (p_3(v_1, v_2, v_3), p_3(v_1, v_2, v'_3)[s_3 \vdash f_g]), \\ & (p_4(v_1, v_2, v_3, k_1), p_4(v_1, v_2, v'_3, k_1)[s_4 \vdash f_g]) \mid \\ & s_1 = s_0[acq \mapsto v_1], s_2 = s_1[order \mapsto v_2], s_3 = s_2[acc \mapsto v'_3], \\ & s_4 = s_3[pubKey(v_1) \mapsto k_1], v'_3 : acc = g(v_3 : acc) \} \end{aligned}$$

Take any such secret permuter  $g$  and let  $v'_3 : acc = g(v_3 : acc)$ . Observe that the relabelled system  $p_0[s_0 \vdash f_g]$  then exhibits the following trace:

$$\begin{aligned} p_0[s_0 \vdash f_g] & \xrightarrow{acq?v_1}_a p_1(v_1)[s_1 \vdash f_g] \xrightarrow{order?v_2}_a p_2(v_1, v_2)[s_2 \vdash f_g] \\ & \xrightarrow{f_g(s_2, acc?v'_3)}_a p_3(v_1, v_2, v'_3)[s_3 \vdash f_g] \\ & \xrightarrow{k_1 := pubKey(v_1)}_a p_4(v_1, v_2, v'_3, k_1)[s_4 \vdash f_g] \\ & \xrightarrow{f_g(s_4, merchant!(v_1, v_2, \{v_2, v'_3 : acc\}_{k_1}))}_a p_0[s_4 \vdash f_g] \end{aligned}$$

Since  $f_g(s_2, acc?v'_3) = acc?v_3$ , we have

$$p_2(v_1, v_2)[s_2 \vdash f_g] \xrightarrow{acc?v_3}_a p_3(v_1, v_2, v'_3)[s_3 \vdash f_g]$$

and, because  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s_4 \vdash merchant!(v_1, v_2, \{v_2, v'_3 : acc\}_{k_1})$  ok (using  $\mathcal{A}$  as given in Example 4.5), we have

$$p_4(v_1, v_2, v'_3, k_1)[s_4 \vdash f_g] \xrightarrow{merchant!(v_1, v_2, \{v_2, v_3 : acc\}_{k_1})}_a p_0[s_4 \vdash f_g]$$

These observations cover most of the cases in checking that  $R$  is a bisimulation relation. It only remains to verify that  $p_0 R p_0[s_4 \vdash f_g]$ . But this is immediate from the observation that  $p_0[s_4 \vdash f_g] \sim p_0[s_0 \vdash f_g]$ , since the values of  $s_4$  over  $\mathcal{E} \cup \mathcal{C}$  will be replaced by new ones before they can be used to decide whether an output is admissible in the future behaviors of  $p_0[s_4 \vdash f_g]$ .

#### 4.4.1 Verifying Admissibility

We can draw some conclusions from the way the example has been analyzed in the previous section. A first observation is that, by giving the abbreviations  $p_i$

( $i = 1 \dots 4$ ), we have actually identified and separated control (i.e. the index  $i$ ) from data (i.e. the parameters of each  $p_i$ ) much like as if we were dealing with an imperative program. A second, very important observation is that the flow of control is not altered by the relabelling. For example, if

$$p_2(v_1, v_2) \xrightarrow{\beta}_a p_3(v_1, v_2, v_3)$$

then

$$p_2(v_1, v_2)[s_2 \vdash f_g] \xrightarrow{\beta}_a p_3(v_1, v_2, v'_3)[s_3 \vdash f_g],$$

which holds only if there is  $\alpha$  such that  $p_2(v_1, v_2) \xrightarrow{\alpha}_a p_3(v_1, v_2, v'_3)$  and  $\beta = f_g(s_2, \alpha)$ .

If fact, it is possible to generalize this procedure to other processes, and thus simplify the verification of our confidentiality property. The following theorem states this result:

**Theorem 4.15** *Let  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  be a confidentiality property. Given  $p \in \text{SecPA}$ , assume there is a partial function  $p_- : \mathcal{N} \times \text{Context} \rightarrow a\text{-SecPA}$  satisfying:*

$$P1) (\mathcal{E} \triangleright p) = p_0(s_0),$$

$$P2) \text{ if } p_i(s) \xrightarrow{\alpha}_a p' \text{ then } \exists j. p' = p_j(s') \text{ where } s \xrightarrow{\alpha} s', \text{ and}$$

$$P3) \text{ if } p_i(s) \xrightarrow{\alpha}_a p_j(s') \text{ then } p_i(\bar{g}(s)) \xrightarrow{f_g(s, \alpha)}_a p_j(\bar{g}(s')), \text{ for all secret per-} \\ \text{muter } g \text{ that preserves admissible outputs.}$$

*Then process  $p$  is admissible w.r.t.  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$ .*

Although we have basically all the elements to give a proof of this statement, this will wait till next chapter, where the machinery developed there will help us produce a rather simple proof (see Section 5.3). This theorem represents a considerable simplification over the direct proof strategy. The specific definition and check of a bisimulation relation is replaced by simple and local conditions which ensure the bisimulation property of some fixed relation.

It is important to notice that Theorem 4.15 can be applied even in cases when the program exhibits branching of control flow based on secret data. This works, of course, provided that the branching is admissible, as described by the confidentiality policy.

As an example, we prove that the server implementation of the Wide-Mouthed Frog protocol of Section 3.1.2 (Table 3.2) is indeed admissible w.r.t. the confidentiality policy given in Example 4.6.

To start with, we define  $p_- : \mathcal{N} \times \text{Context} \rightarrow a\text{-SecPA}$  taking Figure 3.2 as a guide, so that to each state  $S_i$  in the figure we associate an  $a\text{-SecPA}$  process  $p_i(s_i)$  with an appropriate  $\text{Context } s_i$ . There is only one variant though: Since a context keeps only the last input from each channel, and we have to compare

$$\begin{aligned}
p_0(s) &\triangleq \mathcal{E} \triangleright c_S^1 ? x_A . p_1(s_0[c_S^1 \mapsto x_A]) \\
p_1(s) &\triangleq \mathcal{E} \triangleright N_S := \text{nonce}(). p_2(s[\text{nonce}() \mapsto N_S]) \\
p_2(s) &\triangleq \mathcal{E} \triangleright \text{out}!(s(c_S^1), s(\text{nonce}())) . p_3(s) \\
p_3(s) &\triangleq \mathcal{E} \triangleright (p_0(s) + c_S^2 ? x . p_4(s[c_S^2 \mapsto x])) \\
p_4(s) &\triangleq \mathcal{E} \triangleright (p_0(s) + k_A := \text{key}(s(c_S^1)) . p_5(s[\text{key}(s(c_S^1)) \mapsto k_A])) \\
p_5(s) &\triangleq \mathcal{E} \triangleright \left( \begin{array}{l} p_0(s) + \text{let } (x'_A, x_{\text{cipher}}) = s(c_S^2) \text{ in} \\ \quad \text{if } x'_A = s(c_S^1) \text{ then} \\ \quad \quad \text{case } x_{\text{cipher}} \text{ of } \{y\}_{s(\text{key}(s(c_S^1))) : \text{key}(s(c_S^1))} \text{ in} \\ \quad \quad \quad \text{let } (y_A, z_A, x_B, x_{\text{key}}, x_{\text{nonce}}) = y \text{ in} \\ \quad \quad \quad \quad \text{if } (y_A = s(c_S^1) \wedge z_A = s(c_S^1) \wedge \\ \quad \quad \quad \quad \quad x_{\text{nonce}} = s(\text{nonce}())) \\ \quad \quad \quad \quad \text{then} \\ \quad \quad \quad \quad \quad \text{out}!(x_B, *) . p_6(s) \end{array} \right) \\
p_6(s) &\triangleq \mathcal{E} \triangleright (p_0(s) + c_S^3 ? y_{\text{nonce}} . p_7(s[c_S^3 \mapsto y_{\text{nonce}}])) \\
p_7(s) &\triangleq \mathcal{E} \triangleright (p_0(s) + k_b := \text{key}(w_1) . p_8(s[\text{key}(w_1) \mapsto k_B])) \\
p_8(s) &\triangleq \mathcal{E} \triangleright \text{out}!(w_1, \{s', s(c_S^1), w_1, w_2, s(c_S^3)\}_{s(\text{key}(w_1)) : \text{key}(w_1)}) . p_0(s)
\end{aligned}$$

Table 4.1: Control flow for Wide-Mouthed Frog server implementation

different inputs from channel  $c_S$ , we identify each different usage of the channel with a superscript, like in  $c_S^1$ . The result appears in Table 4.1, where we have adopted the following abbreviations

$$\begin{aligned}
w_1 &= \pi_3(\{|\pi_2(s(c_S^2))|\}_{s(\text{key}(s(c_S^1))) : \text{key}(s(c_S^1))}) \\
w_2 &= \pi_4(\{|\pi_2(s(c_S^2))|\}_{s(\text{key}(s(c_S^1))) : \text{key}(s(c_S^1))})
\end{aligned}$$

Furthermore, we take  $p_i(s)$  as defined only if  $s$  contains all the mappings that appear on the definition of  $p_i(s)$ ; and, for  $p_i(s)$  with  $i = 6, 7, 8$ , we also require the existence of values  $v$  and  $v'$  such that

$$\pi_1(s(c_S^2)) = s(c_S^1) \quad (4.3)$$

and

$$\{|\pi_2(s(c_S^2))|\}_{s(\text{key}(s(c_S^1)))} = (s(c_S^1), s(c_S^1), v, v', s(\text{nonce}())) \quad (4.4)$$

(Note that the annotation of channels  $c_S^i$  does not affect the confidentiality policy, since  $c_S \notin \mathcal{E} \cup \mathcal{C}$ . Therefore, if the program in Table 4.1 results admissible, so is the program in Table 3.2).

A simple inspection of the definition of  $p_0, \dots, p_8$  shows that the requirements P1) and P2) of Theorem 4.15 are verified. Notice as well that  $p_0(s)$  is

defined in terms of  $s_0$ , and not of  $s$ , reflecting the fact that the server resets itself before each run.

In order to apply Theorem 4.15, we need to verify requirement P3). If  $\alpha$  contains no reference to  $\mathcal{E}$  (i.e., if  $\alpha$  contains no annotation from  $\mathcal{E} = \{key\}$ ), then  $f_g(s, \alpha) = \alpha$ . In such cases, it is easy to check that if  $p_i(s) \xrightarrow{\alpha}_a p_j(s')$  then  $p_i(\bar{g}(s)) \xrightarrow{\alpha}_a p_j(\bar{g}(s'))$ .

The case of inputs from channels in  $\mathcal{E}$  is also quite easy to verify. For example, consider

$$p_4(s) \xrightarrow{k_1 := key(s(c_S^1))}_a p_5(s[key(s(c_S^1)) \mapsto k_1])$$

If  $g(k_1 : key(s(c_S^1))) = k'_1 : key(g(s(c_S^1)))$ , then

$$f_g(s, k_1 := key(s(c_S^1))) = (k'_1 := key(g(s(c_S^1)))) = (k'_1 := key(\bar{g}(s)(c_S^1)))$$

so that

$$p_4(\bar{g}(s)) \xrightarrow{k'_1 := key(\bar{g}(s)(c_S^1))}_a p_5(\bar{g}(s[key(s(c_S^1)) \mapsto k_1]))$$

since  $\bar{g}(s)[key(\bar{g}(s)(c_S^1)) \mapsto k'_1] = \bar{g}(s[key(s(c_S^1)) \mapsto k_1])$

More interesting are the cases of outputs and function calls that depend on secret data. Consider first the transition

$$p_5(s) \xrightarrow{out!(w_1, *)}_a p_6(s)$$

We can deduce that

$$w_1 = \pi_3(\{\pi_2(s(c_S^2))\}_{s(key(s(c_S^1)) : key(s(c_S^1)))}) \quad (4.5)$$

and that there should exist values  $v$  and  $v'$  so that equations (4.3) and (4.4) are satisfied. (By the way, this shows that the requirements placed on the definition of  $p_6$ ,  $p_7$  and  $p_8$  are fulfilled every time these states are reachable).

The confidentiality policy for the WMF server (Example 4.6) includes the clause

$$\begin{aligned} 0 : out!(\pi_3(\{x\}_{k_A}), *) &\leftarrow nonce(N_S) \wedge key(x_A)(k_A) \\ &\wedge \{x\}_{k_A} = (x_A, x_A, x_B, x_{key}, N_S) \end{aligned}$$

To see that this clause makes  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash out!(w, *)$  ok, let  $\sigma$  be the following substitution:

$$\begin{array}{ll} \sigma(x_A) &= s(c_S^1) & \sigma(N_S) &= s(nonce()) \\ \sigma(k_A) &= s(key(s(c_S^1))) & \sigma(x) &= \pi_2(s(c_S^2)) \\ \sigma(x_B) &= v & \sigma(x_{key}) &= v' \end{array}$$

Then the requirements on the righthand side of the clause hold because of (4.4). Applying  $\sigma$  to the annotated expression on the lefthand side and by means of (4.5):

$$\text{Ann}((\pi_3(\{|x|\}_{k_A}), *)\sigma) = (\pi_3(\{|x|\}_{k_A: \text{key}(x_A)}), *)\sigma = (w_1, *)$$

The secret permuter  $g$  is supposed to preserve admissible outputs. Therefore, if  $s' = \bar{g}(s)$ , we know that  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s' \vdash f_g(s, \text{out}!(w_1, *)) \text{ ok}$ , and since  $\text{out}!(w_1, *)$  is admissible over  $s$ , we get

$$(\mathcal{E}, \mathcal{C}, \mathcal{A}), s' \vdash \text{out}!(g(w_1), *) \text{ ok} \quad (4.6)$$

From (4.5),

$$g(w_1) = \pi_3(\{|\pi_2(s(c_S^2))|\}_{g(s(\text{key}(s(c_S^1)))):\text{key}(s(c_S^1))})$$

Since  $c_S^1, c_S^2 \notin \mathcal{E}$ , we know that  $s(c_S^1) = s'(c_S^1)$  and  $s(c_S^2) = s'(c_S^2)$ . Applying the definition of  $\bar{g}$  (see Def. 4.9),  $g(s(\text{key}(s(c_S^1)))) : \text{key}(s(c_S^1)) = s'(\text{key}(s'(c_S^1))) : \text{key}(s'(c_S^1))$ . Then

$$g(w_1) = \pi_3(\{|\pi_2(s'(c_S^2))|\}_{s'(\text{key}(s'(c_S^1)))):\text{key}(s'(c_S^1))})$$

From (4.6), there are values  $u$  and  $u'$  such that

$$\{|\pi_2(s'(c_S^2))|\}_{s'(\text{key}(s'(c_S^1)))} = (s'(c_S^1), s'(c_S^1), u, u', s'(\text{nonce()}))$$

and from here we can conclude that

$$p_5(s') \xrightarrow{\text{out}!(g(w_1), *)}_a p_6(s')$$

which shows that P3) holds for  $p_5(s) \xrightarrow{\text{out}!(w_1, *)}_a p_6(s)$ . Even though we have worked out the details for this transition, it is easy to notice that most of what we did here can be extended to any admissible transition provided it can be expressed, as it is the case of  $w_1$ , as a function of *Context*. If we write  $w_1(s)$ , then what we have done is simply to verify that  $g(w_1(s)) = w_1(\bar{g}(s))$  and that, for all context  $s$  s.t.  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \text{out}!(w_1(s), *) \text{ ok}$ ,  $p_5(s) \xrightarrow{\text{out}!(w_1(s), *)}_a p_6(s)$ . If then we match this with the fact that for each reachable  $p_5(s)$  we have  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \text{out}!(w_1(s), *) \text{ ok}$ , we have verified P3) for this particular case.

With these same ideas, the remaining two transition schemas involving outputs of secrets,

$$p_7(s) \xrightarrow{k_2 := \text{key}(w_1)}_a p_8(s[\text{key}(w_1) \mapsto k_2])$$

and

$$p_8(s) \xrightarrow{\text{out}!(w_1, \{s', s(c_S^1), w_1, w_2, s(c_S^3)\}_{s(\text{key}(w_1)):\text{key}(w_1)})}_a p_0(s)$$

are proved to respect property P3) using the remaining two clauses of  $\mathcal{A}$ .

Having verified that P3) holds for all cases, we can apply Theorem 4.15 and conclude that our server implementation of the WMF protocol is admissible.



## Chapter 5

# Controlled Information Flow

Technically, a process is admissible with respect to a confidentiality policy if it is invariant under a set of relabellings extracted from the policy. However, admissibility has been so far proposed as the definition of a confidential protocol implementation relation. It has also been said that the dependencies on secrets established by an admissible process are bounded by the policy. *What are the reasons that justify these assertions?*

To explain the confidentiality implications of admissibility, we need to understand what a dependency is. In the late 1970's, Ellis Cohen presented an information theoretic model of secrecy for programs taking the form of state transformers. In this model, a program is considered as a set of communication channels transmitting information from initial to final states. In information theoretic terms, there is information transmission if variety of initial values is conveyed to variety of end values. According to Cohen, a program establishes a *strong dependency* from a set of variables (at the initial state) to a variable (at the final state), if the corresponding communication channel has non-zero capacity.

As Cohen pointed out in the concluding notes of [Coh78], the definition of strong dependency represents a purely quantitative measure of information transmission. He added: "Except for very simple sorts of data, a strict information theoretic measure may not be appropriate; all bits are not equal. For example, the high order bits of a variable containing salary information is likely to be more valuable than the low order bits, and a suitable measure might be weighted accordingly." Our definition of confidentiality policy defines a qualitative measure of information flow. By characterizing (in an intensional way) the communication channels with non-zero capacity, we can actually consider a policy as a bound on the dependencies that a program is allowed to establish.

