Types, compilers, and cryptography for secure distributed programming

Jeremy Planul

Thèse de Doctorat

École Polytechnique

Types, Compilers, and Cryptography for Secure Distributed Programming

Jérémy Planul

Sous la direction de:

Cédric Fournet — Microsoft Research Cambridge
Jean-Jacques Lévy — INRIA

Contents

1 Introduction

We are more and more dependent on our computing infrastructure. For example, most financial transactions are now automated, ranging from cash withdrawals at ATMs to high-frequency algorithmic trading with execution times of microseconds. Health information systems provide another complex example: medical records are now mostly produced, stored, and processed on computers, and they may be shared between various medical and administrative staffs, belonging to various hospitals and institutions. Even so, the security of this computing infrastructure is challenged every day. For example, many frauds result from unintended access to customer accounts over the internet, for instance, because the network communications are insecure, or because the banking application does not correctly restrict the privileges of its different users (including its own employees), or because an attacker got into the computer of the client and leveraged his access to the bank account using a keylogger. Likewise, medical records contain useful information, and many privacy breaches have been much publicised, as they challenge an important aspect of our private live. For instance, an employer may want to know if a future hire is at risk of long-term illness, or a journalist may be curious about the family of a politician. Given the demand for this kind of information, it would be foolish to give all doctors access to all medical records. On the other hand, any medical doctor present at an accident should be granted some access to the health history of the victims (e.g., allergies or coagulation factor anomalies). These examples suggest that, although it is reasonably easy to secure databases and network communications against outsiders, the security of systems whose legitimate users are also potential attackers is much more challenging.

1.1 Enforcing security by construction

From a research viewpoint, a lot of language-based formal techniques have been developed to improve the security of computing systems. A first approach is to review the security of existing systems, either automatically or by manual code inspection, and find and fix any flaws as part of the process. This approach is necessary, since even new systems are at least partly based on legacy systems. However, this approach is limited, in particular, existing systems rely on languages such as C or Javascript which are difficult to verify. In addition, it may rely on a monitoring software that reads the input and outputs of the program and react accordingly, for instance by removing or rewriting messages. A related, but slightly different approach, logs the messages, and then later verifies whether security has been compromised. But, in spite of these results, it is still hard to gain strong confidence that a program is correct and secure. An other, more radical approach, the one followed in this thesis, is to build tools that enable the programmer to build from scratch a new secure system, for example, as a compiler, which transforms a program so that it enforces a given security policy. This has led to a first few small, exemplary verified systems and protocols such as the TLS protocol [Bhargavan et al., 2008], the sel4 microkernel [Klein et al., 2009], some cryptographic algorithms [Barthe et al., 2009], a C compiler [Leroy, 2009].

In this thesis, we focus on language-based security by construction. We take as input the specification of a distributed computation involving multiple participants, together with its expected security properties. We then verify that this specification is sound, using static verification techniques such as type systems, and we then automatically generate a program for each participant. During this compilation process, we select adequate cryptographic and hardware mechanisms, such that the compiled programs correctly implement the computation with the required security properties.

1.2 How to specify security?

Before describing any secure implementations, a first step is to precisely express our security goals. We can distinguish three aspects of computer security: privacy, integrity, and availability.

Confidentiality and Privacy. Confidentiality concerns restrictions over read access to information. Indeed, the information owner generally wants to retain some control over who can access it. Then, security holds when this control is reliably enforced. For example, as we saw before, it is important that health records remain private, but also that they may be selectively read when necessary. Access control is sometimes not sufficient, as the usage of information also needs to be carefully restricted. For example, if an hospital publishes an anonymized database of its patients, it may still be possible to de-anonymize the database by comparing the records with public databases, or more generally, extract more information than intended. Another complication is that, as data from different owners is processed, the result of the computation may be subject to multiple confidentiality policies.

Integrity. Integrity concerns active control over both data and computations. For instance, one may wonder whether some piece of data is the correct result of a computation, and, if the computation is distributed, which participants may have influenced this particular result. Continuing with our banking example, examples of integrity failures include unauthorized transfers, and also transfers whose details are calculated from corrupted data. As for confidentiality, one often needs to enforce integrity beyond direct write access, to also capture other, more subtle dependencies. For example, when generating a new cryptographic key, it is essential to ensure that the pseudo random number generator has collected enough entropy, as otherwise the attacker may cause the key to be generated in a predictable state of the system.

Availability. Availability concerns reliable access to information and guarantees that some computations complete in a timely manner. For instance, denial of service attacks specifically target availability properties of a system. Availability is hard to specify as it involves fairness or timeliness rather than just trace properties. In this thesis, we do not formally consider availability.

These aspects of security are interdependent. For example, an attack on the confidentiality of a password can result in an attack on the integrity of the account protected by this password. Symmetrically, an attack on the integrity of a password, such as its reset to a known factory default, may result in a breach of confidentiality by giving read access to the account. Similarly, an attack on the integrity of a publicly available encryption key, for instance the replacement of the genuine encryption key of a website with a key generated by the attacker, may lead to insecure encryption.

Irrespective of the security goal, any security guarantee can only be expressed for a given system, with some given assumptions on its structure and its participants. For example, one may consider a simple system formed by two participants that trust one another, that locally execute their own programs, and that communicate over an untrusted network. When the system is defined, it is possible to study its security properties, for instance the security of a joint computation on shared secrets.

However, most of the time, the participants of the system do not fully trust one another. In fact, from a security viewpoint, a system is distributed inasmuch as its different components enjoy different levels of trust, and thus can be corrupted independently. For example, a bank may generally be trusted with personal financial information, but not with personal health data. Conversely, the bank generally trusts its clients only for a very small subset of actions; it must protect its security no matter what the client does. A system that consists of the bank, the client, and the network can be defined by specifying the partial trust between its participants. We may first consider the threat of a compromised network, and ensure that the security of the interaction between the bank and the client cannot be compromised by the network. We may also consider the threat of a compromised client, and ensure that the client cannot make unauthorized banking operations. Last, we may consider the threat of a compromised bank, and ensure that the bank do not get any private data from the client besides its financial information.

On the web, the situation becomes more complex since, when a user visit a website, he downloads some code from a remote server (untrusted) and executes it on his computer. Code of the server runs on the user's machine, hence the computer of the user cannot be considered as a unique unit of trust anymore. In this case, the web browser that runs on the user's machine and controls the execution of the code of the server is responsible of the security of the user. However, this raises another problem: the security of the user's computer depends on the security of its browser.

Similarly, when a computer is accessed by multiple users, the programs called by the different users have different levels of trust. The operating system is responsible of this separation. And once again, the security of the different users depends on the security of a complex program. A similar situation arises with virtualization, when multiple operating systems can run on a single machine.

In all these cases, and for a given security goal, we refers to the set of all the components that must be trusted to achieve that goal as the Trusted Computing Base (TCB). An important design goal for secure systems is to minimize the size and complexity of the TCB.

In the following, we reason about active attacks and partly compromised systems by replacing their compromised components with arbitrary programs that have the same privileges as these components, for instance, they can read and write the same variables, and they have access to the same keys.

1.3 How to enforce security?

Once the security goals have been precisely stated, the next step is to select adequate security mechanisms to enforce them, ideally without too much overhead. We detail below the main security mechanisms we consider in this thesis.

1.3.1 Hardware mechanisms

One may protect its computer against remote attacks by disconnecting it from the network, or even shutting it down. These are valid hardware protecting mechanisms, however, they may be incompatible with intended use of the computer. There are many, more subtle hardware protection mechanisms. For instance, operating systems use memory page tables to limit memory access of user programs. A piece of hardware called memory management unit uses pages tables to translate memory addresses seen by the programs into physical address. Distinct page tables can be used for different programs and effectively limit their accessible address ranges. Operating systems can also use an other hardware mechanism called NX bit to separate executable memory from storage memory. Certain parts of the memory can be marked as non executable using this NX bit; the CPU then refuses to execute instructions stored there. In our work, we mostly focus on Trusted Platform Modules, which are present on most computers, and which, in concordance with the CPU, allow for execution of a program in isolation with very strong guarantees on its identity. At

a call to a specific instruction of the CPU, the identity of the program is stored in a specific register in the TPM, and the program is launched with any interruptions disabled. The program can then call specific instructions of the TPM to, for instance, attest its identity with a secret of the TPM, or encrypt some data so that it can only be decrypted by a program with the same identity, and on the same TPM. The program runs at a different level of trust and its TCB is very small since its security is enforced by hardware. This mechanism is used, for instance, by BitLocker to protect an encryption key for the disk from extraction by other programs.

1.3.2 Cryptography

The secrecy and the integrity of a piece of data that is transmitted over an untrusted network (or an untrusted shared memory) can be enforced using various cryptographic mechanisms, such as encryption or signature. These mechanisms effectively shift the problem from securing data to the security of keys. It does not completely solve the problem, but keys can be shared more easily since they are small and can be exchanged only once at the beginning of the computation since they can be used to protect many messages. We now detail some cryptographic mechanisms, which help enforce different aspects of computer security.

Privacy. In the concrete word, a padlock can be used to ensure the privacy of a suitcase. Encryption schemes work similarly for digital information. The scheme is a triple: key generation, encryption and decryption algorithm. The key generation algorithm generate a pair of key, one for encryption, which can be left public, and one for decryption, which should stay private. Strong encryption schemes are generally probabilistic, so that it is not possible to tell if two values are equal by comparing their encryption. The security guarantees of such encryption scheme are also probabilistic, and they assume that the attacker has a limited computation time. Otherwise, he could try all keys, and succeed with certainty.

Integrity. In the concrete word, seals can be used to guarantee the author of a letter. Signature schemes work similarly for digital information. The scheme is a triple: key generation, signature, and signature verification algorithm. The key generation algorithm generate a pair of key, one for signature, which should stay private, and one for verification, which can be left public. As for the encryption schemes, the security guarantees of signature schemes are probabilistic, since the attacker can find the correct signing key by accident.

Availability. It is generally easier to attack availability than integrity or confidentiality. There is generally not much that can be done with cryptography to stop an attacker from cutting a cable, destroying a hard drive, or overloading a server. That being said, there are mitigation techniques, for instance, the replication and repartition of storage devices.

We describe above some examples of cryptographic schemes and their relation with the different aspects of computer security, but there exists a wide range of cryptographic schemes. For instance, some schemes are homomorphic, that is, some functions can be applied directly on ciphertexts, without decrypting them. For example, the ElGamal encryption scheme is homomorphic for multiplications. If we consider a system with a client and a server, which is not trusted by the client. The client can encrypt some data, then send the ciphertexts to the server. The server can then multiply the encrypted data and send the result to the client, even if he cannot learn anything about the contents of the cypertexts. The client can then decrypt the resulting ciphertext and read the result of the multiplication. However, there is no guarantee for the client that the server did the computation as intended; the client may need to use some other mechanism to ensure the integrity of the computation.

1.4 Our work

We focus on the systematic enforcement of language-based security for distributed systems using cryptography and hardware mechanisms. We outline our main results below. One of our recurring technique to express specifications is typing: our type system associates a security level to each part of the program and of its states, and ensures that the program verifies some property by comparing these levels. For instance, a type system may associate confidentiality levels to variables, and verify that private variables are never copied into public variables.

1.4.1 Secure Enforcement for Global Process Specifications

We first consider the enforcement of communication security. The system consists of any number of participants that do not trust one another, and that communicate over an untrusted network. In this first approach, the system is specified by the global pattern of communication between the participants, but not the local computation: the local program of each participant is left unspecified. For example, in a protocol with three participants: a client, a bank, and a merchant, it may be specified that the merchant can send a specific message to the bank (e.g., asking for a money transfer) only after receiving a message of agreement from the client. Corin et al. [2007] develop a compiler that can generate a defensive implementation that uses cryptography to enforce this specification. For example, it ensures that, if the client has not sent his agreement, a compromised merchant cannot convincingly forge a message that the implementation of the bank will accept. However, their compiler only accepts specifications that are sequential, that is, in which only one of participants is authorized to send a message at any given time. We improve on their work by providing support for more elaborate (parallel) specifications and by generalizing the condition for compilation using a new type system. We detail this work in Chapter 2, described below.

To specify our sessions, each participant is associated to a process in the style of CCS [Milner, 1989]. Their global session specification itself consists of the tuple of these processes. The global semantics consists of the possible communications between processes. We specify the local implementation of each participant with an automata. We label their transitions with the message being sent or received. Each message of the implementation corresponds to one high-level communication of the source specification. Concretely, implementation messages may carry an additional payload, such as an anti-replay nonce or some authenticated data about the states of the others participants. The implementations then use the additional information from the payload to accept or reject a message. For example, a local implementation is protected against messages replays if none of its traces receives two messages with the same nonce. Implementations are defined to be secure against particular classes of active adversaries. Classes of adversaries define what messages an adversary can produce depending on its knowledge and the set of compromised participants. For example, the implementation may assume that the adversary cannot forge messages from honest participants, in line with classic symbolic models of cryptography pioneered by Dolev and Yao [1983]. We then specify two properties on implementations: soundness and completeness. Soundness states that for every trace in the implementation, there exist a global source trace in the specification that explains all the messages sent and received by the honest participants. This ensure that the adversary cannot make the honest participants deviate from the specification. In our example, if the client and the bank are honest, a compromised merchant cannot convincingly forge a message that the implementation of the bank will accept if the client has not sent his agreement. Completeness states that, for every global source trace, there exist an implementation trace which messages sent and received are explained by the global source trace. This ensure that the implementation does implement the specification. However, we show that some global session specifications cannot be safely implemented, because implementations try to locally enforce a global specification. For example, consider a source specification with three participants, and where the first participant can send a particular message either to participant two or to participant three but not to both. This specification is unimplementable since, when the first participant is compromised, the participant three cannot know whether he should accept the message, since he cannot know whether the message as already been sent to participant two. We propose a type

system that ensures that source specifications are implementable. This type system is based on a fine tracking of the causality between messages. We then propose a generic implementation for all sessions, which is based on a fine tracking of the dependence between messages. For instance, when a message must be received before another is sent, the second message carries a copy of the first message to prove that it can be sent. Our main result is that this implementation is always complete, and that it is sound for well-typed specifications.

1.4.2 Information flow enforcement

We still consider systems where participants partially trust one another. But now, the programmer writes the full imperative program (not only the patterns of communication), specifies the different levels of confidentiality and integrity of the variables of the program assuming a shared memory, and specifies on which participant each part of the program should run. For example, considering two participants: a bank and a client, the programmer may specify secret variables of the client, and secret variables of the bank, and a computation that runs partially on the client machine, and partially on the bank machine. Fournet *et al.* [2008, 2009] develop a compiler that generates the code for every participant, adding cryptography so that the security specified by the programmer is respected. For instance, the variables exchanged between the bank and the client are encrypted so that they cannot be read by others. In our work, we study more varied cryptographic primitives, and we add support of secure hardware mechanisms. We described the remaining chapters below.

Information Flow and Non-Interference

In Chapter 3, we define our core imperative language and the security specification, and we give some general properties of these security specifications. We describe below the results of the chapter.

Computations are expressed as commands in an imperative while language, which is probabilistic so that we can model cryptography; the mutable memory is a function from variables to values. We define its semantics in terms of Markov chains between configurations, which are pairs of commands and memories. We define security labels as a product lattice of confidentiality and integrity labels. Security policies associate a security label to each variable of the program. We are interested in the security of the flows of information between variables, and not simply the security of access to variables. Intuitively, in a secure program, private variables should not flow into public variables, that is, they should not influence public variables, and similarly, untrusted variables should not flow to trusted variables. This is formalized as non-interference [Goguen and Meseguer, 1982]. A program is non-interferent, if, for two initial memories that coincide on their public (respectively trusted) variables, the two final memories (after execution of the program) coincide on their public (respectively trusted) variables. We enforce non-interference with a standard type system [Volpano et al., 1996]. This type system also assigns a program counter level to commands, recording the level of the variables they write. We then introduce active adversaries, for any fixed level, as commands that can read public variables (variables that are less confidential than the adversary level), that can write untrusted variables (variables that are less trusted than the adversary level), and that run in placeholders (some predefined points in the program). We introduce a new typing rule for adversary placeholders and prove non-interference for well-typed programs against active adversaries. However, the limitation to non-interferent programs is restrictive, and some interesting programs may contain declassification, that is flows from confidential to public variables, and endorsements, that is flows from untrusted to trusted variables. For example, a program that declassifies a secret if a password is guessed might be considered secure, but it is not non-interferent. Similarly, a program that encrypts a variable and declassifies the ciphertext to send it over the internet might be considered secure, since an attacker that does not know the decryption key do not gain any information by reading the ciphertext, but it is not non-interferent. In order to study this kind of programs, we give new 'lax' typing rules that allows for declassification and endorsement, but still enforce two security properties, which are useful, even if weaker than non-interference. The first property, confinement, states that the program counter

level of a command still limits which variables are written by the command. The second property, robust non-interference, states that, if the program counter integrity level is untrusted, then no confidential variable is declassified. With these two lemmas, we know that commands with a high integrity program counter might do privileged operations, and thus need to be considered with care, for instance in our compilers. Finally, in order to better understand what are the capabilities of the adversary in the presence of endorsement and declassification, we introduce a refined active adversary model, namely semi-active adversary model, that further limits which variables the adversary can read and write, and when. For example, given a program with adversary placeholders, our semi-adversary cannot read low confidentiality variables in placeholders were the variable contains no declassified information, since it would not give him any interesting information. We then introduce a new type system and prove that, when the semi-active refinement of the program is well typed, then for every active attack, there exists a semi-active attack of the program that ends with the same memory. Hence, our semi-active adversary is a correct model of what can happened with the active adversary.

We also introduce an abstract notion of program transformations so that we can describe the properties of our different compilers. Source and transformed systems are formed of programs, adversaries, states, and a function that evaluates (runs) them. We introduce two simulation properties for program transformations: security and correctness. Security states that attacks on the transformed programs are not more powerful than attacks on the source programs, hence that the compilation preserves security. This is more general than, for example, proofs that the compiled program is non-interferent if the source program is non-interferent. Conversely, correctness states that some attacks on the source programs (e.g., honest adversaries who cooperate to run the program as intended) have equivalent attacks on the implementation programs. Formally, security states that, for every adversary of the transformed program, there exists an adversary of the source program such that their semantics are equivalent. Conversely, correctness states that for every source adversary, there exists an implementation adversary such that their semantics are equivalent.

Cryptographic Information Flow Compiler (CFLOW)

In Chapter 4, we describe the cryptographic compiler of Fournet et al. [2009] and recall their results. The security of the source program is specified with information flow policies. We first add localities in our language so that we can specify where each part of the code is supposed to run and add a new typing rule for these localities. We then describe the compiler, which takes as input a well-typed source program with localities and security annotations and outputs a distributed program secured with cryptography. To prove the security properties of the compiler, we first define encryption schemes as a triple: key generation, encryption, and decryption function, with functional and security properties, and signature schemes defined as a triple: key generation, signature, and signature verification function, with functional and security properties. The functional properties state, for example, that the decryption function is an inverse of the encryption function when used with correctly generated keys. We then state classic computational cryptographic assumptions (the security properties) for asymmetric encryption and signature in terms of games indistinguishability against adaptive chosen ciphertext attacks (IND-CCA2), indistinguishability against chosen plaintext attacks (IND-CPA), and existential unforgeability under chosen message attacks (CMA). Finally, we state the security and completeness of the compiler as an instance of program transformation.

Information-Flow Types for Homomorphic Encryptions

The CFLOW compiler produces programs using cryptography. We study properties of programs using various cryptographic primitives (more varied than those currently used by the compiler). Using a type system, we prove that, if the cryptography is correctly used, none of the private information leaks to information that is less private and none of the untrusted information leaks to information marked as trusted. For example, we may consider a program that consists of an

encryption of a confidential variable followed by the publication of the ciphertext and its decryption. We can prove by typing that if the encryption key is trusted, if the decryption key is secret, if the decrypted value is written in a confidential variable, then an adversary who only reads public values and write untrusted values cannot get any information on the initial value of the private variable. The goal is to understand how to integrate more diverse cryptographic primitives in our compiler, in order to compile efficiently more programs. We detail this work in Chapter 5 described below.

We first add datatypes to our security types to differentiate between ciphertexts, keys, and non-cryptographic data. Additionally, our security types for ciphertexts and encryptions carry additional information. For instance, the type of a ciphertext indicates the types of data that may be encrypted within such ciphertext. We then define blinding schemes, which contain a pre-encryption and a blinding function instead of a single encryption function. The pre-encryption does not hide its inputting plaintext since it is not randomizing, but prepare the blinding by producing a ciphertext. The blinding then randomizes ciphertexts. A pre-encryption followed by a blinding has the same properties than an encryption. However, the particularity of the blinding function is that it can be applied any number of time without changing the size of the plaintext, and, every time, it hides the relation between the inputting and outputting ciphertext. For example, a participant that is supposed to choose one ciphertext among two can blind its choice so that none knows which was chosen. We then describe the security assumption for these schemes, similar to CPA, which ensure that the blinding is hiding. We then describe homomorphic encryption schemes, which allows some function on plaintext to be computed on the ciphertext. This allows for remote secret computation. We then give a type system for programs using these different schemes (under the CPA or CCA assumption), allowing for limited declassification, for instance, after encryption or blinding. We also add some global conditions so that we can apply our cryptographic hypotheses. First, we need to avoid key cycles. Second, we need to ensure that CPA decryption of untrusted ciphertext never flows to the adversary. The security of the program is expressed as a computational non-interference property since our cryptographic assumptions themselves are computational. A program is computationally non-interferent, if, for two initial memories that coincide on their public (respectively trusted) variables, the probability that a polynomial program distinguishes the public (respectively trusted) restriction of the two final memory is negligible. Our main result is that well typed programs are non-interferent. We then develop two challenging applications of our approach.

Compiling Information-Flow Security to Minimal Trusted Computing Bases

The programs produced with the CFLOW compiler are secure only if they run on participants of the required level of trust. To continue with our example about the client and the bank, if the programmer wants to make a computation on the client that manipulates secret variables from the bank, the compilation will fail since the bank doesn't trust the client with its data. In fact, with the compiler above, it is not possible to enforce the security of a program that computes both on secret variables of the bank and secret variables of the client without a trusted third participant. We model the functionalities of the secure hardware discussed above and then use these functionalities to simulate this trusted third participant on an untrusted machine, with a minimal TCB. We detail this work in Chapter 6 described below.

We first enrich our language with a command that turns data into executable code, so that we can model a secure command that executes data as code. We then describe its semantics and its typing, and restrict its use by the adversary. We then model some security commands available on modern processors with TPMs: a command that increments a monotonic counter; a command that stores the identity of its argument, then runs it in isolation, as executable code; a command that adds information on the current identity; a command that seals data so that it can only be read by a code with a specific identity; a command that certifies some data produced by some code with its identity. We then illustrate the use of these secure commands during secure boots. We prove that the model for active adversary used in Sections 4.3 and 6.6 can be deduced from a more complex model where the active adversary controls the scheduling of machines. Finally, we describe our

new compilation scheme, where we simulate a trusted host using the TPM functionalities. In our example, the computation of secrets of the bank and the client is not possible because they do not trust each other. However, with this compiler, it is possible to simulate a trusted third party to do the computation using the TPM functionalities. The simulation is based on the previously modeled functionalities, which allow us to run some code in isolation with strong identity guarantees and to simulate secure storage. Our main result, expressed as an instance of program transformation is that this compilation scheme is secure.

1.5 Roadmap

This thesis contains revised material from three papers, published at CONCUR [Planul et al., 2009], ESOP [Fournet and Planul, 2011], and CCS [Fournet et al., 2011]. Chapter 2 describes our work on Secure Enforcement for Global Process Specifications; it is independent from the other chapters. Chapter 3 gives the core definitions of our language and policies used in Chapters 4-6. Chapter 4 describes a cryptographic compiler, which we use as a front-end in Chapter 6. Chapter 5 describes our information flow type system for cryptography, and may be read independently from Chapters 4 and 6. Chapter 6 describes our model and compiler for secure hardware. The proofs of the theorems are at the end of the corresponding chapters, and can be skipped without affecting the general understanding.

The CFLOW compiler, and various code samples are available at `http://msr-inria.inria.fr/projects/sec/cflow`.

2

Secure Enforcement for Global Process Specifications

2.1 Introduction

In this chapter, we consider high-level specifications of distributed applications, which focus on global interactions and hide local implementation details. These high-level specifications define a protocol interface, that can be used to program and verify the conformance of local implementations. In an adversarial setting, however, there is no guarantee that every local implementation complies with the protocol. In this chapter, we propose a generic defensive implementation scheme ensuring that the protocol is performed correctly.

The whole thesis concerns the construction of defensive implementation schemes; the next chapters consider lower level specifications that include local implementation details, and use a more concrete model of cryptography (computational instead of semantics). However, the higher level of abstraction of this chapter is adequate for the control-flow problem considered here and we believe that the semantics properties used to prove our theorems are easily implemented (and computationally proved) with basic cryptography.

2.1.1 Distributed Programming with Sessions

Distributed applications may be structured around their global interactions: the pattern in which they exchange messages (often called a *protocol*) is specified, but not their local implementation details.

For example, we consider an e-commerce protocol, involving three participants: a customer, a merchant, and a bank. The customer wants to negotiate a good with the merchant, who wants to verify that the customer is able to pay. We can define the global interactions of this protocol:

- First, the customer requests the price of the good to the merchant.
- Then, the merchant responds with its price.
- Then, the customer accepts or refuse the offer.
- Then, if the customer accepts the offer, the merchant asks the bank for a wiring authorization.
- Last, the bank sends an authorization or a refusal of payment.

This protocol specifies the global interactions, but gives no details on how the merchant chooses the price, how the customer estimates the offer, or how the bank chooses if it authorizes the payment.

When the protocol is specified, each machine that takes part in the application is assigned a *fixed initial process* corresponding to its role in the protocol, e.g., customer (hence, often called a *role*) and may run any local code that implements its process, under the global assumption that all other participating machines will also comply with their respective assigned processes. In our example, the role of the bank specifies that it should send an authorization or refusal of payment when it receives the request of the merchant. Moreover, when the bank receives the request, it can assume that the protocol has been respected until this point. For instance, it can assume that the merchant contactes him only if the customer has accepted the transaction.

This approach yields strong static guarantees, cuts the number of cases to consider at runtime, and thus simplifies distributed programming. It is explored with binary sessions [Honda et al., 1998, Takeuchi et al., 1994, Gay and Hole, 1999] i.e., sessions with only two participants, and, more recently, *multiparty sessions* [Corin et al., 2007, Honda et al., 2008]. Within sessions, machines exchange messages according to fixed, pre-agreed patterns, for instance a sequence of inputs and outputs between a client and a server; these patterns may be captured by types, expressed as declarative contracts (also known as workflows), or more generally specified as processes.

Multiparty session types are an active subject of research. The multiparty session type theory is developed in process calculi [Bettini et al., 2008, Capecchi et al., 2010, Mostrous et al., 2009], and used in several different contexts such as distributed object communication optimizations [Sivaramakrishnan et al., 2010], design by contract [Bocchi et al., 2011], parallel and web service programming [Pernet et al., 2010, Yoshida et al., 2009] and medical guidelines [Nielsen et al., 2010]. Deniélou and Yoshida [2011a] use types to determine the maximum size of message buffers necessary to run the session. Deniélou and Yoshida [2011b] study more dynamic sessions, where the number of participants can change during the session. Yoshida et al. [2010] study sessions parametrized by indexes. Dezani-Ciancaglini and de'Liguoro [2010] do an overview on session types.

2.1.2 The secure implementation problem

These global specifications provide an adequate level of abstraction to address distributed security concerns. Each machine is interpreted as a unit of trust, under the control of a principal authorized to run a particular role in the application. Conversely, these principals do not necessarily trust one another. Indeed, many applications commonly involve unknown parties on open networks, and a machine may participate in a session run that also involves untrustworthy machines (either malicious, or compromised, or poorly programmed). In this situation, we still expect *session integrity* to hold: an "honest" machine should accept a message as genuine only if it is enabled by the global specification. To this end, their implementation may perform cryptographic checks, using for example message signatures, as well as checks of causal dependencies between messages. In our example, the bank should not accept the merchant's message if the customer did not accept the transaction. To this end, the bank may ask for the signed message of the customer as a justification for the merchant's request.

2.1.3 Automated protocol synthesis for sequential sessions

Cryptographic implementation mechanisms need not appear in the global specification. They can sometimes be automatically generated from the specification. Corin et al. [2007] and Bhargavan et al. [2009] perform initial steps in this direction, by securing implementations of n-ary sequential sessions. Their sessions specify sequences of communications between n parties as paths in directed graphs. A global session graph is compiled down to secure local implementations for each role, using a custom cryptographic protocol that protects messages and monitors their dependencies. Their main security result is that session integrity, that is, the respect of the session specification, is guaranteed for all session runs—even those that involve compromised participants. They also report on prototype implementations, showing that the protocol overhead is small. Although their sequential sessions are simple and intuitive, they also lack flexibility, and are sometimes

too restrictive. For instance, concurrent specifications must be decomposed into series of smaller, sequential sessions, and the programmer is left to infer any security properties that span several sub-sessions. More generally, they leave the applicability of their approach to more expressive specification languages as an open problem.

In this chapter, we consider a much larger class of session specifications that enable concurrency and synchronization within session runs; and we systematically explore their secure implementation mechanisms, thereby enforcing more global properties in a distributed calculus. We aim at supporting arbitrary processes, and as such we depart significantly from prior work: our specification language is closer to a generic process algebraic setting, such as CCS [Milner, 1989]. On the other hand, we leave the cryptography implicit and do not attempt to generalize their concrete protocol design. The example below illustrates our session calculus, with concurrent specification.

Example 1 *To illustrate our approach, consider a simple model of an election, with three voting machines (V_1, V_2, and V_3) and one election officer machine (E) in charge of counting the votes for two possible candidates (c_1 and c_2) and sending the result (r_1 or r_2) to a receiver machine (R). We specify the possible actions of each machine using CCS-like processes (defined in Section 2.2), as follows:*

$$V_1 = V_2 = V_3 = \overline{c_1} + \overline{c_2} \qquad E = c_1.c_1.\overline{r_1} \mid c_2.c_2.\overline{r_2} \qquad R = r_1 + r_2$$

Machines running process V_i for $i = 1, 2, 3$ may vote for one of the two candidates by emitting one of the two messages c_1 or c_2; this is expressed as a choice between two output actions. The machine running process E waits for two votes for the same candidate and then fires the result; this is expressed as a parallel composition of two inputs followed by an output action. Finally, the machine running process R waits for one of the two outcomes; this is expressed as a choice between two input actions.

2.1.4 Protocols and applications

We now give an overview of the protocols and the attacker model. Our processes represent protocol specifications, rather than the application code that would run on top of the protocol. Accordingly, our protocol implementation monitors and protects high-level actions driven by an abstract application. The implementation immediately delivers to the application any input action enabled by the specification, and sends to other machines any action output enabled by the specification and selected by the application. The application is in charge of resolving internal choices between different message outputs enabled by the specification, and to provide the message payloads. In our e-commerce example, the application on the bank machine is in charge of accepting or refusing the wiring, while our implementation is in charge of verifying that the merchant's request is legitimate. And in Example 1, the application on the voters machine is in charge of choosing a vote.

We intend that our implementations meet two design constraints: (1) they rely only on the global specification—not on the knowledge of which machines have been compromised, or the mediation of a trusted third party; and (2) they do not introduce any extra message: each high-level communication is mapped to an implementation message. This *message transparency* constraint excludes unrealistic implementations, such as a global, synchronized implementation that reaches a distributed consensus on each action. For instance, the implementation of our e-commerce example cannot add a message from the customer to the bank confirming that he accepted the trade. And in Example 1, the voters cannot directly send a confirmation to the receiver machine.

2.1.5 Attacker Model

We are interested in the security of any subset of machines that run our implementation, under the assumption that the other machines may be corrupted. In Example 1, for instance, the receiver machine should never accept a result from the election officer if no voter has cast its vote.

Our implementations provide protection against active adversaries that controls parts of the session run: We assume an unsafe network, so the adversary may intercept, reorder, and inject messages—this is in line with classic symbolic models of cryptography pioneered by Dolev and Yao [1983]. We also assume partial compromise, so the adversary may control some of the machines supposed to run our implementation, and run instead arbitrary code; hence, those machines need not follow their prescribed roles in the session specification and may instead collude to forge messages in an attempt to confuse the remaining compliant, honest machines.

We focus on global control flow integrity, rather than cryptographic wire formats; thus, we assume that the message components produced by the compliant machines cannot be forged by our adversary. (This can be achieved by standard cryptographic integrity mechanisms, such as digital signatures.) Conversely, the adversary controls the network and may forge any message component from compromised machines. For instance, our implementation of the e-commerce example should ensures that a corrupt merchant cannot forge a convincing message to the bank without the customer agreement. And in Example 1, a corrupt election officer should not be able to forge a result without the corresponding votes. Also, we do not address other security properties of interest, such as payload confidentiality or anonymity.