In this chapter, after reviewing Cohen's model, we translate it from the world

of state transformers to the world of reactive processes written in *a-SecPA*. We then introduce a method that purges an *a-SecPA* process from the dependencies in a policy. If the remaining system still transmits secret information (in the information theoretic sense), then the system cannot be considered a confidential implementation. The chapter concludes by showing that an admissible system transmits no information about secrets once deprived from all admissible dependencies (i.e. the dependencies admitted by the confidentiality policy). From this we conclude that admissibility provides a reasonable definition for the desired confidential implementation relation.

## 5.1 Cohen's Selective Independency

Cohen gave one of the first formalizations of security as lack of information flow at the level of programming languages. He studied under which conditions there is no information flow from a set of variables  $\mathcal{E}$  to variable  $v$  by means of the execution of program  $P$ . If variety in the original values of  $\mathcal{E}$  is not conveyed to variety in the final values of  $v$ , then there is no information flow from  $\mathcal{E}$  to  $v$ . This he called *Strong Independency*.

$$\mathcal{E} \not\vdash_{\varphi}^P v \triangleq \forall \sigma, \sigma'. \sigma =_{-\mathcal{E}} \sigma' \wedge \varphi(\sigma) \wedge \varphi(\sigma') \Rightarrow P(\sigma).v = P(\sigma').v \quad (5.1)$$

where  $\sigma, \sigma' \in St$  (the set of all states), and  $\varphi$  is a state predicate used to select the states of interest. The expression  $\sigma =_S \sigma'$  means “equality over  $S$ ” and is defined as  $\forall v \in S. \sigma.v = \sigma'.v$ . In this context, the complement of variable set  $S$  is written  $\neg S$ , so that  $\sigma =_{\neg S} \sigma'$  denotes  $\forall v \notin S. \sigma.v = \sigma'.v$ .

Cohen also considered a relaxation of this property, called *Selective Independency* which can be used to accommodate partial dependencies. The idea is to use a set of state selectors  $\{\varphi_i\}$  to restrict variety of inputs to within each  $\varphi_i$ .

From [Coh78],  $v$  is selectively independent of  $\mathcal{E}$  over  $p$  w.r.t.  $\{\varphi_i\}$  if

1.  $\{\varphi_i\}$  is a “cover”, i.e.  $\forall \sigma \in St. \exists i. \varphi_i(\sigma)$ ,
2.  $\forall i, \sigma, \sigma'. \sigma =_{\mathcal{E}} \sigma' \Rightarrow \varphi_i(\sigma) = \varphi_i(\sigma')$ , and
3.  $\forall i. \mathcal{E} \not\vdash_{\varphi_i}^p v$ .

Although equation (5.1), strong independency, describes the lack of information flow between variables in a program, it does not really consider leaks through variety in nontermination and probabilistic behavior. Sands and Sabelfeld have given a clear account of these issues, while retaining the connection to Cohen's definitions. Basically, they have expressed Selective Independency as partial equivalence relation (PER) types which keep control of the termination properties of the programs [SS99].

There are two directions in which one might need to extend the definition of Selective Independency. Firstly, instead of taking the program to be a state transformer, one might want to consider reactive systems. Secondly, the model

could be extended to cope with the particular kind of dependencies established by the use of cryptographic operations (e.g. the dependency between a plaintext  $m$  and its corresponding ciphertext  $\{m\}_k$  under symmetric key  $k$ ).

However, before we do so, it is convenient to rephrase Cohen's definition of Selective Independency replacing the set of state selectors  $\{\varphi_i\}$  by a single equivalence relation  $\Delta$  over states. This  $\Delta$  must satisfy a few extra requirements if it is to correspond to a set  $\{\varphi_i\}$ . In first place, it is only interesting to compare the effect of a program on pairs of initial states that differ only in the value of secret variables, i.e.  $\Delta \subseteq =_{-\mathcal{E}}$ . Now, according to the second item in Cohen's definition of Selective Independency, each  $\varphi_i$  depends strictly on the value of secret variables. In terms of relation  $\Delta$  this means that if the same permutation is applied to the values of non-secret variables in a pair of related states  $p \Delta q$  then the resulting permuted states  $p'$  and  $q'$  are also related, i.e.  $p' \Delta q'$ . This is formalized in the following definition:

**Definition 5.1** *Given a set of variables  $\mathcal{E}$ , a relation  $\Delta$  over states is called  $\mathcal{E}$ -strict if*

$$((=_{\mathcal{E}}; \Delta; =_{\mathcal{E}}) \cap =_{-\mathcal{E}}) \subseteq \Delta \subseteq =_{-\mathcal{E}}$$

where “;” indicates usual relational composition.

We can now give an alternative definition of Selective Independency, using  $\mathcal{E}$ -strict equivalence relations:

**Definition 5.2 (Selective Independency for State Transformers)** *Let  $\mathcal{E}$*

*be a set of variables and let  $\Delta$  be an  $\mathcal{E}$ -strict equivalence relation. Then  $v$  is selectively independent of variable set  $\mathcal{E}$  over program  $p$  w.r.t.  $\Delta$ , noted  $\mathcal{E} \not\vdash_{\Delta}^p v$ , iff*

$$\forall \sigma, \sigma'. \sigma \Delta \sigma' \Rightarrow p(\sigma).v = p(\sigma').v \quad (5.2)$$

The following lemma and its corollaries establish the connection between this and Cohen's definition of Selective Independency. Essentially, both definitions are equivalent.

**Lemma 5.3** *Given  $\Delta$ ,  $\{\varphi_i\}$  and  $\mathcal{E}$  such that*

$$(A.1) \quad \forall i, \sigma, \sigma'. \sigma =_{\mathcal{E}} \sigma' \Rightarrow \varphi_i(\sigma) = \varphi_i(\sigma')$$

$$(A.2) \quad \Delta \subseteq St \times St \text{ is the transitive closure of}$$

$$\Delta_1 \triangleq \{(\sigma, \sigma') \mid \exists i. \varphi_i(\sigma) \wedge \varphi_i(\sigma') \wedge \sigma =_{-\mathcal{E}} \sigma'\}$$

*then,  $\mathcal{E} \not\vdash_{\Delta}^p v$  iff  $v$  is selectively independent of  $\mathcal{E}$  over  $p$  w.r.t.  $\{\varphi_i\}$ .*

*Proof.* See Appendix A.

**Corollary 5.4** *If  $\Delta \subseteq St \times St$  is an  $\mathcal{E}$ -strict equivalence relation, then there is a family of state selectors  $\{\varphi_i\}$  such that  $\mathcal{E} \not\vdash_{\Delta}^p v$  iff  $v$  is selectively independent of  $\mathcal{E}$  over  $p$  w.r.t.  $\{\varphi_i\}$ .*

*Proof.* Given  $\Delta$ , let  $\{\Omega_i\}$  be its set of equivalence classes. Then, define  $\{\varphi_i\}$  so that for each  $i$ ,  $\varphi_i(\sigma)$  iff  $\exists \delta =_{\mathcal{E}} \sigma \cdot \delta \in \Omega_i$ .

To see that the resulting family  $\{\varphi_i\}$  satisfies (A.2) it suffices to show that  $\Delta_1$  coincides with  $\Delta$ :

$$\begin{aligned} \sigma \Delta_1 \sigma' &\Leftrightarrow \exists i. \varphi_i(\sigma) \wedge \varphi_i(\sigma') \wedge \sigma =_{\neg \mathcal{E}} \sigma' \\ &\Leftrightarrow \exists i, \delta, \delta'. \sigma =_{\mathcal{E}} \delta \wedge \delta, \delta' \in \Omega_i \wedge \sigma' =_{\mathcal{E}} \delta' \wedge \sigma =_{\neg \mathcal{E}} \sigma' \\ &\Leftrightarrow \sigma (=_{\mathcal{E}}; \Delta; =_{\mathcal{E}}) \sigma' \wedge \sigma =_{\neg \mathcal{E}} \sigma' \\ &\Rightarrow \sigma \Delta \sigma' \text{ (since } \Delta \text{ is } \mathcal{E}\text{-strict)} \end{aligned}$$

For the other direction, take  $\sigma \Delta \sigma'$ . Since  $\Delta \subseteq =_{\neg \mathcal{E}}$ , it must be the case that  $\sigma, \sigma' \in \Omega_i$ , for some  $i$ , and that  $\sigma =_{\neg \mathcal{E}} \sigma'$ . Therefore, we have  $\varphi_i(\sigma) \wedge \varphi_i(\sigma') \wedge \sigma =_{\neg \mathcal{E}} \sigma'$  so that  $\sigma \Delta_1 \sigma'$ .

The definition of  $\{\varphi_i\}$  satisfies (A.1): take  $i, \sigma$  and  $\sigma'$  s.t.  $\sigma =_{\mathcal{E}} \sigma'$  and  $\varphi_i(\sigma)$ . Then, there is  $\delta$  such that  $\delta =_{\mathcal{E}} \sigma$  and  $\delta \in \Omega_i$ . Therefore  $\sigma' =_{\mathcal{E}} \delta$ , which implies  $\varphi_i(\sigma')$ .  $\square$

**Corollary 5.5** *If  $\{\varphi_i\}$  satisfies (A.1), then there exists  $\Delta \subseteq St \times St$  such that  $\mathcal{E} \not\vdash_{\Delta}^p v$  iff  $v$  is selectively independent of  $\mathcal{E}$  over  $p$  w.r.t.  $\{\varphi_i\}$ .*

*Proof.* Simply use (A.2) as the definition of  $\Delta$ .  $\square$

Note that there are obvious connections between this and the PER approach of Sands and Sabelfeld. Indeed, we could have given these definitions, with similar results using partial equivalence relations. The reason why it was requested that  $\Delta$  be reflexive is that it results in simpler proofs for the lemma and corollaries above.

### 5.1.1 Separation of Variety

Cohen [Coh78] established conditions under which a strong independency property can be verified by dividing the set of possible inputs into smaller subsets and proving strong independency within each of them. We restate the theorem here:

**Proposition 5.6 (Cohen's Separation of Variety)** *If*

- $\forall \sigma. \varphi(\sigma) \Rightarrow \exists i. \varphi_i(\sigma)$  (i.e.  $\{\varphi_i\}$  is a cover of  $\varphi$ ), and
- ( $\mathcal{E}$ -independency)  $\forall i, \sigma, \sigma'. (\sigma =_{\neg \mathcal{E}} \sigma' \wedge \varphi(\sigma) \wedge \varphi(\sigma') \Rightarrow \varphi_i(\sigma) = \varphi_i(\sigma'))$

*then*

$$\forall i. \mathcal{E} \not\vdash_{\varphi_i}^p v \text{ implies } \mathcal{E} \not\vdash_{\varphi}^p v$$

When the intention is to prove Selective Independency, the following theorem suggests a similar separation of variety. This result is of fundamental importance to generalize the definition of Selective Independency to *a-SecPA* processes.

**Proposition 5.7** *Let  $\mathcal{E}$  be a set of variables and  $\Delta$  an  $\mathcal{E}$ -strict equivalence relation. Then,  $\mathcal{E} \not\vdash_{\Delta}^p v$  iff there exists a family  $\{\Delta_i\}$  of symmetric relations in  $St \times St$  that satisfies:*

- (1)  $\Delta = (\bigcup_i \Delta_i)^*$ , and
- (2)  $\forall i, \sigma, \sigma'. \sigma \Delta_i \sigma' \Rightarrow p(\sigma).v = p(\sigma').v$

*Proof.*

$\Rightarrow$ ) Take  $\{\Delta\}$  as the family.

$\Leftarrow$ ) According to Definition 5.2, we just need to prove  $\forall \sigma, \sigma'. \sigma \Delta \sigma' \Rightarrow p(\sigma).v = p(\sigma').v$ .

The proof resembles that of (5.2) in Lemma 5.3. Let  $\sigma \Delta \sigma'$ . Assume  $\sigma \neq \sigma'$ , otherwise the desired result is immediate. By assumption (1), there exist  $\sigma_1, \dots, \sigma_n$  and  $i_1, \dots, i_{n-1}$  such that  $\sigma = \sigma_1$ ,  $\sigma' = \sigma_n$  and  $\forall 1 \leq j < n. \sigma_j \Delta_{i_j} \sigma_{j+1}$ .

By applying assumption (2), we get  $p(\sigma_j).v = p(\sigma_{j+1}).v$ , for each  $j$ . The result follows then by transitivity of equality.  $\square$

## 5.2 Selective Independency for a-SecPA

Cohen's definition of Selective Independency is supported on four pillars: (1) a view of a process as a communication channel, (2) a criterion to decide when two inputs to this channel coincide on everything but the secrets ( $=_{-\mathcal{E}}$ ), (3) an assumption on the behavior of the environment ( $\mathcal{E}$ -strictness), and (4) a criterion to identify equivalent outputs from the channel.

Consider the first element: A state transformer can easily be viewed as a memory-less channel communicating data from initial into end states. By contrast, the characterization of the communication channels provided by a reactive (*a-SecPA*) process is quite a subtle matter. Indeed, it represents an open problem. The concepts of information theory have only been extended to specific cases of networks of channels (see chapter 14 of [CT91] for an overview). Moreover, most accounts of confidentiality for computer systems abstract the difficulties away more or less explicitly.

While realizing that the problem is still in need of a satisfactory solution, the approach adopted here is to consider each process at each execution point as a communication channel. This channel inputs a history of execution (a trace) and returns the set of possible ways in which the execution could be continued. The process can decide when to accept inputs, but the actual input value is chosen by the environment. In fact, this approach presents similarities to that behind the definition of Nondeducibility on Compositions [FG95], where high level values (i.e. secrets) are fed into the system by another process (sometimes called "strategy").

In the following sections we analyze the remaining three elements that are necessary to define Selective Independency over *a-SecPA* processes. Before we proceed, it is appropriate to fix some notation: A trace  $\sigma$  in *a-SecPA* is simply a sequence  $\alpha_1\alpha_2\ldots\alpha_n$ , with  $n \geq 0$  and  $\alpha_i \in a\text{-Act}$  for all  $1 \leq i \leq n$ . We use  $Tr$  to denote the set of *a-SecPA* traces, and  $\lambda$  for the empty trace. By  $\sigma_i$  we denote the projection of the  $i$ -th element of a trace  $\sigma$ , and by  $len(\sigma)$  its length.

### 5.2.1 History Indistinguishability

In the original definition of strong independency, the relation  $=_{-\mathcal{E}}$  identifies two states which differ only on the values of secrets. In a way, two states related by  $=_{-\mathcal{E}}$  are indistinguishable by an observer that initially does not know any secret values. When considering the information flow model for *a-SecPA*, as sketched in the previous section, it is important to notice that each communication channel takes traces as input. In other words, in adapting the definition of Selective Independency to reactive systems, it is necessary to define when two traces are indistinguishable for the attacker.

Consider first two annotated actions  $\alpha$  and  $\beta$ . When are they indistinguishable? There are four kinds of actions: silent, input, output and function call. We assume that the attacker can recognize the kind of each action, so  $\alpha$  and  $\beta$  need to be of the same kind. In the case of output actions, we assume that everything is observable, both the channel and the value, and the same can be said of the actual parameter in a function call (see end of this section).

We can formalize the discussion so far by defining when two annotated actions are output equivalent:  $\alpha \approx \beta$  iff

$$\begin{aligned} \alpha = \tau &\Leftrightarrow \beta = \tau \\ \alpha = c!w_1 &\Leftrightarrow \beta = c!w_1 \\ \alpha = c?w_1 &\Leftrightarrow \beta = c?w'_1 \\ \alpha = (w_1 := c(w_2)) &\Leftrightarrow \beta = (w'_1 := c(w_2)) \end{aligned}$$

Note that we have also assumed that the attacker can observe the channel of an input action, but not necessarily the value. If the input channel does not belong to  $\mathcal{E}$  (the set of secret input channels), we can safely assume that the attacker can also observe the input value. If the channel is indeed a secret channel, then it cannot observe anything but the channel name.

We can now define  $=_{-\mathcal{E}}$ , keeping the notation from the previous section to stress the parallelism between the definitions.

**Definition 5.8 (Trace indistinguishability)** *Two traces  $\sigma$  and  $\sigma'$  are (observer) indistinguishable, written  $\sigma =_{-\mathcal{E}} \sigma'$ , when for all  $1 \leq i \leq len(\sigma)$ ,  $\sigma_i \approx \sigma'_i$  and for all  $c \notin \mathcal{E}$ ,*

$$\sigma_i = c?w_1 \vee \sigma_i = (w_1 := c(w_2)) \Rightarrow \sigma'_i = \sigma_i$$

Note that  $=_{-\mathcal{E}}$  is an equivalence.

**On local function calls** To conclude, we comment on the reasons behind the decision to make local function calls (i.e. calls to the TCB) visible to the attacker.

In principle, it is up to the code verifier to decide whether an output channel or a function call should be hidden. If there is a need to control the way local files are affected and local functions are invoked, they will probably not be hidden. On the other side, if the TCB can guarantee that no information is leaked as a result of those outputs or function calls, then they can safely be hidden. However, this last requirement is not easy to fulfill. For example, an attacker could be able to measure the time it takes for the TCB to execute a local function. Since this time may depend on the value of the actual parameter, the attacker would then be able to acquire some knowledge about it. By keeping the function call visible, such an information flow is detected. If, moreover, the amount of information thus leaked is acceptable, such a function call can be included in the policy. In this way, other (ab)uses of the TCB function are prevented.

### 5.2.2 $\Delta$ -Bisimilarity

In Cohen's definition of Selective Independency, the absence of variety at the output side of the communication channel (represented by the state transformer) is measured by comparing values for identity. In converting the definition to reactive systems, we can no longer use identity to compare the output of our communication channels. Indeed, the output is nothing less than a set of behaviors. This, in turn, suggests the use of some sort of behavioral equivalence. However, there is another ingredient to care about: the permutation of secrets continues in the future.

To see this more clearly, suppose that  $p \in a\text{-SecPA}$  can execute two histories  $\sigma, \sigma' \in Tr$  that differ only on secrets, i.e.  $\sigma =_{-\mathcal{E}} \sigma'$ , so that:

$$p \xrightarrow{\sigma}_a q_1 \quad \text{and} \quad p \xrightarrow{\sigma'}_a q_2$$

If we require  $q_1 \sim q_2$  where  $\sim$  is some usual behavioral equivalence (say, strong equivalence), we would be ignoring the fact that the environments of  $q_1$  and  $q_2$  are different (since they have provided different secrets in  $\sigma$  than in  $\sigma'$ ).