2.1.6 Contents

Section 2.2 defines a global semantics for specifications. Section 2.3 describes their generic implementations, and states their soundness and completeness properties. Section 2.4 considers binary specifications. Section 2.5 explains our difficulties with implementing multiparty specifications. Section 2.6 defines our type system to avoid unimplementability. Section 2.7 presents our history-tracking implementation and establishes its correctness for well-typed specifications. Section 2.8 considers sequential n-ary sessions. Section 2.9 concludes.

2.2 Global process specifications

We consider specifications that consist of distributed parallel compositions of local processes, each process running on its own machine. In the remainder of the chapter, we let n be a fixed number of machines, and let $\widetilde{P}$ range over global process specifications, that is, n-tuples of local processes $(P_0, \ldots, P_i, \ldots, P_{n-1})$.

2.2.1 Syntax and informal semantics

Our local processes use a CCS syntax, given below. For simplicity, we choose not to include message payloads in our description of the protocol, since the enforcement of their security is mostly independent to the enforcement of the session control flow (our focus in this chapter). Also, our message outputs are asynchronous (i.e., no action can directly depend on the correct emission of a message) since, in a distributed protocol, it is not possible to know whether a message has been received (without an extra acknowledgment message).

$P ::=$	Local processes
$\mathbf{0}$	inert process
$\overline{a}$	asynchronous send
$a.P$	receive
$P + P'$	choice
$P \mid P'$	parallel fork
$!P$	replication

The inert process cannot do any action. The process $\overline{a}$ can send a message a then stop. The process $a.P$ can receive a then it behaves as process P. The process $P + P'$ can behave as process

P or as process P'. The process $P \mid P'$ can run P and P' in parallel. The process $!P$ behaves as a parallel composition of an infinite number of P. The specification $\widetilde{P} = (P_0, \ldots, P_{n-1})$ sets a global "contract" between n machines; it dictates that each machine i behaves as specified by P_i. For instance, in Example 1 we have $n = 5$ and $\widetilde{P} = (V_1, V_2, V_3, E, R)$.

2.2.2 Local operational semantics ($\rightarrow_{\mathsf{P}}$)

We define standard labeled transitions for local processes, with the rules given below. We write $P \xrightarrow{\beta}_{\mathsf{P}} P'$ when process P evolves to P' with action β ranging over a and $\overline{a}$. We omit the symmetric rules for sum and parallel composition.

$$a.P \xrightarrow{a}_{\mathsf{P}} P \qquad \overline{a} \xrightarrow{\overline{a}}_{\mathsf{P}} \mathbf{0} \qquad \frac{P_0 \xrightarrow{\beta}_{\mathsf{P}} P_0'}{P_0 + P_1 \xrightarrow{\beta}_{\mathsf{P}} P_0'} \qquad \frac{P_0 \xrightarrow{\beta}_{\mathsf{P}} P_0'}{P_0 | P_1 \xrightarrow{\beta}_{\mathsf{P}} P_0' | P_1} \qquad \frac{P \xrightarrow{\beta}_{\mathsf{P}} P'}{!P \xrightarrow{\beta}_{\mathsf{P}} !P | P'}$$

We omit structural rules to simplify parallel compositions with $\mathbf{0}$. In examples, we often implicitly simplify $P \mid \mathbf{0}$ and $\mathbf{0} \mid P$ to P.

We instantiate these definitions with Example 1:

$$V_1 = V_2 = V_3 = \overline{c_1} + \overline{c_2} \xrightarrow{\overline{c_1}}_{\mathsf{P}} \mathbf{0}$$
$$V_1 = V_2 = V_3 = \overline{c_1} + \overline{c_2} \xrightarrow{\overline{c_2}}_{\mathsf{P}} \mathbf{0}$$

Each of the voter can send either the message c_1 or c_2. Also:

$$c_1.c_1.\overline{r_1} \xrightarrow{c_1}_{\mathsf{P}} c_1.\overline{r_1} \xrightarrow{c_1}_{\mathsf{P}} \overline{r_1} \xrightarrow{\overline{r_1}}_{\mathsf{P}} \mathbf{0}$$
$$c_2.c_2.\overline{r_2} \xrightarrow{c_2}_{\mathsf{P}} c_2.\overline{r_2} \xrightarrow{c_2}_{\mathsf{P}} \overline{r_2} \xrightarrow{\overline{r_2}}_{\mathsf{P}} \mathbf{0}$$

Both parallel process in E can independently receive their respective votes and send the result as soon as two vote are casted for the same candidate. Hence, for example, E can successively receive c_2, c_1, and c_2:

$$E = c_1.c_1.\overline{r_1} \mid c_2.c_2.\overline{r_2} \xrightarrow{c_2}_{\mathsf{P}} \xrightarrow{c_1}_{\mathsf{P}} \xrightarrow{c_2}_{\mathsf{P}} \xrightarrow{\overline{r_2}}_{\mathsf{P}} c_1.\overline{r_1}$$

We also have:

$$R = r_1 + r_2 \xrightarrow{r_1}_{\mathsf{P}} \mathbf{0} \qquad R = r_1 + r_2 \xrightarrow{r_2}_{\mathsf{P}} \mathbf{0}$$

2.2.3 Global operational semantics ($\rightarrow_{\mathsf{S}}$)

We also define labeled transitions for global configurations, with the communication rule below. We write $\widetilde{P} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P}'$ when $\widetilde{P}$ evolves to $\widetilde{P}'$ by action a with sender P_i and receiver P_j. (The case $i = j$ is for local, but still observable communications.) We let α range over high-level communication $i\ a\ j$, and let φ range over sequences of these communications (written for instance $(i\ a\ j).\alpha$) representing high-level traces. A source execution is a reduction $\widetilde{P} \xrightarrow{\alpha^0}_{\mathsf{S}} \widetilde{P}^1 \ldots \widetilde{P}^{n-1} \xrightarrow{\alpha^{n-1}}_{\mathsf{S}} \widetilde{P}^n$. In this case, the trace $\varphi = \alpha^0 . \ldots . \alpha^{n-1}$ is a source (execution) trace, and we can write $\widetilde{P} \xrightarrow{\varphi}_{\mathsf{S}} \widetilde{P}^n$. We write $\alpha \in \varphi$ when α occurs in φ. The semantic rule is given below

$$\frac{P_i \xrightarrow{\overline{a}}_{\mathsf{P}} P_i^\circ \quad (P_k = P_k^\circ)^{k \neq i} \quad P_j^\circ \xrightarrow{a}_{\mathsf{P}} P_j' \quad (P_k^\circ = P_k')^{k \neq j}}{\widetilde{P} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P}'}$$

Hypotheses $(P_k = P_k^\circ)^{k \neq i}$ and $(P_k^\circ = P_k')^{k \neq j}$ ensure that only process i and process j change during a communication $i\ a\ j$. When $i \neq j$, the rule instantiate to

$$\frac{P_i \xrightarrow{\overline{a}}_{\mathsf{P}} P_i' \quad P_j \xrightarrow{a}_{\mathsf{P}} P_j'}{\widetilde{P} \xrightarrow{i\ a\ j}_{\mathsf{S}} (P_1, \ldots, P_i', \ldots, P_j', \ldots, P_n)}$$

When $i = j$, the rule instantiate to

$$\frac{P_i \xrightarrow{\overline{a}}_{\mathsf{P}} P_i^\circ \quad P_i^\circ \xrightarrow{a}_{\mathsf{P}} P_i'}{\widetilde{P} \xrightarrow{i\ a\ j}_{\mathsf{S}} (P_1, \ldots, P_i', \ldots, P_n)}$$

In examples, we often use the initial processes names instead of process indexes, e.g., writing $V_1\ c_2\ E$ instead of $0\ c_2\ 3$). For instance, here is one of the possible source executions of Example 1:

$$\begin{aligned}(V_1, V_2, V_3, E, R) &\xrightarrow{V_1\ c_2\ E}_{\mathsf{S}} (\mathbf{0}, V_2, V_3, (c_1.c_1.\overline{r_1} \mid c_2.\overline{r_2}), R)\\ &\xrightarrow{V_3\ c_1\ E}_{\mathsf{S}} (\mathbf{0}, V_2, \mathbf{0}, (c_1.\overline{r_1} \mid c_2.\overline{r_2}), R)\\ &\xrightarrow{V_2\ c_2\ E}_{\mathsf{S}} (\mathbf{0}, \mathbf{0}, \mathbf{0}, (c_1.\overline{r_1} \mid \overline{r_2}), R)\\ &\xrightarrow{E\ r_2\ R}_{\mathsf{S}} (\mathbf{0}, \mathbf{0}, \mathbf{0}, c_1.\overline{r_1}, \mathbf{0})\end{aligned}$$

2.3 Distributed process implementations

We describe distributed implementations of the specifications of Section 2.2, each process being mapped to one machine. A distributed implementation defines the messages the machine can send or receive depending on its implementation state, but it also sets their assumptions on the adversary, i.e., what the adversary is able to do depending on which machines are compromised. Hence, we first specify how we separate compliant (honest) machines from compromised (dishonest) machines, then we define their implementations, their semantics, and their properties (soundness and completeness). The implementation of the compliant machines does not depend on knowing which machines are compromised, as discussed in Section 2.1. However, the global implementation semantics that defines the possible executions in our adversarial model is necessarily parametrized by the set of compliant machines.

We let $\mathcal{C}$ range over subsets of $\{0, \ldots, n-1\}$, representing the indexes of $\widetilde{P}$ whose machines are compliant. We let $\widehat{P}$ range over tuples of processes indexed by $\mathcal{C}$. Intuitively, these machines follow our implementation semantics, whereas the other machines are assumed to be compromised, and may jointly attempt to make the compliant machines deviate from the specification $\widetilde{P}$. For instance, if the election officer of Example 1 is compromised, it may immediately issue a result r_1 without waiting for two votes c_1. Similarly, a compromised voter may attempt to vote twice. We are interested in protecting (the implementations of) compliant machines from such actions.

Next, we give a generic definition of process implementations. In the following sections, we show how to instantiate this definition for a given specification. Informally, an implementation is a set of programs, one for each specification process to implement, plus a definition of the messages that the adversary may be able to construct, in the spirit of symbolic models for cryptography [Dolev and Yao, 1983].

Definition 1 (Distributed implementation) *A* distributed implementation *is a triple* $\langle (Q_i)_{i<n}, (\xrightarrow{\gamma}_i)_{i<n}, (\vdash_{\mathcal{C}})_{\mathcal{C}\subseteq 0..n-1}\rangle$ *(abbreviated* $\langle \widetilde{Q}, \widetilde{\rightarrow}, \vdash_{\mathcal{C}}\rangle$*) where, for each* $i \in 0..n-1$ *and each* $\mathcal{C} \subseteq 0..n-1$*,*

- Q_i *is an implementation process;*
- $\xrightarrow{\gamma}_i$ *is a labeled transition relation between implementation processes;*
- $\vdash_{\mathcal{C}}$ *is a relation between message sets and messages.*

In the definition, each Q_i is an initial implementation processes, and each $\xrightarrow{\gamma}_i$ is a transition relation between implementation processes of machine i, labeled with either the input (M) or output $(\overline{M})$ of a message M (the syntax of M is left abstract). We use the notation Q_i only

for implementation processes. Further processes are denoted Q'_i, Q''_i,.... We let γ range over M and $\overline{M}$, and let ψ range over sequences of γ, representing low-level traces. For each implementation process, these transitions define the messages that may be received and accepted as genuine and the messages that may be sent. We assume that every message M implements a single high-level communication α. (Intuitively, M is an unambiguous wire format for α, carrying additional information.) We write $\rho(\gamma)$ for the corresponding high-level communication α, obtained by parsing γ (with, $\rho(\overline{M}) = \rho(M)$).

The relations $\vdash_{\mathcal{C}}$ model the capabilities of an adversary that controls all machines outside $\mathcal{C}$ to produce (or forge) a message M after receiving (or eavesdropping) the set of messages $\mathcal{M}$. For instance, if the implementation relies on digital signatures, the definition of $\vdash_{\mathcal{C}}$ may reflect the capability of signing arbitrary messages on behalf of the non-compliant machines.

Example 2 *We consider a protocol involving two participants with a request from the first participant followed by a response from the second participant. The processes are* $A_0 = \overline{a} \,|\, b$ *and* $A_1 = a.\overline{b}$*. We may implement this protocol by re-using the syntax of specification processes and labels (*M *carries no additional information, for instance,* $M = 0\ a\ 1$ *or* $\overline{M} = \overline{0\ a\ 1}$*), with initial implementation processes* (A_0, A_1)*, implementation transitions*

$$A_0 \xrightarrow{\overline{0\ a\ 1}}_0 b \xrightarrow{1\ b\ 0}_0 \mathbf{0} \qquad\qquad A_1 \xrightarrow{0\ a\ 1}_1 \overline{b} \xrightarrow{\overline{1\ b\ 0}}_1 \mathbf{0}$$

and an adversary that can replay intercepted messages and produce any messages from compromised principals, modeled with the deduction rules

$$\frac{M \in \mathcal{M}}{\mathcal{M} \vdash_{\mathcal{C}} M} \qquad\qquad \frac{i \notin \mathcal{C} \qquad i, j \in \{0, 1\} \qquad x \in \{a, b\}}{\mathcal{M} \vdash_{\mathcal{C}} i\ x\ j}$$

The transition relation on machine 0 *allows its local program to send a message* a *to machine* 1*; then, it transmits the first message* b *received from machine* 1*. In particular, we choose not to allow the reception of* b *before the emission of* a *since this behavior is not possible in the global specification. Conversely, the transition relation on machine* 1 *transmits the first message* a *received from machine* 0 *to the program on machine* 1*; then, it allows it to send a message* b *to machine* 0*. Concretely, the first adversary rule models an open network, where the adversary can replay messages. The second adversary rule is reminiscent of symbolic cryptography models. Suppose that each participant has its private key and the public key of the other participant. If every participant sign its messages and verify the signature of the messages it receives, the adversary can produces fake messages for compromised participants, as modeled in the second rule.*

$\langle (A_0, A_1), (\xrightarrow{\gamma}_0, \xrightarrow{\gamma}_1), (\vdash_{0,1}, \vdash_0, \vdash_1) \rangle$ *is a distributed implementation.*

2.3.1 Distributed semantics ($\rightarrow_{\mathrm{I}}$)

For a given distributed implementation $\langle \widetilde{Q}, \widetilde{\rightarrow}, \widetilde{\vdash_{\mathcal{C}}} \rangle$ and a given set of compliant machines $\mathcal{C}$, we define a global implementation semantics that collects all message exchanges between compliant machines on the network. In our model, compromised processes do not have implementations; instead, the environment represents an abstract adversary with the capabilities specified by $\vdash_{\mathcal{C}}$. Thus, the global implementation transitions $\xrightarrow{\gamma}_{\mathrm{I}}$ relate tuples of compliant implementation processes $\widehat{Q'}$ and also collects all compliant messages $\mathcal{M}$ exchanged on the network (starting with an empty set). The reductions are of the form $\mathcal{M}, \widehat{Q} \xrightarrow{\gamma}_{\mathrm{I}} \mathcal{M}', \widehat{Q'}$, with the following two rules:

(SendI)
$$\frac{Q'_i \xrightarrow{\overline{M}}_i Q''_i \qquad (Q'_k = Q''_k)^{k \neq i} \qquad i \in \mathcal{C} \qquad \rho(\overline{M}) = i\ a\ j}{\mathcal{M}, \widehat{Q'} \xrightarrow{\overline{M}}_{\mathrm{I}} \mathcal{M} \cup M, \widehat{Q''}}$$

(ReceiveI)
$$\frac{Q'_i \xrightarrow{M}_i Q''_i \qquad (Q'_k = Q''_k)^{k \neq i} \qquad i \in \mathcal{C} \qquad \mathcal{M} \vdash_{\mathcal{C}} M \qquad \rho(M) = j\ a\ i}{\mathcal{M}, \widehat{Q'} \xrightarrow{M}_{\mathrm{I}} \mathcal{M}, \widehat{Q''}}$$

Rule (SendI) simply records the message sent by a compliant participant, after ensuring that its index i matches the sender recorded in the corresponding high-level communication $i\ a\ j$ (so that no compliant participant may send a message on behalf of another). Rule (ReceiveI) enables a compliant participant to input any message M that the adversary may produce from $\mathcal{M}$ (including, for instance, any intercepted message) if its index matches the receiver i in the corresponding high-level communication $j\ a\ i$; this does not affect $\mathcal{M}$, so the adversary may a priori replay M several times. An implementation execution is a reduction $\mathcal{M}, \widehat{Q} \xrightarrow{\gamma^0}_{\mathrm{I}} \mathcal{M}^1, \widehat{Q^1} \ldots \mathcal{M}^{n-1}, \widehat{Q^{n-1}} \xrightarrow{\gamma^{n-1}}_{\mathrm{I}} \mathcal{M}^n, \widehat{Q^n}$. In this case, the corresponding trace $\gamma^0 \ldots . \gamma^{n-1}$ is an implementation (execution) trace. For instance, in Example 2:

- When both participants are honest ($\mathcal{C} = \{0, 1\}$), we have $\widehat{Q} = \widetilde{Q}$. Also, using the first adversary rule, both $\{0\ a\ 1\} \vdash_{\mathcal{C}} 0\ a\ 1$ because $0\ a\ 1 \in \{0\ a\ 1\}$ and $\{0\ a\ 1, 1\ b\ 0\} \vdash_{\mathcal{C}} 1\ b\ 0$ because $1\ b\ 0 \in \{0\ a\ 1, 1\ b\ 0\}$. Hence:

$$\begin{array}{rl} \emptyset, \widehat{Q} \xrightarrow{\overline{0\ a\ 1}}_{\mathrm{I}} & \{0\ a\ 1\}, (b, A_1) \\ \xrightarrow{0\ a\ 1}_{\mathrm{I}} & \{0\ a\ 1\}, (b, \overline{b}) \\ \xrightarrow{\overline{1\ b\ 0}}_{\mathrm{I}} & \{0\ a\ 1, 1\ b\ 0\}, (b, 0) \\ \xrightarrow{1\ b\ 0}_{\mathrm{I}} & \{0\ a\ 1, 1\ b\ 0\}, (0, 0) \end{array}$$

- When only participant 0 is honest ($\mathcal{C} = \{0\}$), we have $\widehat{Q} = A_0$. Also, $\{0\ a\ 1\} \vdash_{\mathcal{C}} 0\ a\ 1$ using the first adversary rule with hypothesis $0\ a\ 1 \in \{0\ a\ 1\}$, and $\{0\ a\ 1\} \vdash_{\mathcal{C}} 1\ b\ 0$ using the second adversary rule with hypothesis $1 \notin \mathcal{C}$. Hence:

$$\begin{array}{rl} \emptyset, \widehat{Q} \xrightarrow{\overline{0\ a\ 1}}_{\mathrm{I}} & \{0\ a\ 1\}, (b) \\ \xrightarrow{1\ b\ 0}_{\mathrm{I}} & \{0\ a\ 1\}, (0) \end{array}$$

- When only participant 1 is honest ($\mathcal{C} = \{1\}$), we have $\widehat{Q} = A_1$. Also, $\emptyset \vdash_{\mathcal{C}} 0\ a\ 1$ using the second adversary rule with hypothesis $0 \notin \mathcal{C}$, and $\{1\ b\ 0\} \vdash_{\mathcal{C}} 1\ b\ 0$ using the first adversary rule with hypothesis $1\ b\ 0 \in \{1\ b\ 0\}$. Hence:

$$\begin{array}{rl} \emptyset, \widehat{Q} \xrightarrow{0\ a\ 1}_{\mathrm{I}} & \{\}, (\overline{b}) \\ \xrightarrow{\overline{1\ b\ 0}}_{\mathrm{I}} & \{1\ b\ 0\}, (0) \end{array}$$

- When both participants are dishonest ($\mathcal{C} = \varepsilon$), we have $\widehat{Q} = \varepsilon$, and the only implementation trace is empty.

The wire format M may include additional information, such as a nonce to detect message replays, or some evidence of a previously-received message. The example below illustrates the need for both mechanisms.

Example 3 *We consider the specification (B_0, B_1), with $B_0 = \overline{b} \,|\, \overline{b}$ and $B_1 = b \,|\, b$. The implementation of B_1 must discriminate between a replay of B_0's first message and B_0's genuine second message, so these two messages must have different wire formats. For instance, an implementation may include a fresh nonce or a sequence number; using $0\ b\ 1\ 0$ as first message and $0\ b\ 1\ 1$ as second message, with $\rho(0\ b\ 1\ 0) = \rho(0\ b\ 1\ 1) = 0\ b\ 1$.*

We consider the specification (C_0, C_1, C_2), with $C_0 = \overline{a}$, $C_1 = a.\overline{b}$, and $C_2 = b$. The implementation of C_2 that receives b from C_1 must check that C_1 previously received a from C_0. For instance, the two messages may be signed by their senders, and C_1's message may also include C_0's message, as evidence that its message on b is legitimate using $0\ a\ 1$ as first message and $(0\ a\ 1)\ 1\ b\ 2$ as second message; with $\rho(0\ a\ 1) = 0\ a\ 1$ and $\rho((0\ a\ 1)\ 1\ b\ 2) = 1\ b\ 2$.

2.3.2 Concrete threat model and adequacy

Distributed implementations (Definition 1) are expressed in terms of processes and transitions, rather than message handlers for a concrete, cryptographic wire format. (We refer to prior work for sample message formats and protocols for sequential specifications [Corin et al., 2007, Bhargavan et al., 2009].)

In the implementation, the use of cryptographic primitives such as digital signatures can prevent the adversary to forge arbitrary messages. On the other hand, we do not want to consider implementations with an adversary so weak that they are not realistically implementable. Since the adversary controls the network, he can resend any intercepted messages and, when it controls a participant, he can at least send any message that this participant would be able to send if it were honest. Accordingly, for any realistic implementation, we intend that the two rules displayed below follow from the definition of $\vdash_{\mathcal{C}}$.

Definition 2 (Adequacy) *We say that an implementation is* adequate *when these two rules are valid.*

$$\frac{M \in \mathcal{M}}{\mathcal{M} \vdash_{\mathcal{C}} M}\ (\textsc{Replay}) \qquad \frac{i \notin \mathcal{C} \qquad Q_i \xrightarrow{\psi}_i Q_i' \xrightarrow{\overline{M_0}}_i Q_i'' \qquad \mathcal{M} \vdash_{\mathcal{C}} M \text{ for each } M \in \psi}{\mathcal{M} \vdash_{\mathcal{C}} M_0}\ (\textsc{Impersonate})$$

In Rule Impersonate, if (1) the adversary controls machine i; (2) a compliant implementation Q_i can send the message M_0 after a trace ψ has been followed; and (3) every message received by i in ψ can be constructed from $\mathcal{M}$ by the adversary; then he can also construct the message M_0 from $\mathcal{M}$.

Example 4 *For instance, in Example 2, Rule* Replay *is one of the adversary rules. We instantiate Rule* Impersonate *for $i = 0$ and $i = 1$, and get:*

$$\frac{0 \notin \mathcal{C} \qquad A_0 \xrightarrow{\overline{0\ a\ 1}}_0 b}{\mathcal{M} \vdash_{\mathcal{C}} \overline{0\ a\ 1}} \qquad \frac{1 \notin \mathcal{C} \qquad A_1 \xrightarrow{0\ a\ 1}_1 Q_i' \xrightarrow{\overline{1\ b\ 0}}_1 Q_i'' \qquad \mathcal{M} \vdash_{\mathcal{C}} 0\ a\ 1}{\mathcal{M} \vdash_{\mathcal{C}} 1\ b\ 0}$$

In the two cases, every message that can be deduced with these rules can be deduced with the second adversary rule given in the example:

$$\frac{i \notin \mathcal{C}}{\mathcal{M} \vdash_{\mathcal{C}} i\ c\ j}$$

Hence, the implementation $\langle (A_0, A_1), (\xrightarrow{\gamma}_0, \xrightarrow{\gamma}_1), (\vdash_{0,1}, \vdash_0, \vdash_1) \rangle$ is adequate.

2.3.3 Soundness and completeness

We now introduce our security properties. We first formally relate high-level traces of the specification, ranged over by φ, which include the communications of all processes, to low-level implementation traces, ranged over by ψ, which record only the inputs and outputs of the compliant processes. We then define our property of soundness, stating that every implementation run corresponds to a run of the specification; and our property of completeness, stating that every specification run corresponds to a run of the implementation. These properties depend only on the implementation and specification traces: trace-equivalent specifications would accept the same sound and complete implementations. The correspondence between high-level and low-level traces is captured by the following definition of *valid explanations*.

Definition 3 (Valid explanations) *A high-level trace $\varphi = \alpha_0 \dots \alpha_{p-1}$ is a* valid explanation *of a low-level trace $\psi = \gamma_0 \dots \gamma_{q-1}$ for a given set $\mathcal{C}$ when there is a partial function ι from (indexes of) low-level messages γ_k of ψ to (indexes of) high-level communications $\alpha_{\iota(k)}$ of φ such that $\rho(\gamma_k) = \alpha_{\iota(k)}$ for $k \in Dom(\iota)$ and*

1. *the restriction of ι on the indexes of the low-level inputs of ψ is an increasing bijection to the indexes of the high-level communications of φ with honest receivers (i a j with $j \in \mathcal{C}$);*
2. *the restriction of ι on the indexes of the low-level outputs of ψ is a partial bijection to the indexes of the high-level communications of φ with honest senders (i a j with $i \in \mathcal{C}$); and*
3. *whenever a low-level input precedes a low-level output, their images by ι are in the same order when defined.*

The definition relates every trace of messages sent and received by honest implementations to a global trace of communications between specification processes (including compromised processes). Suppose that the low-level trace is the trace of a global implementation execution and that the high-level trace is the trace of a global source execution. The first condition guarantees that the messages that have been accepted by honest participants in the implementation correspond to communications in the source execution. The second condition guarantees that the communications of the source execution with an honest emitter correspond to messages sent in the implementation execution. However, since send are asynchronous, not all send need to be reflected in the source trace, and their order is irrelevant. The third condition guarantees that the order of the messages in the source execution is coherent with the order of the message emissions and receptions perceived by the honest participants in the implementation execution. If all the processes are honest, the only possible valid explanation of a trace corresponds to the trace of inputs. Otherwise, the specification trace may have additional communications between compromised processes and may also contain communications corresponding to messages sent by an honest participant to a dishonest participant. Valid explanation guarantees that the implementation messages are received in the same order as the communications in the specification trace. Conversely, since the adversary controls the network, the relation does not guarantee that all low-level outputs are received, or that they are received in order.

For instance, in Example 2, with only honest participant 0 ($\mathcal{C} = \{0\}$), $(\overline{0\ a\ 1})$ is an implementation trace. It is validly explained by the empty source trace, assuming that machine 1 never received a, hence that no high-level communication occurred. It is also validly explained by the source trace $0\ a\ 1$, assuming that the machine 1 received a, hence that the high-level communication occurred. As a more complex example, consider an implementation that uses high-level communications as wire format (that is, M is just α), let $n = 3$, and let $\mathcal{C} = \{0; 2\}$. We may explain the low-level trace

$$\psi = (\overline{0\ b\ 2}).(\overline{0\ c\ 1}).(1\ d\ 2).(\overline{0\ a\ 1}).(0\ b\ 2)$$

with, for instance, the high-level trace $\varphi_1 = (0\ c\ 1).(1\ d\ 2).(0\ b\ 2)$ and the index function ι from ψ to φ_1 defined by $\{0 \mapsto 2, 1 \mapsto 0, 2 \mapsto 1, 4 \mapsto 2\}$. Following Definition 3, we check that

1. the restriction of ι to the indexes of the low-level inputs is $\{2 \mapsto 1; 4 \mapsto 2\}$ and relates the inputs of compliant participants in ψ and in φ;
2. the restriction of ι to the indexes of the low-level outputs is $\{0 \mapsto 2; 1 \mapsto 0\}$ and relates two of the three outputs in ψ to the outputs of compliant participants in φ, out of order; and
3. $(1\ d\ 2)$ precedes $(\overline{0\ a\ 1})$ in ψ, but ι is undefined on that low-level output.

Another valid explanation is $\varphi_2 = (1\ d\ 2).(0\ a\ 1).(0\ c\ 1).(0\ b\ 2)$.

We are now ready to define our main security property, which states that an implementation is *sound* when the compliant machines cannot be driven into an execution disallowed by the global specification.

Definition 4 (Soundness) *$\langle \widetilde{Q}, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is a sound implementation of $\widetilde{P}$ when, for every $\mathcal{C} \subseteq 0..n-1$ and every implementation trace $\emptyset, \widehat{Q} \xrightarrow{\psi}_{\mathsf{I}} \mathcal{M}, \widehat{Q}'$, there exists a source trace $\widetilde{P} \xrightarrow{\varphi}_{\mathsf{S}} \widetilde{P}'$ where φ is a valid explanation of ψ.*

Also, an implementation is *complete* if, when all machines comply, every trace of the global specification can be simulated by a trace of the implementation.

Definition 5 (Completeness) *$\langle \widetilde{Q}, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is a complete implementation of $\widetilde{P}$ when, for $\mathcal{C} = 0..n-1$ and for every source traces $\widetilde{P} \xrightarrow{\varphi}_{\mathsf{S}} \widetilde{P}'$, there exists an implementation trace $\emptyset, \widetilde{Q} \xrightarrow{\psi}_{\mathsf{I}} \mathcal{M}, \widetilde{Q}'$ where φ is a valid explanation of ψ.*

Completeness is necessary to ensure that the implementation corresponds to the source program. For example, an implementation that does nothing is sound for every source program since the only implementation execution is the empty execution, but not complete.

We easily check that, with our definition of valid explanations, an implementation that is both sound and complete also satisfies the message transparency property discussed in Section 2.1.

Example 5 *We instantiate these definitions for the specification of Example 2. We list all the non empty implementation traces and their respective valid explanation source traces:*

implementation trace $(\rightarrow_{\mathsf{I}})$	**explained by the source trace** $(\rightarrow_{\mathsf{S}})$
for $\mathcal{C} = \{0,1\}$	
$(\overline{0\ a\ 1})$	ε
$(\overline{0\ a\ 1}).(0\ a\ 1)$	$(0\ a\ 1)$
$(\overline{0\ a\ 1}).(0\ a\ 1).(\overline{1\ b\ 0})$	$(0\ a\ 1)$
$(\overline{0\ a\ 1}).(0\ a\ 1).(\overline{1\ b\ 0}).(1\ b\ 0)$	$(0\ a\ 1).(1\ b\ 0)$
for $\mathcal{C} = \{0\}$	
$(\overline{0\ a\ 1})$	ε **or** $(0\ a\ 1)$
$(\overline{0\ a\ 1}).(1\ b\ 0)$	$(0\ a\ 1).(1\ b\ 0)$
for $\mathcal{C} = \{1\}$	
$(0\ a\ 1)$	$(0\ a\ 1)$
$(0\ a\ 1).(\overline{1\ b\ 0})$	$(0\ a\ 1)$ **or** $(0\ a\ 1).(1\ b\ 0)$

For every $\mathcal{C} \subseteq \{0,1\}$ and every implementation trace $\emptyset, \widehat{Q} \xrightarrow{\psi}_{\mathsf{I}} \mathcal{M}, \widehat{Q}'$, there exists a source trace $\widetilde{P} \xrightarrow{\varphi}_{\mathsf{S}} \widetilde{P}'$ where φ is a valid explanation of ψ. Hence, $\langle (A_0, A_1), (\xrightarrow{\gamma}_0, \xrightarrow{\gamma}_1), (\vdash_{0,1}, \vdash_0, \vdash_1) \rangle$ is a sound implementation of (A_0, A_1). The implementation process A_0 cannot receive b before sending a. This is key for the soundness of the implementation since, if we had $A_0 \xrightarrow{1\ b\ 0}_0 A_0'$, with $\mathcal{C} = \{0\}$, the resulting implementation trace $(1\ b\ 0)$ has no valid explanation in the possible source traces.

Conversely, for every source traces $\widetilde{P} \xrightarrow{\varphi}_{\mathsf{S}} \widetilde{P}'$, there exists an implementation trace $\emptyset, \widetilde{Q} \xrightarrow{\psi}_{\mathsf{I}} \mathcal{M}, \widetilde{Q}'$ where φ is a valid explanation of ψ. Hence, $\langle (A_0, A_1), (\xrightarrow{\gamma}_0, \xrightarrow{\gamma}_1), (\vdash_{0,1}, \vdash_0, \vdash_1) \rangle$ is a complete implementation of (A_0, A_1).

2.4 Implementing two-party specifications (application)

We instantiate our definitions to specifications with only two participants, such as a client and a server. Such specifications have been much studied using session types [Honda et al., 1998, Takeuchi et al., 1994, Gay and Hole, 1999]. For this section, we set $n = 2$ and implement specifications of the form $\widetilde{P} = (P_0, P_1)$. For simplicity, we also exclude local communications: for any action a, each P_i for $i = 0, 1$ may include either a or $\overline{a}$, but not both.