The difference in the environments can be represented by a relation over traces that not only reflects the permutation of secrets in  $\sigma$  and  $\sigma'$ , but also relates possible continuations of them. We can then define a behavioral relation between processes (provided with their history) that is parameterized by this trace relation. The flavor of the definition is that of a normal bisimulation relation.

**Definition 5.9 ( $\Delta$ -bisimulation)** *Let  $\Delta \subseteq Tr \times Tr$  be a relation over traces. Then, a relation  $R$  over  $a\text{-SecPA} \times Tr$  is a  $\Delta$ -bisimulation iff for any pairs  $(q, \sigma)$  and  $(t, \sigma')$  such that  $(q, \sigma) R (t, \sigma')$ ,*

- if  $q \xrightarrow{\alpha}_a q'$  then there are  $t' \in a\text{-SecPA}$  and  $\beta \in a\text{-Act}$  such that  $\sigma\alpha \Delta \sigma'\beta$ ,  $t \xrightarrow{\beta}_a t'$  and  $(q', \sigma\alpha) R (t', \sigma'\beta)$ , and
- if  $t \xrightarrow{\beta}_a t'$  then there are  $q' \in a\text{-SecPA}$  and  $\alpha \in a\text{-Act}$  such that  $\sigma\alpha \Delta \sigma'\beta$ ,  $q \xrightarrow{\alpha}_a q'$  and  $(q', \sigma\alpha) R (t', \sigma'\beta)$ .

In the usual way, we adapt the definition into a relation between processes:

**Definition 5.10 ( $\Delta$ -bisimilarity)** *Given  $\Delta \subseteq Tr \times Tr$ , two processes  $p, q \in a\text{-SecPA}$  are  $\Delta$ -bisimilar if there exists a  $\Delta$ -bisimulation  $R$  such that*

$$(p, \lambda) R (q, \lambda)$$

*In this case, we write  $p \sim_{\Delta} q$ .*

**Properties** One nice feature of  $\Delta$ -bisimilarity is that it inherits several important properties from  $\Delta$ . In the remainder of this subsection we explore these properties, which include reflexivity, symmetry and transitivity. We conclude with a result that characterizes  $\Delta$ -bisimulation.

**Proposition 5.11** *If  $p \sim_{\Delta} q$  then  $q \sim_{\Delta^{-1}} p$  where  $\Delta^{-1}$  is the relation satisfying  $\sigma'\Delta^{-1}\sigma$  iff  $\sigma\Delta\sigma'$  (i.e.  $(\sim_{\Delta})^{-1} = \sim_{\Delta^{-1}}$ ).*

*Proof.* Let  $R$  be a  $\Delta$ -bisimulation relation such that  $(p, \lambda) R (q, \lambda)$ . Then,  $R^{-1}$  is a  $\Delta^{-1}$ -bisimulation relation satisfying  $(q, \lambda) R^{-1} (p, \lambda)$ .  $\square$

**Corollary 5.12** *If  $\Delta$  is a symmetric relation, then so is  $\sim_{\Delta}$ .*

Let  $\Delta_1; \Delta_2$  be the composition relation satisfying  $\sigma \Delta_1; \Delta_2 \sigma'$  iff there is  $\gamma$  such that  $\sigma \Delta_1 \gamma$  and  $\gamma \Delta_2 \sigma'$ . The following proposition tells us how to compose  $\Delta$ -bisimilarities, a fundamental result in relating Admissibility with Selective Independency (Section 5.4).

**Proposition 5.13** *If  $p \sim_{\Delta_1} q$  and  $q \sim_{\Delta_2} r$  then  $p \sim_{\Delta_1; \Delta_2} r$  (i.e.  $\sim_{\Delta_1}; \sim_{\Delta_2} \subseteq \sim_{\Delta_1; \Delta_2}$ ).*

*Proof.* Let  $R_i$  be a  $\Delta_i$ -bisimulation for each  $i = 1, 2$  such that  $(p, \lambda) R_1 (q, \lambda)$  and  $(q, \lambda) R_2 (r, \lambda)$ . To show that  $p \sim_{\Delta_1; \Delta_2} r$ , define  $R \triangleq R_1; R_2$ . The relation immediately satisfies  $(p, \lambda) R (r, \lambda)$ .

Take any  $(s_0, \sigma_0) R (s_2, \sigma_2)$ . Therefore  $\exists (s_1, \sigma_1)$  s.t.  $(s_0, \sigma_0) R_1 (s_1, \sigma_1)$  and  $(s_1, \sigma_1) R_2 (s_2, \sigma_2)$ . Suppose  $s_0 \xrightarrow{\alpha_0}_a s'_0$ . First, since  $R_1$  is a  $\Delta_1$ -bisimulation, there are  $\alpha_1$  and  $s'_1$  such that

$$s_1 \xrightarrow{\alpha_1}_a s'_1 \text{ and } (s'_0, \sigma_0\alpha_0) R_1 (s'_1, \sigma_1\alpha_1)$$

Then, since  $(s_1, \sigma_1) R_2 (s_2, \sigma_2)$  and  $R_2$  is a  $\Delta_2$ -bisimulation, there are  $\alpha_2$  and  $s'_2$  such that

$$s_2 \xrightarrow{\alpha_2}_a s'_2 \text{ and } (s'_1, \sigma_1 \alpha_1) R_2 (s'_2, \sigma_2 \alpha_2)$$

We can thus conclude that there exist  $\alpha_2$  and  $s'_2$  such that  $s_2 \xrightarrow{\alpha_2}_a s'_2$  and  $(s'_0, \sigma_0 \alpha_0) R (s'_2, \sigma_2 \alpha_2)$ .

The remaining case is analogous.  $\square$

Suppose  $R$  is a  $\Delta_1$ -bisimulation. If  $\Delta_1 \subseteq \Delta_2$ , then it is clear that  $R$  is also a  $\Delta_2$ -bisimulation. This leads to the following proposition:

**Proposition 5.14** *If  $\Delta_1 \subseteq \Delta_2$  and  $p \sim_{\Delta_1} q$  then  $p \sim_{\Delta_2} q$  (i.e.  $\Delta_1 \subseteq \Delta_2 \Rightarrow \sim_{\Delta_1} \subseteq \sim_{\Delta_2}$ ).*

**Corollary 5.15** *If  $\Delta$  is a transitive relation, then so is  $\sim_\Delta$ .*

*Proof.* By Proposition 5.13,  $\sim_\Delta; \sim_\Delta \subseteq \sim_{\Delta; \Delta}$ . Since  $\Delta$  is transitive,  $\Delta; \Delta \subseteq \Delta$ . Therefore, using Proposition 5.14,  $\sim_\Delta; \sim_\Delta \subseteq \sim_\Delta$ .  $\square$

A binary relation is said to be a partial equivalence (PER) if it is symmetric and transitive. Together, Corollaries 5.12 and 5.15 show that  $\sim_\Delta$  is a PER if so is  $\Delta$ . Moreover, since  $\sim_{id}$  coincides with  $\sim$  (i.e. strong bisimulation equivalence), Proposition 5.14 proves that if  $\Delta$  is reflexive, then  $\sim_\Delta$  is reflexive as well.

It is possible to characterize the  $\Delta$ -bisimulations that make two processes  $\Delta$ -bisimilar. In particular, the domain and range of  $\Delta$  must cover the traces of the two bisimilar processes, as Corollary 5.17 shows.

**Lemma 5.16** *Let  $R$  be a  $\Delta$ -bisimulation relation and  $(p, \lambda) R (q, \lambda)$ . If  $p \xrightarrow{\sigma}_a p'$  and  $\sigma \neq \lambda$  then there exist  $q', \sigma'$  such that  $q \xrightarrow{\sigma'}_a q', \sigma \Delta \sigma'$  and  $(p', \sigma) R (q', \sigma')$ .*

*Proof.* By induction on the length of  $\sigma$ .

**Base case:**  $\sigma = \alpha$

Since  $p \xrightarrow{\alpha}_a p'$  and  $(p, \lambda) R (q, \lambda)$ , there are  $q'$  and  $\beta$  such that  $q \xrightarrow{\beta}_a q', \alpha \Delta \beta$  and  $(p', \alpha) R (q', \beta)$ . The result follows taking  $\sigma' = \beta$ .

**Inductive case:** Let  $p \xrightarrow{\sigma}_a p_0 \xrightarrow{\alpha}_a p'$  and  $\text{length}(\sigma) > 0$ . By inductive hypothesis,  $\exists q_0, \sigma'. q \xrightarrow{\sigma'}_a q_0 \wedge \sigma \Delta \sigma' \wedge (p_0, \sigma) R (q_0, \sigma')$ . Now, since  $p_0 \xrightarrow{\alpha}_a p'$ , there are  $q'$  and  $\beta$  such that  $q_0 \xrightarrow{\beta}_a q', \sigma \alpha \Delta \sigma' \beta$  and  $(p', \sigma \alpha) R (q', \sigma' \beta)$ .  $\square$

**Corollary 5.17** *Let  $\text{Tr}(p) \triangleq \{\sigma | p \xrightarrow{\sigma}_a\}$  be the set of traces of process  $p$ . If  $p \sim_\Delta q$  then  $\text{Tr}(p) \subseteq \text{Dom}(\Delta)$  (and  $\text{Tr}(q) \subseteq \text{Rng}(\Delta)$ ).*

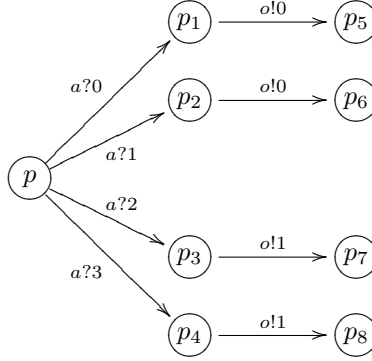
### 5.2.3 Selective Independency

One important element required in translating the definition of Selective Independency into *a-SecPA* concerns the identification of equivalent outputs from the communication channels identified by our information flow model. The intuition behind the definition of  $\Delta$ -bisimilarity, provided  $\Delta \subseteq =_{\mathcal{E}} \setminus id$ , is that  $p \sim_{\Delta} p$  should amount to checking that variety of inputs is not conveyed into variety of outputs, from the standpoint of the observer. However, this is not really true, as the following example illustrates:

Let

$$p \triangleq a?x. ((\text{if } (x = 0 \vee x = 1) \text{ then } o!0) + (\text{if } (x = 2 \vee x = 3) \text{ then } o!1))$$

If  $\mathcal{E} = \{a\}$ , and the only values that can be input on channel  $a$  are 0, 1, 2, 3, then the associated labelled transition system given by the operational semantics of process  $p$  is:



It is clear that process  $p$  leaks partial information about its secret input. This is so, as the process establishes a correlation between outputs and secret inputs.

Now, if we take  $\Delta$  to be exactly  $=_{\mathcal{E}} \setminus id$ , then  $p \sim_{\Delta} p$  holds by means of the following  $\Delta$ -bisimulation:

$$R \triangleq \{((p, \lambda), (p, \lambda)), ((p_1, a?0), (p_2, a?1)), ((p_5, a?0; o!0), (p_6, a?1; o!0)), ((p_3, a?2), (p_4, a?3)), ((p_7, a?2; o!1), (p_8, a?3; o!1)), \dots\}$$

where the dots represent all other pairs that are symmetric to the listed ones.

*What went wrong?* Although  $\Delta$  relates traces where secret inputs are permuted in all possible ways, the definition of  $\sim_{\Delta}$  contains an existential quantification that permits the choice of an appropriate permutation in each case.

In this way, the transition  $p \xrightarrow{a?0}_a p_1$  is simulated by  $p \xrightarrow{a?1}_a p_2$ , thus avoiding the comparison of different outputs.

If the definition of  $\Delta$ -bisimulation were modified to use universal quantification (over  $\beta$ , see Definition 5.9), then  $p \xrightarrow{a?0}_a p_0$  would have to be simulated

by, among others,  $p \xrightarrow{a?2}_a p_3$ . But this would come at a very high cost. Most of the key properties of  $\Delta$ -bisimilarity would be lost. We take a different road. Noticing that the problem could be avoided by reducing the variety tolerated by  $\Delta$ , we require  $\Delta$  to be a “trace permuter” (the definition follows). The full range of variety is then recovered by defining Selective Independency in a way inspired by Proposition 5.7 (Separation of Variety).

**Definition 5.18** *A relation  $\Delta \subseteq Tr \times Tr$  is functional iff*

$$\forall \sigma, \gamma, \gamma' \in Tr. \sigma \Delta \gamma \wedge \sigma \Delta \gamma' \Rightarrow \gamma = \gamma'$$

**Definition 5.19** *A relation  $\Delta \subseteq Tr \times Tr$  is a trace permuter if both  $\Delta$  and  $\Delta^{-1}$  are functional.*

Observe that the class of trace permuters is closed under composition.

By now, we have almost all the elements needed to provide a definition of Selective Independency for *a-SecPA* processes. The last element missing is an appropriate notion of  $\mathcal{E}$ -strictness. From a technical point of view, we just need to define when two traces contain the same secrets (i.e.  $=_{\mathcal{E}}$ ), and then reuse Definition 5.1. However, *what is the intuition behind  $\mathcal{E}$ -strictness in the context of reactive systems?* Again, we draw inspiration from the definition of Nondeducibility on Compositions. A *strategy* is a process which feeds in only high-level data (i.e. secrets) into a system, and the way this data is fed does not prevent the system from engaging in any other low-level actions. Two traces coincide on secrets if they contain the same secret inputs, in the same order and regardless of any other actions. In the tradition of information flow definitions, we use a purge function to define  $=_{\mathcal{E}}$ :

Given a set of secret channels  $\mathcal{E}$ , let  $purge: Tr \rightarrow Tr$  be defined as:

$$\begin{aligned} purge(\tau) &\triangleq \tau \\ purge(\alpha\sigma) &\triangleq \begin{cases} \alpha purge(\sigma) & \text{if } (\alpha = c?v \text{ or } \alpha = (v := c(w))) \text{ and } \alpha \in \mathcal{E} \\ purge(\sigma) & \text{otherwise} \end{cases} \end{aligned}$$

Two traces contain the same secrets if their images under  $purge(\cdot)$  are equal:

$$\sigma =_{\mathcal{E}} \sigma' \text{ iff } purge(\sigma) = purge(\sigma')$$

As for state transformers,  $\mathcal{E}$ -strictness is defined in terms of  $=_{\mathcal{E}}$  and  $=_{\neg\mathcal{E}}$  (cf. Def. 5.1). A relation  $\Delta$  is  $\mathcal{E}$ -strict if

$$((=_{\mathcal{E}}; \Delta; =_{\mathcal{E}}) \cap =_{\neg\mathcal{E}}) \subseteq \Delta \subseteq =_{\neg\mathcal{E}}$$

With all the necessary elements at hand, it is now easy to provide a definition of Selective Independency for *a-SecPA*, using Definition 5.2 as a model with a flavor of Separation of Variety (cf. Prop. 5.7).

**Definition 5.20 (Selective Independency for a-SecPA)** Let  $\mathcal{E}$  be a set of channels, and  $\Delta$  an  $\mathcal{E}$ -strict equivalence relation over traces.

The behavior of program  $p$  is selectively independent of set of channels  $\mathcal{E}$  w.r.t.  $\Delta$ , noted  $\mathcal{E} \not\bowtie_{\Delta}^p$ , iff there is a family of symmetric trace permuters  $\{\Delta_i\}$  such that  $\Delta = (\bigcup_i \Delta_i)^*$  and  $\forall i. p \sim_{\Delta_i} p$ .

Going back to the example at the beginning of this section, consider the following relation over traces:

$$\Delta_1 \triangleq \Delta_1^0 \cup \{(\alpha; o!z, \beta; o!z) \mid z \in \{0, 1\}, \alpha \Delta_1^0 \beta\}$$

where  $\Delta_1^0 \triangleq \{(a?0, a?2), (a?2, a?0), (a?1, a?1), (a?3, a?3)\}$ .

Note that  $\Delta_1$  is a symmetric trace permuter satisfying  $\Delta_1 \subseteq =_{-\mathcal{E}}$ , and that  $=_{-\mathcal{E}}$  is (trivially) an  $\mathcal{E}$ -strict equivalence. It is also easy to check that  $p \not\sim_{\Delta_1} p$ . Moreover, no matter how  $=_{-\mathcal{E}}$  is decomposed as the transitive closure of the union of trace permuters, there will always be at least a trace permuter  $\Delta_i$  making  $p \not\sim_{\Delta_i} p$ . As we wanted,  $p$  does not happen to be selectively independent of  $\{a\}$  w.r.t.  $=_{-\mathcal{E}}$ .

### 5.3 Admissibility as $\Delta$ -Bisimulations

While admissibility is defined in terms of relabelling functions and strong bisimulation, Selective Independency heavily relies on  $\Delta$ -bisimulation. By choosing the  $\Delta$  relation appropriately, all these concepts can be shown to be strongly related. As a result, we can give a simple proof to Theorem 4.15 (which was already used by the end of Chapter 4 to show the admissibility of the proposed implementation of the Wide-Mouthed Frog protocol). Finally, we show how the connection can also be exploited to identify inadmissible processes, something we illustrate on some “malicious” Purchasing Applet examples.

To begin with, notice that there is a natural way to define a relation over traces that mimics the effect of repeatedly applying a relabelling.

**Definition 5.21** Let  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  be a confidentiality property and  $g$  a secret permuter that preserves admissible outputs. Then,  $\Delta_g$  is the smallest relation over traces satisfying:

- $\lambda \Delta_g \lambda$ , and
- if  $\sigma \Delta_g \sigma'$  and  $s_0 \xrightarrow{\sigma} s$  then  $\forall \alpha \in a\text{-Act}. \sigma \alpha \Delta_g \sigma' f_g(s, \alpha)$

Our first result indicates how pairs of traces related by  $\Delta_g$  bring about related contexts.

**Proposition 5.22** If  $\sigma \Delta_g \sigma'$  and  $s_0 \xrightarrow{\sigma} s$  then  $s_0 \xrightarrow{\sigma'} \bar{g}(s)$ .

*Proof.* Direct by induction on the length of  $\sigma$ , applying Lemma 4.12.  $\square$

In the previous section it was noticed that  $\sim_\Delta$  needs a trace permuter in order to make full sense. This is actually the case with  $\Delta_g$ :

**Proposition 5.23** *The relation  $\Delta_g$  is symmetric.*

*Proof.* See Appendix A.

**Observation 5.24** *The relation  $\Delta_g$  is a trace permuter.*

We are now in a position to relate  $\Delta_g$ -bisimulation with the kind of basic propositions appearing in the definition of admissibility:

**Proposition 5.25** *For all  $p \in a\text{-SecPA}$ ,  $p \sim_{\Delta_g} p \Leftrightarrow p \sim p[s_0 \vdash f_g]$*

*Proof.*

$\Rightarrow$ ) Let  $R$  be a  $\Delta_g$ -bisimulation s.t.  $(p, \lambda) R (q, \lambda)$ . Let  $T$  be the binary relation uniquely determined by:  $r T q[s \vdash f_g]$  iff there are  $\sigma$  and  $\sigma'$  such that  $s_0 \xrightarrow{\sigma} s$  and  $(q, \sigma) R (r, \sigma')$ .

We show that  $T$  is a bisimulation relation and that  $p T p[s_0 \vdash f_g]$ . The latter is immediate from our definition since  $s_0 \xrightarrow{\lambda} s_0$  and  $(p, \lambda) R (p, \lambda)$ . In what follows, assume that  $r T q[s \vdash f_g]$ .

- If  $r \xrightarrow{\beta}_a r'$ , then by definition of  $\Delta_g$ -bisimulation, there are  $\alpha$  and  $q'$  satisfying (1)  $q \xrightarrow{\alpha}_a q'$ , (2)  $\sigma\alpha \Delta_g \sigma'\beta$ , and (3)  $(q', \sigma\alpha) R (r', \sigma'\beta)$ .

First, from (2),  $s_0 \xrightarrow{\sigma} s$  and the definition of  $\Delta_g$ , it follows that (4)  $\beta = f_g(s, \alpha)$ . Then, if we call  $s'$  the unique context satisfying (5)  $s \xrightarrow{\alpha} s'$ , we can combine (1), (4) and (5) using the definition of relabelled processes (Def. 4.13), and conclude that

$$q[s \vdash f_g] \xrightarrow{\beta}_a q'[s' \vdash f_g]$$

Finally, from (3) and  $s_0 \xrightarrow{\sigma\alpha} s'$ ,  $r' T q'[s' \vdash f_g]$ .

- If  $q[s \vdash f_g] \xrightarrow{\beta}_a q'[s' \vdash f_g]$ , the definition of relabelled processes (Def. 4.13) allows us to infer that there is an  $\alpha$  satisfying (1)  $q \xrightarrow{\alpha}_a q'$ , (2)  $s \xrightarrow{\alpha} s'$  and (3)  $\beta = f_g(s, \alpha)$ . From (1) and the definition of  $R$ , there are  $\beta'$  and  $r'$  such that (4)  $r \xrightarrow{\beta'}_a r'$ , (5)  $\sigma\alpha \Delta_g \sigma'\beta'$  and (6)

$(q', \sigma\alpha) R (r', \sigma'\beta')$ . From the definition of  $\Delta_g$  and (5), we get that  $\beta' = f_g(s, \alpha)$ , which implies  $\beta = \beta'$ . Finally, from (2) and (6), we obtain  $r' T q'[s' \vdash f_g]$ .

$\Leftarrow$ ) Let  $T$  be a bisimulation relation such that  $p T p[s_0 \vdash f_g]$ . Let  $R$  be the binary relation uniquely determined by:  $(q, \sigma) R (r, \sigma')$  iff  $\sigma \Delta_g \sigma'$  and  $r T q[s \vdash f_g]$  where  $s_0 \xrightarrow{\sigma} s$ .

We show that  $R$  is a  $\Delta_g$ -bisimulation relation and that  $(p, \lambda) R (p, \lambda)$ .

The latter is immediate from our definition since  $s_0 \xrightarrow{\lambda} s_0$ ,  $p T p[s_0 \vdash f_g]$  and  $\lambda \Delta_g \lambda$ .

In what follows, assume that  $(q, \sigma) R (r, \sigma')$ .

– If  $q \xrightarrow{\alpha}_a q'$ , then (1)  $q[s \vdash f_g] \xrightarrow{\beta}_a q'[s' \vdash f_g]$  where (2)  $s \xrightarrow{\alpha} s'$  and (3)  $\beta = f_g(s, \alpha)$ . Since  $T$  is a bisimulation relation, there is  $r'$  such that (4)  $r \xrightarrow{\beta}_a r'$  and (5)  $r' T q'[s' \vdash f_g]$ . Using the definition of  $\Delta_g$ , (3) and the assumption, we obtain (6)  $\sigma\alpha \Delta_g \sigma'\beta$ . Finally, from (2), (5) and (6),  $(q', \sigma\alpha) R (r', \sigma'\beta)$ .

– Suppose  $r \xrightarrow{\beta}_a r'$ . Since  $T$  is a bisimulation relation, there are  $q'$  and  $s'$  such that (1)  $q[s \vdash f_g] \xrightarrow{\beta}_a q'[s' \vdash f_g]$  and (2)  $r' T q'[s' \vdash f_g]$ .

From (1), there must be an  $\alpha$  satisfying  $q \xrightarrow{\alpha}_a q'$ , (3)  $s \xrightarrow{\alpha} s'$  and (4)  $\beta = f_g(s, \alpha)$ . In turn, (4) and the assumptions imply that (5)  $\sigma\alpha \Delta_g \sigma'\beta$ . Putting together (2), (3) and (5), it follows that  $(q', \sigma\alpha) R (r', \sigma'\beta)$ .  $\square$

**Verification of Admissibility** As a first consequence of the previous result, we can give a simple proof of Theorem 4.15. This theorem, presented in Section 4.4.1, localizes the verification of admissibility (provided that the states of the process to be verified can be given a uniform characterization).

**Theorem 4.15** *Let  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  be a confidentiality property. Given  $p \in \text{SecPA}$ , assume there is a partial function  $p_- : \mathcal{N} \times \text{Context} \rightarrow a\text{-SecPA}$  satisfying:*

*P1)  $(\mathcal{E} \triangleright p) = p_0(s_0)$ ,*

*P2) if  $p_i(s) \xrightarrow{\alpha}_a p'$  then  $\exists j. p' = p_j(s')$  where  $s \xrightarrow{\alpha} s'$ , and*

*P3) if  $p_i(s) \xrightarrow{\alpha}_a p_j(s')$  then  $p_i(\bar{g}(s)) \xrightarrow{f_g(s, \alpha)}_a p_j(\bar{g}(s'))$ , for all secret permuter  $g$  that preserves admissible outputs.*

*Then process  $p$  is admissible w.r.t.  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$ .*

*Proof.* Because of Proposition 5.25, we just have to show that  $p_0(s_0) \sim_{\Delta_g} p_0(s_0)$  for any secret permuter  $g$ .

Define

$$R \triangleq \{((p_i(s), \sigma), (p_i(t), \sigma')) \mid i \in \mathcal{N}, \sigma \Delta_g \sigma', p_0(s_0) \xrightarrow{\sigma}_a p_i(s), s_0 \xrightarrow{\sigma} s, \\ p_0(s_0) \xrightarrow{\sigma'}_a p_i(t), s_0 \xrightarrow{\sigma'} t\}$$

To see that  $(p_0(s_0), \lambda) R (p_0(s_0), \lambda)$ , note that  $\lambda \Delta_g \lambda$ ,  $p_0(s_0) \xrightarrow{\lambda}_a p_0(s_0)$  and  $s_0 \xrightarrow{\lambda} s_0$ . In what remains, assume  $(p_i(s), \sigma) R (p_i(t), \sigma')$ . Since  $\Delta_g$  is symmetric (Prop. 5.23),  $R$  is symmetric too. Therefore, it suffices to consider the case  $p_i(s) \xrightarrow{\alpha}_a p'$ . By P2), there is  $j$  such that  $p' = p_j(s')$  where  $s \xrightarrow{\alpha} s'$ , and by Proposition 5.22,  $t = \bar{g}(s)$ . Let  $\beta = f_g(s, \alpha)$  so that  $\sigma \alpha \Delta_g \sigma' \beta$ . Applying P3), we get  $p_i(\bar{g}(s)) \xrightarrow{\beta}_a p_j(\bar{g}(s'))$ . In other words,  $p_i(t) \xrightarrow{\beta}_a p_j(\bar{g}(s'))$ . Finally, it is easy to check that  $(p_j(s'), \sigma \alpha) R (p_j(\bar{g}(s')), \sigma' \beta)$ .  $\square$

**The Malicious Applets are not Admissible** Proposition 5.25 can also be used to identify inadmissible processes. We take a look back at Examples 3.7 and 3.11, two implementations of the purchasing applet idea that leak information about the account number (as input from channel *acc*).

Consider first the process  $MA_1$  (from Example 3.7). If it were admissible, then we would have  $(\mathcal{E} \triangleright MA_1) \sim (\mathcal{E} \triangleright MA_1)[s_0 \vdash f_g]$ . Because of Proposition 5.25, this would imply that  $(\mathcal{E} \triangleright MA_1) \sim_{\Delta_g} (\mathcal{E} \triangleright MA_1)$ .

Let

$$\sigma \triangleq acc?v_1 ; order?v_2 ; acc?v_3 ; k_1 := pubKey(v_1)$$

and

$$\sigma' \triangleq acc?v_1 ; order?v_2 ; acc?v'_3 ; k_1 := pubKey(v_1)$$

where  $(v'_3 : acc) = g(v_3 : acc)$ . Note that  $\sigma \Delta_g \sigma'$ .

Now, suppose there exists a  $\Delta_g$ -bisimulation  $R$  to verify this. Then, it must be the case that

$$\begin{array}{c} (\mathcal{E} \triangleright merchant!(v_1, v_3 : acc, \{(v_2, v_3 : acc)\}_{k_1}).MA_1, \sigma) \\ R \\ (\mathcal{E} \triangleright merchant!(v_1, v'_3 : acc, \{(v_2, v'_3 : acc)\}_{k_1}).MA_1, \sigma') \end{array}$$

However, there is no possible *Context*  $s$  such that

$$(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash merchant!(v_1, v_3 : acc, \{(v_2, v_3 : acc)\}_{k_1}) \text{ ok}$$

and therefore  $\sigma ; \text{merchant}!(v_1, v_3 : \text{acc}, \{(v_2, v_3 : \text{acc})\}_{k_1})$  can only be related by  $\Delta_g$  to  $\sigma' ; \text{merchant}!(v_1, v_3 : \text{acc}, \{(v_2, v_3 : \text{acc})\}_{k_1})$ . Since

$$\mathcal{E} \triangleright \text{merchant}!(v_1, v'_3 : \text{acc}, \{(v_2, v'_3 : \text{acc})\}_{k_1}).MA_1$$

cannot perform the output

$$\text{merchant}!(v_1, v_3 : \text{acc}, \{(v_2, v_3 : \text{acc})\}_{k_1})$$

we conclude that  $R$  is cannot be a  $\Delta_g$ -bisimulation and, therefore, that  $MA_1$  is not admissible w.r.t. to our policy.

The case of  $MA_2$  is analogous. Suppose  $\sigma \Delta_g \sigma'$  with

$$\sigma \triangleq \text{acq}?v_1 ; \text{order}?v_2 ; \text{acc}?v_3 ; k_1 := \text{pubKey}(\text{MerchantId})$$

and

$$\sigma' \triangleq \text{acq}?v_1 ; \text{order}?v_2 ; \text{acc}?v'_3 ; k_1 := \text{pubKey}(\text{MerchantId})$$

where  $(v'_3 : \text{acc}) = g(v_3 : \text{acc})$ .

If there were a  $\Delta_g$ -bisimulation  $R$  verifying  $(\mathcal{E} \triangleright MA_2) \sim_{\Delta_g} (\mathcal{E} \triangleright MA_2)$ , then it should be the case that

$$\begin{array}{c} (\mathcal{E} \triangleright \text{merchant}!(v_1, v_3 : \text{acc}, \{(v_2, v_3 : \text{acc})\}_{k_1}).MA_2, \sigma) \\ R \\ (\mathcal{E} \triangleright \text{merchant}!(v_1, v'_3 : \text{acc}, \{(v_2, v'_3 : \text{acc})\}_{k_1}).MA_2, \sigma') \end{array}$$

Now, if  $s_0 \xrightarrow{\sigma} s$ , then  $s$  does not define a mapping over  $\text{pubKey}(v_1)$  if this is different from  $k_1$ . Then it is not the case that

$$(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \text{merchant}!(v_1, v_3 : \text{acc}, \{(v_2, v_3 : \text{acc})\}_{k_1}) \text{ ok},$$

and the analysis continues just like in the case of  $MA_1$ .

## 5.4 Admissibility vs. Selective Independency

The previous section has shown that there is a common language underlying both the definitions of Admissibility and Selective Independency. The combination of strong equivalence and relabellings in Admissibility is comparable to  $\Delta$ -bisimilarities as they appear in Selective Independency. In spite of the similar bases, these properties have notably different purposes. While Selective Independency pursues the cancellation of all output variety under a restricted variety of inputs, Admissibility aims at controlling the flows of information in an application.

In this section we establish a connection between both properties by essentially associating a selectively independent process to an admissible one. Given an *a-SecPA* process  $p$  and a confidentiality policy  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  we can construct a process  $q$  satisfying the following requirement: its behaviors “mimic” those of  $p$  but, on admissible outputs, it never sends out the actual secret inputs. The way  $p$  and  $q$  are related gives a quite thorough characterization of Admissibility.

**Extended Annotated Values** We extend the set of annotated values with a new, distinguished value  $v_d$  for each channel  $d \in Ch$ . The set of extended annotated values is thus defined as  $e-Val \triangleq a-Val \uplus \{v_d \mid d \in Ch\}$ , where  $\uplus$  indicates disjoint union. We extend, in a similar fashion, the set of annotated actions that are constructed from  $e-Val$ , and call the resulting set  $e-Act$ .

Given an element of  $a-Act$ , we can abstract it into an action where each value input from secret channel  $c \in \mathcal{E}$  is replaced by  $v_c$ . We formalize this idea by means of the abstraction function  $abs: a-Act \rightarrow e-Act$ :

$$\begin{aligned} abs(k) &= k \\ abs(w_1, \dots, w_n) &= (abs(w_1), \dots, abs(w_n)) \\ abs(\{w_1\}_{w_2}) &= \{abs(w_1)\}_{abs(w_2)} \\ abs(p_i(w)) &= p_i(abs(w)) \\ abs(w_1 : c(w_2)) &= \begin{cases} v_c : c(abs(w_2)) & \text{if } c \in \mathcal{E} \text{ and } w_1 \text{ is a value} \\ abs(w_1) : c(abs(w_2)) & \text{otherwise} \end{cases} \end{aligned}$$

In the same way we have associated a relabelling function  $f_g$  to a secret permuter  $g$ , we can associate a relabelling  $\phi: Context \times a-Act \rightarrow e-Act$  to the abstraction function relabelling  $\phi$ :

$$\phi(s, \alpha) = \begin{cases} o!abs(w) & \text{if } (\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \alpha \text{ ok, and } \alpha = o!w \\ v := c(abs(w)) & \text{if } (\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \alpha \text{ ok, and } \alpha = (v := c(w)) \\ \alpha & \text{otherwise} \end{cases}$$

**Abstracted Processes** The abstraction function, when immersed in relabelling  $\phi$ , gives us a simple way of defining a process that abstracts  $p$  in the desired way. Define:

$$q \triangleq p[s_0 \vdash \phi]$$

Intuitively, if  $q$  does not leak any information about the secrets it inputs, then  $p$  leaks them at most through admissible outputs. More formally, we show that if  $p$  is admissible w.r.t. some policy, then its associated abstracted process  $q$  is selectively independent w.r.t. some equivalence relation  $\Gamma$  associated with the policy.

**Definition 5.26** *Let  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  be a confidentiality property. Then,  $\Pi$  is the smallest relation over traces satisfying:*

- $\lambda \Pi \lambda$ , and
- if  $\sigma \Pi \sigma'$  and  $s_0 \xrightarrow{\sigma} s$  then  $\forall \alpha \in a\text{-Act}. \sigma \alpha \Pi \sigma' \phi(s, \alpha)$

**Observation 5.27** *Since it is defined in terms of function  $\phi$ ,  $\Pi$  is a functional relation. Moreover, since  $\phi$  is injective in its second argument (Lemma A.2),  $\Pi^{-1}$  is functional too. Finally, this implies  $\Pi; \Pi^{-1} = id_{a\text{-Act}^*}$  because  $\Pi$  is total over  $a\text{-Act}^*$ .*

The most important property of  $\Pi$  is that, no matter what the confidentiality policy is, it is always the case that  $p$  and  $q$  are  $\Pi$ -bisimilar, as the following proposition shows:

**Proposition 5.28** *Let  $p$  be an  $a\text{-SecPA}$  process, then  $p \sim_{\Pi} p[s_0 \vdash \phi]$ .*

*Proof.* Let  $R \triangleq \{((q, \sigma), (q[s \vdash \phi], \sigma')) \mid p \xrightarrow{\sigma}_a q \wedge s_0 \xrightarrow{\sigma} s \wedge \sigma \Pi \sigma'\}$ . We will show that  $R$  is a  $\Pi$ -bisimulation s.t.  $(p, \lambda) R (p[s_0 \vdash \phi], \lambda)$ .