2.4.1 A simple (insecure) implementation

An implementation that re-uses the syntax of specification processes and labels with initial implementation processes (P_0, P_1) (that is, Q_i is just P_i, and M is just α) is generally not sound.

Example 6 *Consider for instance the specification* (D_0, D_1) *with:*

$$D_0 = a.\overline{e} \,|\, b.(\overline{e} + \overline{c}) \qquad\qquad D_1 = \overline{a} \,|\, \overline{b} \,|\, e \,|\, c$$

We have the following source execution:

$$\begin{array}{rl} (D_0, D_1) \xrightarrow{1\ a\ 0}_{\mathsf{S}} & (\overline{e} \,|\, b.(\overline{e} + \overline{c}), \overline{b} \,|\, e \,|\, c) \\ \xrightarrow{0\ e\ 1}_{\mathsf{S}} & (b.(\overline{e} + \overline{c}), \overline{b} \,|\, c) \\ \xrightarrow{1\ b\ 0}_{\mathsf{S}} & (\overline{e} + \overline{c}, c) \\ \xrightarrow{0\ c\ 1}_{\mathsf{S}} & (\mathbf{0}, \mathbf{0}) \end{array}$$

Hence, in order to have a complete implementation, we need to have the following reduction of process 1*:*

$$\begin{array}{rl} D_1 \xrightarrow{\overline{1\ a\ 0}}_1 & \overline{b} \,|\, e \,|\, c \\ \xrightarrow{0\ e\ 1}_1 & \overline{b} \,|\, c \\ \xrightarrow{\overline{1\ b\ 0}}_1 & c \\ \xrightarrow{0\ c\ 1}_1 & \mathbf{0} \end{array}$$

We deduce that in state c*, machine* 1 *can receive the message* c*. We also have the following source execution:*

$$\begin{array}{rl} (D_0, D_1) \xrightarrow{1\ b\ 0}_{\mathsf{S}} & (a.\overline{e} \,|\, (\overline{e} + \overline{c}), \overline{a} \,|\, e \,|\, c) \\ \xrightarrow{0\ e\ 1}_{\mathsf{S}} & (a.\overline{e}, \overline{a} \,|\, c) \\ \xrightarrow{1\ a\ 0}_{\mathsf{S}} & (\overline{e}, c) \end{array}$$

In this source execution, machine 1 *cannot receive the message* c *. In order to have a complete implementation, we need to have the following reduction of process* 1*:*

$$\begin{array}{rl} D_1 \xrightarrow{1\ b\ 0}_{\mathsf{S}} & \overline{a} \,|\, e \,|\, c \\ \xrightarrow{0\ e\ 1}_{\mathsf{S}} & \overline{a} \,|\, c \\ \xrightarrow{1\ a\ 0}_{\mathsf{S}} & c \end{array}$$

We know from the first source execution that $c \xrightarrow{0\ c\ 1}_1 \mathbf{0}$*. Thus, when only participant* 1 *is honest* $(\mathcal{C} = \{1\})$*, we have:*

$$\begin{array}{rl} \emptyset, \widehat{Q} \xrightarrow{\overline{1\ b\ 0}}_{\mathrm{I}} & \{(1\ b\ 0)\}, \overline{a} \,|\, e \,|\, c \\ \xrightarrow{0\ e\ 1}_{\mathrm{I}} & \{(1\ b\ 0)\}, \overline{a} \,|\, c \\ \xrightarrow{\overline{1\ a\ 0}}_{\mathrm{I}} & \{(1\ b\ 0), (1\ a\ 0)\}, c \\ \xrightarrow{0\ c\ 1}_{\mathrm{I}} & \{(1\ b\ 0), (1\ a\ 0)\}, \mathbf{0} \end{array}$$

This implementation trace has no valid explanation source trace because the last input is not possible after the actions $0\ e\ 1$*,* $\overline{1\ a\ 0}$*, and* $0\ c\ 1$ *in this order. In state* c*, machine* 1 *does not know whether a message* c *from machine* 0 *is legitimate or not. Therefore, an implementation accepting the message* c *is unsound, and an implementation refusing it is incomplete.*

2.4.2 History-tracking Implementations

Our implementation relies on a refinement of the specification syntax and semantics to keep track of past communications: local processes are of the form $P : \psi$ where ψ is a sequence of global communications, each tagged with a fresh nonce ℓ used to prevent message replays. (The following rules ensure that no message can be received twice with the same tag.)

- We use $P_0 : \varepsilon$ and $P_1 : \varepsilon$ as initial processes (where ε is the empty sequence).
- We define local implementation transitions $\rightarrow_i$ from the initial specification $\widetilde{P}$ and the specifications traces:

$$\frac{\widetilde{P} \xrightarrow{\rho(\psi)}_{\mathsf{S}} \widetilde{P'} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P''} \qquad i\ a\ j\ \ell \notin \psi}{P'_i : \psi \xrightarrow{\overline{i\ a\ j\ \ell}}_i P''_i : \psi.(i\ a\ j\ \ell)} \qquad \frac{\widetilde{P} \xrightarrow{\rho(\psi)}_{\mathsf{S}} \widetilde{P'} \xrightarrow{i\ a\ j}_{\mathsf{S}} \widetilde{P''} \qquad i\ a\ j\ \ell \notin \psi}{P'_j : \psi \xrightarrow{i\ a\ j\ \ell}_j P''_j : \psi.(i\ a\ j\ \ell)}$$

  where i, j is either $0, 1$ or $1, 0$ and where ρ yields an high-level trace by erasing all nonces ℓ in ψ.
- We define the adversary capability $\vdash_{\mathcal{C}}$ by $\dfrac{M \in \mathcal{M}}{\mathcal{M} \vdash_{\mathcal{C}} M}$ and $\dfrac{i \notin \mathcal{C}}{\mathcal{M} \vdash_{\mathcal{C}} i\ a\ j\ \ell}$.

Hence, an action is locally enabled only when it extends the specification trace recorded so far and the nonce ℓ is locally fresh. The adversary may send a message either after eavesdropping it or by constructing it with a compromised sender. (Pragmatically, a concrete implementation may generate ℓ at random, or increment a message sequence number, and may use a more compact representation of ψ.)

We illustrate this scheme with an implementation of example 2. The main source execution is

$$(A_0, A_1) \xrightarrow{0\ a\ 1}_{\mathsf{S}} (b, \overline{b}) \xrightarrow{1\ b\ 0}_{\mathsf{S}} (\mathbf{0}, \mathbf{0})$$

Hence, we get (for any $\ell \neq \ell'$)

$$\begin{array}{llll} A_0 : \varepsilon \xrightarrow{\overline{0\ a\ 1\ \ell}}_0 & b : (0\ a\ 1\ \ell) & \xrightarrow{1\ b\ 0\ \ell'}_0 \mathbf{0} : (0\ a\ 1\ \ell).(1\ b\ 0\ \ell') \\ A_1 : \varepsilon \xrightarrow{0\ a\ 1\ \ell}_1 & \overline{b} : (0\ a\ 1\ \ell) & \xrightarrow{\overline{1\ b\ 0\ \ell'}}_1 \mathbf{0} : (0\ a\ 1\ \ell).(1\ b\ 0\ \ell') \end{array}$$

In the implementation defined above, the history carried by the implementation processes forbids the first process to wrongly receive the response b before sending the request a. This implementation is sound and complete.

Our implementation scheme is sound and complete:

Theorem 1 *Let (P_0, P_1) be a specification. $\langle (P_0 : \varepsilon, P_1 : \varepsilon), \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is a sound and complete implementation of (P_0, P_1).*

The soundness of our implementation above relies on every machine recording every communication (since every message is sent by one machine and received by the other one); this approach does not extend to specifications with more than two machines, inasmuch as these machines do not directly observe actions between two remote machines.

The proof of Theorem 1 is at the end of this chapter, in Subsection 2.10.1.

2.5 Unimplementability

In the preceding section, we presented a complete and sound implementation for binary sessions. We now illustrate some difficulties in the general n-ary case and introduce restrictions that aim at avoiding these problematic specification.

Example 7 *Consider a variant of Example 1 with the same V_1, V_2, V_3 and R but with the election officer E split into E_1 and E_2:*

$$\begin{array}{rl} V_1 = V_2 = V_3 = & \overline{c_1} + \overline{c_2} \\ E_1 = & c_1.c_1.\overline{r_1} \\ E_2 = & c_2.c_2.\overline{r_2} \\ R = & r_1 + r_2 \end{array}$$

One of the voters (say, V_1) may cheat, and send both c_1 *to* E_1 *and* c_2 *to* E_2*. To prevent this unsound execution,* E_1 *and* E_2 *would need to communicate with one another, thereby breaking message transparency. Therefore no adequate implementation (Definition 2) of this example can be both sound and complete.*

This unimplementability problem, and those discussed below, originate in the linearity of our choice, send, and receive operators. In the specification, the choice operator allows two mutually exclusive behaviors; however, in the implementation, it is sometime not possible to enforce that both behaviors are not taken at the same time.

2.5.1 Unimplementable patterns

Next, we further investigate the problem and identify a number of minimal specification patterns for which there is no adequate, complete and sound implementation.

Simple-fork Consider the following specification

$$\begin{aligned} A &= \overline{b} + \overline{c} \\ B &= b \\ C &= c \end{aligned}$$

The process A can send a message to either B or C. A dishonest machine in charge of running A can send both messages. Any complete adequate implementation of this specification is unsound.

Proof: By completeness, the implementation must have two traces that are validly explained by $0\ b\ 1$ and $0\ c\ 2$, respectively. Hence, there exist A_1, A'_1, B_1, C_1 such that:

$$A \xrightarrow{\overline{M_b}}_A A_1 \qquad A \xrightarrow{\overline{M_c}}_A A'_1 \qquad B \xrightarrow{M_b}_B B_1 \qquad C \xrightarrow{M_c}_C C_1$$

with $\rho(M_b) = A\ b\ B$ and $\rho(M_c) = A\ c\ C$. Now, with $\mathcal{C} = \{B; C\}$, as the implementation is adequate, $\emptyset \vdash_{\mathcal{C}} M_b$ and $\emptyset \vdash_{\mathcal{C}} M_c$. We get $\emptyset, (B, C) \xrightarrow{M_b}_{\mathrm{I}} (B_1, C) \xrightarrow{M_c}_{\mathrm{I}} (B_1, C_1)$, which has no valid explanation: the implementation is unsound. □

To prevent this pattern, we demand that the two sides of a sum always emit to the same participant. This excludes some implementable specifications, for instance when one of the send can never be received (because the reception is never reached).

Duplicate-message Consider the following specification

$$\begin{aligned} A &= \overline{a} \\ B &= a \\ C &= a \end{aligned}$$

The process A can send a message to either B or C. A dishonest machine in charge of running A can send the same message to both. Any complete adequate implementation of this specification is unsound

Proof: By completeness, the implementation must have two traces that are validly explained by $A\ a\ B$ and $A\ a\ C$. Hence, there exist A_1, A'_1, B_1, C_1 such that:

$$A \xrightarrow{\overline{M_a}}_A A_1 \qquad A \xrightarrow{\overline{M'_a}}_A A'_1 \qquad B \xrightarrow{M_a}_B B_1 \qquad C \xrightarrow{M'_a}_C C_1$$

with $\rho(M_a) = A\ a\ B$ and $\rho(M'_a) = A\ a\ C$. Now, with $\mathcal{C} = \{B; C\}$, as the implementation is adequate, $\emptyset \vdash_{\mathcal{C}} M_a$ and $\emptyset \vdash_{\mathcal{C}} M'_a$. We get $\emptyset, (B, C) \xrightarrow{M_a}_{\mathrm{I}} (B_1, C) \xrightarrow{M'_a}_{\mathrm{I}} (B_1, C_1)$, which has no valid explanation: the implementation is unsound. □

To prevent this pattern, we demand that receives of each action a occur only at one participant.

Delayed-fork Consider the following specification

$$\begin{aligned} A &= \overline{a} + \overline{b} \\ B &= a.\overline{c} \,|\, b.\overline{d} \\ C &= c \\ D &= d \end{aligned}$$

The process A can send either a or b to B; B subsequently sends a message to C or D. Here, dishonest machines in charge of running A and B can collude: A sends both messages to B who sends in turn both messages to C and D. Any complete adequate implementation of this specification is unsound.

PROOF: By completeness, the implementation must have two traces that are validly explained by $A\ a\ B.B\ c\ C$ and $A\ b\ B.B\ d\ D$. Hence, there exist A_1, A'_1, B_1, B_2, B_1, B_2, C_1, D_1 such that:

$$A \xrightarrow{\overline{M_a}}_A A_1 \qquad A \xrightarrow{\overline{M_b}}_A A'_1 \qquad B \xrightarrow{M_a}_B B_1 \xrightarrow{\overline{M_c}}_B B_2 \qquad B \xrightarrow{M_b}_B B'_1 \xrightarrow{\overline{M_d}}_B B'_2$$
$$C \xrightarrow{M_c}_C C_1 \qquad D \xrightarrow{M_d}_D D_1$$

with $\rho(M_a) = A\ a\ B$, $\rho(M_b) = A\ b\ B$, $\rho(M_c) = B\ c\ C$, $\rho(M_d) = B\ d\ D$. Now, with $\mathcal{C} = \{C; D\}$, as the implementation is adequate, $\emptyset \vdash_{\mathcal{C}} M_a, M_b$ and $M_a, M_b \vdash_{\mathcal{C}} M_c, M_d$. We get $\emptyset, (C, D) \xrightarrow{M_c}_{\mathrm{I}} (C_1, D) \xrightarrow{M_d}_{\mathrm{I}} (C_1, D_1)$, which has no valid explanation: the implementation is unsound. □

To prevent this pattern, we demand that the two sides of a sum affect the same participants in the same order (the restriction on the two sides of a sum should not stop at the first level).

Early-duplicate-fork Consider the following specification

$$\begin{aligned} A &= \overline{a} \\ B &= a.\overline{c} \,|\, a.\overline{d} \\ C &= c \\ D &= d \end{aligned}$$

The process A sends a message to B. Depending on what part of B receives it, B can sends a message to either C or D. A dishonest machine in charge of running B can send both messages to C and D after a single receive. Any complete adequate implementation of this specification is unsound.

PROOF: By completeness, the implementation must have two traces that are validly explained by $A\ a\ B.B\ c\ C$ and $A\ a\ B.B\ d\ D$. Hence, there exist A_1, A'_1, B_1, B'_1, B_2, B'_2, C_1, D_1 such that:

$$A \xrightarrow{\overline{M_a}}_A A_1 \qquad A \xrightarrow{\overline{M'_a}}_A A'_1 \qquad B \xrightarrow{M_a}_B B_1 \xrightarrow{\overline{M_c}}_B B_2 \qquad B \xrightarrow{M'_a}_B B'_1 \xrightarrow{\overline{M_d}}_B B'_2$$
$$C \xrightarrow{M_c}_C C_1 \qquad D \xrightarrow{M_d}_D D_1$$

with $\rho(M_a) = A\ a\ B$, $\rho(M'_a) = A\ a\ B$, $\rho(M_c) = B\ c\ C$, $\rho(M_d) = B\ d\ D$. Now, with $\mathcal{C} = \{C; D\}$, as the implementation is adequate, $\emptyset \vdash_{\mathcal{C}} M_a, M'_a$ and $M_a, M'_a \vdash_{\mathcal{C}} M_c, M_d$. We get $\emptyset, (C, D) \xrightarrow{M_c}_{\mathrm{I}} (C_1, D) \xrightarrow{M_d}_{\mathrm{I}} (C_1, D_1)$, which has no valid explanation: the implementation is unsound. □

To prevent this pattern, we demand that each occurrence of an action affect the same participants, in the same order.

Mixed-fork Consider the following specification

$$\begin{array}{lcl} A & = & (\overline{b_0} \,|\, \overline{c_0}) + (\overline{b_1} \,|\, \overline{c_1}) \\ B & = & b_0 \,|\, b_1 \\ C & = & c_0 \,|\, c_1 \end{array}$$

The process A can send either b_0 to B and c_0 to C, or b_1 to B and c_1 to C. A dishonest machine in charge of running A can send b_0 to B and c_1 to C. Any complete adequate implementation of this specification is unsound.

Proof: By completeness, the implementation must have four traces that are validly explained by $A\ b_0\ B$, $A\ b_1\ B$, $A\ c_0\ C$ and $A\ c_1\ C$. Hence, there exist A_1, A'_1, A''_1, A'''_1, B_1, B'_1, C_1, C'_1 such that:

$$A \xrightarrow{\overline{M_{b_0}}}_A A_1 \qquad A \xrightarrow{\overline{M_{c_0}}}_A A'_1 \qquad A \xrightarrow{\overline{M_{b_1}}}_A A''_1 \qquad A \xrightarrow{\overline{M_{c_1}}}_A A'''_1 \qquad B \xrightarrow{M_{b_0}}_B B_1$$
$$B \xrightarrow{M_{b_1}}_B B'_1 \qquad C \xrightarrow{M_{c_0}}_C C_1 \qquad C \xrightarrow{M_{c_1}}_C C'_1$$

with $\rho(M_{b_0}) = A\ b_0\ B$, $\rho(M_{b_1}) = A\ b_1\ B$, $\rho(M_{c_0}) = A\ c_0\ C$, $\rho(M_{c_1}) = A\ c_1\ C$. Now, with $\mathcal{C} = \{B; C\}$, as the implementation is adequate, $\emptyset \vdash_{\mathcal{C}} M_{b_0}, M_{c_0}, M_{b_1}, M_{c_1}$. We get $\emptyset, (B, C) \xrightarrow{M_{b_0}}_{\mathrm{I}} (B_1, C) \xrightarrow{M_{c_1}}_{\mathrm{I}} (B_1, C'_1)$, which has no valid explanation: the implementation is unsound. □

To prevent this pattern, we demand in some cases that actions in parallel affect the same participants in the same order.

2.6 Implementability by typing

We develop a type system that guarantees implementability by tracking of causality in process actions. Corin et al. [2007] and Bhargavan et al. [2009] use a list of semantic rules to prevent unimplementability. In comparison, our type system is more uniform and efficient. However, as most type systems, it may refuse some specifications that are implementable (e.g., programs with unimplementable subprograms that can never be reached). Yet, we believe that practical specifications are typable.

This type system uses two kinds of types, for sequential processes and (possibly) parallel processes:

$\sigma ::=$	sequential types	$\pi ::=$	parallel types
$\mathbf{0}$	completion	σ	sequential type
$i.\sigma$	sequence	$i.\pi$	sequence
		$\pi \,|\, \pi$	parallel composition

Intuitively, our types indicate (with machine indexes i) which other participants may be affected by each action, and in what order. In Example 1, with $E = c_1.c_1.\overline{r_1} \,|\, c_2.c_2.\overline{r_2}$ and $R = r_1 + r_2$, the action c_1 is of type $E.R$ since it is received by process E and this reception may contribute to the emission of r_1 to process R.

We define subtyping with three base rules and two context rules:

$$\mathbf{0} \leq \sigma \qquad \pi \,|\, \pi \leq \pi \qquad \pi \leq \pi' \,|\, \pi \qquad \pi \leq \pi \,|\, \pi' \qquad \frac{\pi \leq \pi'}{i.\pi \leq i.\pi'} \qquad \frac{\pi_1 \leq \pi'_1 \quad \pi_2 \leq \pi'_2}{\pi_1 \,|\, \pi_2 \leq \pi'_1 \,|\, \pi'_2}$$

For example, with $\mathbf{0} \leq \sigma$, we state that an action that is never received can be considered, if necessary, as a potential dangerous action that sequentially affect processes in σ. This set of

rules ensure that, for a sequential type, we can "forget" potential future actions and obtain a less precise type and, for parallel types, we can duplicate or merge parallel copies carrying the same information. For simplicity, we inlined structural equivalence (e.g., we would not need the rule $\pi \leq \pi \,|\, \pi'$ if $\pi' \,|\, \pi$ was structurally equivalent to $\pi' \,|\, \pi$).

We type local processes at each machine $i \in 0..n-1$, in a given environment Γ that map channels to types. The typing judgment $\Gamma \vdash_i P : \pi$ indicates that P can be given type π at machine i in environment Γ, with the rules below:

$$\text{(Send)}\quad \Gamma, a : \pi \vdash_i \overline{a} : i.(\pi \backslash i) \qquad \text{(Receive)}\quad \frac{\Gamma, a : \pi \vdash_i P : \pi' \quad \pi' \leq \pi}{\Gamma, a : \pi \vdash_i a.P : \pi'} \qquad \text{(Sub)}\quad \frac{\Gamma \vdash_i P : \pi \quad \pi \leq \pi'}{\Gamma \vdash_i P : \pi'} \qquad \text{(Nil)}\quad \Gamma \vdash_i \mathbf{0} : i$$

$$\text{(Plus)}\quad \frac{\Gamma \vdash_i P_0 : \sigma \quad \Gamma \vdash_i P_1 : \sigma}{\Gamma \vdash_i P_0 + P_1 : \sigma} \qquad \text{(Par)}\quad \frac{\Gamma \vdash_i P_0 : \pi \quad \Gamma \vdash_i P_1 : \pi'}{\Gamma \vdash_i P_0 \,|\, P_1 : \pi \,|\, \pi'} \qquad \text{(Repl)}\quad \frac{\Gamma \vdash_i P : \pi}{\Gamma \vdash_i !P : \pi}$$

where $\pi \backslash i$ is π after erasure of every occurrence of i.

Rule (Send) gives to the output $\overline{a}$ the type of action a (minus i) preceded by i. This records that $\overline{a}$ at host i affects any process that receives on a. Conversely, rule (Receive) gives to $a.P$ the type of the continuation process P, and checks that it is at least as precise as the type of action a. Rule (Sub) enables subtyping. Rule (Nil) gives type i to an empty process, since it has no impact outside i. Rule (Plus) ensures that the two branches of a choice have the same effect, a sequential type, excluding e.g., the typing of the specifications in Example 7. Rules (Par) and (Repl) deal with parallel compositions.

For instance, in the environment $\Gamma = r_1 : R, r_2 : R, c_1 : E.R, c_2 : E.R$, the processes of Example 1 have types

$$\Gamma \vdash_{V_1} V_1 : V_1.E.R \qquad \Gamma \vdash_{V_2} V_2 : V_2.E.R \qquad \Gamma \vdash_{V_3} V_3 : V_3.E.R \qquad \Gamma \vdash_E E : E.R \qquad \Gamma \vdash_R R : R$$

Conversely, the processes $V_1 = V_2 = V_3 = \overline{c_1} + \overline{c_2}$ are not typable within the unsafe specification of Example 7 (in which E is replaced by $E_1 = c_1.c_1.\overline{r_1}$ and $E_2 = c_2.c_2.\overline{r_2}$), because $\overline{c_1}$ (e.g.,) and $\overline{c_2}$ necessarily have incompatible types, e.g., $\Gamma \vdash_{V_1} \overline{c_1} : V_1.E_1.R$ and $\Gamma \vdash_{V_1} \overline{c_2} : V_1.E_2.R$.

We end this section by defining typability for global specifications, with a shared environment for all machines and a technical condition to ensure consistency on channels with parallel types.

Definition 6 (Well-typed) *A global specification $\widetilde{P}$ is* well-typed *when, for some environment Γ and each $i \in 0..n-1$, we have $\Gamma \vdash_i P_i : \pi_i$ and, for each $(a : \pi) \in \Gamma$, either π is (a subtype of) a sequential type, or $\widetilde{P}$ has at most one receive on a.*

We informally check that the unimplementable patterns are not well-typed.

Simple-fork By Rule Nil, $\vdash_B \mathbf{0} : B$, $\vdash_C \mathbf{0} : C$. In order to type $B = b.\mathbf{0}$ and $C = c.\mathbf{0}$ with Rule Receive, we need $\Gamma(b) = B$ and $\Gamma(c) = C$. Then, $\Gamma \vdash_A b : A.B$ and $\Gamma \vdash_A c : A.C$ by Rule Send. We cannot apply Rule Plus and type process A since $A.B$ and $A.C$ are different.

Duplicate-message By Rules Receive and Nil, in order to type B and C, we need $\Gamma(a) = B$ and $\Gamma(a) = C$. This is not possible.

Delayed-fork By Rules Receive and Nil, in order to type C and D, we need $\Gamma(c) = C$ and $\Gamma(d) = D$. By Rules Receive and Send, in order to type B, we need $\Gamma(a) = B.C$ and $\Gamma(b) = B.D$. Then, $\Gamma \vdash_A a : A.B.C$ and $\Gamma \vdash_A b : A.B.D$ by Rule Send. We cannot apply Rule Plus and type process A.

Early-duplicate-fork By Rules Receive and Nil, in order to type C and D, we need $\Gamma(c) = C$ and $\Gamma(d) = D$. By Rule Send, $\Gamma \vdash_B c : B.C$ and $\Gamma \vdash_A d : B.D$. Then, by Rule Receive, in order to type B, we need $\Gamma(a) = B.C$ and $\Gamma(a) = B.D$. This is not possible.

Mixed-fork By Rules Receive and Nil, in order to type B and C, we need $\Gamma(b_0) = B$, $\Gamma(b_1) = B$, $\Gamma(c_0) = C$ and $\Gamma(c_1) = C$. By Rules Send and Par $\vdash_A \overline{b_0} \,|\, \overline{c_0} : A.(B \,|\, C)$ and $\vdash_A \overline{b_1} \,|\, \overline{c_1} : A.(B \,|\, C)$. We cannot apply Rule Plus and type process A in this case since the types are parallels.

2.7 History-tracking implementations

In this section we present an implementation for session specifications. We prove that the implementation is complete, and that it is sound when the specification is well-typed (Definition 6). The resulting family of implementations subsumes the one presented in the special case of binary sessions (Section 2.4).

2.7.1 Multiparty specifications and history-tracking implementations

As seen in Example 1, in a multiparty system, a local action at one machine may causally depend on communications between other machines. To prevent cheating, we embed evidence of past execution history in our implementation messages. Thus, to implement Example 1, the code for the election officer E explicitly forwards evidence of receiving c_1 twice in order to convince R that it can send the result r_1.

As a preliminary step, we enrich processes with histories of prior communications. Then, we equip these processes with a refined semantics, with rules that define how histories are collected and communicated. Finally, the presence of histories allows us to constrain each local implementation by prescribing what messages may be sent and received at runtime. (Our history-tracking implementation is related to locality semantics for CCS; for instance Boudol and Castellani [1994] use *proved labeled transitions* that keep track of causality by recording where each action occurs in a process.)

Histories and messages are defined by the following grammar:

$H ::=$	Histories
$\quad \varepsilon$	empty history
$\quad H.M$	recorded receive
$M ::=$	Messages
$\quad (H\ i\ a\ j\ \ell)$	

Histories are sequences of messages. Each message $H\ i\ a\ j\ \ell$ records an action a between sender i and receiver j (where i and j are indexes of processes in the global specification), with a history H that provides evidence that action a is indeed enabled. In addition, ℓ denotes a unique nonce, freshly generated for this message, used to avoid replays.

The syntax of local processes is extended with histories as follows:

$T ::=$	Threads
$\quad \mathbf{0}$	inert thread
$\quad \overline{a}$	asynchronous send
$\quad a.P$	receive
$\quad P + P'$	choice
$\quad !P$	replication
$R ::=$	History-tracking processes
$\quad (T_0 : H_0 \,\|\, T_1 : H_1 \,\|\, \ldots \,\|\, T_{k-1} : H_{k-1})$	parallel composition history-annotated threads
$\widetilde{R} ::=$	Global history-tracking specifications
$\quad (R_0, R_1, \ldots, R_{n-1})$	tuple of n history-tracking processes

where P ranges over the local processes of Section 2.2. Our specification processes are split into different parallel components, each with its own history. For example, when $P = a.\overline{b} \,|\, \overline{c}$ receives a, this receive enables action $\overline{b}$ (and is tracked in its history) but is independent from action $\overline{c}$. So, a *thread* T is a (specification) process without parallel composition at top-level, a *history-tracking process* R is a collection of threads in parallel, each annotated with its history, and a *global history-tracking specification* $\widetilde{R}$ is a tuple of n history-tracking processes. We write $M \in \widetilde{R}$ when M appear in $\widetilde{R}$.

When P is not a single thread, we write $P : H$ for the parallel composition of its threads, each annotated with the same history H (i.e., $P_0 \,|\, P_1 : H$ stands for $P_0 : H \,|\, P_1 : H$). Further, the function $[\![\cdot]\!]_0$ adds empty histories to a global specification, effectively generating a global history-tracking specification. Conversely, since any thread is a local process, an history-tracking process (resp. a global history specification) stripped of its histories is a local process (resp. a global specification).

2.7.2 Semantics of history specifications ($\rightarrow_{\mathsf{h}}$ and $\rightarrow_{\mathsf{H}}$)

We define labeled transitions for history specifications. We write $R \xrightarrow{\gamma}_{\mathsf{h}} R'$ when R can evolve to R' with action γ. It corresponds to an input or an output on one of its threads.

$$\text{(SendH)}\ \frac{T \xrightarrow{\overline{a}}_{\mathsf{P}} P}{T : H \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} P : H} \qquad \text{(ReceiveH)}\ \frac{T \xrightarrow{a}_{\mathsf{P}} P}{T : H' \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} P : H'.(H\ i\ a\ j\ \ell)} \qquad \text{(ParH)}\ \frac{R \xrightarrow{\gamma}_{\mathsf{h}} R'}{R \,|\, R'' \xrightarrow{\gamma}_{\mathsf{h}} R' \,|\, R''}$$

Rule (ReceiveH) records the message in the thread history. In contrast, rule (SendH) does not record the message, since our semantics is asynchronous. Rule (ParH) is a rule for parallel contexts; we omit the symmetric rule.

We write $\widetilde{R} \xrightarrow{M}_{\mathsf{H}} \widetilde{R}'$ to represent communications between history-tracking specifications, with a single global rule:

$$\frac{\begin{array}{ll} R_i \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} R_i^\circ & (R_k = R_k^\circ)^{k \neq i} \\ R_j^\circ \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} R_j' & (R_k^\circ = R_k')^{k \neq j} \quad H\ i\ a\ j\ \ell \notin \widetilde{R} \end{array}}{\widetilde{R} \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{H}} \widetilde{R}'}$$

A communication step consists of a send (at machine i) followed by a receive (at machine j) for local history-tracking processes (possibly with $i = j$). The condition $H\ i\ a\ j\ \ell \notin \widetilde{R}$ excludes multiple usage of the same message.

The example below illustrates the semantics of history specifications.

Example 8 *We consider a protocol involving three participant, with a request form the first participant to the second, followed by a response from the second participant to the third. The processes are $C_0 = \overline{a}$, $C_1 = a.\overline{b}$, and $C_2 = b$. The following is a trace of its global history specification.*

$$\begin{aligned} [\![(C_0, C_1, C_2)]\!] = (\overline{a} : \varepsilon, a.\overline{b} : \varepsilon, b : \varepsilon)_0 &\xrightarrow{\varepsilon\ 0\ a\ 1\ \ell}_{\mathsf{H}} (\mathbf{0} : \varepsilon, \overline{b} : \varepsilon\ 0\ a\ 1\ \ell, b : \varepsilon) \\ &\xrightarrow{(\varepsilon\ 0\ a\ 1\ \ell) 1\ b\ 2\ \ell'}_{\mathsf{H}} (\mathbf{0} : \varepsilon, \mathbf{0} : \varepsilon\ 0\ a\ 1\ \ell, \mathbf{0} : (\varepsilon\ 0\ a\ 1\ \ell)\ 1\ b\ 2\ \ell') \end{aligned}$$

2.7.3 Local semantics ($\rightarrow_i$)

We are now ready to define distributed implementation transitions locally, for each machine $i \in 0..n-1$, written $R_i \xrightarrow{M}_i R'_i$:

$$\frac{[\![\widetilde{P}]\!]_0 \xrightarrow{\psi}_{\mathsf{H}} \widetilde{R'} \quad H\ i\ a\ j\ \ell \notin \widetilde{R'} \qquad R'_i \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} R^\circ_i \quad (R'_k = R^\circ_k)^{k\neq i} \qquad R^\circ_j \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} R''_j \quad (R^\circ_k = R''_k)^{k\neq j}}{R'_i \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_i R^\circ_i} \qquad \frac{[\![\widetilde{P}]\!]_0 \xrightarrow{\psi}_{\mathsf{H}} \widetilde{R'} \quad H\ i\ a\ j\ \ell \notin \widetilde{R'} \qquad R'_i \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} R^\circ_i \quad (R'_k = R^\circ_k)^{k\neq i} \qquad R^\circ_j \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{h}} R''_j \quad (R^\circ_k = R''_k)^{k\neq j}}{R^\circ_j \xrightarrow{H\ i\ a\ j\ \ell}_j R''_j}$$

These rules (with identical premises) prescribe that a distributed implementation can send or receive a message when the corresponding communication is enabled in some global history specification state that is reachable from the initial history specification process $[\![\widetilde{P}]\!]_0$.

For instance, with the specification of Example 8, we get

$$\begin{array}{rl} \overline{a} : \varepsilon & \xrightarrow{\overline{\varepsilon\ 0\ a\ 1\ \ell}}_0 0 \\ a.\overline{b} : \varepsilon & \xrightarrow{\varepsilon\ 0\ a\ 1\ \ell}_1 b : (\varepsilon\ 0\ a\ 1\ \ell) \xrightarrow{\overline{(\varepsilon\ 0\ a\ 1\ \ell) 1\ b\ 2\ \ell'}}_1 0 : (\varepsilon\ 0\ a\ 1\ \ell) \\ b : \varepsilon & \xrightarrow{(\varepsilon\ 0\ a\ 1\ \ell)\ 1\ b\ 2\ \ell'}_2 0 : ((\varepsilon\ 0\ a\ 1\ \ell)\ 1\ b\ 2\ \ell') \end{array}$$

A naive concrete implementation may enumerate all possible runs at every communication. A more efficient implementation may cache this computation and perform incremental checks, or perform this computation at compile-time. (See Corin et al. [2007], Bhargavan et al. [2009] for optimized implementations in the sequential case.)

2.7.4 Distributed implementation

We finally define our distributed implementation, with $\widetilde{Q} = [\![\widetilde{P}]\!]_0$ as initial implementation processes, with $(\xrightarrow{\gamma}_i)_i$ defined above as transition relations between implementation processes, and with capabilities $\vdash_{\mathcal{C}}$ for the adversary defined by

$$\frac{M \in \mathcal{M}}{\mathcal{M} \vdash_{\mathcal{C}} M} \qquad \frac{(\mathcal{M} \vdash_{\mathcal{C}} M_m)^{m<k} \qquad i \notin \mathcal{C}}{\mathcal{M} \vdash_{\mathcal{C}} M_0.\cdots.M_{k-1}\ i\ a\ j\ \ell}$$

The first rule states that the adversary can eavesdrop messages on the network. The second rule states that the adversary can build any message sent by a dishonest participant, with an history composed of sequence of messages previously obtained. Conversely, the adversary cannot forge any message from a compliant machine (i.e., a machine $i \in \mathcal{C}$). This can be cryptographically enforced by signing messages and their histories. Our implementation is adequate, in particular a dishonest participant can behave as an honest participant.

(The global transition rules $\rightarrow_{\mathrm{I}}$ for our distributed implementation follow from the general definitions of Section 2.3.)

2.7.5 Soundness and completeness

Our implementation is complete, that is, it can simulate any specification trace:

Theorem 2 (Completeness) *$\langle [\![\widetilde{P}]\!]_0, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is a complete implementation of $\widetilde{P}$.*

The proof of Theorem 2 is at the end of this chapter, in Subsection 2.10.2.

Our main result states that our implementation is also sound *when applied to well-typed specifications*; as explained in Section 2.6, many other specifications cannot be safely implemented.

Theorem 3 (Soundness by Typing) *If $\widetilde{P}$ is well-typed, then $\langle [\![\widetilde{P}]\!]_0, \widetilde{\rightarrow}, \vdash_{\mathcal{C}} \rangle$ is sound.*

The proof of Theorem 3 is at the end of this chapter, in Subsection 2.10.3.

2.8 Sequential multiparty sessions (application)

We consider secure implementations of sequential multiparty sessions, as defined by Corin et al. [2007]. Their sessions are a special case of process specifications. We recall their grammar, which defines a session as a parallel composition of *role processes*, each process specifying the local actions for one role of the session.

$\tau ::=$	Payload types
$\quad \mathrm{int} \mid \mathrm{string}$	$\quad$ base types
$p ::=$	Role processes
$\quad !(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k}$	$\quad$ send
$\quad ?(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k}$	$\quad$ receive
$\quad \mu\chi.p$	$\quad$ recursion declaration
$\quad \chi$	$\quad$ recursion
$\quad 0$	$\quad$ end
$S ::=$	Sequential session (with n roles)
$\quad (r_i = p_i)_{i \in 0..n-1}$	

Their role processes must alternate between send and receive actions, and moreover only the initiator role process (p_0) begins with a send. Thus, there is always at most one role that can send the next message, as expected of a sequential session. From a more global viewpoint, a session is represented as a directed graph, whose nodes represent roles and whose arrows are indexed by unique communication labels. The paths in the graph correspond to the global execution traces for the session. Given an additional implementability property on these paths, named "no blind fork", they construct a cryptographic implementation that guarantees *session integrity*, even for sessions with dishonest participants, a property closely related to Soundness (Definition 4).

For example, the session graph:

$$A \xrightarrow{\quad a \quad} B \xrightarrow{\quad b \quad} A$$

corresponds to the source code:

$$A_0 = !(a : int\ ;\ ?(b\ :\ int\ ;\ 0)) \qquad A_1 = ?(a : int\ ;\ !(b\ :\ int\ ;\ 0))$$

In our grammar, it can be modeled with the two processes:

$$A_0 = \overline{a} \mid b \qquad A_1 = a.\overline{b}$$

To illustrate our approach, we show that our generic implementation directly applies to every session that they implement (although with less compact message formats). We translate their role processes into our syntax as follows:

$$\begin{aligned}
[\![p]\!] = {} & \textstyle\sum_{i<k} (\overline{f_i}) \qquad \text{when } p = !(f_i : \widetilde{\tau_i}\ ;\ p_i)_{i<k} \\
& \mid \textstyle\prod_{(?(f_i:\widetilde{\tau_i}\ ;\ p_i)_{i<k}) \in p,\ i<k,\ p_i = !(f'_j:\widetilde{\tau'_j}\ ;\ p'_j)_{j<l}} !f_i.\textstyle\sum_{j<l}(\overline{f'_j}) \\
& \mid \textstyle\prod_{(?(f_i:\widetilde{\tau_i}\ ;\ p_i)_{i<k}) \in p,\ i<k,\ p_i = \chi,\ (\mu\chi.!(f_j:\widetilde{\tau_j}\ ;\ p_j)_{j<l}) \in p} !f_i.\textstyle\sum_{j<l}(\overline{f'_j}) \\
[\![(r_i = p_i)_{i \in 0..n-1}]\!] = {} & ([\![p_i]\!])_{i \in 0..n-1}
\end{aligned}$$

Each node in a session graph has an input arrow and one or several output arrows, representing an internal choice between outputs. Accordingly, our translation associates to each node a replicated input followed by a choice between asynchronous outputs. In addition, the initial role for the session is an internal choice between outputs, translated to an internal choice of asynchronous outputs. We can check that our translation behaves as the initial sequential session. (The sequentiallity of the session follows from the presence of a single choice between outputs in every reachable state, so we can replicate all inputs, whether they occur in recursive loops or not.) Also, for any given sequential session, typability of the translation (Definition 6) coincides with the "no blind fork" property Corin et al. [2007]. Hence, every sequential session supported by their implementation is typable, and can also be implemented in our general framework.

2.9 Conclusion

We have given an account of distributed specifications and their implementations in three steps: (1) a global specification language; (2) a distributed implementation semantics; (3) correctness and completeness results, depending on an implementability condition. In combination, this yields a general framework for designing and verifying n-ary communication abstractions with strong, guaranteed security properties. (In comparison, the work on sequential multiparty sessions Corin et al. [2007] can now be seen as a specialized cryptographic implementation for the sequential case.)

2.10 Proofs

2.10.1 Proof of Theorem 1

In order to prove soundness, we first show some properties of low level traces when both participants are honest ($\mathcal{C} = \{0; 1\}$):

Lemma 1 *Let P'_0 and P'_1 be two processes, $\mathcal{M}$ be a set of low level messages and ψ, ψ_0, ψ_1 be three low level traces such that*

$$0, (P_0 : 0, P_1 : 0) \xrightarrow{\psi}_{\mathrm{I}} \mathcal{M}, (P'_0 : \psi_0, P'_1 : \psi_1)$$

Let γ be a low level action and $i\ a\ j\ \ell$ a low level message. We have:

1. *if $\overline{i\ a\ j\ \ell} \in \psi$ then $i\ a\ j\ \ell \in \psi_i$;*
2. *if $i\ a\ j\ \ell \in \psi$, then $i\ a\ j\ \ell \in \psi_j$;*
3. *if $i\ a\ j\ \ell \in \psi$, then $\overline{i\ a\ j\ \ell}$ occurs in ψ before $i\ a\ j\ \ell$;*
4. *γ occurs at most once in ψ.*

Proof: The proof is by induction on the length of ψ. If $\psi = \varepsilon$, the lemma holds. Otherwise, there exists ψ', ψ'_0, ψ'_1, γ, $\mathcal{M}'$, P''_0 and P''_1 such that

$$0, (P_0 : 0, P_1 : 0) \xrightarrow{\psi'}_{\mathrm{I}} \mathcal{M}', (P''_0 : \psi'_0, P''_1 : \psi'_1) \xrightarrow{\gamma}_{\mathrm{I}} \mathcal{M}, (P'_0 : \psi_0, P'_1 : \psi_1)$$

We consider three cases for γ:

- If $\gamma = \overline{i\ a\ j\ \ell}$, by definition of $\rightarrow_{\mathrm{I}}$, and $\rightarrow_i$, we have $\psi_i = \psi'_i.i\ a\ j\ \ell$, $\psi'_j = \psi_j$, and $\mathcal{M} = \mathcal{M}'.i\ a\ j\ \ell$. Using the induction hypothesis, we prove items 1-3 on ψ, ψ_0, and ψ_1. By definition of $\rightarrow_i$, we have $i\ a\ j\ \ell \notin \psi'_i$. Using induction hypothesis applied to item 1, we deduce that $\overline{i\ a\ j\ \ell} \notin \psi$. We thus prove item 4 for ψ'.

- If $\gamma = i\ a\ j\ \ell$, by definition of $\rightarrow_{\mathrm{I}}$, and $\rightarrow_j$, we have $\psi'_i = \psi_i$, $\psi_j = \psi'_j.i\ a\ j\ \ell$, and $\mathcal{M}' = \mathcal{M}$. Using the induction hypothesis, we prove items 1-3 on ψ, ψ_0, and ψ_1. By definition of $\rightarrow_j$, we have $i\ a\ j\ \ell \notin \psi'_j$. Using induction hypothesis applied to item 2, we deduce that $i\ a\ j\ \ell \notin \psi'$. We thus prove item 4 for ψ.
- Otherwise, if $\gamma = \overline{i'\ a'\ j'\ \ell}$ or $\gamma = i'\ a'\ j'\ \ell$ with $i'\ a'\ j'\ \ell \neq i\ a\ j\ \ell$, we conclude by induction.

□

To prove soundness when both participants are honest ($\mathcal{C} = \{0; 1\}$), we introduce an intermediate semantics that independently gathers the inputs and outputs of a couple of process. Given two processes (P'_0, P'_1), its rules are:

$$\frac{P'_0 \xrightarrow{\overline{a}}_{\mathsf{P}} P''_0}{(P'_0, P'_1) \xrightarrow{\overline{0\ a\ 1\ \ell}} (P''_0, P'_1)} \qquad \frac{P'_0 \xrightarrow{a}_{\mathsf{P}} P''_0}{(P'_0, P'_1) \xrightarrow{1\ a\ 0\ \ell} (P''_0, P'_1)}$$

and symmetrically for P'_1. We first show that implementation traces are traces of this semantic:

Lemma 2 *Let P'_0 and P'_1 be two processes, $\mathcal{M}$ be a set of low level messages and ψ, ψ_0, ψ_1 be three low level traces such that*

$$0, (P_0 : 0, P_1 : 0) \xrightarrow{\psi}_{\mathrm{I}} \mathcal{M}, (P'_0 : \psi_0, P'_1 : \psi_1)$$

Then

$$(P_0, P_1) \xrightarrow{\psi} (P'_0, P'_1)$$

Proof: From rules on $\rightarrow_i$ and $\rightarrow_{\mathsf{P}}$. □

This semantics has some reordering properties:

Lemma 3 *Let P'_0, P''_0, P'_1, and P''_1 be four processes, $i\ a\ j\ \ell$ and $i'\ a'\ j'\ \ell$ be two low level messages, and ψ_1 and ψ_2 be two low level traces.*

- *If $(P'_0, P'_1) \xrightarrow{\overline{i\ a\ j\ \ell}} \xrightarrow{\overline{i'\ a'\ j'\ \ell}} (P''_0, P''_1)$ then $(P'_0, P'_1) \xrightarrow{\overline{i'\ a'\ j'\ \ell}} \xrightarrow{\overline{i\ a\ j\ \ell}} (P''_0, P''_1)$ and a valid explanation of a trace $\psi_1.(\overline{i'\ a'\ j'\ \ell}).(\overline{i\ a\ j\ \ell}).\psi_2$ is also a valid explanation of the trace $\psi_1.(\overline{i\ a\ j\ \ell}).(\overline{i'\ a'\ j'\ \ell}).\psi_2$.*
- *If $(P'_0, P'_1) \xrightarrow{\overline{i\ a\ j\ \ell}} \xrightarrow{i'\ a'\ j'\ \ell} (P''_0, P''_1)$ and $i\ a\ j\ \ell \neq i'\ a'\ j'\ \ell$ then $(P'_0, P'_1) \xrightarrow{i'\ a'\ j'\ \ell} \xrightarrow{\overline{i\ a\ j\ \ell}} (P''_0, P''_1)$ and a valid explanation of the trace $\psi_1.(i'\ a'\ j'\ \ell).(\overline{i\ a\ j\ \ell}).\psi_2$ is also a valid explanation of a trace $\psi_1.(\overline{i\ a\ j\ \ell}).(i'\ a'\ j'\ \ell).\psi_2$.*

Proof: From rules on $\rightarrow_{\mathsf{P}}$ and definition of valid explanations. □

We are interested in low-level traces that follow a particular order, which makes them easier to relate to high-level traces.

Definition 7 *We say that a low-level trace ψ is* sorted *when*

- *every input in ψ is immediately preceded by its unique corresponding output;*
- *outputs not received in ψ are at the end of the trace.*

We now show that we can sort traces of our intermediary semantic if they follow certain properties.

Lemma 4 *Let P_0' and P_1' be two processes, and ψ a low level trace such that:*

- $(P_0, P_1) \xrightarrow{\psi} (P_0', P_1')$;
- *any low level action γ occurs at most once in ψ;*
- *for any low level message $i\ a\ j\ \ell$ occurring in ψ, $\overline{i\ a\ j\ \ell}$ occurs in ψ before $i\ a\ j\ \ell$.*

Then, there exists a sorted trace ψ' that follows the same properties and is such that every valid explanation of ψ' is a valid explanation of ψ.

Proof: In a trace following the hypotheses of Lemma 4, we call *uncoupled* outputs that have no corresponding input. We define an algorithm sorting the trace:

- If there exists an uncoupled output immediately followed by an action which is not an uncoupled output, we swap them. We repeat this operation until the uncoupled outputs are at the end of the trace. This terminates since uncoupled outputs are only moved toward the end of the trace.
- Otherwise, if there exist an input not immediately preceded by its corresponding output, we consider the output corresponding to the last of these inputs and we swap it with the action immediately after. We repeat this operation until there are no input immediately preceded by its corresponding output. This terminates since each step lower the distance between an input and its corresponding output, while leaving the end of the trace (containing already sorted actions) unchanged.

After each step, the resulting trace respects the hypotheses of the lemma (by hypotheses on ψ and Lemma 3). Also, every valid explanation of this new trace is a valid explanation of the original trace ψ.

□

Now, we show that we can build a valid explanation for low-level traces that are sorted.

Lemma 5 *Let P_0' and P_1' be two processes, and ψ a sorted low-level trace such that $(P_0, P_1) \xrightarrow{\psi} (P_0', P_1')$; Let ρ_0 be the function from low level messages to high levels messages (and extended to traces) defined by $\rho_0(\overline{i\ a\ j\ \ell}) = \varepsilon$ and $\rho_0(i\ a\ j\ \ell) = i\ a\ j$. Then, the high level trace $\varphi = \rho_0(\psi)$ is a valid explanation of ψ. There exists P_0'' and P_1'' such that $(P_0, P_1) \xrightarrow{\varphi}_{\mathsf{S}} (P_0'', P_1'')$. Furthermore, if there is no uncoupled output in ψ, then $(P_0, P_1) \xrightarrow{\varphi}_{\mathsf{S}} (P_0', P_1')$.*

Proof: The proof is by induction on the length of the trace ψ. Let ψ be a low level trace following the hypotheses of Lemma 5. we distinguish three possible cases:

- The last action of ψ is an output, of the form $\overline{i\ a\ j\ \ell}$: there exists a low level trace ψ' such that $\psi = \psi'.\overline{i\ a\ j\ \ell}$. The trace ψ' verifies the hypotheses of the lemma and is smaller that ψ, hence, by induction, there exists P_0'' and P_1'' such that $(P_0, P_1) \xrightarrow{\rho_0(\psi')}_{\mathsf{S}} (P_0'', P_1'')$. Moreover, $\rho_0(\psi) = \rho_0(\psi').\rho_0(\overline{i\ a\ j\ \ell}) = \rho_0(\psi')$, hence $(P_0, P_1) \xrightarrow{\rho_0(\psi')}_{\mathsf{S}} (P_0'', P_1'')$, and we conclude.

- The last action of ψ is an input, of the form $i\ a\ j\ \ell$, by hypothesis, it must be preceded by its unique corresponding output $\overline{i\ a\ j\ \ell}$: there exists a low level trace ψ' such that $\psi = \psi'.\overline{i\ a\ j\ \ell}.i\ a\ j\ \ell$, and there exists P_0'' and P_1'' such that $(P_0, P_1) \xrightarrow{\psi'} (P_0'', P_1'') \xrightarrow{\overline{i\ a\ j\ \ell}} \xrightarrow{i\ a\ j\ \ell} (P_0', P_1')$. Also, ψ does not finish by an output, hence, there is no uncoupled output in ψ or in ψ'. ψ' verifies the hypotheses of the lemma and is smaller that ψ, hence, by induction, $(P_0, P_1) \xrightarrow{\rho_0(\psi')}_{\mathsf{S}} (P_0'', P_1'')$. By definition of $\rightarrow$, we have $P_i'' \xrightarrow{\overline{a}}_{\mathsf{P}} P_i'$ and $P_j'' \xrightarrow{a}_{\mathsf{P}} P_j'$. Hence, by definition of $\rightarrow_{\mathsf{S}}$, $(P_0, P_1) \xrightarrow{\rho_0(\psi').\rho(i\ a\ j\ \ell)}_{\mathsf{S}} (P_0', P_1')$. By definition of ρ_0, we have $\rho_0(\psi) = \rho_0(\psi').\rho_0(\overline{i\ a\ j\ \ell}).\rho_0(i\ a\ j\ \ell) = \rho_0(\psi').i\ a\ j$. We conclude.
- Otherwise, ψ is empty and the lemma holds.

□

Our last lemma states (and prove by induction) a property slightly stronger than completeness.

Lemma 6 *For $\mathcal{C} = \{0; 1\}$ and for every source execution $\widetilde{P} \xrightarrow{\varphi}_{\mathsf{S}} \widetilde{P}'$, there exist a memory $\mathcal{M}$, and three low-level traces ψ, ψ_0, ψ_1 such that φ is a valid explanation of ψ, $\varphi = \rho(\psi_0) = \rho(\psi_1)$ and*

$$\varepsilon, (P_0 : \varepsilon, P_1 : \varepsilon) \xrightarrow{\psi}_{\mathrm{I}} \mathcal{M}, (P_0' : \psi_0, P_1' : \psi_1)$$

Proof: The proof is by induction on the trace φ. If $\varphi = \varepsilon$, φ is a valid explanation of $\psi = \varepsilon$. Otherwise, there exist φ', α, P_0'' and P_1'' such that $\varphi = \varphi'.\alpha$ and:

$$\widetilde{P} \xrightarrow{\varphi'}_{\mathsf{S}} \widetilde{P''} \xrightarrow{\alpha}_{\mathsf{S}} \widetilde{P'}$$

By induction on φ', there exists ψ', ψ_0', ψ_1' such that φ' valid explanation of ψ', $\varphi' = \rho(\psi_0') = \rho(\psi_1')$ and

$$\varepsilon, (P_0 : \varepsilon, P_1 : \varepsilon) \xrightarrow{\psi'}_{\mathrm{I}} \mathcal{M}, (P_0'' : \psi_0', P_1'' : \psi_1')$$

From the definition of $\rightarrow_i$ and $\rightarrow_{\mathrm{I}}$, we get

$$\varepsilon, (P_0 : \varepsilon, P_1 : \varepsilon) \xrightarrow{\psi' \cdot (\overline{\alpha\ \ell}) \cdot (\alpha\ \ell)}_{\mathrm{I}} \mathcal{M}', (P_0' : \psi_0 \cdot (\alpha\ \ell), P_1' : \psi_1 \cdot (\alpha\ \ell))$$

Furthermore, $\varphi' \cdot \alpha$ is a valid explanation of $\psi' \cdot (\overline{\alpha\ \ell}) \cdot (\alpha\ \ell)$; and $\varphi' \cdot \alpha = \rho(\psi_0 \cdot (\alpha\ \ell)) = \rho(\psi_1 \cdot (\alpha\ \ell))$, we conclude. □

Proof of Theorem 1 Completeness follows from Lemma 6.

We prove soundness separately for the three kinds of sets $\mathcal{C}$ of honest participants:

- When both participants are honest ($\mathcal{C} = \{0; 1\}$), we conclude by applying Lemmas 1, 2, 4, and 5.
- When exactly one of the participants is honest (assume $C = \{0\}$, the other case is symmetric), we have

  $$\emptyset, P_0 : 0 \xrightarrow{\psi}_{\mathrm{I}} \mathcal{M}, P_0' : \psi$$

  We focus on the last action of ψ, and consider three possible cases:
  - The last action of ψ is an output: there exists ψ' such that $\psi = \psi'.(\overline{1\ a\ 0\ \ell})$ (the case $\psi = \psi'.(\overline{0\ a\ 1\ \ell})$ is similar). From the definition of $\rightarrow_{\mathrm{I}}$ and $\rightarrow_0$, there exists P_1' and $\widetilde{P''}$ such that

    $$\frac{\widetilde{P} \xrightarrow{\rho(\psi')}_{\mathsf{S}} \widetilde{P''} \xrightarrow{1\ a\ 0}_{\mathsf{S}} \widetilde{P'} \quad 1\ a\ 0\ \ell \notin \psi'}{P_0' : \psi' \xrightarrow{1\ a\ 0\ \ell}_0 P_0'' : \psi'.(1\ a\ 0\ \ell)}$$

We have $\widetilde{P} \xrightarrow{\rho(\psi').(1\ a\ 0)}_{\mathsf{S}} \widetilde{P'}$ and the high level trace $\rho(\psi').(1\ a\ 0)$ is equal to $\rho(\psi)$, which is a valid explanation of ψ.

– Similarly, if the last action of ψ is an input: there exists ψ' such that $\psi = \psi'.(1\ a\ 0\ \ell)$ (the case $\psi = \psi'.(\overline{0\ a\ 1\ \ell})$ is similar). From the definition of $\rightarrow_{\mathrm{I}}$ and $\rightarrow_0$, there exists P'_1 and $\widetilde{P''}$ such that

$$\frac{\widetilde{P} \xrightarrow{\rho(\psi')}_{\mathsf{S}} \widetilde{P''} \xrightarrow{1\ a\ 0}_{\mathsf{S}} \widetilde{P'} \quad \overline{1\ a\ 0\ \ell} \notin \psi'}{P'_0 : \psi' \xrightarrow{\overline{1\ a\ 0\ \ell}} P''_0 : \psi'.(\overline{1\ a\ 0\ \ell})}$$

We have $\widetilde{P} \xrightarrow{\rho(\psi').(\overline{1\ a\ 0})}_{\mathsf{S}} \widetilde{P'}$ and the high level trace $\rho(\psi').(1\ a\ 0)$ is equal to $\rho(\psi)$, which is naturally a valid explanation of ψ.

– If $\psi = \varepsilon$, we conclude.

- When no participant is honest, the only implementation trace is ε, we conclude.

□

2.10.2 Proof of Theorem 2

We first introduce a lemma that states (and prove by induction) a property slightly stronger than completeness.

Lemma 7 *For $\mathcal{C} = \{0 \dots n\}$ and for every high level trace $\widetilde{P} \xrightarrow{\varphi}_{\mathsf{S}} \widetilde{P'}$, there exist a memory $\mathcal{M}$, an implementation trace ψ, and an history-tracking specification $\widetilde{R'}$ such that φ is a valid explanation of ψ, $\widetilde{P'}$ is $\widetilde{R'}$ after erasing of the histories and*

$$\varepsilon, [\![\widetilde{P}]\!]_0 \xrightarrow{\psi}_{\mathrm{I}} \mathcal{M}, \widetilde{R'}$$

Proof: The proof is by induction on the trace φ. If $\varphi = \varepsilon$, φ is a valid explanation of $\psi = \varepsilon$. Otherwise, there exist φ', α, and $\widetilde{P''}$ such that $\varphi = \varphi'.\alpha$ and:

$$\widetilde{P} \xrightarrow{\varphi'}_{\mathsf{S}} \widetilde{P''} \xrightarrow{\alpha}_{\mathsf{S}} \widetilde{P'}$$

We conclude by induction on φ' and definitions of $\rightarrow_i$ and $\rightarrow_{\mathrm{I}}$. □

Proof of Theorem 2 Completeness follows from Lemma 7. □

2.10.3 Proof of Theorem 3

The proof relies on auxiliary lemmas. In these lemma we assume a specification $\widetilde{P}$, its history-tracking implementation $\widetilde{R}$, and a set of honest participants $\mathcal{C}$. We first define the notion of inclusion for messages in traces or history-tracking process.

Definition 8 (History inclusion) *Let R be an history-tracking process, ψ a low-level trace, $\mathcal{M}$ a memory, H and H' two history, and M and M' two low-level messages. We use $\lhd$ for the relation of subterm:*

- *$M \lhd R$ when M is a subterm of R;*
- *$M \lhd \psi$ when M is a subterm of ψ;*

- $M \lhd \mathcal{M}$ *when* M *is a subterm of* $\mathcal{M}$*;*
- $M \lhd H$ *when* M *is a subterm of* H*;*
- $M \lhd M'$ *when* M *is a subterm of* M'*;*
- $H \lhd R$ *when* H *is a subterm of* R*;*
- $H \lhd \psi$ *when* H *is a subterm of* ψ*;*
- $H \lhd \mathcal{M}$ *when* H *is a subterm of* $\mathcal{M}$*;*
- $H \lhd H'$ *when* H *is a subterm of* H'*;*
- $H \lhd M$ *when* H *is a subterm of* M*.*

We use $\in$ *for inclusion* at top level*:*

- $M \in R$ *when* $M \in H$ *and* $H \in R$*;*
- $M \in \psi$ *and* $\overline{M} \in \psi$ *are already defined;*
- $M \in \mathcal{M}$ *when* M *is one of the message of the set* $\mathcal{M}$*;*
- $M \in H$ *when* H *is of the form* $\ldots.M.\ldots$*;*
- $M \in M'$ *when* $M \in H$ *and* $H \in M'$*;*
- $H \in R$ *when* $T : H$ *appear in* R *for some* T*;*
- $H \in \psi$ *when* $H \in M$ *and* $M \in \psi$*;*
- $H \in \mathcal{M}$ *when* $H \in M$ *and* $M \in \mathcal{M}$*;*
- $H \in H'$ *when* $H \in M$ *and* $M \in H'$*;*
- $H \in M$ *when* M *is of the form* $H\ i\ a\ j\ \ell$*.*

We say that $H\ i\ a\ j\ \ell$ *is* from i *and* addressed to j*.*

In the next lemma, we show that each message that is part of an history of one of the history tracking process appear in the trace of the history tracking execution, and reciprocally. Also, we show that each message that is a subterm of an history of one of the history tracking process appear in the trace of the history tracking execution.

Lemma 8 *Let* ψ *be a low-level trace, let* M *be a message addressed to* j*, let* $\widetilde{R'}$ *be an history-tracking specification. If* $\widetilde{R} \xrightarrow{\psi}_{\mathsf{H}} \widetilde{R'}$*, then*

- $M \in R'_j \Leftrightarrow M \in \psi$*;*
- $M \lhd R'_{j'} \Rightarrow M \in \psi$*.*

Proof: We let $M = H\ i\ a\ j\ \ell$. The proof is by induction on the length of ψ. If $\psi = \varepsilon$, $\widetilde{R'} = \widetilde{R}$, and we conclude. Otherwise, there exist ψ', M', and $\widetilde{R''}$ such that

$$\widetilde{R} \xrightarrow{\psi'}_{\mathsf{H}} \widetilde{R''} \xrightarrow{M'}_{\mathsf{H}} \widetilde{R'}$$

We consider two cases (we assume $i \neq j$, the other case is similar):

- If $M' = M$, by definition of $\rightarrow_{\mathsf{H}}$, there exist R_i''', R_j''', H' and P such that $R_i'' = \overline{a} : H|R_i'''$, $R_i' = \mathbf{0} : H|R_i'''$ $R_j'' = a.P : H'|R_i'''$, $R_j' = P : H'.M|R_j'''$, and $R_k' = R_k''$ for $k \neq i'$ and $k \neq j'$. Thus $M \in R'$ and $M \in \psi$. We conclude.

- Otherwise, $M \in \psi \Leftrightarrow M \in \psi'$. We let $M' = H''\ i''\ a''\ j''\ \ell''$ and, by definition of $\rightarrow_{\mathsf{H}}$, there exist $R_{i''}'''$, $R_{j''}'''$, H''' and P such that $R_{i''}'' = \overline{a''} : H''|R_{i''}'''$, $R_{i''}' = \mathbf{0} : H''|R_{i''}'''$, $R_{j''}' = a.P : H'''|R_{j''}'''$, $R_{j''}' = P : H'''.(M')|R_{j''}'''$, and $R_k' = R_k''$ for $k \neq i''$ and $k \neq j''$. Hence, $M \in R_j' \Leftrightarrow M \in R_j''$, we conclude our first point. For the second point:

  - If $M \lhd R_{j'}'$, we conclude by induction.
  - If $M \lhd R_{j'}''$ but $M \not\lhd R_{j'}'$, we have $M \lhd H''$ (and $j' = j''$). Hence, we have $M \lhd R_{i''}''$, and we conclude by induction.

□

In the next lemma, we show that an action can only occurs once in an implementation trace.

Lemma 9 *Let ψ be an implementation trace, let $\widetilde{R'}$ be an history-tracking specification and let $\mathcal{M}$ be a memory such that $\widetilde{R} \xrightarrow{\psi}_{\mathsf{H}} \widehat{R'}$ or $\widehat{R}, \varepsilon \xrightarrow{\psi}_{\mathsf{I}} \widehat{R'}, \mathcal{M}$. Let γ be an action, γ occurs at most once in ψ.*

Proof: By definition of $\rightarrow_{\mathsf{H}}$, and Lemma 8, if $\widetilde{R} \xrightarrow{\psi}_{\mathsf{H}} \widehat{R'}$, γ occurs at most once in ϕ. The other case is a corollary, from the definition of $\rightarrow_i$ and $\rightarrow_{\mathrm{I}}$. □

In the next lemma, we show the consistency of histories.

Lemma 10 *Let φ be an implementation trace, let $\widetilde{R'}$ be an history tracking specification such that $\widetilde{R} \xrightarrow{\varphi}_{\mathsf{H}} \widetilde{R'}$, let H_0 be an history, let M and M' be two messages. If $H_0 \lhd \widetilde{R'}$, $M.M' \lhd \widetilde{R'}$ and $M' \in H_0$ then $M.M' \in H_0$.*

Proof: Let $M = H\ i\ a\ j\ \ell$ and $M' = H'\ i'\ a'\ j'\ \ell'$. The proof is by induction on the length of φ. If $\varphi = \varepsilon$, $\widetilde{R'}$ contains no messages, and we conclude. Otherwise, there exist φ', M'', and $\widetilde{R''}$ such that

$$\widetilde{R} \xrightarrow{\varphi'}_{\mathsf{H}} \widetilde{R''} \xrightarrow{M''}_{\mathsf{H}} \widetilde{R'}$$

We consider two cases (we assume that M'' is from i and addressed to j with $i \neq j$, the other case is similar):

- If $M'' = M = H\ i\ a\ j\ \ell$, by definition of $\rightarrow_{\mathsf{H}}$, there exist R_i''', R_j''', H'' and P such that $R_i'' = \overline{a} : H|R_{i'}'''$, $R_i' = \mathbf{0} : H''|R_{i'}'''$, $R_{j'}' = a.P : H''|R_{j'}'''$, $R_j' = P : H''.M|R_{j'}'''$, and $R_k' = R_k''$ for $k \neq i$ and $k \neq j$. By Lemma 9, $M \notin \varphi'$, hence, by Lemma 8, $M \not\lhd R''$. Hence $M.M' \lhd \widetilde{R'}$ is not possible. We conclude.