First, note that  $p \xrightarrow{\lambda}_a p$ ,  $s_0 \xrightarrow{\lambda} s_0$  and  $\lambda \Pi \lambda$ , so that  $(p, \lambda) R (p[s_0 \vdash \phi], \lambda)$ .

Suppose then that  $(q, \sigma) R (q[s \vdash \phi], \sigma')$ . If  $q \xrightarrow{\alpha}_a q'$ , then  $q[s \vdash \phi] \xrightarrow{\beta}_a q'[s' \vdash \phi]$  where  $s \xrightarrow{\alpha} s'$  and  $(1) \beta = \phi(s, \alpha)$ . From (1),  $\sigma \alpha \Pi \sigma' \beta$  and, therefore,  $(q', \sigma \alpha) R (q'[s' \vdash \phi], \sigma' \beta)$ .

If, on the other side,  $q[s \vdash \phi] \xrightarrow{\beta}_a q'[s' \vdash \phi]$ , then there must exist an  $\alpha$  such that  $q \xrightarrow{\alpha}_a q'$ ,  $s \xrightarrow{\alpha} s'$  and  $\beta = \phi(s, \alpha)$ . The rest is similar to the previous case.  $\square$

The importance of the previous result becomes evident when combined with the compositional properties of  $\Delta$ -bisimilarity:

**Corollary 5.29** *Let  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  be a confidentiality policy and  $g$  a secret permuter that preserves admissible outputs. Moreover, let  $p$  be an  $a\text{-SecPA}$  process and let  $q$  be  $p[s_0 \vdash \phi]$ . Then,  $p \sim p[s_0 \vdash f_g]$  iff  $q \sim_{(\Pi^{-1}; \Delta_g; \Pi)} q$ .*

*Proof.* From the proposition, we know that  $p \sim_{\Pi} q$ , and using Proposition 5.11 that  $q \sim_{\Pi^{-1}} p$ . From Proposition 5.25,  $p \sim p[s_0 \vdash f_g]$  iff  $p \sim_{\Delta_g} p$ . If  $p \sim_{\Delta_g} p$ , by the compositionality of  $\Delta$ -bisimilarities (Prop. 5.13),  $q \sim_{(\Pi^{-1}; \Delta_g; \Pi)} q$ . Composing again,  $p \sim_{(\Pi; \Pi^{-1}; \Delta_g; \Pi; \Pi^{-1})} p$ . Since  $\Pi; \Pi^{-1} = id_{a\text{-Act}^*}$  (Obs. 5.27),  $p \sim_{\Delta_g} p$ .  $\square$

By means of Corollary 5.29, if  $p$  is admissible,  $q$  does not convey variety of input (restricted by  $\Pi^{-1}; \Delta_g; \Pi$ ) into variety of behavior. However, even if each  $\Pi^{-1}; \Delta_g; \Pi$  is a symmetric trace permuter (see Lemma A.4), we cannot directly apply the definition of Selective Independency. The reason is mainly technical:

Let  $G$  be the set of secret permuters that preserve admissible outputs. Then, the relation  $(\bigcup_{g \in G} (\Pi^{-1}; \Delta_g; \Pi))^*$  is not  $\mathcal{E}$ -strict.

To circumvent this difficulty, define

$$\Gamma_g \triangleq (=_{\mathcal{E}}; \Pi^{-1}; \Delta_g; \Pi; =_{\mathcal{E}}) \cap =_{\neg \mathcal{E}}$$

and, consequently, call  $\Gamma$  the reflexive transitive closure of their union, i.e.

$$\Gamma \triangleq \left( \bigcup_{g \in G} \Gamma_g \right)^*$$

That is all we need. Indeed, the following proposition shows that it makes sense to ask whether  $\mathcal{E} \not\sim_{\Gamma}^q$  when  $p$  is admissible.

**Proposition 5.30** *For all  $g \in G$ ,  $\Gamma_g$  is a symmetric trace permuter satisfying  $\Pi^{-1}; \Delta_g; \Pi \subseteq \Gamma_g$ . Moreover, the resulting  $\Gamma$  is an  $\mathcal{E}$ -strict equivalence relation.*

*Proof.* See Appendix A.

The main result of this chapter can now be stated and proved. If a process is admissible, abstracting its admissible outputs renders it selectively independent w.r.t. an  $\mathcal{E}$ -strict trace relation.

**Proposition 5.31** *Let  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  be a confidentiality policy and  $p$  a SecPA process. Let  $\phi$  be the abstraction relabelling associated with the policy and let  $q \triangleq (\mathcal{E} \triangleright p)[s_0 \vdash \phi]$ .*

*If  $p$  is admissible w.r.t.  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  then  $\mathcal{E} \not\sim_{\Gamma}^q$ .*

*Proof.* By  $p$ 's admissibility,  $\mathcal{E} \triangleright p \sim (\mathcal{E} \triangleright p)[s_0 \vdash f_g]$  for all secret permuters  $g \in G$ . By Corollary 5.29,  $q \sim_{(\Pi^{-1}; \Delta_g; \Pi)} q$ . Since  $\Pi^{-1}; \Delta_g; \Pi \subseteq \Gamma_g$  (Prop. 5.30), by Proposition 5.14 we conclude that  $q \sim_{\Gamma_g} q$  for all  $g \in G$ . By the definition of Selective Independence (Def. 5.20),  $\mathcal{E} \not\sim_{\Gamma}^q$ .  $\square$

We conclude by observing that the permutation of secret inputs indicated by  $\Gamma$  is precisely defined by  $G$ , the set of secret permuters that is derived from the confidentiality policy. In other words, an admissible process does not convey any variety of secret input (as indicated by the confidentiality policy) but through concrete admissible outputs. Not only does this characterize admissibility, but hopefully justifies it, thus answering the question posed at the beginning of this chapter.



## Chapter 6

# Experimentation: An Architecture for Confidentiality

The definition of confidential protocol implementation constitutes a theoretical incursion into the problem of making sure that a piece of downloaded code preserves the confidentiality properties of a given protocol. Rather unsurprisingly, the concepts developed so far have necessarily dealt with abstract models, certainly quite far from providing a practical answer to the problem above. It is evident that a comprehensive solution demands that a number of concrete issues be addressed. These include, for example, the construction of an appropriate annotated semantics for the concrete programming language to use, the correct identification of interfaces, the specification of confidentiality properties, the assignment of meaning to these properties, and the actual verification mechanism. Criteria like performance, and ease of use and of deployment, are all imperative.

The agenda is certainly quite challenging, and many issues remain still open to be able to give a comprehensive solution. However, in this chapter, we investigate a possible path towards carrying admissibility into practice. This path takes the form of a confidentiality architecture. As usual, an architecture shows how different components interact in order to complete a task, which in this case aims at providing confidentiality guarantees to users of mobile code.

We first study the context of application trying to derive minimum requirements for the architecture. Taking those requirements, we then suggest a candidate architecture. Finally, we report on some experiments, draw conclusions on the applicability of the proposed architecture, and identify some open problems.

## 6.1 Requirements

Among other features, an architecture serves practitioners in structuring and applying abstract concepts. It can be seen, therefore, as a middle point between a theory and its application. We are interested in an architecture suggesting how to apply the ideas around admissibility.

Our architecture has to cope with code, supposedly implementing cryptographic protocols, as well as with confidentiality policies. It also has to define the rôles of the various participants, like the code producer, the issuer of the policy, and the user (code consumer). For the sake of illustration, assume that the code corresponds to applets written in Java/JVM. Each applet is requested to comply with an admissibility property. Each admissibility property is associated with a confidentiality policy.

There are three main aspects to consider: The formal modelling process, the enforcement of admissibility, and the management of confidentiality policies. In each case, we discuss the requirements that they put on the architecture.

**Semantics** In order to apply admissibility to JVM applets, we need to give them an annotated semantics, as in Chapter 3. This, in turn, calls for a concrete identification of input and output channels. That is, the architecture should identify sources of secret data. This is not as easy as it may originally seem, specially if the source of the secret is the user himself (e.g. in the Purchasing Applet example, the user provides the secret account number). The architecture should therefore make provisions for the correct identification of channels even in specific cases.

**Enforcement of Admissibility** The architecture should not describe the actual mechanism used to verify admissibility, but it may well identify the agent(s) responsible of performing the verification, and suggest a procedure. Moreover, since the code consumer does not necessarily trust the code producer, the architecture should clearly identify which parts are trustable and which are not.

With respect to the verification procedure, it is worth considering whether the current mechanisms provided by the Java Runtime Environment could be adapted to such a task. The mechanism are two: A Bytecode Verifier, which performs a series of program analyses on the code –at loading time– to make sure that, when executed, it will not violate certain (fixed) safety restrictions; and a Security Manager, in charge of enforcing access control at runtime. Since admissibility is a property of sets of behaviors, it cannot be checked dynamically. This excludes the Security Manager. On the other hand, the Bytecode Verifier could be considered as a possibility only if we could reshape the verification of admissibility as a static analysis.

An alternative is to use a mechanism based on *Proof-Carrying Code* [NL98b]: PCC was originally designed to certify safety properties of untrusted code.

These properties include, among others, memory safety, type safety and conformance to resource usage bounds. The fundamental idea behind PCC consists in attaching to the code an easily checkable proof of compliance to a safety policy. This policy is expected to have been agreed upon, by both the receiver and the sender, before the transmission of the code.

**Policies** An architecture for admissibility properties should also address the generation and transmission of confidentiality policies. Contrary to the safety properties considered by Necula and Lee, our policies are highly application dependent. Therefore, we must not assume that the user can easily determine which policy to use, before the actual application is considered.

Moreover, we cannot assume in general that the user is able to interpret the meaning of any given policy. Therefore, the architecture should make provisions for some sort of policy advisor in charge of assisting the user on the interpretation of policies. This interpretation includes the association of the policy to a protocol, a task that, in some cases, could be handed over to an authentication authority.

In a way, a policy assumes that the user is part of the TCB. For example, if a user is requested to enter an address, it is expected the data provided does not correspond to an account number, or anything else but the address itself. We could say that the confidentiality properties embodied by a policy depend upon the correct behavior of the user. Since this behavior cannot be constrained, we can only expect to inform the user correctly on which behavior is actually expected. In the case of the purchasing applet example, the user has to be informed of the text field where to feed in the account number. The fact that this text field might appear in a window next to a tag saying “User PIN number input” does not really identify the channel. The successful application of our confidentiality policy for the purchasing applet depends on the ability of the policy advisor to inform the user.

## 6.2 A PCC Architecture for Confidentiality

We propose an architecture based on Proof-Carrying Code, with confidentiality policies as specifications of admissibility. The use of PCC serves two purposes: it lets us experiment with this technology in the context of confidentiality properties (instead of the more traditional safety ones), and it frees us from the need of automatizing the verification of admissibility. Even if there were such an automatic mechanism, we could embed it in our PCC architecture (as part of a certifying compiler), thus showing the flexibility of this proposal.

Instead of using PCC information as a code filter, weeding out from execution those applets that fail some statically determined property we use successful PCC checks to support a security assistant which will let applet users inquire about its security properties, such as the destination of data which is input into the different fields.

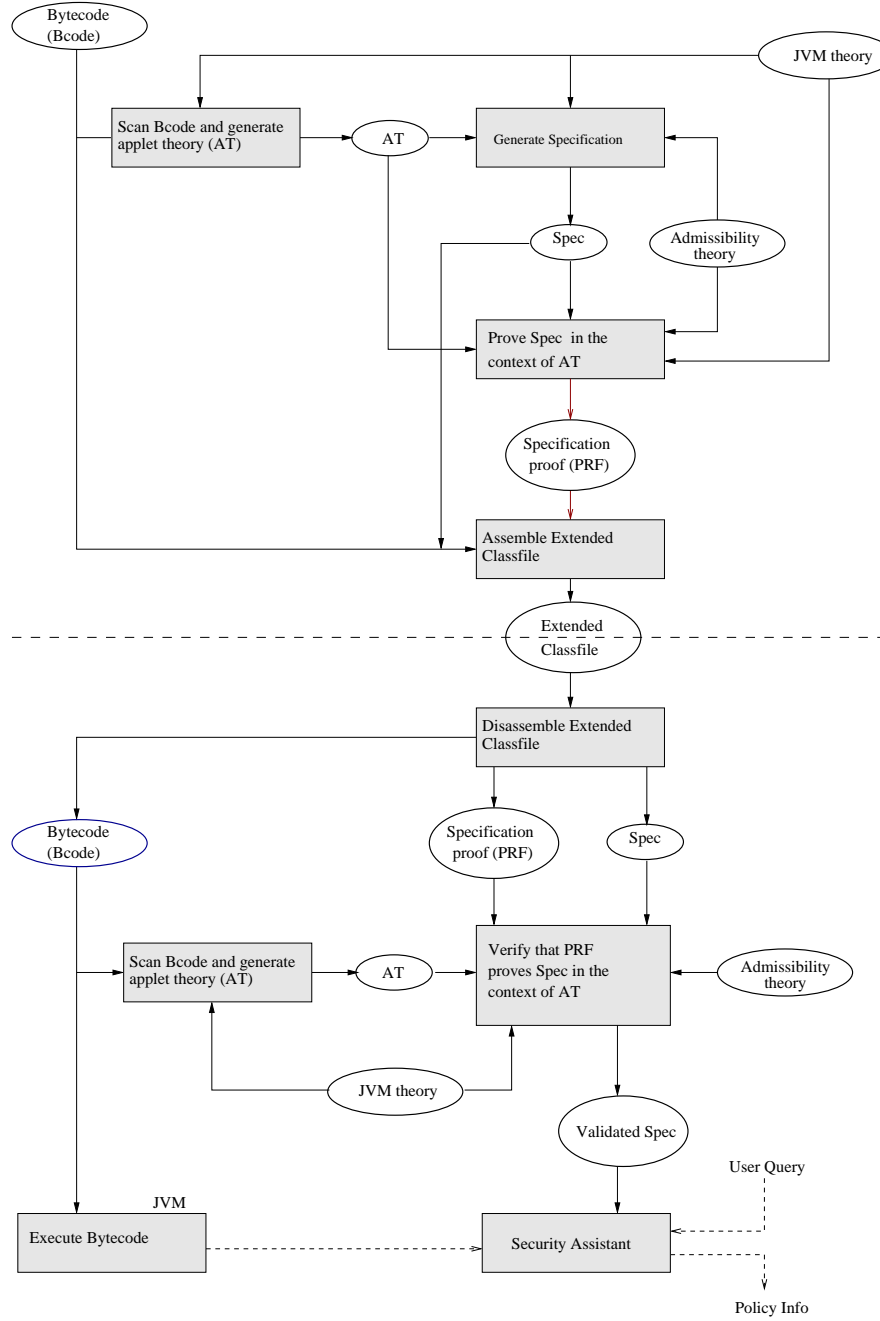


Figure 6.1: A Proof-Carrying Code Architecture for Confidentiality

Figure 6.1 shows the different elements of the architecture, distinguishing between trusted and untrusted components. The code producer, who is not trusted, is responsible for providing a proof that the applet satisfies a specification (a confidentiality policy), and for putting all components together into an extended applet (classfile). The user's trusted system can extract the different elements from this extended classfile, verify the proof of the property and initialize the security assistant.

The architecture assumes that all participants share some common knowledge, although it does not assume that this is necessarily used in a correct way. For example, it is important that the proof producer and the proof verifier map the actual applet code to the same formal language, and with identical results (represented in the graph by AT, Applet Theory). To do so, it is convenient that both share the same "theory" about the abstract machine (JVM Theory). However, the translation of the applet into AT done by the code producer is not trusted by the code consumer, who performs its own conversion. At both sides, the meaning of confidentiality policies is derived from an Admissibility Theory which reflects the definitions of Chapter 4.

Although it is the responsibility of the code producer to provide the extended classfile, the task can easily be delegated to third parties. For example, the actual proof could be performed by another person or system, or by a certifying compiler. Even the specification could be obtained from other sources. For example, a certification authority could be responsible for associating a policy to a protocol, which in the case of the purchasing applet could give the user evidence that the policy corresponds to the protocol requested by the acquirer bank or the credit card issuer.

In practice, a specification consists of a confidentiality policy and a set of channel identifications. In the case of the purchasing applet, it is necessary to identify entry points for acquirers and account numbers, which is why the Applet Theory is needed for the full specification.

The security assistant is responsible for informing the user on the result of the verification process, and of which channels should receive each piece of secret data. As written above, the fact that a text field appears on a window labelled "User PIN number input" cannot be trusted. The security assistant fills in the gap by telling the user whether what the label says is true. Finally, if the policy has been signed by an authentication authority, the assistant is responsible for verifying the signature.

## 6.3 Experiments

We report here on a few experiments done to test the proposed PCC architecture. The idea was to evaluate the difficulties in carrying the suggestions of the architecture into practice.

```

if (button ‘Submit form’ was pressed) {
    get acq from applet window;
    get K, the Acquirer’s public key from local keystore;
    get order from applet window;
    get acc from applet window;
    enc = encrypt (order, acc) with K;
    data = convert (acq, order, enc) to proper type for output;
    create Socket connection with Merchant;
    get OutputStream associated with Socket;
    write data to OutputStream;
    close Socket;
}

```

Table 6.1: Partial pseudo-code for the purchasing applet

### 6.3.1 Modeling the Java Virtual Machine

When applying the architecture, one of the most important decisions to make concerns the semantics of the language of interest. Not only should this semantics be of formal nature, but it should permit the kinds of analyses done over the annotated semantics of Chapter 3. In the case of JVM code, the former amounts to identifying states and labelled transition systems among them. While this is rather straightforward, providing the annotations is perhaps more difficult. Data has to be tracked as it is stored in objects of the most varied types, and subject to modifications by a full range of methods.

How to give a process algebraic semantics to an object-oriented language is still a matter of dispute among researchers. We have chosen instead a more traditional approach to operational semantics. For example, the representation of states actually follows that of the states in the JVM. Although there is only space for an informal presentation here, we take a quick look at the representation of states and transitions.