- If $M'' = M' = H'\ i'\ a'\ j'\ \ell'$, by definition of $\rightarrow_{\mathsf{H}}$, there exist $R_{i'}'''$, $R_{j'}'''$, H'' and P such that $R_{i'}'' = \overline{a'} : H'|R_{i'}'''$, $R_{i'}' = \mathbf{0} : H'|R_{i'}'''$, $R_{j'}' = a'.P : H''|R_{j'}'''$, $R_{j'}' = P : H''.M'|R_{j'}'''$, and $R_k' = R_k''$ for $k \neq i'$ and $k \neq j'$. By Lemma 9, $M' \notin \varphi'$, hence, by Lemma 8, $M' \not\lhd R''$. Hence $M' \in H_0$ and $H_0 \lhd \widetilde{R'}$ implies that $H''.M'$ ends with H_0. Also, $M.M' \lhd \widetilde{R'}$ implies that $H''.M'$ ends with $M.M'$. We conclude.

- Otherwise, we conclude by induction on φ'.

□

In the next lemma, we show that, under certain conditions, we can swap messages in history-tracking traces.

Lemma 11 *Let $\widetilde{R'}$,$\widetilde{R''}$, and $\widetilde{R'''}$ be history-tracking specifications, let M and M' be two messages. If $\widetilde{R'} \xrightarrow{M}_{\mathsf{H}} \widetilde{R''} \xrightarrow{M'}_{\mathsf{H}} \widetilde{R'''}$, $M \notin H$ and $(M.M') \notin \widetilde{R'''}$, then, $\widetilde{R'} \xrightarrow{M'}_{\mathsf{H}} \xrightarrow{M}_{\mathsf{H}} \widetilde{R'''}$.*

Proof: We let $M = H\ i\ a\ j\ \ell$ and $M' = H'\ i'\ a'\ j'\ \ell'$. By definition of $\rightarrow_{\mathsf{H}}$ (assuming that $i \neq j$ and $i' \neq j'$), there exist R''_{i}, $R''_{j'}$, H° and P such that $R'_{i'} = \overline{a} : H | R''_{i'}$, $R''_{i'} = \mathbf{0} : H | R''_{i'}$, $R'_{j'} = a.P : H^\circ | R''_{j'}$, $R''_{j'} = P : H^\circ.M | R''_{j'}$, and $R'_k = R''_k$ for $k \neq i'$ and $k \neq j'$, and there exist R''_{i}, R''_{j}, $H^{\circ\circ}$ and P' such that $R''_i = \overline{a'} : H' | R''_i$, $R'''_i = \mathbf{0} : H' | R''_i$, $R''_j = a'.P' : H^{\circ\circ} | R''_j$, $R'''_j = P' : H^{\circ\circ}.(M') | R''_j$, and $R''_k = R'''_k$ for $k \neq i'$ and $k \neq j'$. $M \notin H$, hence $H' \neq H.M$ and $\overline{a'} : H'$ is not in $P : H.M$. $(M.M') \notin \widetilde{R'''}$, hence $H^{\circ\circ}.(M') \neq H^\circ.M.M'$, hence $H^{\circ\circ} \neq H^\circ.M$ and $a'.P' : H^{\circ\circ}$ is not in $P : H'.M$. The part consumed in R'' are not part of P, hence, by definition of $\rightarrow_{\mathsf{H}}$, $\widetilde{R'} \xrightarrow{H\ i\ a\ j\ \ell}_{\mathsf{H}} \xrightarrow{M}_{\mathsf{H}} \widetilde{R'''}$. □

We extend typing on messages, threads and histories; checking that those are compatible. For example, a thread i should contains, at top level of his history, only messages that are addressed to him.

Definition 9 (Implementation typing) *Let Γ be an environment, H an history, and $H\ i\ a\ j\ \ell$ and T a thread, we extend typing for message, histories, and threads:*

$$\frac{\Gamma, a : \pi \vdash H : i.(\pi \backslash i) \quad j.\pi' \leq \pi}{\Gamma, a : \pi \vdash H\ i\ a\ j\ \ell : j.\pi'}\ (\textsc{Message}) \qquad \frac{\Gamma \vdash M : \pi \quad \Gamma \vdash H : \pi}{\Gamma \vdash H.M : \pi}\ (\textsc{History}) \qquad \frac{\Gamma \vdash H : \pi \quad \Gamma \vdash_i T : \pi}{\Gamma \vdash_i (T : H) : \pi}\ (\textsc{Thread})$$

We also write $\vdash X \geq \pi$ when for all π' such that $\vdash X : \pi'$, $\pi \leq \pi'$.

In the next lemma, we show subject reduction for this extended typing.

Lemma 12 (Subject Reduction) *Let $\widetilde{R'}$ and $\widetilde{R''}$ be two history tracking specifications, let M be a message, let γ be a low level action (either M or $\overline{M}$, let $\mathcal{M}$ and $\mathcal{M}'$ be two memories. If $\Gamma \vdash \widetilde{R'}$ and $\widetilde{hp'} \xrightarrow{M}_{\mathsf{H}} \widetilde{R''}$ then $\Gamma \vdash M$ and $\Gamma \vdash \widetilde{R''}$. If $\Gamma \vdash \widetilde{R'}$, $\Gamma \vdash \mathcal{M}$ and $\mathcal{M}, \widetilde{hp'} \xrightarrow{\gamma}_{\mathrm{I}} \mathcal{M}', \widetilde{R''}$ then $\Gamma \vdash M$, $\Gamma \vdash \widetilde{R''}$, and $\Gamma \vdash \mathcal{M}'$.*

Proof: We let $M = H\ i\ a\ j\ \ell$ and $\Gamma(a) = \pi$. By definition of $\rightarrow_{\mathsf{H}}$ (assuming that $i \neq j$), there exist R''_i, R''_j, H° and P such that $R'_i = \overline{a} : H | R''_i$, $R''_i = \mathbf{0} : H | R''_i$, $R'_j = a.P : H^\circ | R''_j$, $R''_j = P : H^\circ.(M) | R''_j$, and $R'_k = R''_k$ for $k \neq i$ and $k \neq j$. By Rule Send, $\Gamma \vdash_i \overline{a} : i.(\pi \backslash i)$. Since $\Gamma \vdash R'$, we have by Rule Thread $\Gamma \vdash H : i.(\pi \backslash i)$. Also, by Rule Receive, $\Gamma \vdash_j a.P : \pi'$, hence $\Gamma \vdash H^\circ : \pi'$. $\Gamma \vdash_j a.P : \pi'$, hence $\pi' \leq \pi$ from Rule Receive. Also, $\pi' = j.\pi''$ (since all types typed by $\vdash_j$ start by j). Thus $j.\pi'' \leq \pi$ and $\Gamma \vdash M : \pi'$ with Rule Message. Finally, $\Gamma \vdash_j (P : H^\circ.(M)) : \pi'$ and we conclude.

The proof for $\rightarrow_{\mathrm{I}}$ is similar. □

In the next lemma, we show that if a message sent by an honest participant is a subterm of a message known from the memory, then this message is contained in the memory.

Lemma 13 *Let $\mathcal{M}$ be a memory, let M and M' be two messages such that M is from an honest participant and $M \lhd M'$. If $\mathcal{M} \vdash_{\mathcal{C}} M'$ then $M \lhd \mathcal{M}$.*

PROOF: The proof is by induction on the derivation of $\mathcal{M} \vdash_{\mathcal{C}} M'$. If $M' \in \mathcal{M}$ we conclude. Otherwise, it is obtained by the rule $\dfrac{(\mathcal{M} \vdash_{\mathcal{C}} M_m)^{m<k} \qquad i \notin \mathcal{C}}{\mathcal{M} \vdash_{\mathcal{C}} M_0.\cdots.M_{k-1}\ i\ a\ j\ \ell}$ with $M' = M_0.\cdots.M_{k-1}\ i\ a\ j\ \ell$. If $M_m = M$ for some $m < k$, $\mathcal{M} \vdash_{\mathcal{C}}$ implies $M \in \mathcal{M}$ since M is from an honest participant. Otherwise , $M \in M_m$ for some $m < k$, and we conclude by induction. □

In the next lemma, we show that if a message is typable, its history should contain, at top level, only messages that are addressed to its sender.

Lemma 14 *Let M and M' be two messages such that $\Gamma \vdash M'$. If $M \in M'$ and M' is from i then M is addressed to i.*

PROOF: By Rule MESSAGE and HISTORY, we need $\vdash M : i.(\pi \backslash i)$ to type M', hence M is addressed to i. □

In the next lemma, we show that if a message is received by an honest participant, is known from the memory, and contains another message, then this message is contained in the memory.

Lemma 15 *Let $\mathcal{M}$ be a typable memory, let M and M' be two typable messages such that $\mathcal{M} \vdash_{\mathcal{C}} M'$ and M is addressed to an honest participant. If $M \lhd M'$ then, there exists $M'' \in \mathcal{M}$ such that $M \lhd M''$ and $M'' \lhd M'$.*

PROOF: The proof is by induction on the derivation of $\mathcal{M} \vdash_{\mathcal{C}} M'$. If $M' \in \mathcal{M}$ we conclude. Otherwise, it is obtained by the rule $\dfrac{(\mathcal{M} \vdash_{\mathcal{C}} M_m)^{m<k} \qquad i \notin \mathcal{C}}{\mathcal{M} \vdash_{\mathcal{C}} M_0.\cdots.M_{k-1}\ i\ a\ j\ \ell}$ with $M' = M_0.\cdots.M_{k-1}\ i\ a\ j\ \ell$. If $M_m = M$ for some $m < k$, by Lemma 14 we have $i \in \mathcal{C}$ since M is addressed to an honest participant. Hence $M' \in \mathcal{M}$ (this case is already treated). Otherwise, $M \lhd M_m$ for some $m < k$, and we conclude by induction. □

In the next lemma, we show relation between contents of the implementation traces and history-tracking processes.

Lemma 16 *Let ψ be an implementation trace, let $\widetilde{R'}$ be an history tracking specification, let $\mathcal{M}$ be a memory, let M be a message from i addressed to j, and let H be an history such that $H \in M$. If $\widehat{R}, \varepsilon \xrightarrow{\psi}_{\mathrm{I}} \widehat{R'}, \mathcal{M}$, then*

1. *$M \in \psi \Leftrightarrow M \in R'_j$;*
2. *$\overline{M} \in \psi \Leftrightarrow M \in \mathcal{M}$;*
3. *$M \in \psi \Rightarrow \mathcal{M} \vdash M$;*
4. *$\overline{M} \in \psi \Rightarrow H \in R'_i$.*

PROOF: We let $M = H\ i\ a\ j\ \ell$. The proof is by induction on the length of ψ. If $\psi = \varepsilon$, $\widehat{R'} = \widehat{R}$, and we conclude. Otherwise, there exist ψ', γ, $\mathcal{M}'$, and $\widehat{R''}$ such that

$$\widehat{R}, \varepsilon \xrightarrow{\psi'}_{\mathrm{I}} \widehat{R''}, \mathcal{M}' \xrightarrow{\gamma}_{\mathrm{I}} \widehat{R'}, \mathcal{M}$$

We consider four cases:

1. If $\gamma = M$, by definition of $\rightarrow_{\mathrm{I}}$, and $\rightarrow_j$, there exist R'''_j, H' and P such that $R''_j = a.P : H'|R'''_j$, $R'_j = P : H'.M|R'''_j$, and $R'_k = R''_k$ for $k \neq j$. Also, $\mathcal{M} \vdash M$ and $\mathcal{M} = \mathcal{M}'$.

2. If $\gamma = \overline{M}$, by definition of $\rightarrow_{\mathrm{I}}$, and $\rightarrow_i$, there exist R_i''' such that $R_i'' = \overline{a} : H|R_i'''$, $R_i' = \mathbf{0} : H|R_i'''$, and $R_k' = R_k''$ for $k \neq i$. Also, $\mathcal{M}' = \mathcal{M}.M$.

3. Otherwise, if $\gamma = \overline{H'\ i'\ a'\ j'\ \ell'} = M'$, by definition of $\rightarrow_{\mathrm{I}}$, and $\rightarrow_{i'}$, there exists $R_{i'}'''$ such that $R_{i'}'' = \overline{a'} : H'|R_{i'}'''$, $R_{i'}' = \mathbf{0} : H'|R_{i'}'''$, and $R_k' = R_k''$ for $k \neq i'$. Also, $\mathcal{M} = \mathcal{M}'.M'$.

4. Otherwise $\gamma = H'\ i'\ a'\ j'\ \ell' = M'$, by definition of $\rightarrow_{\mathrm{I}}$, and $\rightarrow_{j'}$, there exist $R_{j'}'''$, H'', and P such that $R_{j'}'' = a'.P : H''|R_{j'}'''$, $R_{j'}' = P : H''.M'|R_{j'}'''$, and $R_k' = R_k''$ for $k \neq j'$. Also, $\mathcal{M} \vdash M'$ and $\mathcal{M} = \mathcal{M}'$.

We now prove the four points of the lemma depending on these cases:

1. In Case 1, $M \in R_j'$ and $M \in \psi$; in Cases 2,3,4 $M \in R_j' \Leftrightarrow M \in R_j''$ and $M \in \psi \Leftrightarrow M \in \psi'$, and we conclude by induction.

2. In Case 2, $\overline{M} \in \psi$ and $M \in \mathcal{M}$; in Cases 1,3,4 $\overline{M} \in \psi \Leftrightarrow \overline{M} \in \psi'$ and $M \in \mathcal{M} \Leftrightarrow M \in \mathcal{M}'$, and we conclude by induction.

3. In Case 1, $\mathcal{M} \vdash M$ (and $M \in \psi$); in Cases 2,3,4 $M \in \psi \Rightarrow M \in \psi'$ and $\mathcal{M}' \vdash M \Rightarrow \mathcal{M} \vdash M$, and we conclude by induction.

4. In Case 2, $H \in R_i'$ (and $\overline{M} \in \psi$); in Cases 1,3,4 $\overline{M} \in \psi \Rightarrow \overline{M} \in \psi'$ and $H \in R_i'' \Rightarrow H \in R_i'$, and we conclude by induction.

□

We define a notion on consistency on memories, that ensure that histories of messages present in the memory are formed from other messages present in the memory.

Definition 10 (Memory consistency) *Let $\mathcal{M}$ be a memory, we say that $\mathcal{M}$ is consistent if, $\vdash \mathcal{M}$ and, if, for every message $M \in \mathcal{M}$ and $M' \lhd M$ such that M' is from an honest participant, $M' \in \mathcal{M}$.*

In the next lemma, we show that memory generated by an execution are consistent.

Lemma 17 *Let ψ be an implementation trace, let $\mathcal{M}$ be a memory, let $\widetilde{R'}$ be an history-tracking specification. If $\widehat{R}, \varepsilon \xrightarrow{\psi}_{\mathrm{I}} \widehat{R'}, \mathcal{M}$, then $\mathcal{M}$ is consistent.*

Proof: The proof is by induction on the length of ψ. If $\psi = \varepsilon$, $\mathcal{M} = \emptyset$, and we conclude. Otherwise, there exist ψ', γ, $\mathcal{M}'$, and $\widehat{R''}$ such that

$$\widehat{R}, \varepsilon \xrightarrow{\psi'}_{\mathrm{I}} \widehat{R''}, \mathcal{M}' \xrightarrow{\gamma}_{\mathrm{I}} \widehat{R'}, \mathcal{M}$$

We consider two cases:

- If $\gamma = M$, $\mathcal{M} = \mathcal{M}'$, we conclude by induction on $\mathcal{M}'$.
- If $\gamma = \overline{H\ i\ a\ j\ \ell}$, by definition of $\rightarrow_{\mathrm{I}}$, and $\rightarrow_i$, $\mathcal{M} = \mathcal{M}.H\ i\ a\ j\ \ell$ and for every message $M' \in H$, we have $M' \in R''$. Hence, by Lemma 16, $M' \in \psi'$ and by Lemma 12, $\mathcal{M}' \vdash M'$. If M' is from an honest participant, by definition $M' \in \mathcal{M}'$. Otherwise, by Lemma 15, for every message $M'' \in H'$ such that M'' is from an honest participant, there exists $M''' \in \mathcal{M}'$ such that $M'' \lhd M'''$, and we conclude by induction on $\mathcal{M}'$.

□

In the next lemma, we show that if a message is sent by an honest participant and contains another message that is known from a consistent memory, then the first message is present in the memory.

Lemma 18 *Let $\mathcal{M}$ be a consistent memory, let M and M' be two messages such that M is from an honest participant and $M \lhd M'$. If $\mathcal{M} \vdash_{\mathcal{C}} M'$ then $M \in \mathcal{M}$.*

PROOF: The proof is direct from the definition of consistent memory and Lemma 13. □

In the next lemma, we show that the order of messages in implementation traces and histories is consistent.

Lemma 19 *Let ψ be an implementation trace, $\mathcal{M}$ a memory, $\widetilde{R'}$ an history-tracking specification such that $\widehat{R}, \varepsilon \xrightarrow{\psi}_{\mathrm{I}} \widehat{R'}, \mathcal{M}$. Let M be a message such that $M \lhd R'$.*

- *If M is from an honest participant, then $\overline{M} \in \psi$.*
- *If M is addressed to an honest participant, then $M \in \psi$.*

PROOF: $M \lhd R'$, hence, there exists $M' \in R'_i$ such that $M \lhd M'$ for some i we consider the smaller of such M'. By Lemma 16, $M' \in \psi$ and $\mathcal{M} \vdash M'$. Also, by Lemma 17, $\mathcal{M}$ is consistent.

- Suppose that M is from an honest participant. By Lemma 18, $M \in \mathcal{M}$. Then, by Lemma 16, $\overline{M} \in \psi$.
- Suppose that M is addressed to an honest participant. If $M = M'$, we conclude. Otherwise, by Lemma 15, there exists a message M'' such that $M'' \in \mathcal{M}$ such that $M \lhd M''$ and $M'' \lhd M'$. By definition of $\lhd$, and since $M \neq M'$, there exist $H \in M''$ such that $M \lhd H$. Hence, by Lemma 16, $H \in R$. Since M' was the smaller message in R' containing M, it is not possible.

□

In the next lemma, we show an other property on the order of messages in an implementation trace.

Lemma 20 *Let ψ be an implementation trace, $\mathcal{M}$ a memory, $\widetilde{R'}$ an history-tracking specification such that $\widehat{R}, \varepsilon \xrightarrow{\psi}_{\mathrm{I}} \widehat{R'}, \mathcal{M}$. Let M and M' be two messages such that $M \lhd M'$ or $M.M' \in R'$. Let $\gamma = \overline{M}$ or $\gamma = M$. If $\gamma \in \psi$ and $M' \in \psi$, M' appear after γ.*

PROOF: $M' \in \psi$, hence there exist an implementation trace ψ', memories $\mathcal{M}'$ and $\mathcal{M}''$, and history-tracking specifications $\widetilde{R''}$ and $\widetilde{R'''}$ such that $\widehat{R}, \varepsilon \xrightarrow{\psi'}_{\mathrm{I}} \widehat{R''}, \mathcal{M}' \xrightarrow{M'}_{\mathrm{I}} \widehat{R'''}, \mathcal{M}''$. By Lemma 16 and hypothesis, $M \lhd R'''$, we conclude with Lemma 19 and Lemma 9. □

We define a partition on sequence of messages. Each part of this partition contains the messages that are eventually received by a process i after a sequence of adversary messages. These are defined on source and implementation traces.

Definition 11 (Trace projection) *Let ψ be a low-level trace. ψ can be a trace of actions (send and receive), e.g., an implementation trace or ψ can be a trace of communications, e.g., a trace on $\rightarrow_{\mathsf{H}}$. We define projection on such trace. $\phi_i^{in}(\psi)$ is the list of the messages received by the process*

i (keeping their order). $\phi_i^{out}(\psi)$ is the list of honest messages that appear in dishonest message received by the process i(in their order). $\phi_i(\psi)$ is their union (keeping the order of the actions.

$$
\begin{array}{lll}
\phi_i^{in}(\psi) = & \{\gamma \in \psi \mid \gamma \text{ is addressed to } i\} & \\
\phi_i^{out}(\psi) = & \{\gamma \mid & \gamma \in \psi \text{ with } \gamma = \overline{M} \text{ or } \gamma = M, \\
 & & \gamma \text{ is addressed to a dishonest participant}, \\
 & & \exists M_1, \dots M_k \mid \quad M \in M_1 \\
 & & \qquad\qquad M_j \in M_{j+1}, j < k \\
 & & \qquad\qquad M_j \text{ is addressed to a dishonest participant}, j < k \\
 & & \qquad\qquad M_k \text{ is addressed to } i \\
 & & \qquad\qquad M_k \in \psi
\end{array}
$$

We also let $\phi_i(\psi)$ be the union of these two projection, and $\phi_i^{all}(\psi)$ be the union of these two projections, plus the intermediate adversary messages.

In the next lemma, we show that messages in $\phi_i^{out}(\psi)$ have a particular type.

Lemma 21 *Let ψ be a typed implementation trace, let M be a message. If $M \in \phi_i^{out}(\psi)$, then $\vdash M : \ominus.i$ (we let $\ominus$ be any sequence of dishonest indexes).*

Proof: By definition of $\phi_i(\psi)$, there exists $M_1, \dots M_k$, $M_j \in M_{j+1}$ and M_j is addressed to a dishonest participant for $j < k$, M_k is addressed to i. $\vdash \psi$ and $M_k \in \psi$, hence, using Rules Message and History, $\vdash M_k : i$, then, for every $j < k$, we have $\vdash M_j : \ominus.i$, finally $\vdash M : \ominus.i$. □

In the next lemma, we show that if a process send a message, then it contains a thread with the corresponding history that can emit the corresponding action, with the corresponding history.

Lemma 22 *Let ψ be an implementation trace, let R' and R'' be two history-tracking specifications, M and M' be two messages such that M' is a message on action a, $M \notin \psi$, and $M \in M'$. If $R' \xrightarrow{\psi}_{\mathsf{h}} \xrightarrow{M}_{\mathsf{h}} R''$, there exist T, H_0, and R''' such that $R' = T : H_0 | R'''$, $= M \in H_0$, and $\vdash T \geq \Gamma(a)$.*

Proof: We let $M' = H\ i\ a\ j\ \ell$. The proof is by induction on the number of actions in ψ. If $\psi = \varepsilon$, by definition of $\rightarrow_{\mathsf{h}}$, there exists R° such that $R' = \overline{a} : H | R^\circ$. We have $\vdash \overline{a} \geq \Gamma(a)$: we conclude. Otherwise, there exist ψ', γ and R''' such that $R' \xrightarrow{\psi'}_{\mathsf{h}} R''' \xrightarrow{\gamma}_{\mathsf{h}} \xrightarrow{\overline{H\ i\ a\ j\ \ell}}_{\mathsf{h}} R''$. There are two cases:

- If $\gamma = \overline{H'\ i'\ a'\ j'\ \ell'}$, by definition of $\rightarrow_{\mathsf{h}}$, there exists R° such that $R' = \overline{a'} : H' | R^\circ$ and $R''' = \mathbf{0} : H | R^\circ$. We conclude by induction.
- If $\gamma = H'\ i'\ a'\ j'\ \ell'$, by definition of $\rightarrow_{\mathsf{h}}$, there exist R°, H'' and P such that $R' = a'.P : H'' | R^\circ$ and $R''' = P : H''.(H'\ i'\ a'\ j'\ \ell') | R^\circ$. By induction, there exist T, H''', and $R^{\circ\circ}$ such that $R''' = T : H''' | R^{\circ\circ}$, $M \in H'''$, and $\vdash T \geq \Gamma(a)$. If T is part of $R^{\circ\circ}$, we conclude. If T is part of P, and since $H'\ i'\ a'\ j'\ \ell' \neq M$, $M \in H''$. Also, $\vdash P \geq \Gamma(a)$, hence $\vdash a'.P \geq \Gamma(a)$.

□

We are interested in low-level traces that follow a particular order, which makes them easier to relate to high-level traces.

Definition 12 *We say that two low-level trace Ω and ψ are* compatible *for i when*

- $\phi_i^{in}(\Omega) = \phi_i^{in}(\psi)$;

- $M \in \phi_i^{out}(\Omega) \Leftrightarrow \overline{M} \in \phi_i^{out}(\psi)$;
- *for every* $M \in \phi_i^{out}(\psi)$ *and* $M' \in \phi_i^{in}(\psi)$ *such that* M *appear after* M' *in* ψ, M *appear after* M' *in* Ω.

We say that two low-level trace Ω *and* ψ *are* opposite compatible *for* i *when*

- $\overline{\phi_i^{in}(\Omega)} = \phi_i^{in}(\psi)$;
- $M \in \phi_i^{out}(\Omega) \Leftrightarrow \overline{M} \in \phi_i^{out}(\psi)$;
- *for every* $M \in \phi_i^{out}(\psi)$ *and* $M' \in \phi_i^{in}(\psi)$ *such that* M *appear after* M' *in* ψ, M *appear after* M' *in* Ω.

In this lemma, we show that an implementation trace that share a particular set of actions with another implementation trace can be reordered so that their order on these actions are the same.

Lemma 23 *Let* ψ *and* Ω *be two low-level traces, let* $\widetilde{R'}$ *and* $\widetilde{R''}$ *be two history-tracking specification and let* $\mathcal{M}$ *be a memory such that*

- $\widehat{R}, \varepsilon \xrightarrow{\psi}_{\mathrm{I}} \widehat{R'}, \mathcal{M}$;
- $\widetilde{R} \xrightarrow{\Omega}_{\mathsf{H}} \widetilde{R''}$;
- $R'_i = R''_i$;
- $M \in \phi_i^{in}(\Omega) \Leftrightarrow M \in \phi_i^{in}(\psi)$.
- $M \in \phi_i^{out}(\Omega) \Leftrightarrow \overline{M} \in \phi_i^{out}(\psi)$.

There exists a low-level trace Ω *compatible with* ψ *for* i *and such that* $\widetilde{R} \xrightarrow{\Omega'}_{\mathsf{H}} \widetilde{R''}$

Proof: We define an algorithm sorting the trace: We consider the first action in $\phi_i^{in}(\psi)$ that is not sorted in Ω (M). We say that a message M depends on M' if $M' \lhd M$, if $M'.M \lhd \widetilde{R''}$, or if there exists M'' such that M depends on M'' and M'' depends on M. We consider M plus all the messages that it depends on. One by one, we swap all these messages with the messages preceding them in Ω, in order to place them before the corresponding actions following M in $\phi_i^{in}(\psi)$. From Lemma 21, and the typing rules, we verify that, since $R'_i = R''_i$, if M depends on $M''' \in \phi_i^{out}(\Omega)$, then M depends on $M''' \in \psi$. From Lemma 20, we know that none of the messages after M in $\phi_i^{in}(\psi)$ depend on him. So we can always swap one of these actions until M is sorted. After each step, the trace respect the hypothesis of the lemma (by induction hypothesis and Lemma 10 and 11). This terminates since we sort each action of $\phi_i(\Omega)$ one by one, and each sorting take a finite time since the group of actions only goes backward.

□

We extend $\rightarrow_{\mathsf{h}}$ on tuple of processes.

Definition 13 (Intermediate semantics) *Let* $\widehat{R'}$ *and* $\widehat{R''}$ *be two global history specifications, let* γ *be and action. We say that* $\widehat{R'} \xrightarrow{\gamma}_{\mathsf{h}} \widehat{R''}$ *when there exists* i *such that* $R'_i \xrightarrow{\gamma}_{\mathsf{h}} R''_i$ *and* $R''_j = R'_j$ *for* $j \neq i$.

In the next lemma, we show that, under certain conditions, we can swap messages in traces of $\rightarrow_{\mathsf{h}}$.

Lemma 24 *Let $\widehat{R'}$ and $\widehat{R''}$ be history-tracking specifications, let M and M' be two messages such that $i, j, i', j' \in \mathcal{C}$.*

1. *If $\widehat{R'} \xrightarrow{\overline{M}}_{\mathsf{h}} \xrightarrow{\overline{M'}}_{\mathsf{h}} \widehat{R''}$ then $\widehat{R'} \xrightarrow{\overline{M'}}_{\mathsf{h}} \xrightarrow{\overline{M}}_{\mathsf{h}} \widehat{R''}$ and a valid explanation of a trace $\psi_1.\overline{M'}.\overline{M}.\psi_2$ is also a valid explanation of the trace $\psi_1.\overline{M}.\overline{M'}.\psi_2$.*

2. *If $\widehat{R'} \xrightarrow{\overline{M}}_{\mathsf{h}} \xrightarrow{M'}_{\mathsf{h}} \widehat{R''}$ then $\widehat{R'} \xrightarrow{M'}_{\mathsf{h}} \xrightarrow{\overline{M}}_{\mathsf{h}} \widehat{R''}$ and a valid explanation of a trace $\psi_1.M'.\overline{M}.\psi_2$ is also a valid explanation of the trace $\psi_1.\overline{M}.M'.\psi_2$.*

Proof:

1. We let $M = H\ i\ a\ j\ \ell$ and $M' = H'\ i'\ a'\ j'\ \ell'$. By definition of $\rightarrow_{\mathsf{h}}$, there exist $\widehat{R'''}$ such that $\widehat{R'} \xrightarrow{\overline{M}}_{\mathsf{h}} \widehat{R'''} \xrightarrow{\overline{M'}}_{\mathsf{h}} \widehat{R''}$. By definition of $\rightarrow_{\mathsf{h}}$, there exists R_i°, such that $R'_i = \overline{a} : H|R_i^\circ$, $R'''_i = \mathbf{0} : H|R_i^\circ$, and $R'''_k = R'_k$ for $k \neq i$, and there exists $R_{i'}^{\circ\circ}$ such that $R'''_{i'} = \overline{a'} : H'|R_{i'}^{\circ\circ}$, $R''_{i'} = \mathbf{0} : H'|R_{i'}^{\circ\circ}$, and $R''_k = R'''_k$ for $k \neq i'$. The part consumed in $\widehat{R'''}$ is not modified by the first action, we conclude. Following the definition of valid explanations, we can check that a valid explanation of a trace $\psi_1.\overline{M'}.\overline{M}.\psi_2$ is also a valid explanation of the trace $\psi_1.\overline{M}.\overline{M'}.\psi_2$.

2. Similarly, by definition of $\rightarrow_{\mathsf{h}}$ there exist $\widehat{R'''}$ such that $\widehat{R'} \xrightarrow{\overline{M}}_{\mathsf{h}} \widehat{R'''} \xrightarrow{\overline{M'}}_{\mathsf{h}} \widehat{R''}$. By definition of $\rightarrow_{\mathsf{h}}$, there exists $R_{i'}^\circ$ such that $R'_{i'} = \overline{a} : H|R_{i'}^\circ$, $R'''_{i'} = \mathbf{0} : H|R_{i'}^\circ$, and $R'''_k = R'_k$ for $k \neq i'$, R_j°, H° and P such that $R'''_j = a'.P : H^\circ|R_j^\circ$, $R''_j = P : H^\circ.M|R_j^\circ$, and $R'_k = R''_k$ for $k \neq j$, The part consumed in R''' is not modified by the first action, we conclude. Following the definition of valid explanations, we can check that a valid explanation of a trace $\psi_1.M'.\overline{M}.\psi_2$ is also a valid explanation of the trace $\psi_1.\overline{M}.M'.\psi_2$.

□

In this lemma, we show that an implementation trace can be reordered so that is honest outputs precede the corresponding inputs.

Lemma 25 *Let ψ be a low-level traces, let $\widehat{R'}$ be an history-tracking specification, let $\mathcal{M}$ be a memory such that $\widehat{R}, \varepsilon \xrightarrow{\psi}_{\mathrm{I}} \widehat{R'}, \mathcal{M}$; Then, there exists a low level trace ψ' such that*

- *$\widehat{R} \xrightarrow{\psi'}_{\mathsf{h}} \widehat{R'}$;*
- *every valid explanation of ψ' is a valid explanation of ψ;*
- *every input in ψ' from and addressed to an honest participant is immediately preceded by its unique corresponding output;*
- *the order of actions that are not from and addressed to an honest participant is left unchanged.*

Proof: We define an algorithm sorting the trace: We consider the last output in ψ which is not immediately followed by its corresponding input. We swap the output messages forward until it is well-placed. From Lemma 20, we know that it is before any message containing it in ψ. So we can always swap the output until it is sorted. After each step, the trace respect the hypothesis of the lemma (by induction hypothesis and Lemma 24). This terminates we sort actions one by one.

□

Definition 14 *We say that a low-level trace ψ is* well-formed *when every input in ψ with a dishonest sender is immediately preceded by its unique corresponding output.*

In the next lemma, we show that, since messages with a parallel type can only be received once, it is possible to force the different alternatives of this reception to agree on the message received.

Lemma 26 *Let φ and φ' be two well-formed implementation traces, let M be a message on action a, let $\widetilde{R'}$, $\widetilde{R''}$, and $\widetilde{R^a}$ be three history-tracking specification such that:*

- $\widetilde{R} \xrightarrow{\varphi}_{\mathsf{h}} \widetilde{R'} \xrightarrow{\varphi'}_{\mathsf{h}} \widetilde{R''}$;
- *there exist only one a in $\widetilde{R}$;*
- – $\widetilde{R'} \xrightarrow{M}_{\mathsf{h}} \widetilde{R^a}$ *and M is from an honest participant; or*
  – $\widetilde{R'} \xrightarrow{\overline{M}}_{\mathsf{h}} \xrightarrow{M}_{\mathsf{h}} \widetilde{R^a}$.

We define two functions on messages (by extension on low-level traces) and history-tracking specifications:

$$\begin{array}{lll} f(H'\ i'\ a'\ j'\ \ell') & = \varepsilon & \text{if } a' = a \\ & = g(H')\ i'\ a'\ j'\ \ell & \text{otherwise} \\ g(H'\ i'\ a'\ j'\ \ell') & = M & \text{if } a' = a \\ & = g(H')\ i'\ a'\ j'\ \ell & \text{otherwise} \\ h(\overline{a} : H') & = h(\mathbf{0} : H') & \\ h(a.P : H') & = h(P : H'.M) & \\ h(T : H') & = T : g(H') & \text{otherwise} \end{array}$$

There exists $\widetilde{R'''}$ such that $h(\widetilde{R'''}) = h(\widetilde{R''})$ and

- $\widetilde{R'} \xrightarrow{M}_{\mathsf{h}} \widetilde{R^a} \xrightarrow{f(\varphi')}_{\mathsf{h}} \widetilde{R'''}$ *and M is from an honest participant; or*
- $\widetilde{R'} \xrightarrow{\overline{M}}_{\mathsf{h}} \xrightarrow{M}_{\mathsf{h}} \widetilde{R^a} \xrightarrow{f(\varphi')}_{\mathsf{h}} \widetilde{R'''}$.

Proof: We treat the case where M is not from an honest participant, the other case is similar. We proceed by induction on the length of φ'. If $\varphi' = \varepsilon$, by definition of $\rightarrow_{\mathsf{h}}$ (assuming that $i \neq j$), there exist R_i°, R_j°, H° and P such that $R'_i = \overline{a} : H | R_i^\circ$, $R_i^a = \mathbf{0} : H | R_i^\circ$, $R'_j = a.P : H^\circ | R_j^\circ$, $R_j^a = P : H^\circ.(H\ i\ a\ j\ \ell) | R_j^\circ$, and $R'_k = R_k^a$ for $k \neq i$ and $k \neq j$. We have: $h(R'_k) = h(R_k^a)$ for $k \neq i$ and $k \neq j$;

$$\begin{array}{ll} h(R'_i) & = h(\overline{a} : H) | h(R_i^\circ) \\ & = h(\mathbf{0} : H) | h(R_i^\circ) \\ & = h(R_i^a) \\ h(R'_j) & = h(a.P : H^\circ) | h(R_j^\circ) \\ & = h(P : H^\circ.H\ i\ a\ j\ \ell) | h(R_j^\circ) \\ & = h(R_j^a) \end{array}$$

and $\varepsilon = f(\varepsilon)$:we conclude. Otherwise, let γ be the last action of φ', we assume that $\gamma = \overline{H'\ i'\ a'\ j'\ \ell'}$ (we let $H'\ i'\ a'\ j'\ \ell' = M'$), with $i \notin \mathcal{C}$ (the other cases are similar). Hence, the action before γ in φ' is M'. Hence, there exist φ'', γ and $\widetilde{R''''}$ such that $\widetilde{hp} \xrightarrow{\varphi}_{\mathsf{h}} \widetilde{R'} \xrightarrow{\varphi''}_{\mathsf{h}} \widetilde{R''''} \xrightarrow{\overline{M'}}_{\mathsf{h}} \xrightarrow{M'}_{\mathsf{h}} \widetilde{R''}$. By induction hypothesis, there exists $\widetilde{R'''}$ such that $h(\widetilde{R'''}) = h(\widetilde{R''''})$ and $\widetilde{hp} \xrightarrow{\varphi}_{\mathsf{h}} \widetilde{R'} \xrightarrow{M}_{\mathsf{h}} \xrightarrow{f(\varphi'')}_{\mathsf{h}} \widetilde{R'''}$. By definition of $\rightarrow_{\mathsf{h}}$ (assuming that $i' \neq j'$), there exist R_i°, R_j°, H° and P such that $R''''_i = \overline{a'} : H' | R_i^\circ$, $R''_i = \mathbf{0} : H' | R_i^\circ$, $R''''_j = a'.P : H^\circ | R_j^\circ$, $R''_j = P : H^\circ.M' | R_j^\circ$, and $R''''_k = R''_k$ for $k \neq i$ and $k \neq j$.

- If $a = a'$, we have $h(R''_k) = h(R''''_k) = h(R'''_k)$ for $k \neq i$ and $k \neq j$;

$$\begin{array}{rl} h(R'''_i) & = h(R''''_i) \\ & = h(\overline{a} : H')|h(R^\circ_i) \\ & = h(\mathbf{0} : H')|h(R^\circ_i) \\ & = h(R''_i) \\ h(R'''_j) & = h(R''''_j) \\ & = h(a.P : H^\circ)|h(R^\circ_j) \\ & = h(P : H^\circ.M)|h(R^\circ_j) \\ & = h(P : H^\circ.M')|h(R^\circ_j) \\ & = h(R''_j) \end{array}$$

  and $f(\varphi'') = f(\varphi'.H'\ i'\ a\ j'\ \ell)$: we conclude.

- Otherwise, we know that $h(R'''_i) = h(\overline{a'} : H'|R^\circ_i)$, hence, either there exist P, H'' and $R^{\circ\circ}_i$ such that $R'''_i = a.P : H''|R^{\circ\circ}_i$ with $\overline{a'} \in P$ or there exist $R^{\circ\circ}_i$ and H'' such that $R'''_i = \overline{a'} : H''|R^{\circ\circ}_i$, $g(H'') = g(H')$, and $h(R^{\circ\circ}_i) = h(R^\circ_i)$. There is only one a in $\widetilde{R}$, there is one reception on a before $\widetilde{R'''}$, hence, there is no a in $\widetilde{R'''}$, the first case is not possible. Similarly, $h(R'''_j) = h(a'.P : H^\circ|R^\circ_j)$,hence, there exist P', $R^{\circ\circ}_j$, and $H^{\circ\circ}$ such that $R'''_j = a'.P' : H^{\circ\circ}|R^{\circ\circ}_j$, $h(P') = h(P)$, $g(H^{\circ\circ}) = g(H^\circ)$, and $h(R^{\circ\circ}_j) = h(R^\circ_j)$. By definition of $\rightarrow_{\mathsf{h}}$, we have $\widetilde{R'''} \xrightarrow{H''\ i'\ a'\ j'\ \ell}_{\mathsf{h}} \xrightarrow{H''\ i'\ a'\ j'\ \ell}_{\mathsf{h}} \widetilde{R''''}$ with $R''''_i = \mathbf{0} : H''|R^{\circ\circ}_i$, $R''''_j = P : H^{\circ\circ}.M|R^{\circ\circ}_j$, and $R''''_k = R'''_k$ for $k \neq i$ and $k \neq j$. By Lemma 8, if $H'''\ i''\ a\ j''\ \ell \triangleleft H''$, we have $H'''\ i''\ a\ j''\ \ell \in \varphi.H\ i\ a\ j\ \ell.f(\varphi')$. There is only one a in $\widetilde{R}$, hence $H'''\ i''\ a\ j''\ \ell = H\ i\ a\ j\ \ell$, hence $g(H'') = H''$. Thus $H''\ i'\ a'\ j'\ \ell = f(H'\ i'\ a'\ j'\ \ell')$. We have $h(R''_k) = h(R''''_k) = h(R'''_k) = h(R''''_k)$ for $k \neq i$ and $k \neq j$;

$$\begin{array}{rl} h(R''_i) & = h(\mathbf{0} : H'|R^\circ_i) \\ & = h(\mathbf{0} : H''|R^{\circ\circ}_i) \\ & = h(R''''_i) \\ h(R''_j) & = h(P : H^\circ.M'|R^\circ_j) \\ & = h(P : H^{\circ\circ}.M|R^{\circ\circ}_j) \\ & = h(R''''_j) \end{array}$$

  and $f(\varphi'').f(M') = f(\varphi'.M')$: we conclude.

□

Proof of Theorem 3 The soundness theorem states that for all execution $\emptyset, \widehat{R} \xrightarrow{\psi_0}_{\mathrm{I}} \mathcal{M}, \widehat{R^0}$, there is a valid explanation φ of ψ_0 such that $\widetilde{P} \xrightarrow{\varphi}_{\mathsf{S}} \widetilde{P'}$.

2.10.4 Building a local trace

We now focus on one process i chosen among the compliant ones, and show that

(Ω-property) if $\emptyset, \widehat{R} \xrightarrow{\psi_0}_{\mathrm{I}} \mathcal{M}, \widehat{R^0}$, there is a trace Ω_i compatible with ψ for i and such that $\widetilde{R} \xrightarrow{\Omega_i}_{\mathsf{H}} \widetilde{R''}$.

Let γ be the last message of this process in ψ_0. By definition of $\rightarrow_{\mathrm{I}}$ and $\rightarrow_i$, there exist an high-level trace ω, and two history-tracking process $\widetilde{R'}$ and $\widetilde{R''}$ such that $\widetilde{R} \xrightarrow{\omega}_{\mathsf{H}} \widetilde{R'} \xrightarrow{M}_{\mathsf{H}} \widetilde{R''}$, with $R''_i = R^0_i$. We let $\Omega_i = \omega.M$.

Using Lemma 8 and 16, as $R''_i = R^0_i$, the actions received by i in Ω and in ψ_0 are the same: $M \in \phi^{in}_i(\Omega) \Leftrightarrow M \in \phi^{in}_i(\psi)$ For the same reasons, plus Lemma 17 and 18: $M \in \phi^{out}_i(\psi) \Leftrightarrow M \in \phi^{out}_i(\Omega)$.

We conclude by applying Lemma 23.

2.10.5 A correct explanation of ψ_0

We use the Ω_is from (Ω-property) to build a correct explanation of ψ_0 by induction. We keep an invariant on growing sub-parts of the implementation trace, the valid explanation and traces extracted from the Ω_is.

Let Ψ, ψ, θ, $(\Upsilon_k)_{k\in\mathcal{C}}$, and $(\omega_k)_{k\in\mathcal{C}}$ be low-level well-formed traces; let $\widetilde{R'}$, $(\widetilde{R^k})_{k\in\mathcal{C}}$, and $(\widetilde{R'}^k)_{k\in\mathcal{C}}$ be history-tracking processes. Our invariant states that:

$$\widehat{R} \xrightarrow{\Psi}_{\mathsf{h}} \widehat{R^0}$$
$$(\widetilde{R} \xrightarrow{\Upsilon_k}_{\mathsf{h}} \widetilde{R'}^k)_{k\in\mathcal{C}}$$
$$\widehat{R} \xrightarrow{\psi}_{\mathsf{h}} \widehat{R'}$$
$$\widetilde{R} \xrightarrow{\Theta}_{\mathsf{H}} \widetilde{R'}$$
$$(\widetilde{R} \xrightarrow{\omega_k}_{\mathsf{h}} \widetilde{R}^k)_{k\in\mathcal{C}}$$

- ψ is a prefix of Ψ;
- for all k, ω_k is a prefix of Υ_k;
- every valid explanation of Ψ is a valid explanation of ψ_0;
- every input in Ψ from and addressed to an honest participant is immediately preceded by its unique corresponding output;
- Θ is a valid explanation of ψ;
- for all k, Υ_k is opposite compatible with Ψ for k;
- for all k, ω_k is opposite compatible with ψ for k;
- for all k, $\widetilde{R}^k$ coincide with $\widetilde{R'}$ on its restriction to threads T such that $\vdash T \geq \ominus.k$, and such that :
  - it is of the form $a.P : H$ and there exists a message $M \lhd \phi_i(\Psi)$ on a;
  - it is of the form $\overline{a} : H$.

The initial Ψ is obtained by Lemma 25 on ψ_0. The initial Υ_k are obtained by flattening of the Ω_k. The initial ψ and ω_k are empty.

We build an algorithm that makes ψ, ϕ and ω_i grow while conserving the invariants. By Lemma 12, all the traces and messages here type. We assume that the next action in Ψ is $\gamma = M$ or $\gamma = \overline{M}$.

If there exist no $k \in \mathcal{C}$ such that $M \in \phi_k^{in}(\Psi)$ or $\overline{M} \in \phi_k^{out}(\Psi)$, the invariant is valid with the new trace $\psi' = \psi.\gamma$. Otherwise, we let $S = \{k \in \mathcal{C} \mid M \in \phi_k(\Psi)\}$. We assume that, for every $k \in S$, the next action in Υ_k is γ_k:

$$\widehat{R} \xrightarrow{\psi}_{\mathsf{h}} \widehat{R'} \xrightarrow{\gamma}_{\mathsf{h}} \widehat{R''}$$
$$\widetilde{R} \xrightarrow{\theta}_{\mathsf{H}} \widetilde{R'}$$
$$(\widetilde{R} \xrightarrow{\omega_k}_{\mathsf{h}} \widetilde{R}^k \xrightarrow{\gamma_k}_{\mathsf{H}} \widetilde{R''}^k)_{k\in S}$$
$$(\widetilde{R} \xrightarrow{\omega_k}_{\mathsf{h}} \widetilde{R}^k)_{k\notin S}$$

- If $\gamma_k \notin \phi_k^{all}(\Psi)$, γ_k is a send from an honest participant, or γ_k is a reception addressed to an honest participant for some k, the invariant is valid with the new trace $\omega'_k = \omega_k.\gamma_k$. Otherwise, we let $\gamma_k = \overline{M_K} = \overline{H_k\ i_k\ a_k\ j_k\ \ell_k}$ or $\gamma_k = M_K = H_k\ i_k\ a_k\ j_k\ \ell_k$ and $M = H\ i\ a\ j\ \ell$ and we continue.

- Suppose that there exist $k_0 \in S$ such that $(i_{k_0}, j_{k_0}) \notin \mathcal{C}$. Since Υ is well-formed, $\gamma_k = \overline{M_k}$ and the action after is M_k. By definition of $\rightarrow_{\mathsf{h}}$ (assuming that $i_{k_0} \neq j_{k_0}$), there exist $R^\circ_{i_{k_0}}$, $R^\circ_{j_{k_0}}$, H° and P such that $R^{k_0}_{i_{k_0}} = \overline{a_{k_0}} : H_{k_0}|R^\circ_{i_{k_0}}$, $R'^{k_0}_{i_{k_0}} = \mathbf{0} : H_{k_0}|R^\circ_{i_{k_0}}$, $R^{k_0}_{j_{k_0}} = a_{k_0}.P : H^\circ|R^\circ_{j_{k_0}}$, $R''_{j_{k_0}} = P : H^\circ.M_{k_0}|R^\circ_{j_{k_0}}$. Let $S' = \{k \in \mathcal{C} \mid M_{k_0} \in \phi_k(\Psi)\}$. For every $k \in S'$, we have $\vdash M_{k_0} : \ominus.k$ by Lemma 21, hence $\vdash \overline{a_{k_0}} \geq \ominus.k$.

  - If S' is the singleton $\{k_0\}$, then, by Lemma 22, $\vdash a_{k_0}.P \geq \ominus.k$. Also, $\vdash \overline{a_{k_0}} \geq \ominus.k$. Hence, by hypothesis, $\widetilde{R}^{k_0}$ coincide with $\widetilde{R'}$ on these threads. Hence, $\widetilde{R'} \xrightarrow{M_{k_0}}_{\mathsf{H}} \widetilde{R''}$ for some $\widetilde{R''}$ that coincide on the threads of the invariants.
  - Otherwise, a_{k_0} is a parallel action. Hence, it appear only once. By Lemma 22, for every $k \in S'$, $\vdash a_{k_0}.P \geq \ominus.k$. Also, $\vdash \overline{a_{k_0}} \geq \ominus.k$. Hence, by hypothesis, $\widetilde{R}^{k_0}$ coincide with $\widetilde{R'}$ on these threads, and, for all $k \in S'$, $\widetilde{R}^k$ coincide with $\widetilde{R'}$ on these threads. Hence, $\widetilde{R'} \xrightarrow{M_{k_0}}_{\mathsf{H}} \widetilde{R''}$ and $\widetilde{R}^k \xrightarrow{\overline{M_{k_0}}}_{\mathsf{h}} \xrightarrow{M_{k_0}}_{\mathsf{h}} \widetilde{R^{ka}}$ for some $\widetilde{R^{ka}}$, and $\widetilde{R''}$ that coincide on the threads of the invariants. We apply Lemma 26 and get, for every $k \in S'$, Υ'_k respecting the invariant and such that their next action corresponds to $\widetilde{R}^k \xrightarrow{\overline{M_{k_0}}}_{\mathsf{h}} \xrightarrow{M_{k_0}}_{\mathsf{h}} \widetilde{R^{ka}}$.

  We conclude with $\psi' = \psi.M_{k_0}$ and $\omega'_k = \omega_k.\overline{M_{k_0}}.M_{k_0}$ for every $k \in S'$.

- Suppose that $i \in \mathcal{C}$ and $j \notin \mathcal{C}$, then $\gamma = \overline{M}$. If there exist no $k_0 \in S$ such that $M = \gamma_{k_0}$, we swap γ with the next action in Ψ (we may have to swap more than one action at a time to ensure progress). Otherwise, By definition of $\rightarrow_{\mathsf{h}}$ and $\rightarrow_{\mathrm{I}}$ (assuming that $i \neq j$), there exist R°_i, R°_j, H° and P such that $R_i = \overline{a_k} : H_k|R^\circ_i$, $R'_i = \mathbf{0} : H|R^\circ_i$, R°_j, H° and P such that $R^k_j = a_{k_0}.P : H^\circ|R^\circ_j$, $R''_j = P : H^\circ.M_{k_0}|R^\circ_j$. Let $S' = \{k \in \mathcal{C} \mid M \in \phi_k(\Psi)\}$. For every $k \in S'$, we have $\vdash M : \ominus.k$ by Lemma 21.

  - If S' is the singleton $\{k_0\}$, then, by Lemma 22, $\vdash a_{k_0}.P \geq \ominus.k$. Hence, by hypothesis, $\widetilde{R}^{k_0}$ coincide with $\widetilde{R'}$ on these threads. Hence, $\widetilde{R'} \xrightarrow{M_{k_0}}_{\mathsf{H}} \widetilde{R''}$ for some $\widetilde{R''}$ that coincide on the threads of the invariants.
  - Otherwise, a_{k_0} is a parallel action. Hence, it appear only once. By Lemma 22, for every $k \in S'$, $\vdash a_{k_0}.P \geq \ominus.k$. Hence, by hypothesis, $\widetilde{R}^{k_0}$ coincide with $\widetilde{R'}$ on this thread, and, for all $k \in S'$, $\widetilde{R}^k$ coincide with $\widetilde{R'}$ on this threads. Hence, $\widetilde{R'} \xrightarrow{M_{k_0}}_{\mathsf{H}} \widetilde{R''}$ and $\widetilde{R}^k \xrightarrow{M_{k_0}}_{\mathsf{h}} \widetilde{R^{ka}}$ for some $\widetilde{R^{ka}}$, and $\widetilde{R''}$ that coincide on the threads of the invariants. We apply Lemma 26 and get, for every $k \in S'$, Υ'_k respecting the invariant and such that their next action corresponds to $\widetilde{R}^k \xrightarrow{M_{k_0}}_{\mathsf{h}} \widetilde{R^{ka}}$.

  We conclude with $\psi' = \psi.M_{k_0}$ and $\omega'_k = \omega_k.M_{k_0}$ for every $k \in S'$.

- Suppose that $i \notin \mathcal{C}$ and $j \in \mathcal{C}$, then $\gamma = M$. Also, $M \in \phi_k(\Psi)$ only for $k = j$. Since Υ_j is compatible with Ψ for j, and $\gamma_j \in \phi_j(\Psi)$, $\gamma_j = \overline{M}$. By definition of $\rightarrow_{\mathsf{h}}$ and $\rightarrow_{\mathrm{I}}$, and since $\widetilde{R}^i$ coincide with $\widetilde{R'}$ on thee emitting thread, $\widetilde{R'} \xrightarrow{M}_{\mathsf{H}} \widetilde{R''}$ for some R'', and we conclude with $\psi' = \psi.M$ and $\omega'_j = \omega_j.M$ for every $k \in S'$.

- Suppose that $i \in \mathcal{C}$ and $j \in \mathcal{C}$, we conclude similarly.

We stop when $\Psi = \psi$, which result in Θ being a global explanation of ψ_0.

□

3

Information-Flow and Non-Interference

In the following chapters, we consider full descriptions of distributed applications. They are specified as an imperative programs with shared memory. The communications that implement the shared memory and the control flow, and (for now) the locality of the code is left abstract in these specifications. Instead, the shared memory is annotated with security levels specifying which parts the participants can read or write. Then, their security is defined as information flow policies, specifying the allowed flows of data in the program. These specifications are more precise than the session specifications studied above since they cover the computation and not only the interactions between participants.

In this chapter, we define our core language, which is used in the next chapters (with some extensions). We also define the information flow policies, and state standard non-interference theorems. We then define our adversaries, and a refinement that precises the security of programs that are not non-interferent. We then define the settings of our main simulation theorems. These definitions pave the way for the main result presented in the following chapters.

In an adversarial setting, there is no guarantees that every participant complies with the specified computation and security policies. Using cryptography, it is possible to provide guarantees on the security of the data, even when it flows through unsecure networks and compromised participants. In the chapters below, we develop a type system to verify the correct use of cryptography in accordance to information flow policies, and we propose a defensive implementation that uses secure hardware and cryptography to enforce information flow policies, even when no participant is trusted enough to handle the data.

3.1 An imperative probabilistic While language

We want precise and realistic guarantees, hence we need a more concrete model of cryptography. A probabilistic semantics is necessary for modeling encryption security [Goldwasser and Micali, 1982]. Encryptions are modeled as probabilistic functions that follow cryptographic hypothesis proven by cryptographers, stated in term of probabilistic polynomial-time games. Our core language is a probabilistic while-language, (a small core of the C programming language) with the following grammar:

$$\begin{array}{rcl} e & ::= & x \mid op(e_1, \ldots, e_n) \\ P & ::= & x := e \mid x_1, \ldots, x_m := f(e_1, \ldots, e_n) \\ & & \mid P; P \mid \text{if } e \text{ then } P \text{ else } P \mid \text{while } e \text{ do } P \mid \mathit{skip} \end{array}$$

where x ranges over variables, v ranges over literal values (bitstrings, which may represent integers, strings, booleans), and op ranges and f range over deterministic and probabilistic n-ary functions, respectively, with arity $n \geq 0$. Expressions e consist of variables and operations. We assume

$$\text{(Assign)}\ \frac{[\![e]\!](\mu) = v}{\langle x := e, \mu\rangle \rightsquigarrow_1 \langle \surd, \mu\{x \mapsto v\}\rangle} \qquad \text{(Fun)}\ \frac{p = [\![f]\!](\mu(e_1), \ldots, \mu(e_n))(\widetilde{v}) \qquad p > 0}{\langle \widetilde{x} := f(\widetilde{e}), \mu\rangle \rightsquigarrow_p \langle \surd, \mu\{\widetilde{x} \mapsto \widetilde{v}\}\rangle}$$

$$\text{(SeqS)}\ \frac{\langle P, \mu\rangle \rightsquigarrow_p \langle P_1, \mu_1\rangle \qquad P_1 \neq \surd}{\langle P; P', \mu\rangle \rightsquigarrow_p \langle P_1; P', \mu_1\rangle} \qquad \text{(SeqT)}\ \frac{\langle P, \mu\rangle \rightsquigarrow_p \langle \surd, \mu_1\rangle}{\langle P; P', \mu\rangle \rightsquigarrow_p \langle P', \mu_1\rangle} \qquad \text{(Cond)}\ \frac{P' = \text{ if } [\![e]\!](\mu) \neq 0 \text{ then } P_1 \text{ else } P_0}{\langle \text{if } e \text{ then } P_1 \text{ else } P_0, \mu\rangle \rightsquigarrow_1 \langle P', \mu\rangle}$$

$$\text{(SkipS)}\ \langle \mathit{skip}, \mu\rangle \rightsquigarrow_1 \langle \surd, \mu\rangle \qquad \text{(Stable)}\ \langle \surd, \mu\rangle \rightsquigarrow_1 \langle \surd, \mu\rangle \qquad \text{(While)}\ \frac{P' = \text{ if } [\![e]\!](\mu) \neq 0 \text{ then } P_1; \text{while } e \text{ do } P_1 \text{ else } \surd}{\langle \text{while } e \text{ do } P_1, \mu\rangle \rightsquigarrow_1 \langle P', \mu\rangle}$$

Figure 3.1: Probabilistic operational semantics

given (polynomial-time) functions for standard boolean and arithmetic constants (0, 1, ...) and operators (||, +, ...). We let $=_0$ be comparison on booleans, true when both its arguments are either 0 or non-0. Commands P consist of variable assignments, using deterministic expressions and probabilistic functions, composed into sequences, conditionals, and loops. We write if e then P for if e then P else *skip*. We write $\widetilde{e}$ (respectively $\widetilde{P}$) for a tuple of variables $e_1, \ldots, e_n$ (respectively commands $P_1, \ldots, P_n$) for some $n \geq 0$. We let $rv(P)$ and $wv(P)$ be the sets of variables that are syntactically read and written by P: $x \in rv(P)$ when x occurs in an expression of P; and $x \in wv(P)$ when a command of the form $x := e$ occurs in P. We let $v(P) = rv(P) \cup wv(P)$.

3.1.1 Probabilistic Semantics

Every function f is equipped with an associated parametric probability distribution $[\![f]\!]$. Hence, in the special case f is a deterministic function, the distribution $[\![f]\!](\widetilde{v})$ gives probability 1 to $f(\widetilde{v})$ and 0 to any other output. We write $\{0,1\}$ for the fair "coin-tossing" function that returns either 0 or 1 with probability $\frac{1}{2}$. We use probabilistic functions mainly to model cryptographic algorithms as commands.

Program configurations (s) are of the form $\langle P, \mu\rangle$ where P is a program and μ is a *memory*, that is, a function from variables to values. We write $\mu\{x \mapsto v\}$ for the memory that maps x to v and $y \neq x$ to $\mu(y)$. The special program $\surd$ represents termination. Figure 3.1 gives the operational semantics of commands as Markov chains between program configurations, with probabilistic steps $s \rightsquigarrow_p s'$ induced by the probabilistic functions. We omit the usual rules for expressions $[\![e]\!](\mu)$. We lift these reduction steps to configuration distributions (d), and write $d \rightsquigarrow d'$ when, for all configurations s', $d'(s') = \sum_{s \rightsquigarrow_p s'} p \times d(s)$. We need Rule Stable to have full distributions ($\sum_s d'(s) = \sum_s d(s) = 1$).

We define the semantics of a configuration $\langle P, \mu\rangle$ by a series of distributions $(d_i)_{i \geq 0}$ such that $d_0(\langle P, \mu\rangle) = 1$ and $d_i \rightsquigarrow d_{i+1}$ for $i \geq 0$. We write $\Pr[\langle P, \mu\rangle; \varphi]$ for the probability that P terminates with a final memory that meets condition φ: $\Pr[\langle P, \mu\rangle; \varphi] = \lim_{i \to \infty} \sum_{s = \langle \surd, \mu\rangle \mid \varphi} d_i(s)$ (the limit exists because the sum increases with i and is bounded by 1). When P always terminates, that is, $\Pr[\langle P, \mu\rangle; \mathit{true}] = 1$, we let $d_\infty(d_0)$ be the limit of $(d_i)_{i \geq 0}$. Since all the configurations of d_∞ are of the form $\langle \surd, \mu'\rangle$, we write $\rho_\infty(d_0)$ for the corresponding final distribution of memories μ'. For a given domain S, we write $\mu_{|S}$ for μ restricted to S, $\rho_{|S}$ for the projection of ρ on S, that is, $\rho_{|S}(\mu_{|S}) = \sum_{\mu' \mid \mu_{|S} = \mu'_{|S}} \rho(\mu')$, and $d_{|S}$ for the projection of d on S, that is, $d_{|S}(\langle P, \mu_{|S}\rangle) = \sum_{\mu' \mid \mu_{|S} = \mu'_{|S}} d(\langle P, \mu'\rangle)$. When it is clear from the context, we may write μ for the single point distribution ρ defined by $\rho(\mu) = 1$ and $\rho(\mu') = 0$ for $\mu' \neq \mu$.

Example 9 *As an example of our language and probabilistic semantic, we consider the command*

$$P \doteq \begin{array}{l} x := 1; \\ \text{while } x \text{ do } \{ \\ \quad y := y + 1; \\ \quad x := \{0,1\}\} \end{array}$$

In this command, $x := 1$ *and* $y := y + 1$ *are deterministic assignments and* $x := \{0,1\}$ *is a probabilistic function application. For convenience, we define*

$$P_1 \doteq \begin{array}{l} \text{while } x \text{ do } \{ \\ \quad y := y + 1; \\ \quad x := \{0,1\}\} \end{array} \quad P_2 \doteq \begin{array}{l} y := y + 1; \\ x := \{0,1\}; \\ P_1 \end{array} \quad P_3 \doteq \begin{array}{l} x := \{0,1\}; \\ P_1 \end{array}$$

We now consider the initial program configuration

$$s = \langle P, \{x \mapsto 0, y \mapsto 0\}\rangle$$

We have $\langle x := 1, \{x \mapsto 0, y \mapsto 0\}\rangle \rightsquigarrow_1 \langle \surd, \{x \mapsto 1, y \mapsto 0\}\rangle$ *by Rule* Assign. *Hence, by Rule* SeqT,

$$\langle P, \{x \mapsto 0, y \mapsto 0\}\rangle \rightsquigarrow_1 \langle P_1, \{x \mapsto 1, y \mapsto 0\}\rangle$$

By Rule While, *since* $x \mapsto 1$ *in the memory,*

$$\langle P_1, \{x \mapsto 1, y \mapsto 0\}\rangle \rightsquigarrow_1 \langle P_2, \{x \mapsto 1, y \mapsto 0\}\rangle$$

Then, by Rules Assign *and* SeqT,

$$\langle P_2, \{x \mapsto 1, y \mapsto 0\}\rangle \rightsquigarrow_1 \langle P_3, \{x \mapsto 1, y \mapsto 1\}\rangle$$

$\{0,1\}$ *is the fair coin-tossing function, with* $[\![\{0,1\}]\!](0) = 1/2$ *and* $[\![\{0,1\}]\!](1) = 1/2$. *By Rules* Fun *and* SeqT,

$$\langle P_3, \{x \mapsto 1, y \mapsto 1\}\rangle \rightsquigarrow_{\frac{1}{2}} \langle P_1, \{x \mapsto 1, y \mapsto 1\}\rangle$$

and

$$\langle P_3, \{x \mapsto 1, y \mapsto 1\}\rangle \rightsquigarrow_{\frac{1}{2}} \langle P_1, \{x \mapsto 0, y \mapsto 1\}\rangle$$

Then, by Rule While,

$$\langle P_1, \{x \mapsto 0, y \mapsto 1\}\rangle \rightsquigarrow_1 \langle \surd, \{x \mapsto 0, y \mapsto 1\}\rangle$$

and by Rule Stable,

$$\langle \surd, \{x \mapsto 0, y \mapsto 1\}\rangle \rightsquigarrow_1 \langle \surd, \{x \mapsto 0, y \mapsto 1\}\rangle$$

Finally, we let

$$\begin{array}{rcl} s_i & \doteq & \langle P_1, \{x \mapsto 1, y \mapsto i\}\rangle \\ s_i^2 & \doteq & \langle P_2, \{x \mapsto 1, y \mapsto i\}\rangle \\ s_i^3 & \doteq & \langle P_3, \{x \mapsto 1, y \mapsto i\}\rangle \\ q_i & \doteq & \langle P_1, \{x \mapsto 0, y \mapsto i\}\rangle \\ r_i & \doteq & \langle \surd, \{x \mapsto 0, y \mapsto i\}\rangle \end{array}$$

and we get

$$\begin{array}{rcccc} s & \rightsquigarrow_1 & s_0 & & \\ s_i & \rightsquigarrow_1 & s_i^2 & \rightsquigarrow_1 & s_i^3 \\ s_i^3 & \rightsquigarrow_{\frac{1}{2}} & s_{i+1} & & \\ s_i^3 & \rightsquigarrow_{\frac{1}{2}} & q_{i+1} & & \\ q_i & \rightsquigarrow_1 & r_i & \rightsquigarrow_1 & r_i \end{array}$$

If we consider the series of distributions $(d_i)_{i\geq 0}$*, we have*

$$d_0(s) = 1 \qquad d_1(s_0) = 1 \qquad d_2(s_0^2) = 1 \qquad d_3(s_0^3) = 1 \qquad \begin{array}{l} d_4(s_1) = \frac{1}{2} \\ d_4(q_1) = \frac{1}{2} \end{array} \qquad \begin{array}{l} d_5(s_1^2) = \frac{1}{2} \\ d_5(r_1) = \frac{1}{2} \end{array}$$

$$\begin{array}{l} d_6(s_1^3) = \frac{1}{2} \\ d_6(r_1) = \frac{1}{2} \end{array} \qquad \begin{array}{l} d_7(s_2) = \frac{1}{4} \\ d_7(q_2) = \frac{1}{4} \\ d_7(r_1) = \frac{1}{2} \end{array} \qquad \ldots \qquad \begin{array}{l} d_{10}(s_3) = \frac{1}{8} \\ d_{10}(q_3) = \frac{1}{8} \\ d_{10}(r_2) = \frac{1}{4} \\ d_{10}(r_1) = \frac{1}{2} \end{array} \qquad \ldots \qquad \begin{array}{l} d_{13}(s_4) = \frac{1}{16} \\ d_{13}(q_4) = \frac{1}{16} \\ d_{13}(r_3) = \frac{1}{8} \\ d_{13}(r_2) = \frac{1}{4} \\ d_{13}(r_1) = \frac{1}{2} \end{array} \qquad \ldots$$

We have $\Pr[s; y = 3] = \lim_{i\to\infty} d_i(r_3)$ *since* r_3 *is the only possible final distribution with a memory in which* $y \mapsto 3$*. Hence* $\Pr[s; y = 3] = \frac{1}{8}$*.*

We have

$$\begin{array}{rl} \Pr[s; \mathit{true}] & = \lim_{i\to\infty} \sum_{s=\langle\surd,\mu\rangle} d_i(s) \\ & = \lim_{i\to\infty} \sum_j d_i(r_j) \\ & = \lim_{i\to\infty} \sum_{j\leq(i-4)/3} \frac{1}{2^j} \\ & = 1 \end{array}$$

Hence P *always terminates, and we have*

$$\rho_\infty(d_0)(x)(0) = 1 \qquad \rho_\infty(d_0)(y)(1) = \frac{1}{2} \qquad \rho_\infty(d_0)(y)(2) = \frac{1}{4} \qquad \rho_\infty(d_0)(y)(3) = \frac{1}{8}$$

$$\rho_\infty(d_0)(y)(4) = \frac{1}{16} \qquad \ldots \qquad \rho_\infty(d_0)(y)(i) = \frac{1}{2^i}$$

In the following, we only consider programs that terminates in a bounded time, for simplicity and in previsions of limitations of computational cryptography. Most of our proofs that do not depend on computational assumptions can be generalized by taking the limit.