States are made up of the following three components:

1. *A Classfile Area* containing a definition for each class, its constant pool, fields and method codes.
2. *A Heap* containing a representation for each object or array.
3. *A Frame stack for each running thread* where each frame identifies the current executing method, the class it belongs to, the current value of the program counter, the values of local variables, and the operand stack.

To illustrate how the transitions of this state transition system are defined, consider a possible implementation of the purchasing applet: Table 6.1 delineates the part responsible for the applet side of the protocol where, for the sake of presentation, we have replaced the original JVM instructions by a less detailed and, hopefully, easier to read pseudo-code. Each step involves a mix of primitive JVM instructions and library method calls. The library methods are

implemented, in turn, by other methods and instructions, native or in bytecode form, belonging to JVM or the local operating system. Since our task is to analyze the confidentiality property of that part of the code which is mobile, it is natural to draw the trust borderline at the level of library method calls. Thus each library method invocation will, in the model, give rise to a “virtual instruction”, a labelled transition representing the effect of the corresponding library method call. Two sorts of effects are involved. The effect on the external world (socket creation, input and output) is captured by transition labels (i.e., actions, that in this case correspond to local function calls in *a-SecPA*), and the effect on applet execution is captured by replacing the library objects by object “stubs” which maintain the required data structures. For example, a `java.net.Socket` stub contains a field of class `java.net.InetAddress` to keep track of the internet address the socket instance is connected to. Virtual instructions which have no externally observable effect are regarded as internal and labelled  $\tau$ . To avoid indeterminate states we assume that the applet has passed the bytecode verifier and all library methods are well-behaved, so that each instruction has a well defined successor state.

For instance, step 2, Table 6.1, is the input event  $acq?x$  in the model of Section 3.1.1. An object of class `NamedTextField` is used to identify the entry point of each datum. Class `NamedTextField`, defined as a subclass of `java.awt.TextField`, associates a name to a textfield object. As an example, the name of the textfield intended for account number input may be, simply, “account number.” Reading a string from a `NamedTextField` object is modelled as a virtual instruction labelled `GETTEXT textfield string`. This string is allocated in the heap and annotated with the name of the textfield, i.e. “account number.”

Similarly, in step 3, a keystore containing name-key pairs is accessed using some local method. This is represented as a virtual instruction labelled `GETKEY principal key`, with `key` the principal’s key. Clearly, this transition corresponds to event  $k := pubKey(x)$  in the model of Section 3.1.1.

Steps 7, 8 and 9 can be modelled as virtual instructions, but here we have assumed that they do not involve any observable communication. Therefore, they are labelled with the silent action. On the other hand, step 8, socket creation, and step 11, socket closure, are clearly observable, so they are abstracted by appropriately labelled virtual instructions: `MAKESOCKET ipaddr portno` and `CLOSE socket`.

Step 10 is associated with virtual instruction `WRITE dest st`, and corresponds to the purchase order event  $merchant!(x, y, \{(y, z)\}_k)$  in the model of Chapter 3. While the destination of the message (`dest`) can directly be recovered from the `OutputStream` object to which the method is applied, determining the value of `st` is more involved. The reason is that `st` should support a symbolic representation of data, sufficient to recover the values of  $acq$ ,  $order$ ,  $acc$  and  $K$ , as well as the operations performed on them to obtain the actual value which is transmitted to `dest`. This is done by annotating, in the model, every byte array in the heap, and extending the transitions that correspond to conversions

| Step | Virtual instruction label | Events in model                 |
|------|---------------------------|---------------------------------|
| 2    | GETTEXT textfield string  | $acq?x$                         |
| 3    | GETKEY principal key      | $k := pubKey(x)$                |
| 4    | GETTEXT textfield string  | $order?y$                       |
| 5    | GETTEXT textfield string  | $acc?z$                         |
| 6    | $\tau$                    | -                               |
| 7    | $\tau$                    | -                               |
| 8    | MAKESOCKET ipaddr portno  | -                               |
| 9    | $\tau$                    | -                               |
| 10   | WRITE dest st             | $merchant!(x, y, \{(y, z)\}_k)$ |
| 11   | CLOSE socket              | -                               |

Table 6.2: Virtual instructions and corresponding events in the model of Section 3.1.1

from string, concatenation (pairing) and encryption so that the annotation is updated accordingly.

Table 6.2 contains a list of the virtual instructions used in each step of the pseudo-code, together with a reference to the corresponding event, if any, in the model of Section 3.1.1.

### 6.3.2 The Prototype

Once an appropriate semantics is chosen, we can proceed to implement the different components of the architecture (i.e. the square boxes in Figure 6.1). In fact, we have implemented an experimental platform for proof-carrying JVM applets based on Sun's Java Plug-in running inside Netscape Navigator 4.5. This platform permits us to experiment with concrete applets, equipped with proofs and specifications in the form of admissibility predicates. Proofs and specifications are produced using the Isabelle theorem prover [Pau98b], based on earlier work by Pusch [Pus98]. The Isabelle formalization uses the ideas of Section 6.3.1 and Chapter 4 rather directly. The formalization presents no essential problems. However, we have not been interested in any performance optimizations. Arriving at a good structure of the JVM specification and the proof which permits the checking speeds required for real applications will probably require a complete redesign of the prototype. For this reason we do not regard it very meaningful to report on performance aspects at this stage, but refer instead to the work of Necula and Lee (cf. [NL98b, NL98a]) which has gone some way to indicate the practical realizability of the general PCC scheme.

**PCC Data Assembly** The code producer is responsible for assembling an *extended classfile*, which is just a standard Java classfile containing specific attributes for the specification and proof of each applet. A Java program loads an applet generated by `javac`, the Java Compiler and converts it into the Applet

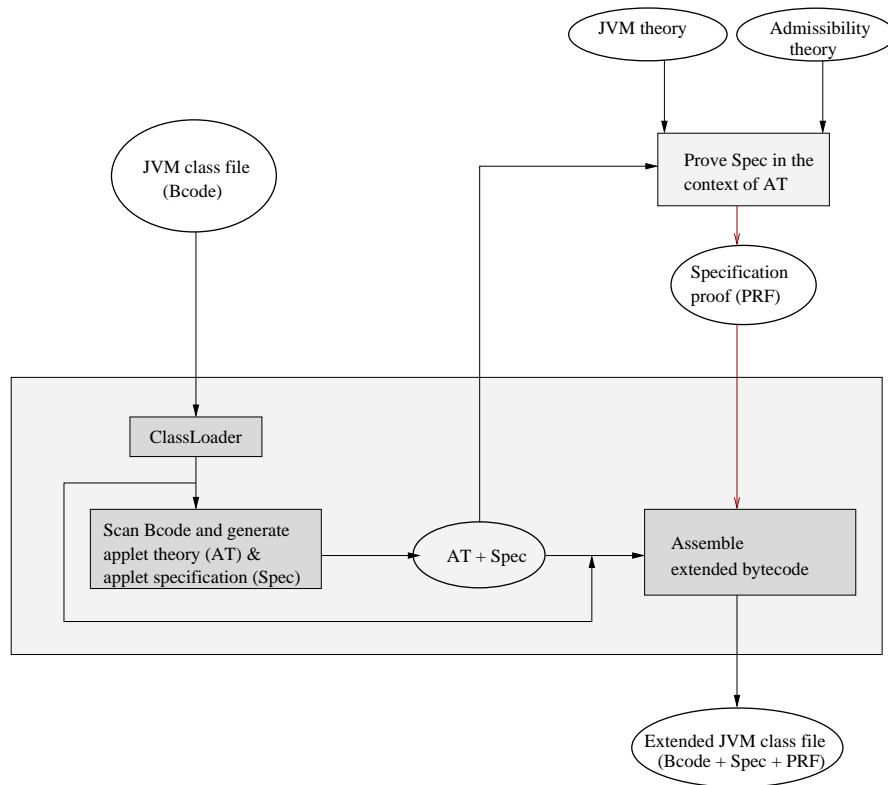


Figure 6.2: Proof-Carrying Code Assembler

Theory, an Isabelle theory representing the code. This is handed to the Isabelle theorem prover together with the admissibility specification. The proof script obtained is assembled into the applet, with the code and the specification to obtain a classfile that can be transmitted to the code consumer. The procedure is summarized in Fig. 6.2. Note that the prototype takes specifications to be properties in a format understandable to the theorem prover. This is, of course, too simple. For example, regarding the naming of textfields, we have put this information in the applet itself (using objects of class `NamedTextField`, instead of the standard `TextField`). A more general solution would separate the code from the channel naming conventions required for the adequate interpretation of the policy. Finally, the specifications in the prototype are not signed by any certification authority.

**Enforcement of Admissibility** The *extended classfile*, which otherwise remains completely readable to the standard implementation of the Java Virtual

Machine, allows a slightly modified virtual machine perform extra tests on the untrusted code to reduce the risks associated with its execution. Besides executing the applet, this machine is required to:

- Download a web page with a reference to an extended class file to be executed at the browser site,
- verify the specifications (and their proofs) provided together with the class file,
- decide whether to run the downloaded code or not, and
- provide the user of the browser with information flow security data extracted from the specifications which had been successfully verified.

In view of this, the user side of the prototype consists of:

1. A *Java-enabled web browser* capable of downloading web pages and providing the necessary environment for the execution of embedded Java applets,
2. a *theorem checker* able to verify the proofs provided by the code producer, and
3. a modified *Java Runtime Environment* capable of handling extended class files, invoking the theorem checker, keeping track of the security specifications and providing this information to the user of the browser.

This part of the prototype, with its three subcomponents, is depicted in Figure 6.3. The rest of the section describes these components and their interplay in more detail.

### The Web Browser Component

For the prototype, we used Netscape 4.5 as browser to access applets on the internet. Besides being able to execute Java applets, nice feature of this browser is that it can be configured to employ Sun's Java Plug-in. This means that HTML files can be slightly modified to instruct Netscape to run a plug-in acting as an interface with the Java Runtime Environment (JRE). The modification of the HTML file amounts to replacing the usual APPLET tags with EMBED tags referring to objects of mime type "application/x-java-applet;version=1.1.2".

### The Theorem Checker Component

Although in a realistic situation the theorem checker would probably be tailored to check a particular kind of proofs (for performance reasons), we have taken a simpler and more direct path: We used the same Isabelle theorem prover employed to do the actual proofs. In fact, given that we have decided to represent the proofs as Isabelle scripts, this was an obvious choice.

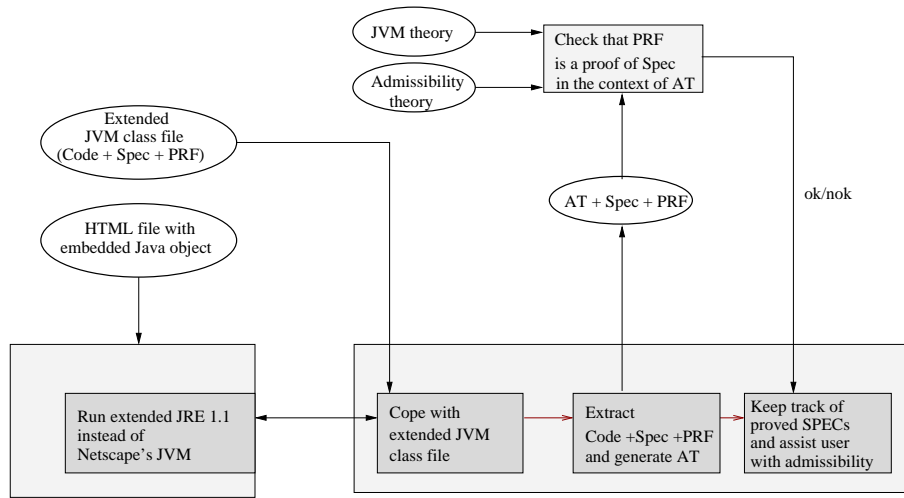


Figure 6.3: Proof-Carrying Code Checker

Notice that even though the proof producer needs and uses all three items input to the theorem checker, none can be trusted by the applet consumer. Therefore, local (trusted) versions of the JVM semantics, the Admissibility theory and the Applet Theory (AT) have to be used. Of all, the first two can be kept in local storage. The Applet Theory, instead, depends on the actual code of the applet and has to be generated on the fly (see next component) upon reception of the applet.

### The Extended Java Runtime Environment Component

In the prototype, the JRE was modified to fulfill three tasks: The disassembly of extended class files, the interface with the theorem checker, and the assistance of the user.

The first task requires the introduction of a wedge into the loading process of the JRE. The code for each class in Java is loaded into the JVM by a Class Loader. Since class loaders have the responsibility to determine when a class can be added to the running Java environment, they were the natural candidates to make the connection with the two added modules (see below) responsible for implementing the PCC concept.

There are two varieties of class loaders: Primordial Class Loaders and Class Loader objects [MF99]. The first load system classes from the local file system. Since these classes are local, they are trusted and they contain no PCC extensions. The second, Class Loader objects, load all other classes, which are untrusted by default, and thus may indeed contain PCC extensions.

Every class loader object is necessarily an instance of a subclass of `ClassLoader`. Method `defineClass` of class `java.lang.ClassLoader`, which is final

and thus not possible to override, has to be invoked before any loaded class can be used. Precisely because of this, we have placed a link to our checking methods inside the `defineClass` method. This let us guarantee<sup>1</sup> that no class load will avoid the checks which are mandated by our PCC architecture.

As a result of these changes, whenever a class file containing PCC extensions is loaded, the Isabelle Manager module (see Figure 6.3) is invoked to extract the code, the specification (Spec) and the proof (PRF). The code is then automatically translated into a theory file (Applet Theory) written in terms understandable to the theorem checker (Isabelle). This translation takes into account the definition of the semantics of JVM in Isabelle and has to be done by a local component to maintain confidence in it and avoid malicious JVM code to pass the PCC tests by providing a translation into a fake Applet Theory.

After this, the Isabelle Manager starts the Isabelle theorem prover (if it was not already running) and instructs it to check that PRF is a correct proof of Spec in the context of the Applet Theory.

The PCC Manager module takes care of specifications with a correct proof. Once the proof of a specification is found to be valid, this module parses and stores the specification. Subsequently, it remains in charge of assisting users, informing them of the admissibility property satisfied by the applet and the correct use of channels (including fields meant for secret input). In our prototype, a query can be performed on every input field on the applet window. In each case, if there is an admissibility property recorded for the applet, the parsed relabelling function contains information about the intended destination of the value input at the selected field. In the case of the Purchasing Applet example, this lets the PCC Manager tell the user that the credit card number information will only be sent out encrypted using the Acquirer's public key.

### 6.3.3 Some Conclusions on the Experiments

The road from theory to practice is usually a long one. Our experiments should then be taken basically as the source of inspiration they have been, more than as a thorough solution to the problems motivating our research. We can say though that there is some potential in the PCC architecture presented here. A realistic application would require the development of an infrastructure for producing, certifying and distributing confidentiality policies. Automatic proof methods, compact proofs and quick and efficient proof verification procedures would also be needed. Furthermore, even if our early experiments have indicated that the suggested form of user support is both natural and helpful, it is clear that it would not scale well to complex confidentiality specifications. In those cases, one could consider an alternative assistant that would rely on an external authority to certify a high-level, human-readable description of the confidentiality property accompanying the applet. This authority would not have

<sup>1</sup>This can be guaranteed for as long as no hole in the standard JRE is exploited to defeat Java's normal security mechanisms.

to certify the code itself, but just a description of the guarantees implied by the admissibility property. More work is needed, though, to determine what forms assistant output should take once we begin to address more complex applets and protocols.

The experiments have been extremely helpful in identifying some “problems” introduced by our way of expressing confidentiality. The need to correctly name input channels whose data is provided by a human user was not apparent at the beginning. Moreover, it was noticed that the design of an annotated semantics presents several yet unmatched challenges in the case of a real and complex programming language like Java.



## Chapter 7

# Conclusions and Future Work

The study and enforcement of secrecy properties of code is a complex matter. For years, researchers have proposed models and solutions only to find them faulty shortly after. In several cases, the faults were found by slightly shifting the perspective of study. In this respect, this thesis is probably no exception. The good news is that most of the attempts at taming confidentiality have helped advance our understanding of the subject.

In this context, the importance of finding semantical foundations for the proposed methods cannot be underestimated. As a matter of fact, these foundations provide a solid background against which the appropriateness and correctness of these methods can be measured. The problem of modelling, specifying and verifying secrecy of mobile implementations of security protocols, demands a semantical characterization of secrecy in the presence of admitted information flows. When a piece of code implements a security protocol, it is expected to have access to secret data. The code has then the possibility of leaking the secret information directly or indirectly by exploiting the freedom given by the protocol specification. This situation does not really correspond to the “transmission” view in Wittbold and Johnson’s terminology (see the discussion in the introduction). The purpose there is to design systems without covert channels, making it impossible for any Trojan Horse to leak information. It neither corresponds to the “eavesdropping” view, which is the approach taken to analyze the protocol and where the only attacker is assumed to be outside the system. The approach taken here is to relate the confidentiality properties of an implementation to those of the protocol, thus avoiding the need of applying at the code level the techniques that are usually applied at the level of the protocol. In order to do so, a number of features were described and small problems solved throughout this thesis.

In first place, we presented a language that is rich enough to be able to encode

implementations of various interesting protocols and simple enough to allow for a clear and manageable treatment. The choice of *SecPA* seems convenient from different points of view: its semantics is relatively close to the semantic model used to model confidentiality properties (i.e. labelled transition systems), and its standard features, derived from CCS and other process algebras which represent cryptographic operations explicitly, have been studied profusely.

A security protocol was taken as the measure of the amount and quality of information that an implementation is allowed to leak. Our confidentiality policies therefore extract and isolate that information contained in the protocol. Moreover, their compact representation makes them suitable for transmission and manipulation.

The definition of the desired confidentiality property demanded the use of a non-standard, extended semantics (*a-SecPA*) for the implementation language. Although it is an “instrumented semantics”, we have avoided the problems of other, previous semantics of this sort by providing results relating the extended semantics to a more standard one (Section 3.3). In general, the semantics of a program represents an abstraction of its execution on a real machine. Consequently, we cannot pretend to exclude all possible attacks, for there can always be attacks that exploit features abstracted away in our models. Two such features, not present in our extended semantics, are time and probabilistic behaviour, but there are others too. Some timed and probabilistic attacks can nevertheless be prevented using our definition of Admissibility (c.f. next section).

Once the extended semantic was defined, we could provide a confidential protocol implementation relation (i.e. Admissibility) between the confidentiality properties of the implementation and the policy. This relation was defined in terms of well-understood mechanisms, like strong bisimulation and relabelling functions. The use of bisimulation equivalences had previously been advocated in the definition of several secrecy properties, for computer systems, programming languages and protocols; while relabellings provide a twist to purge functions and permutation of initial states. Although we have preferred strong bisimulation equivalence, the definition of Admissibility is independent of this choice. However, strong equivalence has the advantages that it has a simple theory, its verification techniques have been studied extensively, and it implies most other reasonable equivalences, including observational ones. By relying on the protocol analysis phase, that determines the implications of the policy, there is no strong need to consider an observational equivalence that assumes computationally-bounded observers.

That Admissibility, a property defined using strong bisimulation equivalence and relabelling functions over an instrumented semantics, relates the secrecy of an implementation to a policy is a strong claim that needs justification. The strategy adopted in this thesis was to show that an admissible process, when deprived of all admissible outputs, leaks no secret information. The last item demanded a variation of noninterference for nondeterministic systems where the only thing to protect was high-level values, and no high-level behaviors

(reflecting the differences between the “transmission” problem and the problem addressed here). The resulting definition of Selective Independency is by itself interesting and deserves further study.

Admissibility, like other information flow properties, has a coinductive definition. To complicate matters further, it uses universal quantification over an infinite set of conditional relabelling functions. Fortunately, it also has an unwinding theorem, a result giving sufficient conditions and reducing the proof of admissibility to a set of local verifications (Theorem 4.15). Although this result is of extreme importance for the derivation of program analysis techniques, Admissibility is just a piece in a much bigger puzzle. Indeed, the enforcement of confidentiality in the scenarios considered in the introduction involves a whole infrastructure, with features ranging from the protocol production and analysis, their confidentiality policies, production, transmission and consumption of code, its verification, user interface issues, etc., some of which were considered and discussed in the Proof-Carrying Code architecture of Chapter 6.

## 7.1 Future Work

The study of confidentiality is presently going through a fascinating period where different theories and approaches seem to converge, giving us a complex and elaborate picture of the field. Although in this thesis we have barely touched upon a small area, many questions remain unanswered, motivating a number of possible continuations to this work.

Standard semantical models and languages have helped to understand better the essence of confidentiality properties. While we have used them in this thesis, there is still place for more standardization. For example, it might be worth exploring replacing *SecPA* with languages like spi-calculus or the recent applied-pi calculus [AF01].

Regarding the confidentiality policies defined in this work, we have assumed it to be relatively easy to extract them from the protocols. However, this need not be so when considering elaborate protocol suites. That is, there is a need to further study how policies relate to protocols: could policies be automatically extracted from protocols?

It is now commonly regarded that an information flow property should not disregard flows originating from timed or probabilistic behaviors. In either case, the underlying semantical model must be enriched with the corresponding information (time and probabilities) before the property can properly be stated. It would be interesting to investigate how a property like admissibility could scale to those models. All the same, in the current definition, strong bisimulation might be too stringent but it is able to detect those timing channels that do not exploit variations in the execution times of library functions (e.g. encryption). If a program contains a function call that depends on a secret value and that is not admitted by the policy, then the program does not satisfy Admissibility. Therefore, the variations in execution time of a program can only depend

on secrets if admitted by the policy, which means that an admissible implementation preserves the timed confidentiality properties of the protocol when analyzed together with the particular implementation of the functions (something already considered in computational models for cryptographic protocols, see Section 2.2.2).

The definition of Selective Independency for *a-SecPA* processes (Section 5.2) extends Cohen’s definition to nondeterministic processes. In the context of this thesis, it was required to permute secret data while otherwise preserving high-level actions, a feature not present in most information flow properties within the “transmission” view. However, in Selective Independency, the definition of  $\mathcal{E}$ -strictness models our assumptions about the behavior of high-level users. Understanding the definition of  $\mathcal{E}$ -strictness could help understand the relation between Selective Independency and more established notions of information flow.

Concurrently with the effort to achieve a better understanding of the defined notions of confidential protocol implementation, more work is needed to determine the practical significance of the proposed technology. In the context of EOARD Project SPC 01-4025 we plan to develop program analysis techniques to verify admissibility properties over a range of languages of increasing complexity. We also plan to experiment with an architecture similar to that described in Chapter 6, but instead relying on program analysis.

# Bibliography

- [Aba97] M. Abadi. Secrecy by typing in security protocols. In M. Abadi and T. Ito, editors, *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software*, volume 1281 of *LNCS*, pages 611–638, Sendai, Japan, 1997. Springer.
- [AF01] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 104–115, London, UK, January 2001.
- [AG98a] M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5(4):267–303, 1998.
- [AG98b] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi Calculus. Technical Report SRC-149, Digital Systems Research Center, January 1998.
- [Aga00] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000. ACM.
- [AJ01] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In *Fourth International Symposium on Theoretical Aspects of Computer Software (TACS2001)*, LNCS, Sendai, Japan, October 29–31 2001. Tohoku University, Springer. To appear.
- [AR80] Andrews and Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.
- [AR00] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In J. van

- Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses, and T. Ito, editors, *Proc. of the Int. Conf. IFIP TCS 2000*, volume 1872 of *LNCS*, pages 3–22. Springer, August 2000.
- [BAN89] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. In *Proceedings of the Royal Society of London A*, pages 426:233–271, 1989.
- [BGH<sup>+</sup>95] M. Bellare, J. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, and M. Waidner. iKP – a family of secure electronic payment protocols. In *First USENIX Workshop on Electronic Commerce*, May 1995.
- [BK01] P. Boury and N. El Kadhi. Static analysis of java cryptographic applets. In *ECOOP2001 Workshop on Java Formal Verification*, Budapest, June 2001.
- [BL76] D.E. Bell and L.J. LaPadula. Secure computer systems: Unified exposition and MULTICS interpretation. Technical Report MTR-2997, Mitre Corp., Bedford, Mass., USA, June 1976.
- [Bol97] D. Bolignano. Towards a mechanization of cryptographic protocol verification. In *9th International Conference on Computer Aided Verification*, number 1254 in *LNCS*, pages 131–142, Berlin, 1997. Springer.
- [BR93] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the First Annual Conference on Computer and Communications Security*. ACM, 1993.
- [BS98] E. Börger and W. Schulte. A modular design for the Java Virtual Machine architecture. In E. Börger, editor, *Architecture Design and Validation Methods*. Springer, December 1998.
- [CG00] J. Cederquist and P. Giambiagi. Implementations that preserve confidentiality. Extended Abstract at the IEEE Symposium on Logic in Computer Science, June 2000.
- [Che98] David M. Chess. Security issues in mobile code systems. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 1–14. Springer, 1998.
- [Coh77] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.
- [Coh78] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

- [CT91] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley series in telecommunication. John Wiley & Sons, Inc., 1991.
- [DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
- [Dea97] D. Dean. Secure mobile code: Where do we go from here? In *DARPA Workshop on Foundations for Secure Mobile Code*, Monterey, CA, USA, March 1997.
- [Den76] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
- [DFG99] A. Durante, R. Focardi, and R. Gorrieri. CVS: A compiler for the analysis of cryptographic protocols. In *Proceedings of 12th IEEE Computer Security Foundations Workshop*, pages 203–212, Mor-dano, Italy, June 1999. IEEE.
- [DG00] M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, pages 233–244, Cambridge, England, July 2000. IEEE.
- [DY83] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, March 1983.
- [FG95] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.
- [FG97] R. Focardi and R. Gorrieri. The compositional security checker: a tool for the verification of information flow security properties. *IEEE Transaction on Software Engineering*, 23(9):550–571, 1997.
- [FGM00] R. Focardi, R. Gorrieri, and F. Martinelli. Secrecy in security protocols as noninterference. In S. Schneider and P. Ryan, editors, *Proceedings of DERA/RHUL Workshop on Secure Architectures and Information Flow*, volume 32 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.
- [FHG98] F. J. Thayer Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 160–171, Oakland, CA, May 1998.
- [FLR77] R. Feiertag, K. Levitt, and L. Robinson. Proving multi-level security of system design. *ACM Operating Systems Review*, 11(5):57–65, November 1977.

- [FM99a] R. Focardi and F. Martinelli. A uniform approach for the definition of security properties. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99, Vol. I*, volume 1708 of *LNCS*, pages 794–813. Springer, 1999.
- [FM99b] Stephen N. Freund and John C. Mitchell. A formal framework for the Java Bytecode language and verifier. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'99)*, November 1999.
- [GM82] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, 1982.
- [GM84a] J.A. Goguen and J. Meseguer. Inference control and unwinding. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 75–86, Oakland, CA, April 1984.
- [GM84b] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, April 1984.
- [Gol97] A. Goldberg. A specification of Java loading and bytecode verification. Technical report, Kestrel Institute, Palo Alto, CA, 1997.
- [Gra90] J. W. Gray, III. Probabilistic interference. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, CA, May 1990.
- [Gra92] J. W. Gray, III. Toward a mathematical foundation for information flow security. *Journal of Computer Security*, 1:255–294, 1992.
- [HR98] N. Heintze and J. G. Riecke. The SLam Calculus: Programming with secrecy and integrity. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, CA, January 1998. ACM.
- [Jö1] J. Jürjens. Secrecy-preserving refinement. In *Formal Methods Europe (FME'01)*, LNCS. Springer, March 2001.
- [JL75] A. K. Jones and R. J. Lipton. The enforcement of security policies for computation. In *Proceedings of the 5th Symposium on Operating Systems Principles*, pages 197–206, November 1975.
- [JMT97] T. Jensen, D. Le Métayer, and T. Thorn. Security and dynamic class loading in Java: A formalisation. Technical Report 1137, IRISA, Rennes Cedex, France, October 1997.

- [Kad01] N. El Kadhi. Automatic verification of confidentiality properties of cryptographic program. *Networking and Information Systems Journal*, 6, 2001. To appear.
- [Koc96] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO’96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
- [Koz99] Dexter Kozen. Language-based security. In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *Proceedings of Conf. Mathematical Foundations of Computer Science (MFCS’99)*, volume 1672 of *LNCS*, pages 284–298. Springer, September 1999.
- [LJ00] K. R. M. Leino and R. Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, May 2000.
- [LMMS99] P.D. Lincoln, J.C. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security protocols. In *FM’99 World Congress On Formal Methods in the Development of Computing Systems, Toulouse, France*, 1999.
- [Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In Margaria and Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996. Also in *Software Concepts and Tools*, 17:93-102, 1996.
- [Low99] G. Lowe. Defining information flow. Technical Report TR1999/3, Department of Mathematics and Computer Science. University of Leicester, 1999.
- [Man00] H. Mantel. Possibilistic definitions of security – an assembly kit –. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, pages 185–199, Cambridge, England, July 2000. IEEE.
- [Man01] H. Mantel. Preserving information flow properties under refinement. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 78–91, Oakland, CA, May 2001.
- [MC92] I. S. Moskowitz and O. L. Costich. A classical automata approach to noninterference type problems. In *Proceedings of 5th IEEE Computer Security Foundations Workshop*, Franconia, New Hampshire, 1992.
- [McC87] D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 161–166, Oakland, CA, April 1987.

- [McC88] D. McCullough. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 177–186, Oakland, CA, May 1988.
- [McC90] D. McCullough. A hookup theorem for multilevel security. *IEEE Transaction on Software Engineering*, 16(6):563–568, 1990.
- [McL90] J. McLean. Security models and information flow. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 180–187, Oakland, CA, May 1990.
- [McL92] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1, 1992.
- [McL94] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 1136–1145. John Wiley & Sons, 1994.
- [MF99] Gary McGraw and Ed Felten. *Securing Java*. John Wiley & Sons, Inc., 1999.
- [Mil87] J. K. Millen. Covert channel capacity. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 60–66, Oakland, CA, April 1987.
- [Mil89] Robin Milner. *Communication and concurrency*. Prentice-Hall, 1989.
- [Mil99] J. Millen. 20 years of covert channel modelling and analysis. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 113–114, Oakland, CA, May 1999.
- [MK94] I. S. Moskowitz and M. H. Kang. Covert channels – Here to stay? In *Proceedings of COMPASS’94, Gaithersburg, MD*, pages 235–243. IEEE Press, 1994.
- [ML98] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, CA, May 1998.
- [MMS97] J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Mur $\phi$ . In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–153, Oakland, CA, May 1997.
- [MR00] J. Millen and H. Ruess. Protocol-independent secrecy. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

- [MS01] H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, pages 126–142, Nova Scotia, Canada, June 2001. IEEE.
- [MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. Series on Discrete Mathematics and its applications. CRC Press LLC, 1997.
- [MWCG98] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 85–97, San Diego, CA, January 1998. ACM.
- [Mye99] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999. ACM.
- [Nec97] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 106–119, Paris, France, January 1997. ACM.
- [NL98a] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Proceedings of the Annual Symposium on Logic in Computer Science (LICS)*, Indianapolis, U.S.A., June 1998. IEEE, Computer Society Press.
- [NL98b] George C. Necula and Peter Lee. Safe, untrusted agents using Proof-Carrying Code. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 61–91. Springer, 1998.
- [Pau98a] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
- [Pau98b] L. C. Paulson. Introduction to Isabelle. Technical report, Computer Laboratory, University of Cambridge, 1998.
- [PSW00] B. Pfizmann, M. Schunter, and M. Waidner. Cryptographic security of reactive systems (extended abstract). In S. Schneider and P. Ryan, editors, *Proceedings of DERA/RHUL Workshop on Secure Architectures and Information Flow*, volume 32 of *Electronic Notes in Theoretical Computer Science*. Elsevier, April 2000.
- [Pus98] Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für Informatik, Technische Universität München, 1998.

- [Qia99] Zhenyu Qian. A formal specification of Java(tm) Virtual Machine instructions for objects, methods and subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer, 1999.
- [RG99] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proceedings of 12th IEEE Computer Security Foundations Workshop*, pages 228–238, Mordano, Italy, June 1999. IEEE.
- [Ros95] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 114–127, May 1995.
- [Ros98] A. W. Roscoe. Proving security protocols with model checkers by data independence techniques. In *Proceedings of 11th IEEE Computer Security Foundations Workshop*, Rockport, MA, June 1998.
- [RS99] P. Y. A. Ryan and S. A. Schneider. Process algebra and non-interference. In *Proceedings of 12th IEEE Computer Security Foundations Workshop*, pages 214–227, Mordano, Italy, June 1999. IEEE.
- [Rus92] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-2, Stanford Research Institute, 1992.
- [SA98] Raymie Stata and Martin Abadi. A type system for Java Bytecode subroutines. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 149–160, San Diego, CA, January 1998. ACM.
- [Sab01] A. Sabelfeld. The impact of synchronisation on secure information flow. In *Proceedings of the Andrei Ershov 4th International Conference on Perspectives of System Informatics*, LNCS, Novosibirsk, July 2001. Springer.
- [SG95] P. Syverson and J. Gray, III. The epistemic representation of information flow security in probabilistic systems. In *Proceedings of 8th IEEE Computer Security Foundations Workshop*, pages 152–166, Kenmare, Ireland, 1995.
- [SMH00] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics – 10 Years Back, 10 Years Ahead. Conference on the Occasion of Dagstuhl’s 10th Anniversary.*, volume 2000 of *LNCS*, pages 86–101, Saarbrücken, Germany, August 2000. Springer.

- [Son99] D. Song. Athena: A new efficient automatic checker for security protocol analysis. In *Proceedings of 12th IEEE Computer Security Foundations Workshop*, Mordano, Italy, June 1999. IEEE.
- [SS99] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. In *Proceedings of the 8th European Symposium on Programming*, volume 1576 of *LNCS*, pages 40–58, Amsterdam, March 1999. Springer.
- [SS00] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, Cambridge, England, July 2000. IEEE.
- [SS01] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1), 2001. Extended version of [SS99].
- [Sut86] D. Sutherland. A model of information. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, 1986.
- [SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 355–364, San Diego, CA, January 1998. ACM.
- [Vol00] D. Volpano. Secure introduction of one-way functions. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, pages 246–254, Cambridge, England, July 2000. IEEE.
- [VS98a] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proceedings of 11th IEEE Computer Security Foundations Workshop*, pages 34–43, Rockport, MA, June 1998.
- [VS98b] Dennis Volpano and Geoffrey Smith. Language issues in mobile program security. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 25–43. Springer, 1998.
- [VS00] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 268–276, Boston, MA, January 2000. ACM.
- [VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

- [WJ90] J. T. Wittbold and M. D. Johnson. Information flow in nondeterministic systems. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 140–161, Oakland, CA, May 1990.

# Appendix A

## Proofs

### A.1 Proofs for Chapter 4

**Lemma 4.3** *Let  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \alpha \text{ ok}$  where  $\alpha = o!w$  or  $\alpha = (v := o(w))$ . If  $v_0 : c(w_0)$  is a subterm of  $w$  and  $c \in \mathcal{E}$ , then  $s(c(w_0)) = v_0$ .*

*Proof.* Since  $c \in \mathcal{E}$  and  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \alpha \text{ ok}$ , there must be a clause  $c!e \leftarrow c_1(e_1)(x_1) \wedge \dots \wedge c_n(e_n)(x_n) \wedge b$  in the policy deeming  $\alpha$  admissible by means of substitution  $\sigma$ . We know then that  $\text{Ann}(e)\sigma = w$  and that, for every  $i$ ,  $s(c_i(\text{Ann}(e_i)\sigma)) = x_i\sigma$ .

That  $v_0 : c(w_0)$  is a subterm of  $w$  can easily be expressed by requiring the existence of an  $a\text{-Expr}$   $\rho$  containing a single occurrence of a distinct variable, say  $y$ , such that  $\rho[v_0 : c(w_0)/y] = w$ . By the observations above, we have  $\rho[v_0 : c(w_0)/y] = \text{Ann}(e)\sigma$ .