3.2 Security policies

Security is often about protecting information. For example, military files have confidentiality levels that can be read with different levels of security clearance. Conversely, military plan have different levels of integrity, and may only be modifiable by top officials. We use similar security labels that indicate both the confidentiality of the value (ex: "secret" or "public") and its integrity (ex:"trusted" or "untrusted"). Those are two different (but linked) aspects of security. For example, if a military plan is protected by a secret password, breaking the secrecy of the password may allow for an unauthorized modification of the plan. Conversely, breaking the integrity of a password, by setting it to its default value, may break the secrecy of protected files. Formally, the different labels are ordered in a lattice, thus defining their relation. For example, a value is more secret if its label is higher in the confidentiality lattice. (Different instances of security lattices have been studied, in particular, the Decentralized Label Model of Myers and Liskov [1998])

3.2.1 Security labels

We annotate each program variable with a security label. These labels specify the programmer's security intent. They do not affect the operational semantics of programs. The security labels form a lattice $(\mathcal{L}, \leq)$ obtained as the product of two lattices, for confidentiality levels $(\mathcal{L}_C, \leq_C)$ and for integrity levels $(\mathcal{L}_I, \leq_I)$. We write $\bot_\mathcal{L}$ and $\top_\mathcal{L}$ for the smallest and largest elements of $\mathcal{L}$, and $\sqcup$ and $\sqcap$ for the least upper bound and greatest lower bound of two elements of $\mathcal{L}$, respectively. We write $\bot_C$, $\bot_I$, $\top_C$, and $\top_I$ for the smallest and largest elements of $\mathcal{L}_C$ and $\mathcal{L}_I$, respectively. In examples, we often use a four-point lattice defined by $LH < HH < HL$ and $LH < LL < HL$, where LH for instance is low-confidentiality high-integrity.

For a given label $\ell = (\ell_C, \ell_I)$ of $\mathcal{L}$, the confidentiality label ℓ_C specifies a read level for variables, while the integrity label ℓ_I specifies a write level; the meaning of $\ell \leq \ell'$ is that ℓ' is at least as confidential (can be read by fewer entities) and at most as trusted (can be written by more entities) than ℓ. We let $C(\ell) = \ell_C$ and $I(\ell) = \ell_I$ be the projections that yield the confidentiality and integrity parts of a label. Hence, the partial order on $\mathcal{L}$ is defined as $\ell \leq \ell'$ if and only if $C(\ell) \leq_C C(\ell')$ and $I(\ell) \leq_I I(\ell')$. We overload $\leq_C$ and $\leq_I$, letting $\ell \leq_C \ell'$ be $C(\ell) \leq_C C(\ell')$ and $\ell \leq_I \ell'$ be $I(\ell) \leq_I I(\ell')$. We let $\ell^I \doteq (\bot_C, I(\ell))$ be the label with low confidentiality and the integrity of ℓ.

The example below illustrates a simple security lattice (a variant of the DLM for two participants)

Example 10 *As an example, we consider two participants A and B that do not trust each other. To each of these participants, we associate a confidentiality label (ℓ^a_C for A and ℓ^b_C for B) and an integrity label (ℓ^a_I for A and ℓ^b_I for B). A value written by A and that only A is allowed to read has the label ℓ^a_C, ℓ^a_I. A value written by A that only B is allowed to read has the label ℓ^b_C, ℓ^a_I. These labels do not form a proper lattice by themselves; we introduce:*

- *An integrity label for values that are untrusted (that can come from any of the participants): $\top_I$*
- *An integrity label for values that are trusted by both participants: $\bot_I$*
- *A confidentiality label for values that are public: $\bot_C$*
- *A confidentiality label for values that are secret (that cannot be read by any of the participants): $\top_C$*

and we obtain a lattice defined by the rules:

$$\bot_C \leq \ell^a_C \leq \top_C \qquad \bot_C \leq \ell^b_C \leq \top_C \qquad \bot_I \leq \ell^a_I \leq \top_I \qquad \bot_I \leq \ell^b_I \leq \top_I$$

3.2.2 Policies

We now define a security policy by labeling the variables of the program. Formally, memory policies are functions Γ from variables to security labels. (The variables are untyped for now, but we add datatypes to the policy in one of the following chapters.) Once a security policy is defined, it is possible to partially compare two memories relatively to this policy. For example, if we determine that two memory are equal on their public variables, it is not possible distinguish them by accessing only the public variables. Formally, we define two variants *low equality* between memories relative to a label $\ell \in \mathcal{L}$:

- Letting $S = \{x \mid \Gamma(x) \leq \ell\}$, we define $\mu =_\ell \mu'$ as $\mu_{|S} = \mu'_{|S}$. We also define low equality between distributions $\rho =_\ell \rho'$ as $\rho_{|S} = \rho'_{|S}$.
- Similarly, letting $S' = \{x \mid \ell \not\leq \Gamma(x)\}$, we define $\mu =^\ell \mu'$ as $\mu_{|S'} = \mu'_{|S'}$. We also define low equality between distributions $\rho =^\ell \rho'$ as $\rho_{|S'} = \rho'_{|S'}$.

The first variant relates couple of memories that are equal on all variables below a security label. The second variant relates couple of memories that are equal on all variables that are not above a security label. These two notions are similar, but one may be more practical depending on the situation.

3.3 Information-flow and Non-interference

The security policies defined above do not influence the semantic of programs, however, it is possible to define access control policies (e.g., based on the memory policy) that limit which part of the program or which participant can access each variable. Still, a trusted program can access a secret variable and leak (by mistake) its value to a public variable. Thus, these access control policies are unsatisfactory for protecting information on large systems, because they control access to information but not its propagation. Therefore, it may be useful to control information flows through the program and to verify, for instance, that secret variables only flows to secret variables. We want to enforce *information-flow* policies.

When a program is secure, we expect the public final memory not to be influenced by the private initial memory. This is formalized as a property of *non-interference* [Goguen and Meseguer, 1982]: if two initial memories coincide on their public (respectively trusted) variables, the two final memories (after execution of the program) should coincide on their public (respectively trusted) variables. Formally, our property of non-interference is parametrized by a label ℓ; and memories are compared using low equality for ℓ (thus, for example, public variables are variables that are more public than ℓ). A program is then non-interferent when it is non-interferent for any label ℓ.

In a concrete setting, private information can flow by other channels than the memory. For example, a program can be considered insecure if does not terminate (or terminate slowly) on a particular private input, since it may leak information on the private input if its termination is observable. The most famous example of a side channel attack is the timing attack on implementation of cryptographic operations from Kocher [1996] that allows to retrieve encryption keys by timing decryptions. In this thesis, we consider only termination insensitive non-interference, we assume that the adversary cannot observe termination or any other side channel. Other works study information-flow non-interference in the presence of side channels. For instance, Volpano and Smith [1997] tackles termination sensitive non-interference.

Denning [1976] is the first to study static checking of potential information flows to verify the security of a programs. Dynamic methods to track information flows have been developed, for instance by Suh et al. [2004]. The current preferred static method, first introduced by Volpano et al. [1996], uses an information-flow type system. We present such type system in the subsection below.

3.3.1 A strict type system for non-interference

We equip our language with a type system that enforces (termination-insensitive) non-interference. Typing judgments for commands are of the form $\Gamma \vdash P : \ell$. Typing judgments for expressions are of the form $\Gamma \vdash e : \ell$. We often omit the policy Γ when it is clear from the context (because it is fixed).

The typing rules for commands and expressions appear in Figure 3.2. This type system is similar to those typically used for non-interference [see e.g. Volpano et al., 1996].

We have $\vdash e : \ell$ when e contains only variables at a level lower than ℓ, i.e., variables that are less confidential and more trusted than ℓ. This is enforced by the rules Var, TOp and SubE. We have $\vdash P : \ell$ when P is non-interferent, and when it only writes variables with a level higher than ℓ (sometimes called its 'program counter' level), i.e., variables that are more confidential and less trusted than ℓ. We explain some of the rules:

- When executing a program that consists of a simple assignment ($x := e$), the final value of x depends on e. In Rule TAssign Strict, $\vdash e : \Gamma(x)$ ensures that all the variables of e are less confidential and at least as trusted as x, hence guaranteeing non-interference of the command. The resulting type of the command corresponds to the type of the only variable written: $\Gamma(x)$.

$$\frac{}{\vdash x : \Gamma(x)}\ (\textsc{Var}) \qquad \frac{\vdash e : \ell \ \text{ for } e \in \widetilde{e}}{\vdash op(\widetilde{e}) : \ell}\ (\textsc{TOp}) \qquad \frac{\vdash e : \ell \qquad \ell \leq \ell'}{\vdash e : \ell'}\ (\textsc{SubE})$$

$$\frac{\vdash e : \Gamma(x)}{\vdash x := e : \Gamma(x)}\ (\textsc{TAssign Strict}) \qquad \frac{\vdash e : \ell \ \text{ for } e \in \widetilde{e} \quad \ell \leq \Gamma(x) \ \text{ for } x \in \widetilde{x}}{\vdash \widetilde{x} := f(\widetilde{e}) : \ell}\ (\textsc{TFun})$$

$$\frac{\vdash P : \ell \qquad \vdash P' : \ell}{\vdash P; P' : \ell}\ (\textsc{TSeq}) \qquad \frac{\vdash e : \ell \qquad \vdash P : \ell \qquad \vdash P' : \ell}{\vdash \text{if } e \text{ then } P \text{ else } P' : \ell}\ (\textsc{TCond})$$

$$\frac{\vdash e : \ell \qquad \vdash P : \ell}{\vdash \text{while } e \text{ do } P : \ell}\ (\textsc{TWhile}) \qquad \frac{}{\vdash \mathit{skip} : \top}\ (\textsc{TSkip}) \qquad \frac{}{\vdash \surd : \top}\ (\textsc{TInert}) \qquad \frac{\vdash P : \ell \qquad \ell' \leq \ell}{\vdash P : \ell'}\ (\textsc{TSubC})$$

Figure 3.2: Security type system (for a fixed policy Γ)

- When executing a program that consists of a conditional branch (if e then P else P'), the final values of the variables written in P or P' may be influenced by the result of the conditional expression, hence by the values of the variables in e. In Rule TCond, $\vdash e : \ell$, $\vdash P : \ell$, and $\vdash P' : \ell$ ensure that all the variable in e are less confidential and at least as trusted as the variables written in P and P', hence guarantying the non-interference of the command.
- The Rule TSeq is a subtyping rule. It states that, if P is non-interferent, P only writes variables with a level higher than ℓ, and $\ell' \leq \ell$, then P only writes variables with a level higher than ℓ'. It is necessary, for example, to type with Rule TCond a conditional branch with two commands of different type. In this case, the level considered is a level below the level of the two commands, e.g., the greatest lower bound.

The example below illustrates typing.

Example 11 *Let Γ be defined by $\Gamma(s) = HH$ (s is secret) and $\Gamma(p) = LH$ (p is public). As an example of typing, we consider the command*

$$P \ \doteq \ \text{if } p \leq s \text{ then } s := s - p$$

We have $\Gamma(p) = LH$, hence $\vdash p : LH$ using Rule Var. Then, since $LH \leq HH$, we have $\vdash p : HH$ using Rule SubE. Similarly, $\vdash s : HH$ using Rule Var. Hence $\vdash p \leq s : HH$ and $\vdash s - p : HH$ using Rule TOp. Hence $\vdash s := s - p : HH$ using Rule TAssign Strict and $\vdash$ if $p \leq s$ then $s := s - p : HH$ using Rule TCond. P types at level HH, is non interferent, and only writes confidential variables.

A program that types with the typing rules of Figure 3.2 is (termination insensitive) non-interferent:

Theorem 4 (Non-interference) *Let Γ be a security policy, P a well-typed command, $\ell \in \mathcal{L}$, and μ_0 and μ_1 two initial memories such that P terminates.*

If $\mu_0 =_\ell \mu_1$, then $\rho_\infty(\langle P, \mu_0\rangle) =_\ell \rho_\infty(\langle P, \mu_1\rangle)$. If $\mu_0 =^\ell \mu_1$, then $\rho_\infty(\langle P, \mu_0\rangle) =^\ell \rho_\infty(\langle P, \mu_1\rangle)$.

The proof of Theorem 4 is at the end of this chapter, in Subsection 3.6.1.

We illustrate our non interference theorem in the example below.

Example 12 *By Theorem 4, the program of Example 11 is non-interferent (with the corresponding policy). We instantiate this definition for some $\ell = LL$ and some memories $\mu_0 = \{s \mapsto 7, p \mapsto 10\}$ and $\mu_1 = \{s \mapsto 12, p \mapsto 10\}$. The two memories differ only on their secret values: $\mu_{0|\ell} = \{s \mapsto 10\} = \mu_{1|\ell}$. Also, $\rho_\infty(\langle P, \mu_0\rangle)$ is a single point distribution, with $\rho_\infty(\langle P, \mu_0\rangle)(\{s \mapsto 2, p \mapsto 10\}) = 1$ and $\rho_\infty(\langle P, \mu_1\rangle)$ is a single point distribution, with $\rho_\infty(\langle P, \mu_1\rangle)(\{s \mapsto 7, p \mapsto 10\}) = 1$. The two memory distributions differ only on their secret values: $\rho_\infty(\langle P, \mu_0\rangle) =_\ell \rho_\infty(\langle P, \mu_1\rangle)$.*

3.3.2 Active adversaries

In this thesis, we consider the security of systems against active adversaries. In our information-flow settings, we consider a compromise level, and we model these adversary with commands that read and write variables according to this level. If we consider a single machine, the compromise level may correspond to the users that are compromised, allowing the adversary to read or write data that these user can read or write. In a distributed model, the compromise level may correspond to the machines that are compromised, the networking traffic that can be spied, etc. These restrictions on the adversary commands are access control policies, but the adversary does not necessarily follows information-flow policies (i.e., it is not typed). As a first step, these commands are interleaved with the legitimate parts of the program, but we will later consider adversaries that control the scheduling of the programs.

Formally, for a given $\alpha \in \mathcal{L}$, we let α-adversaries range over tuple of commands $\widetilde{A}$ that read variables in a set $V_{\mathcal{A}}^{C} = \{x \mid x \leq_C \alpha\}$ with confidentiality level less than or equal to $C(\alpha)$ and write variables in a set $V_{\mathcal{A}}^{I} = \{x \mid \alpha \leq_I x\}$ with integrity level more than or equal to $I(\alpha)$. We consider programs obtained by composing commands with α-adversaries. To this end, we add command variable to the grammar of P:

$$P ::= \cdots \mid X$$

Command variables X are placeholders for commands, bound in command contexts. As usual, we often use anonymous command variables in command contexts, writing $P[\widetilde{Q}]$ instead of $P[\widetilde{Q}/\widetilde{X}]$. For instance, $(X_1; X_2; X_1)[P_1, P_2]$ stands for $P_1; P_2; P_1$. In the following, we sometimes use $\_$ for anonymous command variables that are filled with adversary commands. Command context are commands with free command variables. We add a typing rule for command variables X:

$$\frac{}{\vdash X : (C(\alpha), \top_I)}\ (\textsc{TAdversary})$$

With this additional rule, we state that command variables, which are placeholders for unknown adversary commands, might leak the program counter.

We now update our theorem on non-interference (Theorem 4) to account for active adversaries. Since the adversary commands are not restricted on flow of information but only on variables it can read or write, non-interference is limited to labels more secret and more trusted than α.

Theorem 5 (Non-interference against active adversary) *Let Γ be a security policy, P a well-typed command context, $\ell, \alpha \in \mathcal{L}$, $\widetilde{A}$ an α-adversary, and μ_0 and μ_1 two initial memories such that $P[\widetilde{A}]$ terminates.*

If $\mu_0 =_\ell \mu_1$, and $C(\alpha) \leq C(\ell)$ or $I(\alpha) \not\leq I(\ell)$, then $\rho_\infty(\langle P[\widetilde{A}], \mu_0\rangle) =_\ell \rho_\infty(\langle P[\widetilde{A}], \mu_1\rangle)$. If $\mu_0 =^\ell \mu_1$, and $C(\ell) \not\leq C(\alpha)$ or $I(\ell) \leq I(\alpha)$, then $\rho_\infty(\langle P[\widetilde{A}], \mu_0\rangle) =^\ell \rho_\infty(\langle P[\widetilde{A}], \mu_1\rangle)$.

The proof of Theorem 5 is at the end of this chapter, in Subsection 3.6.2.

With a variant of Example 11, we illustrate an active attack that is correctly banned by our type system.

Example 13 *Let $\alpha = LL$, we consider a variant of command P of Example 11 with an additional placeholder for the adversary:*

$$P' \doteq \textit{if } p \leq s \textit{ then } \{s := s - p; \_\}$$

P' does not type since $\vdash \_ : LL$. Indeed, we choose $\ell = LL$, $\alpha = LL$, $A \doteq l := 1$ ($\Gamma(l) = LL$), $\mu_0 = \{s \mapsto 7, p \mapsto 10, l \mapsto 0\}$ and $\mu_1 = \{s \mapsto 12, p \mapsto 10, l \mapsto 0\}$. $\rho_\infty(\langle P'[A], \mu_0\rangle)$ is a single point distribution, with $\rho_\infty(\langle P'[A], \mu_0\rangle)(\{s \mapsto 2, p \mapsto 10, l \mapsto 1\}) = 1$ and $\rho_\infty(\langle P'[A], \mu_1\rangle)$ is a single point distribution, with $\rho_\infty(\langle P'[A], \mu_1\rangle)(\{s \mapsto 7, p \mapsto 10, l \mapsto 0\}) = 1$. The two memory distributions differ on the public value of l: $\rho_\infty(\langle P'[A], \mu_0\rangle) \neq_\ell \rho_\infty(\langle P'[A], \mu_1\rangle)$.

3.4 Declassifications and endorsements

Our strict type system yields non-interference, hence it necessarily excludes many useful programs that (by design) selectively allow some flows from private to public values (declassifications) or flows from untrusted to trusted values (endorsements). We illustrate one of those programs below.

Example 14 (Password protection) *Consider a program that releases a secret after verifying a password entered in variable* guess*:*

$$P \doteq \text{if } guess = pwd \text{ then } r := secret$$

with $\Gamma(guess) = LL$, $\Gamma(pwd) = HH$, $\Gamma(r) = LH$, *and* $\Gamma(secret) = HH$. *The password is highly trusted and confidential. The guess is untrusted and public: it is a tentative to log-in, for example from a terminal. The secret is secret an trusted, but may be leaked to the result* r *if the guess match the password. Although this program is arguably secure, it endorses* guess *(which is a priori untrusted), declassifies the outcome of the test (which is secret, since* pwd *is), then possibly declassifies secret.*

In some of the following chapters, we intend to compile such programs, considering bad flows as part of the specification of the program and letting the programmer take responsibility for the source properties of his program, but still ensuring that the compilation process does not introduce any further potentially-unsafe information flows. Hence, we need a laxer type system that does not enforce non-interference, but still guarantees interesting properties.

Robustness

The research on information flow properties of programs that are not non-interferent is active. Zdancewic and Myers [2001] consider systems that include declassification and characterize what information is leaked. In particular, they define a notion of robustness, which ensure that an attacker is not able to influence what information is declassified. Sabelfeld and Myers [2004] consider programs were possible leaks of information are specified by the initial value of the declassified expression, this implies that values declassified are not controlled by an attacker. Askarov and Myers [2010] compare traces that have the same endorsement, also with a notion of robustness for release events. Chong and Myers [2006] also consider that the attacker should not control the information that is declassified. They introduce a function that defines what is the necessary integrity level of the declassification so that it is robust. They also allows for some endorsement (considering them as non-deterministic assignment). Sabelfeld and Sands [2005] wrote a review of declassifications approaches.

As we see in Example 14 and the works discussed above, declassification is useful, but may need to be controlled. In our example, in particular, the declassification depends on the low-integrity variable *guess*, letting the adversary influence what is declassified. In this case, it is arguably safe, but it is generally dangerous. For instance, the example would be entirely broken with $\Gamma(pwd) = HL$, since the untrusted password could be set to 0. As in the works discussed above, we say that a declassification is *robust* when it does not depend on low-integrity data. We rely on a monotonic *robustness function*, $R : \mathcal{L}_C \to \mathcal{L}_I$, that indicates the minimum integrity level required to declassify each confidentiality level. For instance, in a system where a participant B is associated with a confidentiality level ℓ_C^b and an integrity level ℓ_I^b, we may choose $R(\ell_C^b) = \ell_I^b$ to ensure that only B (and participants trusted by B) has the authority to declassify values that only him can read. We also support non-robust declassifications, treating them as a high-integrity endorsement followed by a robust declassification,

When using the lax type system with active adversaries, we only consider robustness functions that ensure that adversaries can read any robust level they can write.

$$\frac{\vdash e : (c, \_) \quad c \leq C(x)}{\vdash x := e : \Gamma(x)} \text{ (TAssign Endorse)} \qquad \frac{\vdash e : (c, \_) \quad c \not\leq C(x)}{\vdash x := e : \Gamma(x) \sqcap (\top, R(c))} \text{ (TAssign Robust)}$$

Figure 3.3: Lax rules for the security type system (for a fixed policy Γ)

Definition 15 *R is robust against an adversary level α when, for all confidentiality levels $c \in \mathcal{L}_C$, if $I(\alpha) \leq R(c)$, then $c \leq C(\alpha)$.*

3.4.1 A more permissive type system

For typing unsafe programs, we define a new type system, whose typing rules allow declassifications and endorsements but take them into account to compute the level of the command. The new type system uses two new 'lax' typing rules appearing in Figure 3.3 instead of Rule TAssign Strict. Every typing judgment in this subsection uses this lax type system. The Rules TAssign Endorse and TAssign Robust are two generalization of rule TAssign Strict. TAssign Endorse is TAssign Strict only when e has at least the integrity of x; otherwise, the assignment endorses e. Irrespective of e, the command is typed at the level of x. TAssign Robust enables the declassification of e into x ($c \not\leq C(x)$) but it records this privileged operation by raising the command type up to the associated robust integrity level $R(c)$.

The lax type system enforces two fundamental properties. First, if a command has level ℓ, then it does not write variables below ℓ. The lax type system enforces same containment properties than the strict type system (see Lemma 28).

Theorem 6 (Lax-Containment) *Let Γ be a policy, $\ell \in \mathcal{L}$ a label, P a command, and μ a memory such that P terminates.*

If $\vdash P : \ell$, then $\rho_\infty(\langle P, \mu\rangle) =^\ell \mu$ and for any ℓ' such that $\ell \not\leq \ell'$, we have $\rho_\infty(\langle P, \mu\rangle) =_{\ell'} \mu$.

The proof of Theorem 6 is at the end of this chapter, in Subsection 3.6.3.

Second, a command at level ℓ may declassify values of confidentiality c only if $I(\ell) \leq R(c)$. This theorem is weaker than Theorem 4 of non-interference for the strict type system: it only guarantees the non-interference of low-integrity commands for high-confidentiality values.

Theorem 7 (Robust non-interference) *Let Γ be a policy, $\ell \in \mathcal{L}$ a label, $c \in \mathcal{L}_C$ such that $I(\ell) \not\leq R(c)$. Let P be a command and μ_0, μ_1 memories such that P terminates.*

If $\vdash P : \ell$ and $\mu_0 =^{(c,H)} \mu_1$, then $\rho_\infty(\langle P, \mu_0\rangle) =^{(c,H)} \rho_\infty(\langle P, \mu_1\rangle)$.

The proof of Theorem 7 is at the end of this chapter, in Subsection 3.6.4.

3.4.2 Semi-active adversaries

The guarantees of a program that contains declassification and endorsement in an active adversary model are thin. A programmer who wants to estimate the security of its program has to consider all possible adversaries and all possible executions. We intend to ease this process by limiting the number of possible adversaries to consider, for example, by enforcing that these adversaries only write variables that are going to be endorsed. To this end, we study a refined version of our active adversaries, which is more constrained. We then prove that, under certain circumstances, constrained adversaries cover all attacks.

We define semi-active command context $\overline{P}$ with a grammar obtained from that of P by adding annotations to command variables. We write $X_{d,n}$ instead of X, where d and n range over sets of variables. As in the section above, we may write $[\_]_{d,n}$ instead of $X_{d,n}$ for anonymous command variables. The set d limits the variables the adversary can read (in this placeholder) and the set n limits the variables the adversary can write (in this placeholder). For a given command context $\overline{P}$, we let semi-active α-adversaries range over α-adversaries $\widetilde{A}$ such that, for each placeholder $[\_]_{d,n}$ and each command A_i filling this placeholder in $\overline{P}[\widetilde{A}]$:

- $\mathrm{rv}(A_i) \cap v(P) \subseteq d$ and
- $\mathrm{wv}(A_i) \cap v(P) \subseteq n$.

We illustrate this model below.

Example 15 *Let $\alpha = LL$, we consider a variant of the command of Example 14 with an intermediary variable* test *such that $\Gamma(\mathrm{test}) = LH$ and additional placeholders for the adversary :*

$$\overline{P} \;\doteq\; [\_]_{\emptyset,\{guess\}}; test := (guess = pwd); [\_]_{\{test\},\emptyset}; \text{if } test \text{ then } r := secret; [\_]_{\{r\},\emptyset}$$

A semi-active adversary for this command can only write guess *at the start of the program, read the intermediate result* test*, and read r at the end of the program. However, it is still as powerful as active adversaries for the command context*

$$P \;\doteq\; \_; test := (guess = pwd); \_; \text{if } test \text{ then } r := secret; \_$$

Our semi-active adversary model gives a good idea of the weak points of the program.

3.4.3 Refining active attacks to semi-active attacks

We intend to prove that, while a semi-active command context limit the number of attacks to consider, it may still covers all attacks of the corresponding unrestricted command context. We do not directly generate the best correct semi-active command context. However, we use a type system to check them, and their generation becomes an inference problem.

The typing rules for semi-active command context appear in Figure 3.4. The typing rules for expressions are left unchanged from Figure 3.2. Our type system uses local policies that track the levels of values inside the variables. For instance, a public variable is marked as secret if its current contents results from a declassification, thus indicating that the adversary may want to read it. On the contrary, when the variable is zeroed, its level is updated to public. We still consider a global policy Γ, and we use γ (with the same domain) instead of Γ for those policies that can be different from the global policy. Typing judgments for commands are of the form $\gamma \vdash \overline{P} : \ell, \gamma'$. In this case, we say that $\overline{P}$ types at level ℓ with initial policy γ and final policy γ'.

We generalize operation on labels to operations on security policies, and we set a default policy γ_0:

$$C(\gamma) = \{x \mapsto C(\gamma(x)) \mid x \in \mathrm{dom}(\Gamma)\} \qquad I(\gamma) = \{x \mapsto I(\gamma(x)) \mid x \in \mathrm{dom}(\Gamma)\}$$
$$\gamma \sqcap \gamma' = \{x \mapsto \gamma \sqcap \gamma' \mid x \in \mathrm{dom}(\Gamma)\} \qquad \gamma \sqcup \gamma' = \{x \mapsto \gamma \sqcup \gamma' \mid x \in \mathrm{dom}(\Gamma)\}$$
$$\gamma \leq \gamma' \Leftrightarrow \forall x \in \mathrm{dom}(\Gamma), \gamma(x) \leq \gamma'(x) \qquad \gamma =_S \gamma' \Leftrightarrow \forall x \in S, \gamma(x) = \gamma'(x)$$
$$\gamma_0 = \{x \mapsto (\bot_I, \top_C) \mid x \in \mathrm{dom}(\Gamma)\}$$

for $S \subseteq \mathrm{dom}(\Gamma)$. We write $\overline{S}$ for the complement of set S in $\mathrm{dom}(\Gamma)$, that is $\overline{S} = \Gamma \backslash S$.

In Rule TsAssign, the hypothesis binds a type for expression e in the initial policy γ. Then, it updates the type of x with the flow from e. There is no specific rule for declassification or endorsement in this type system system with local policies:

$$\text{(TsAssign)}\ \frac{\gamma \vdash e : \ell}{\gamma \vdash x := e : \ell, \gamma\{x \mapsto \ell\}} \qquad \text{(TsFun)}\ \frac{\gamma \vdash \widetilde{e} : (H, i)}{\gamma \vdash x_1, \ldots, x_m := f(e_1, \ldots, e_n) : (i, \top_C), \gamma\{x \mapsto (H, i)\}}$$

$$\text{(TsSeq)}\ \frac{\gamma \vdash P : \ell, \gamma' \quad \gamma' \vdash Q : \ell, \gamma''}{\gamma \vdash P; Q : \ell, \gamma''} \qquad \text{(TsCond)}\ \frac{\gamma \vdash P : \ell, \gamma' \quad \gamma \vdash P' : \ell, \gamma' \quad \gamma \vdash e : \ell}{\gamma \vdash \text{if } e \text{ then } P \text{ else } P' : \ell, \gamma'} \qquad \text{(TsWhile)}\ \frac{\gamma \vdash P : \ell, \gamma \quad \gamma \vdash e : \ell}{\gamma \vdash \text{while } e \text{ do } P : \ell, \gamma}$$

$$\text{(TsSkip)}\ \frac{}{\gamma \vdash \mathit{skip} : \ell, \gamma} \qquad \text{(TsInert)}\ \frac{}{\gamma \vdash \surd : \ell, \gamma} \qquad \text{(TsSubC)}\ \frac{\gamma_1' \leq \gamma_1 \quad \gamma_2 \leq \gamma_2' \quad \ell' \leq \ell \quad \gamma_1 \vdash P : \ell, \gamma_2}{\gamma_1' \vdash P : \ell', \gamma_2'}$$

$$\text{(TsAdversary)}\ \frac{\begin{array}{c} I(\gamma') =_{\overline{n}} I(\Gamma) \sqcup I(\gamma) \\ I(\gamma) \leq R(C(\gamma)) \\ C(\gamma) =_{\overline{d}} C(\Gamma) \sqcap C(\gamma') \\ C(\gamma') =_d C(\Gamma) \\ i = \sqcap_{x \in n} I(\gamma'(x)) \sqcap_{x \in d} R(C(\gamma(x))) \end{array}}{\gamma \vdash [\_]_{d,n} : (C(\alpha), i), \gamma'}$$

Figure 3.4: Type system for semi-active adversaries

- If the command is a declassification compared to the global Γ, $\gamma'(x)$ has higher confidentiality than $\Gamma(x)$ to keep track of the high-confidentiality contents of x;
- If the command is an endorsement compared to the global Γ, the variables endorsed in e have an higher integrity in γ to keep track of their high integrity footprint.