We need now to prove the following auxiliary statement:

- (1) If  $\rho[v_0 : c(w_0)/y] = \text{Ann}(e)\sigma$  then there is a  $\rho' \in a\text{-Expr}$  such that  $\rho = \rho'\sigma$  and  $\text{Ann}(e) = \rho'[x_i : c_i(\text{Ann}(e_i))/y]$  for some  $i$  s.t.  $c_i = c$ .

If (1) holds, then

$$\begin{aligned} \rho[v_0 : c(w_0)/y] &= \text{Ann}(e)\sigma = (\rho'[x_i : c_i(\text{Ann}(e_i))/y])\sigma \\ &= (\rho'\sigma)[x_i\sigma : c_i(\text{Ann}(e_i)\sigma)/y] \\ &= \rho[x_i\sigma : c_i(\text{Ann}(e_i)\sigma)/y] \end{aligned}$$

Therefore  $v_0 = x_i\sigma$  and  $w_0 = \text{Ann}(e_i)\sigma$ , from which we conclude  $s(c(w_0)) = s(c(\text{Ann}(e_i)\sigma)) = x_i\sigma = v_0$ .

Assertion (1) can be proved by induction on the structure of expression  $e$ . Most other cases being direct, we consider the case  $e = x$ : Since  $v_0 : c(w_0)$  is assumed to be a subterm of  $\text{Ann}(e)\sigma$ , we can deduce that there is an  $i$  such that  $c_i = c$ , and  $\text{Ann}(e)\sigma = x_i\sigma : c_i(\text{Ann}(e_i)\sigma) = v_0 : c(w_0)$ . Therefore,  $\rho$  must be  $y$ . Finally,  $\rho' = y$  makes (1) hold.  $\square$

**Lemma 4.12** *If  $s \xrightarrow{\alpha} s'$  then  $\bar{g}(s) \xrightarrow{f_g(s, \alpha)} \bar{g}(s')$*

*Proof.* If  $\alpha$  is  $\tau$  or an output, then  $s = s'$  and  $f_g(s, \alpha)$  is not an input. Therefore,  $\bar{g}(s) \xrightarrow{f_g(s, \alpha)} \bar{g}(s)$ , and the results follows immediately.

Assume first  $\alpha = (w_1 := c(w_2))$ , which implies  $s' = s[c(w_2) \mapsto \llbracket w_1 \rrbracket]$ . From definition 4.10, there are annotated values  $w'_1$  and  $w'_2$  such that  $f_g(s, \alpha) = (w'_1 := c(w'_2))$  and  $w'_1 : c(w'_2) = g(w_1 : c(w_2))$ . From definition 4.8,  $w'_2 = g(w_2)$ . Then, our transition relation between contexts renders:

$$\bar{g}(s) \xrightarrow{f_g(s, \alpha)} \bar{g}(s)[c(g(w_2)) \mapsto \llbracket w'_1 \rrbracket]$$

We should prove then that  $\bar{g}(s)[c(g(w_2)) \mapsto \llbracket w'_1 \rrbracket]$  and  $\bar{g}(s')$  are equal contexts. We apply both over  $d(w)$ . Note that  $\bar{g}(s')(d(w)) = \llbracket g(s'(d(w'))) : d(w') \rrbracket$  where  $w' = g(w)$ .

- Case  $d(w) = c(g(w_2))$ :

$$\begin{aligned} \bar{g}(s)[c(g(w_2)) \mapsto \llbracket w'_1 \rrbracket](d(w)) &= \llbracket w'_1 \rrbracket = \llbracket w'_1 : c(w'_2) \rrbracket \\ &= \llbracket g(w_1 : c(w_2)) \rrbracket \\ &= \llbracket g(s'(c(w_2)) : c(w_2)) \rrbracket \\ &= \bar{g}(s')(d(w)) \end{aligned}$$

- Case  $d(w) \neq c(g(w_2))$

$$\begin{aligned} \bar{g}(s)[c(g(w_2)) \mapsto \llbracket w'_1 \rrbracket](d(w)) &= \bar{g}(s)(d(w)) \\ &= \llbracket g(s(d(g(w))) : d(g(w))) \rrbracket \\ &= \llbracket g(s'(d(g(w))) : d(g(w))) \rrbracket \\ &= \bar{g}(s')(d(w)) \end{aligned}$$

□

## A.2 Proofs for Chapter 5

**Lemma 5.3** *Given  $\Delta$ ,  $\{\varphi_i\}$  and  $\mathcal{E}$  such that*

$$(A.1) \quad \forall i, \sigma, \sigma'. \sigma =_{\mathcal{E}} \sigma' \Rightarrow \varphi_i(\sigma) = \varphi_i(\sigma')$$

$$(A.2) \quad \Delta \subseteq St \times St \text{ is the transitive closure of}$$

$$\Delta_1 \triangleq \{(\sigma, \sigma') \mid \exists i. \varphi_i(\sigma) \wedge \varphi_i(\sigma') \wedge \sigma =_{\neg \mathcal{E}} \sigma'\}$$

*then,  $\mathcal{E} \not\vdash_{\Delta}^p v$  iff  $v$  is selectively independent of  $\mathcal{E}$  over  $p$  w.r.t.  $\{\varphi_i\}$ .*

*Proof.*

$\Rightarrow$ ) We have to prove (1) that  $\{\varphi_i\}$  is a cover, and (2)  $\forall i. \mathcal{E} \not\vdash_{\varphi_i}^p v$ .

For (1), take any  $\sigma \in St$ . The definition of  $\mathcal{E} \not\sim_{\Delta}^p v$  indicates that  $\Delta$  is an equivalence relation, and therefore  $\sigma \Delta \sigma$ . Assumption (A.2) implies the existence of a chain  $\sigma \Delta_1 \sigma_1 \dots \sigma_n \Delta_1 \sigma$ . Finally, from  $\sigma \Delta_1 \sigma_1$ , there is at least an index  $i$  such that  $\varphi_i(\sigma)$ .

For (2), take  $i, \sigma$  and  $\sigma'$  s.t.  $\sigma =_{-\mathcal{E}} \sigma' \wedge \varphi_i(\sigma) \wedge \varphi_i(\sigma')$ . By (A.2),  $\sigma \Delta \sigma'$  and the result follows from  $\mathcal{E} \not\sim_{\Delta}^p v$ .

$\Leftarrow$ ) We have to prove that  $\Delta$  is an  $\mathcal{E}$ -strict equivalence relation, as well as equation (5.2).

Symmetry and transitivity of  $\Delta$  follow immediately from (A.2). To see that it is reflexive, take any  $\sigma \in St$ . From the definition of Cohen's selective independency,  $\{\varphi_i\}$  is a cover, and therefore there is an index  $i$  s.t.  $\varphi_i(\sigma)$ . Finally, from (A.2),  $\sigma \Delta_1 \sigma$ , which in turn implies  $\sigma \Delta \sigma$ .

For (5.2), let  $\sigma \Delta \sigma'$ . From (A.2), there is a chain  $\sigma_1 \Delta_1 \dots \Delta_1 \sigma_n$ , where  $\sigma_1 = \sigma$  and  $\sigma_n = \sigma'$ . Moreover, for each consecutive pair  $(\sigma_i, \sigma_{i+1})$ , we know that  $\sigma_i =_{-\mathcal{E}} \sigma_{i+1}$  and that there is  $j_i$  such that  $\varphi_{i_j}(\sigma_i)$  and  $\varphi_{i_j}(\sigma_{i+1})$ . In other words, we can apply the definition of Cohen's selective independency to each consecutive pair in the chain and obtain:  $\forall 1 \leq i < n. p(\sigma_i).v = p(\sigma_{i+1}).v$ . Equation (5.2) follows immediately.

To see that  $\Delta$  is  $\mathcal{E}$ -strict, let  $\delta (=_{\mathcal{E}} \Delta; =_{\mathcal{E}}) \delta'$  and  $\delta =_{-\mathcal{E}} \delta'$ . Then, there must exist two states  $\sigma$  and  $\sigma'$  such that  $\delta =_{\mathcal{E}} \sigma$ ,  $\sigma \Delta \sigma'$  and  $\sigma' =_{\mathcal{E}} \delta'$ . Like in the previous paragraph, there is a chain  $\sigma = \sigma_1 \Delta_1 \dots \Delta_1 \sigma_n = \sigma'$ . From (A.2), for each  $1 \leq j < n$  there is an index  $i_j$  such that  $\varphi_{i_j}(\sigma_j)$  and  $\varphi_{i_j}(\sigma_{j+1})$ .

Define

$$\sigma'_j.v = \begin{cases} \sigma_j.v & \text{if } v \in \mathcal{E} \\ \delta.v & \text{otherwise} \end{cases}$$

so that  $\sigma'_j =_{\mathcal{E}} \sigma_j$  and  $\sigma'_j =_{-\mathcal{E}} \delta$ . The former implies  $\varphi_{i_j}(\sigma'_j)$  as well as  $\varphi_{i_j}(\sigma'_{j+1})$  (using (A.1)); and the latter, that  $\sigma'_j =_{-\mathcal{E}} \sigma'_{j+1}$ ,  $\delta = \sigma'_1$  and  $\delta' = \sigma'_n$ . We can then apply (A.2) to each consecutive pair to obtain  $\delta \Delta_1 \sigma'_2 \Delta_1 \dots \delta'$ , which clearly implies  $\delta \Delta \delta'$ , as wanted.

We also have to check that  $\Delta \subseteq =_{-\mathcal{E}}$ , but this follows directly from  $\Delta_1 \subseteq =_{-\mathcal{E}}$  and that  $=_{-\mathcal{E}}$  is an equivalence.  $\square$

**Proposition 5.23** *The relation  $\Delta_g$  is symmetric.*

*Proof.* Suppose  $\sigma \Delta_g \sigma'$ . We prove by induction on the length of  $\sigma$  that  $\sigma' \Delta_g \sigma$ .

**Base case:**  $\sigma = \lambda$ . Then  $\sigma'$  must be  $\lambda$  too, and the results follows trivially.

**Inductive case:** Suppose  $\sigma = \sigma_0 \alpha$ , then by the definition of  $\Delta_g$ ,  $\sigma' = \sigma'_0 \beta$  for

some  $\sigma'_0$  and  $\beta$ . Moreover,  $\sigma_0 \Delta_g \sigma'_0$  and  $\beta = f_g(s, \alpha)$  where  $s_0 \xrightarrow{\sigma_0} s$ . By

the inductive hypothesis,  $\sigma'_0 \Delta_g \sigma_0$ . According to Proposition 5.22,  $s_0 \xrightarrow{\sigma'_0} \bar{g}(s)$ . Then, using Proposition 4.11,  $f_g(\bar{g}(s), \beta) = f_g(\bar{g}(s), f_g(s, \alpha)) = \alpha$ , which implies that  $\sigma'_0 \beta \Delta_g \sigma_0 \alpha$  (i.e.  $\sigma' \Delta_g \sigma$ ).  $\square$

We give first conditions under which the abstraction function *abs* results injective.

**Lemma A.1** *If  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash o!w_0 \text{ ok}$ ,  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash o!w_1 \text{ ok}$  and  $\text{abs}(w_0) = \text{abs}(w_1)$  then  $w_0 = w_1$ .*

*Proof.* Define a function  $\kappa: e\text{-Val} \times \text{Context} \rightarrow a\text{-Val}$  as

$$\begin{aligned} \kappa(k, s) &= k \\ \kappa((w_1, \dots, w_n), s) &= (\kappa(w_1, s), \dots, \kappa(w_n, s)) \\ \kappa(\{w_1\}_{w_2}, s) &= \{\kappa(w_1, s)\}_{\kappa(w_2, s)} \\ \kappa(p_i(w), s) &= p_i(\kappa(w, s)) \\ \kappa(w_1 : c(w_2), s) &= \begin{cases} s(c(\kappa(w_2, s))) : c(\kappa(w_2, s)) & \text{if } w_1 = v_c \\ \kappa(w_1, s) : c(\kappa(w_2, s)) & \text{otherwise} \end{cases} \end{aligned}$$

To prove the lemma, it suffices to verify the following: if  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash o!w_0 \text{ ok}$  then  $\kappa(\text{abs}(w_1), s) = w_1$  when  $w_1$  is a subterm of  $w_0$ .

The proof is by induction on the structure of  $w_1$ . The main case to consider is when  $w_1 = v : c(w_2)$  and  $c \in \mathcal{E}$ . Then  $\text{abs}(w_1) = v_c : c(\text{abs}(w_2))$  so that  $\kappa(\text{abs}(w_1), s) = \kappa(v_c : c(\text{abs}(w_2)), s) = s(c(\kappa(\text{abs}(w_2), s))) : c(\kappa(\text{abs}(w_2), s))$ . By the inductive hypothesis, this is equal to  $s(c(w_2)) : c(w_2)$ . The result follows from Lemma 4.3.  $\square$

**Lemma A.2**  *$\phi$  is injective on its second argument.*

*Proof.* Suppose  $s$  is a *Context*, and  $\alpha$  and  $\beta$  are actions in *a-Act* such that  $\phi(s, \alpha) = \phi(s, \beta)$ . We want to show that  $\alpha = \beta$ .

- Case  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \alpha \text{ ok}$ , and  $\alpha = o!w$ :

By the definition of  $\phi$ ,  $\phi(s, \alpha) = \phi(s, \beta) = o!\text{abs}(w)$ . Then, either (1)  $\beta = o!w'$ ,  $\text{abs}(w) = \text{abs}(w')$  and  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \beta \text{ ok}$ , or (2)  $\beta = o!\text{abs}(w)$  and  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \not\vdash \beta \text{ ok}$ .

In (1), by Lemma A.1,  $w = w'$ , and therefore  $\alpha = \beta$ . In (2), since  $\beta \in a\text{-Act}$ ,  $\text{abs}(w) = w$  which implies  $\beta = o!w = \alpha$ .

- Case  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \alpha \text{ ok}$ , and  $\alpha = (v := c(w))$ :

Very similar to the previous case. It uses a version of Lemma A.1 slightly adapted to function calls  $(v := c(w))$ .

- All other cases:

If neither  $\alpha$  nor  $\beta$  falls in any of the cases above, then  $\alpha = \phi(s, \alpha) = \phi(s, \beta) = \beta$ .  $\square$

**Lemma A.3** *Let  $(\mathcal{E}, \mathcal{C}, \mathcal{A})$  be a confidentiality policy,  $abs$  the associated abstraction function,  $g$  a secret permuter, and  $w \in a\text{-Val}$ . Then,  $abs(w) = abs(g(w))$ .*

*Proof.* By induction on the structure of  $w$ .  $\square$

**Lemma A.4** *The relation  $\Pi^{-1}; \Delta_g; \Pi$  is a symmetric trace permuter and a subset of  $=_{-\mathcal{E}}$ .*

*Proof.* From Observation 5.27 and the fact that the class of trace permuters is closed under composition, we conclude that  $\Pi^{-1}; \Delta_g; \Pi$  is a trace permuter. That it is symmetric is immediate from  $(\Pi^{-1}; \Delta_g; \Pi)^{-1} = \Pi^{-1}; \Delta_g^{-1}; \Pi = \Pi^{-1}; \Delta_g; \Pi$  (cf. Prop. 5.23).

Now assume that  $\sigma \Pi^{-1} \delta \Delta_g \delta' \Pi \sigma'$ . We prove, by induction on the length of  $\sigma$ , that  $\sigma =_{-\mathcal{E}} \sigma'$ .

- **Base case:** If  $len(\sigma) = 0$ , then  $\sigma = \lambda$ . Then  $\delta = \lambda$  by the definition of  $\Pi$  and similarly  $\delta' = \sigma' = \lambda$ . Since  $\lambda =_{-\mathcal{E}} \lambda$ , we have established the base case.
- **Inductive case:** Suppose  $\sigma = \sigma_0 \alpha$  and that the result holds for  $len(\sigma_0)$ . By the definition of  $\Pi$ , there must be  $\delta_0$  and  $\beta$  such that  $\delta = \delta_0 \beta$ ,  $\sigma_0 \Pi^{-1} \delta_0$  and  $\alpha = \phi(s, \beta)$  where  $s_0 \xrightarrow{\delta_0} s$ . By the definition of  $\Delta_g$ , there must be  $\delta'_0$  and  $\beta'$  such that  $\delta' = \delta'_0 \beta'$ ,  $\delta_0 \Delta_g \delta'_0$  and  $\beta' = f_g(s, \beta)$ . Again, by the definition of  $\Pi$ , there must be  $\sigma'_0$  and  $\alpha'$  such that  $\sigma' = \sigma'_0 \alpha'$ ,  $\delta'_0 \Pi \sigma'_0$  and  $\alpha' = \phi(s', \beta')$  where  $s_0 \xrightarrow{\delta'_0} s'$ . Notice that, by Proposition 5.22,  $s' = \bar{g}(s)$ . Therefore, we should compare  $\phi(s, \beta)$  (i.e.  $\alpha$ ) against  $\phi(s', f_g(s, \beta))$  (i.e.  $\alpha'$ ).

1. If  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \beta \text{ ok}$  and  $\beta = o!w$ :  
 $\phi(s', f_g(s, \beta)) = \phi(s', o!g(w)) = o!abs(g(w))$ , using that  $g$  preserves admissible outputs (c.f. Def. 4.9). By Lemma A.3, this is equal to  $o!abs(w)$ , i.e.  $\phi(s, \beta)$ .
2. If  $(\mathcal{E}, \mathcal{C}, \mathcal{A}), s \vdash \beta \text{ ok}$  and  $\beta = (v := c(w))$ :  
Let  $v'$  be such that  $v' : c(g(w)) = g(v : c(w))$ . Then,  $\phi(s', f_g(s, \beta)) = \phi(s', v' : c(g(w))) = v' := c(abs(g(w)))$ , using that  $g$  preserves admissible outputs (c.f. Def. 4).