These variations from the global Γ are accounted for when typing semi-active placeholders (see also Example 16).

In Rule TsAdversary, we type a semi-active placeholder, ensuring that the capabilities given in the sets d and n are sufficient to cover all attacks. In particular, the adversary may want to read low confidentiality variables with high-confidentiality payloads (i.e., x such that $C(\Gamma(x)) \leq C(\gamma(x))$), but has no interest in reading low confidentiality variables with low-confidentiality payload. Similarly, the adversary may want to write low integrity variables with high integrity footprint (i.e., x when $I(\gamma'(x)) \leq I(\Gamma(x))$). The first hypothesis covers endorsement: the values that the adversary cannot write (i.e., outside of n) need to respect the global policy Γ and the preceding flows of the program. The second hypothesis is a robustness conditions covering all variables (see also Examples 17 and 18). The third hypothesis ensures that the variables that are not declassified respect the global policy Γ and the preceding flows of the program. The fourth hypothesis ensures that variables are not declassified more than the global policy.

Rule TsFun, is similar to Rule TsAssign. However, outputs of random functions cannot be inferred from their input and they carries some additional information that may need to be read by the adversary (see also Example 19).

In Rule TsSeq, the first hypothesis types a first command at level ℓ with initial policy γ and updated policy γ'. The second hypothesis types a second command at level ℓ with initial policy γ' and updated policy γ''. We can type the sequence of the two commands at level ℓ, with initial policy γ and updated policy γ''.

In Rule TsCond, we ensure that the guard can type at level ℓ and that both commands can type at level ℓ with initial policy γ and updated policy γ'. The symmetry of the rule gives strong guarantees against implicit flows, as seen in Example 18.

In Rule TsWhile, we ensure that the guard can type at level ℓ and that the command can type at level ℓ with initial and updated policy γ, both policies have to be the same since it is a loop.

In Rule TsSubC, we allows for weakening of a typing statement, with a stronger initial policy or a weaker updated policy.

A semi-active command context $\overline{P}$ is well-typed when there exists ℓ such that $\gamma_0 \vdash \overline{P} : \ell, \gamma_0$. The confidentiality of the initial policy ensures that all initial variables that are public can be read by the first adversary commands (this is similar to the restriction on Rule TsFun that ensure that public randomly generated variables can be read by the adversary). The integrity of the final policy ensures that all the variables that are untrusted can be written by the last adversary commands.

3.4.4 Examples

We first give two small examples on how the typing inference constraints placeholder annotations (hence on adversary capabilities) .

Example 16 (Semi-active placeholders) *Consider the command context*

$$p := s; p' := 0; [\_]_{n,d}$$

Let $\Gamma(p) = \Gamma(p') = HL$ *and* $\Gamma(s) = HH$. *If* $\gamma_0 \vdash p := s; p' := 0 : \ell, \gamma'$, *then* $C(\gamma'(p)) = H \not\leq C(\Gamma(p))$. *Hence* $p \in d$*: it is a low confidentiality variable with a high confidentiality content that the adversary may want to read. However, we can type this command with* $C(\gamma'(p')) = L \leq C(\Gamma(p'))$ *and have* $p' \notin d$*: the adversary does not gain anything by reading* p'.

Consider the command context

$$[\_]_{n,d}; h := l; l' := 0; h' := l'$$

Let $\Gamma(h) = \Gamma(h') = HL$ *and* $\Gamma(l) = \Gamma(l') = LL$. *If* $\gamma \vdash h := l; l' := 0; h' := l' : \ell, \gamma_0$, *we have* $I(\gamma(l)) = H \not\geq I(\Gamma(l))$, *hence* $l \in n$*: it is a low integrity variable with a high integrity footprint that the adversary may want to write. However, we can type this command with* $I(\gamma(l')) = L \geq I(\Gamma(l'))$ *and have* $l' \notin n$*: the adversary does not gain anything by writing* l'.

In the example below, we illustrate the need for the robustness condition in Rule TsAdversary. In particular, we show that some variables might be indirectly endorsed because of confidentiality issues.

Example 17 (Carry-on endorsement) *Consider the command context*

$$\begin{array}{l} [\_]_{n_1,d_1}; s' := s \oplus l; \\ [\_]_{n_2,d_2}; p := s'; \\ [\_]_{n_3,d_3} \end{array}$$

Let $\Gamma(p) = HL$, $\Gamma(s') = LH$, *and* $\Gamma(l) = LL$. *In general, the adversary only needs to write endorsed variables* (s') *as late as possible and, if* $\Gamma(s) = HL$, *there is no reason for him to write* l *since it can write* s' *and simulate any computation appearing before. But if* $\Gamma(s) = HH$, *the adversary cannot simulate the computation anymore since it does not know* s. *The condition* $I(\gamma) \leq R(C(\gamma))$ *in Rule* TsAdversary *ensure that he is able to write* l *(in* n_1*) as well as* s' *(in* n_2*) in this program, even if* l *is not directly endorsed.*

In the example below, we illustrate how some variables may be indirectly endorsed for their exploitation in indirect flows.

Example 18 (Robustness and conditional branches) *Consider the command context*

$$\begin{array}{l} l := 0; [\_]_{n_1,d_1}; \\ \text{if } s \text{ then } l := 0 \text{ else } skip; \\ [\_]_{n_2,d_2} \end{array}$$

Let $\Gamma(s) = HH$ and $\Gamma(l) = LL$. If we only consider the implicit flow from s to l, we may suppose that the adversary only gains in reading l at the end of the program but does not need anything else. However, if the adversary does not modify l at the start (e.g., writing 1 in l), it cannot exploit the implicit flow from s (since l is always 0). Hence, semi-active adversaries need to be able to write in l, even if there is no explicit endorsement of l. This emerges from the robustness condition in Rule TsAdversary *and the symmetry of Rule* TsCond.

We detail its typing. Suppose that $\gamma \vdash$ if s then $l := 0$ else skip $: \ell, \gamma'$, *with $\gamma(s) = HH$. By Rule* TsCond, *$\gamma \vdash l := 0 : \ell, \gamma'$, hence we have $C(\gamma'(l)) = H$. With the robustness condition of Rule* TsAdversary *applied on the next command, we need $I(\gamma'(l)) \leq C(\gamma'(l))$, hence $I(\gamma'(l)) = H$. By Rule* TsSkip, *$\gamma \vdash$* skip $: \ell, \gamma'$ *if $\gamma = \gamma'$. In particular $\gamma(l) = H$, hence, by Rule* TsAdversary, *$l \in n_1$, the semi-active adversary can write l in the first placeholder.*

In the example below, we illustrate the need to consider random sampling as a declassification.

Example 19 (Random assignments) *Consider the command context*

$$\begin{array}{l} [\_]_{n_1,d_1}; \\ r := rand(); [\_]_{n_2,d_2}; \\ \text{if } r = g \text{ then } p := s; [\_]_{n_3,d_3} \end{array}$$

Let $\Gamma(r) = \Gamma(p) = HL$, $\Gamma(g) = LL$, and $\Gamma(s) = HH$. The value r does not explicitly depend on any confidential variable, but it is still an interesting read for the adversary. This justifies the high confidentiality output of probabilistic functions in Rule TsFun.

3.4.5 Simulation theorem

Our theorem states that a well-typed semi-active command context, which is only susceptible to a very limited set of attacks, still covers all the attacks of the corresponding active command context. We consider the program transformation $[\![\overline{P}]\!]$ that replaces every semi-active placeholder $[\_]_{d,n}$ in $\overline{P}$ with $\_$.

Theorem 8 *Let Γ be a policy. Let $\alpha \in \mathcal{L}$ be such that R is robust against α. Let $\overline{P}$ be a well-typed semi-active command context.*

For every α-adversary $\widetilde{A}$, such that $[\![\overline{P}]\!][\widetilde{A}]$ terminates, there exists a semi-active α-adversary $\widetilde{A'}$ such that $[\![\overline{P}]\!][\widetilde{A'}]$ terminates and, for every memory μ,

$$\rho_\infty\langle\overline{P}[\widetilde{A'}], \mu\rangle = \rho_\infty\langle[\![\overline{P}]\!][\widetilde{A}], \mu\rangle$$

Example 20 *For instance, in the commands of Example 15 ($\overline{P}$ and P), the active adversary*

$$A_0 \doteq guess := \text{“}foo\text{”} \qquad A_1 \doteq r := 0 \qquad A_2 \doteq \text{if } r = 0 \text{ then } g := 0 \text{ else } g := 1$$

can be replaced by the following semi-active adversary that never writes r

$$\begin{array}{c} A'_0 \doteq guess := \text{“}foo\text{”} \qquad A'_1 \doteq testcopy := test \\ A'_2 \doteq \text{if } testcopy = 0 \text{ then } g := 0 \text{ else if } r = 0 \text{ then } g := 0 \text{ else } g := 1 \end{array}$$

For every memory μ,

$$\rho_\infty\langle\overline{P}[\widetilde{A'}], \mu\rangle = \rho_\infty\langle P[\widetilde{A}], \mu\rangle$$

The proof of Theorem 8 is at the end of this chapter, in Subsection 3.6.5.

3.5 Properties of program transformations

In the section above, we proved the security of a simple program transformation that only modify the allowed adversaries. In the chapters below, we consider similar properties on more complex transformations (compilations) that affect the programs themselves. We present the main properties we establish for our different compilers, as regards both security and functional correctness. The properties are stated below for an abstract program transformation between source and target programs; they are instantiated before each of our main theorems in chapters 4 and 6.

We are interested in transformations that operate on programs that are (possibly) not perfectly secure, so, as in the case above, we cannot define security as the preservation of non-interference. Instead, for each of our transformations, we demand, as in the section above, that there is an inverse map from target adversaries to source adversaries, essentially showing how to 'decompile' each attack into an attack already present before the transformation is applied.

We consider system configurations obtained by composing a program (e.g. a command context) $P \in \mathcal{P}$, an adversary (e.g. a tuple of commands) $A \in \mathcal{A}$, and an initial state (e.g. a memory) $\mu \in \mathcal{M}$. Their semantics is given by an evaluation function, written $\langle\!\langle \cdot \rangle\!\rangle : \mathcal{P} \to \mathcal{A} \to \mathcal{M} \to \mathcal{M}$, and an observational equivalence on states, written $\approx \subseteq \mathcal{M}^2$. For a given program and an arbitrary adversary, we are interested in the properties of final states up to $\approx$.

For instance, if we consider commands for a client and a bank scheduled by an adversary that controls the network, we may let programs range over pairs of commands Q_c, Q_b, let adversaries range over binary command contexts, let $\approx$ be low-equality on memory distributions, and use the evaluation function

$$\langle\!\langle (Q_c, Q_b), A, \mu \rangle\!\rangle \doteq \rho_\infty(\langle A[Q_c, Q_b], \mu \rangle)$$

Given definitions for source $(\mathcal{P}, \mathcal{A}, \mathcal{M}, \approx, \langle\!\langle\rangle\!\rangle)$ and target $(\mathcal{P}', \mathcal{A}', \mathcal{M}', \approx', \langle\!\langle\rangle\!\rangle')$ configurations, we consider *program transformations*, written $[\![\cdot]\!] : \mathcal{P} \to \mathcal{P}'$, together with *state projections* (that is, surjective functions), written $\pi : \mathcal{M}' \to \mathcal{M}$. (The role of π is to erase any auxiliary variable introduced by the transformation.) We arrive at the following definitions:

Definition 16 (Security) $([\![\cdot]\!], \pi)$ *is secure when, for every source program $P \in \mathcal{P}$ and target adversary $A' \in \mathcal{A}'$, there is a source adversary $A \in \mathcal{A}$ such that, for every target initial memory $\mu' \in \mathcal{M}'$, we have*

$$\langle\!\langle P, A, \pi(\mu') \rangle\!\rangle \approx \pi \langle\!\langle [\![P]\!], A', \mu' \rangle\!\rangle'$$

Definition 17 (Correctness) $([\![\cdot]\!], \pi)$ *is correct when, for every source program $P \in \mathcal{P}$ and source adversary $A \in \mathcal{A}$, there is a target adversary $A' \in \mathcal{A}'$ such that, for every target initial memory $\mu' \in \mathcal{M}'$, we have*

$$\langle\!\langle P, A, \pi(\mu') \rangle\!\rangle \approx \pi \langle\!\langle [\![P]\!], A', \mu' \rangle\!\rangle'$$

The two definitions differ in their quantification on adversaries. Informally, all attacks must be reflected, but not all of them need to be preserved, so we expect functional correctness only for a well-behaved subset of adversaries, acting for instance as reliable networks and fair schedulers, and we will use a smaller set $\mathcal{A}$ for Definition 17 than for Definition 16.

3.6 Proofs

3.6.1 Proof of Theorem 4

We now prove the theorem. We first state a semantic property of sequential evaluation:

Lemma 27 *Let P and Q be two commands and μ_0 a memory such that $P;Q$ terminates. We have*

$$\rho_\infty(\langle P;Q,\mu_0\rangle) = \sum_\mu \rho_\infty(\langle P,\mu_0\rangle)(\mu) * \rho_\infty(\langle Q,\mu\rangle)$$

PROOF: Let n be a bound on the maximum number of steps for $\langle P,\mu_0\rangle$ to terminate. We proceed by induction on n.

- If $n=0$, we have $P=\surd$, hence $\rho_\infty(\langle P,\mu_0\rangle)=\mu_0$, and by Rule SEQT we have $\rho_\infty(\langle P;Q,\mu_0\rangle) = \rho_\infty(\langle Q,\mu_0\rangle)$.
- Otherwise, decomposing the first reduction step of P, we get $\rho_\infty(\langle P,\mu_0\rangle) = \sum_i p_i\rho_\infty(\langle P_i,\mu_i\rangle)$ for some $k\geq 1$ and p_i, P_i, and μ_i with $i=1..k$ where each $\langle P_i,\mu_i\rangle$ terminates in at most $n-1$ step. By Rule SEQS, we also have $\rho_\infty(\langle P;Q,\mu_0\rangle) = \sum_i p_i\rho_\infty(\langle P_i;Q,\mu_i\rangle)$. By inductive hypothesis, we obtain

$$\begin{aligned}\rho_\infty(\langle P;Q,\mu_0\rangle) &= \sum_i p_i\rho_\infty(\langle P_i;Q,\mu_i\rangle)\\ &= \sum_i p_i \sum_\mu \rho_\infty(\langle P_i,\mu_i\rangle)(\mu) * \rho_\infty(\langle Q,\mu\rangle)\\ &= \sum_\mu (\sum_i p_i\rho_\infty(\langle P_i,\mu_i\rangle)(\mu)) * \rho_\infty(\langle Q,\mu\rangle)\\ &= \sum_\mu \rho_\infty(\langle P,\mu_0\rangle)(\mu) * \rho_\infty(\langle Q,\mu\rangle)\end{aligned}$$

□

In this lemma, we show that a typable command does not influence memory at a level below its type.

Lemma 28 (Containment) *Let Γ be a policy, $\ell\in\mathcal{L}$ a label, P a command, and μ a memory such that P terminates.*

If $\vdash P:\ell$, then $\rho_\infty(\langle P,\mu\rangle) =^\ell \mu$ and for any ℓ' such that $\ell\not\leq\ell'$, we have $\rho_\infty(\langle P,\mu\rangle) =_{\ell'} \mu$.

PROOF: The proof is by induction on the maximum number of reduction steps for $\langle P,\mu\rangle$ to terminate. The base case $P=\surd$ is trivial. The inductive case is by analysis on P:

Case P is $x := e$. By Rules TASSIGN STRICT and TSUBC, we have $\ell\leq\Gamma(x)$, hence $\rho_\infty(\langle P,\mu\rangle) =^\ell \mu$. Additionally, if $\ell\not\leq\ell'$, then $\Gamma(x)\not\leq\ell'$ and $\rho_\infty(\langle P,\mu\rangle) =_{\ell'} \mu$.

Case P is $\widetilde{x} := f(\widetilde{e})$. By Rules TFUN and TSUBC, we have (for all x in $\vec{x}$) $\ell\leq\Gamma(x)$. The conclusion follows as for the case above.

Case P is if e then P_1 else P_2. By rules TCOND and TSUBC, if $\vdash P:\ell'$ then $\vdash P_1:\ell'$ and $\vdash P_2:\ell'$. Also, either $\langle P,\mu\rangle \rightsquigarrow_1 \langle P_1,\mu\rangle$ or $\langle P,\mu\rangle \rightsquigarrow_1 \langle P_2,\mu\rangle$. We conclude by induction on $\langle P_1,\mu\rangle$ and $\langle P_2,\mu\rangle$.

Case P is while e do P'. By rules WHILE, TSUBC, and TSEQ, if $\vdash P:\ell'$ then $\vdash P'$; while e do $P':\ell'$. Also, either $\langle P,\mu\rangle \rightsquigarrow_1 \langle P'$; while e do $P',\mu\rangle$ or $\langle P,\mu\rangle \rightsquigarrow_1 \langle\surd,\mu\rangle$. We conclude by induction on $\langle P'$; while e do $P',\mu\rangle$ and $\langle\surd,\mu\rangle$.

Case P is $P;P'$. By Lemma 27, we have $\rho_\infty(\langle P;P',\mu\rangle) = \sum_{\mu'} \rho_\infty(\langle P',\mu'\rangle) * \rho_\infty(\langle P,\mu\rangle)(\mu')$. We first consider the case $=^\ell$. By inductive hypothesis on $\langle P,\mu\rangle$, $\rho_\infty(\langle P,\mu\rangle) =^\ell \mu$. Hence $\rho_\infty(\langle P,\mu\rangle)(\mu')\neq 0$ if and only if $\mu'=^\ell\mu$. By inductive hypothesis on each of the $\langle P',\mu'\rangle$, $\rho_\infty(\langle P',\mu'\rangle) =^\ell \mu' =^\ell \mu$. We conclude that $\rho_\infty(\langle P;P',\mu\rangle) =^\ell \mu$. Similarly, if $\ell\not\leq\ell'$, then $\rho_\infty(\langle P,\mu\rangle) =_{\ell'} \mu$.

Case P is *skip*. $\langle P, \mu\rangle \rightsquigarrow_1 \langle \surd, \mu\rangle$. We conclude.

□

In the next lemma, we show that the evaluation of a typed expression on two memories equal below its type yield the same values.

Lemma 29 *Let Γ be a policy, $\ell, \ell' \in \mathcal{L}$ two labels, e an expression such that $\vdash e : \ell$, and μ_0 and μ_1 two memories. If $\mu_0 =_\ell \mu_1$, then $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$. If $\ell' \not\leq \ell$ and $\mu_0 =^{\ell'} \mu_1$, then $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$.*

Proof: The proof is by induction on the typing derivation:

- e is x. By typing hypothesis, Rule Var, and Rule SubE, we have $\Gamma(x) \leq \ell$, hence, if $\mu_0 =_\ell \mu_1$, then $\mu_0(x) = \mu_1(x)$. Also, if $\ell' \not\leq \ell$, then $\ell' \not\leq \Gamma(x)$. Hence, if $\mu_0 =^{\ell'} \mu_1$, then $\mu_0(x) = \mu_1(x)$. We conclude.
- e is $op(\widetilde{e})$. We conclude by induction hypothesis on $\vdash e : \ell$ for $e \in \widetilde{e}$ by Rule SubE (Rule SubE commutes with Rule TOp).

□

We now prove our theorem.

Proof of Theorem 4 For the case $\mu_0 =^\ell \mu_1$, if $P \vdash \ell$ then we conclude by Lemma 28 that $\rho_\infty(\langle P, \mu_0\rangle) =^\ell \rho_\infty(\langle P, \mu_1\rangle)$. For the case $\mu_0 =_\ell \mu_1$, if there exists ℓ' such that $\vdash P : \ell'$ and $\ell' \not\leq \ell$ then we conclude with Lemma 28 that $\rho_\infty(\langle P, \mu_0\rangle) =_\ell \rho_\infty(\langle P, \mu_1\rangle)$. In all the other cases, the proof is by induction on the number of reduction steps for P to terminate. We assume that $\vdash P : \ell'$.

Case P is $x := e$. By Rule TAssign Strict, $\vdash e : \ell'$. For the case $\mu_0 =_\ell \mu_1$, we have $\ell' \leq \ell$, hence, by Rule SubE $\vdash e : \ell$ and $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by lemma 29. For the case $\mu_0 =^\ell \mu_1$, we have $\ell' \not\leq \ell$, hence $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by lemma 29. We conclude by the semantic definition Assign.

Case P is if e then P_1 else P_2. By rule TCond, $\vdash e : \ell'$, similarly $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by lemma 29. Thus, $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) = b$ for some $b \in \{0, 1\}$ and

$$\rho_\infty(\langle \text{if } e \text{ then } P_1 \text{ else } P_2, \mu_0\rangle) = \rho_\infty(\langle P_b, \mu_0\rangle)$$
$$\rho_\infty(\langle \text{if } e \text{ then } P_1 \text{ else } P_2, \mu_1\rangle) = \rho_\infty(\langle P_b, \mu_1\rangle)$$

We conclude by induction on $\langle P_b, \mu_0\rangle$ and $\langle P_b, \mu_1\rangle$.

Case P is while e do P'. By rule TWhile, $\vdash e : \ell'$, similarly $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by lemma 29. Thus, either

- $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) = 0$ and

$$\rho_\infty(\langle \text{while } e \text{ do } P', \mu_0\rangle) = \mu_0$$
$$\rho_\infty(\langle \text{while } e \text{ do } P', \mu_1\rangle) = \mu_1$$

  and we conclude from hypothesis $\mu_0 =_\ell \mu_1$;
- or $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) \neq 0$ and

$$\rho_\infty(\langle \text{while } e \text{ do } P', \mu_0\rangle) = \rho_\infty(\langle P'; \text{while } e \text{ do } P', \mu_0\rangle)$$
$$\rho_\infty(\langle \text{while } e \text{ do } P', \mu_1\rangle) = \rho_\infty(\langle P'; \text{while } e \text{ do } P', \mu_1\rangle)$$

  and we conclude by induction on $\langle P'; \text{while } e \text{ do } P', \mu_0\rangle$ and $\langle P'; \text{while } e \text{ do } P', \mu_1\rangle$.

Case P is $P_0; P_1$. We first consider the case $=_\ell$. We write

$$\begin{aligned}\rho_0 = &\quad \rho_\infty(\langle P_0, \mu_0\rangle)\\ \rho_1 = &\quad \rho_\infty(\langle P_0, \mu_1\rangle)\end{aligned}$$

By induction on P_0, we know that $\rho_0 =_\ell \rho_1$. By definition of low-equality on distributions, for every μ_i, p_i such that $\rho_{0|\ell}(\mu_i) = p_i$, there exist $(\mu^0_{i,j})_{j\leq n^0_i}$ and $(p^0_{i,j})_{j\leq n^0_i}$ such that

- $\mu^0_{i,j} =_\ell \mu_i$,
- $\rho_0(\mu^0_{i,j}) = p^0_{i,j}$,
- $\sum_j p^0_{i,j} = p_i$

Similarly, there exist $(\mu^1_{i,j})_{j\leq n^1_i}$ and $(p^1_{i,j})_{j\leq n^1_i}$ such that

- $\mu^0_{i,j} =_\ell \mu^1_{i,j} =_\ell \mu_i$,
- $\rho_0(\mu^0_{i,j}) = p^0_{i,j}$,
- $\rho_1(\mu^1_{i,j}) = p^1_{i,j}$,
- $\sum_j p^0_{i,j} = \sum_j p^1_{i,j} = p_i$

By Lemma 27,

$$\begin{aligned}\rho_\infty(\langle P_0; P_1, \mu_0\rangle) &= \textstyle\sum_\mu \rho_0(\mu) * \rho_\infty(\langle P_1, \mu\rangle)\\ &= \textstyle\sum_i \sum_j \rho_\infty(\langle P_1, \mu_{i,j}\rangle) * p^0_{i,j}\\ \rho_\infty(\langle P_0; P_1, \mu_1\rangle) &= \textstyle\sum_\mu \rho_1(\mu) * \rho_\infty(\langle P_1, \mu\rangle)\\ &= \textstyle\sum_i \sum_j \rho_\infty(\langle P_1, \mu_{i,j}\rangle) * p^1_{i,j}\end{aligned}$$

For every couple (i, j), we have $\rho_\infty(\langle P_1, \mu^0_{i,j}\rangle) =_\ell \rho_\infty(\langle P_1, \mu^1_{i,j}\rangle)$ and we conclude. We conclude similarly for the case $=^\ell$

□

3.6.2 Proof of Theorem 5

We first update our containment lemma for active adversaries.

Lemma 30 (Containment against active adversary) *Let Γ be a policy, P a command context , $\ell, \ell', \alpha \in \mathcal{L}$ three labels, $\widetilde{A}$ an α-adversary, and μ a memory such that $P[A]$ terminates.*

If $\vdash P : \ell$, and $C(\ell) \not\leq C(\alpha)$ or $I(\ell) \leq I(\alpha)$, then $\rho_\infty(\langle P[\widetilde{A}], \mu\rangle) =^\ell \mu$. Additionally, if $\vdash P : \ell$, $\ell \not\leq \ell'$, and $C(\alpha) \leq C(\ell)$ or $I(\alpha) \not\leq I(\ell)$, then $\rho_\infty(\langle P[\widetilde{A}], \mu\rangle) =_{\ell'} \mu$.

Proof: The proof is by induction on the maximum number of reduction steps for $\langle P[\widetilde{A}], \mu\rangle$ to terminate. The base case $P = \surd$ is trivial. The inductive case is by analysis on P. The proof is similar to the one of Theorem 28, so we only consider the additional case

Case $P[\widetilde{A}]$ is A. By Rules TAdversary and TSubC, for the first case, either $C(\ell) \not\leq C(\alpha)$ in which case $rv(A) \subseteq \{x \mid \ell \not\leq \Gamma(x)\}$, hence $\rho_\infty(\langle A, \mu\rangle) =^\ell \mu$; or $I(\ell) \leq I(\alpha)$ in which case $wv(A) \cap \{x \mid \ell \not\leq \Gamma(x)\} = \emptyset$, hence $\rho_\infty(\langle A, \mu\rangle) =^\ell \mu$. By Rules TAdversary and TSubC, for the second case, either $C(\alpha) \leq C(\ell)$ in which case $rv(A) \in \{x \mid \Gamma(x) \leq \ell\}$, hence $\rho_\infty(\langle A, \mu\rangle) =_\ell \mu$; or $I(\alpha) \not\leq I(\ell)$ in which case $wv(A) \cap \{x \mid \Gamma(x) \leq \ell\} = 0$, hence $\rho_\infty(\langle A, \mu\rangle) =_\ell \mu$. We conclude.

□

PROOF OF THEOREM 5 If $P \vdash \ell$ and $\mu_0 =^{\ell} \mu_1$, then, we conclude with Lemma 28 that $\rho_\infty(\langle P[\widetilde{A}], \mu_0\rangle) =^{\ell} \rho_\infty(\langle P[\widetilde{A}], \mu_1\rangle)$. If there exist ℓ' such that $\vdash P : \ell'$, $\ell' \not\leq \ell$, and $\mu_0 =_\ell \mu_1$, then we conclude with Lemma 28 that $\rho_\infty(\langle P[\widetilde{A}], \mu_0\rangle) =_\ell \rho_\infty(\langle P[\widetilde{A}], \mu_1\rangle)$. In all the other cases, the proof is by induction on the number of reduction steps for $P[\widetilde{A}]$ to terminate. We suppose that $\vdash P : \ell'$. The proof is similar to the one of Theorem 4 (with the updated containment lemma), so we only consider the additional case

Case $P[\widetilde{A}]$ is A. The case is the same than in Lemma 30.

□

3.6.3 Proof of Theorem 6

PROOF: In Rules TASSIGN ENDORSE and TASSIGN ENDORSE, the type of the command is the type of the variable being written (as in TASSIGN STRICT). Hence, the proof of Lemma 28 works as well with the lax type system. □

3.6.4 Proof of Theorem 7

PROOF OF THEOREM 7 If $\vdash P : (c, H)$, then, we conclude with Theorem 6 that $\rho_\infty(\langle P, \mu_0\rangle) =^{(c,H)} \rho_\infty(\langle P, \mu_1\rangle)$. Otherwise, for every ℓ such that $\vdash P : \ell$, we have $(c, H) \not\leq \ell$ and the proof is by induction on the number of reduction steps for P to terminate. The proof is similar to the one of Theorem 4, so we only consider the cases that differ:

Case P is $x := e$. Type with Rule TASSIGN ENDORSE. By typing rule TASSIGN ENDORSE, $\vdash e : (C(\ell), i)$ for some i. Hence $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by lemma 29. We conclude by the semantic definition ASSIGN.

Case P is $x := e$. Type Rule TASSIGN ROBUST. By typing rule TASSIGN ROBUST, $\vdash e : (c', i)$ (for some c' and i'), with $I(\ell) \leq R(c')$. Since R is monotonic and $I(\ell) \not\leq R(c)$, we have $c \not\leq c'$. Hence $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by lemma 29. We conclude by the semantic definition ASSIGN.

□

3.6.5 Proof of Theorem 8

We first extend our definitions of $=_\ell$ and $=^{\ell}$ to deal with dynamic policies. We write $\mu_0 =_{\ell,\gamma} \mu_1$ (respectively $\mu_0 =^{\ell,\gamma} \mu_1$) when $\mu_0 =_\ell \mu_1$ (respectively $\mu_0 =^{\ell} \mu_1$) with policy γ.

In the next lemma, we prove that commands of high confidentiality do not contain command variables.

Lemma 31 *Let Γ be a policy, $\overline{P}$ a semi-active command context, $\ell, \alpha \in \mathcal{L}$ two labels.*

If $\vdash \overline{P} : \ell$ and $C(\ell) \not\leq C(\alpha)$, then $\overline{P}$ does not contain command variables.

PROOF: The proof is by induction on the typing of $\overline{P}$, and deduces from the fact that the only rule to type $[\_]_{d,n}$ is Rule TSADVERSARY (and TSSUBC). We have $\ell \not\leq (C(\alpha), L)$, hence $\not\vdash [\_]_{d,n} : \ell$. □

In the next lemma, we show a property of containment for confidentiality. Also, we show a property on the set of variables readable that the adversary such that $=_{(C(\alpha),L),\gamma} \subseteq =_{(C(\alpha),L),\gamma'}$.

Lemma 32 (Containment-Confidentiality) *Let Γ be a policy, P a command, $\ell, \alpha \in \mathcal{L}$ two labels. Let μ be a memory such that P terminates.*

If $\gamma \vdash P : \ell, \gamma'$, $C(\ell) \not\leq C(\alpha)$, then $\{x \mid C(\gamma'(x)) \leq C(\alpha)\} \subseteq \{x \mid C(\gamma(x)) \leq C(\alpha)\}$, and

$$\rho_\infty(\langle P, \mu\rangle) =_{(C(\alpha),L),\gamma'} \mu$$

Proof: The proof is by induction on the maximum number of reduction steps for $\langle P, \mu\rangle$ to terminate. The base case $P = \surd$ is trivial. The proof is similar to the one of Lemma 28, so we only consider the cases that differ:

Case P is $x := e$. By Rules TsAssign and TsSubC, we have $\ell \leq \gamma'(x)$, hence $C(\gamma'(x)) \not\leq C(\alpha)$. Hence $\{x \mid C(\gamma'(x)) \leq C(\alpha)\} \subseteq \{x \mid C(\gamma(x)) \leq C(\alpha)\}$ and $\rho_\infty(\langle P, \mu\rangle) =_{(C(\alpha),L),\gamma'} \mu$.

Case P is $\widetilde{x} := f(\widetilde{e})$. By Rules TsFun and TsSubC, we have $C(\ell) = L$. Our hypothesis states $C(\ell) \not\leq C(\alpha)$, this is not possible.

□

In the next lemma, we show a property of containment for integrity. Also, we show a property on the set of variables that are not writable by the adversary such that, for instance, $=^{(L,I(\alpha)),\gamma} \subseteq =^{(L,I(\alpha)),\gamma'}$.

Lemma 33 (Containment-Integrity) *Let Γ be a policy, $\overline{P}$ be a semi-active command context, $\ell, \alpha \in \mathcal{L}$ two labels. Let A be a semi-active α-adversary . Let A' be an α-adversary. Let μ be a memory such that $\overline{P}[A]$ and $[\![\overline{P}]\!][A']$ terminates.*

If $\gamma \vdash \overline{P} : \ell, \gamma'$ and $I(\alpha) \leq I(\ell)$, then $\{x \mid I(\alpha) \not\leq I(\gamma'(x))\} \subseteq \{x \mid I(\alpha) \not\leq I(\gamma(x))\}$, and

$$\rho_\infty(\langle [\![\overline{P}]\!][A'], \mu\rangle) =^{(L,I(\alpha)),\gamma'} \rho_\infty(\langle \overline{P}[A], \mu\rangle) =^{(L,I(\alpha)),\gamma'} \mu$$

Proof: The proof is by induction on the maximum number of reduction steps for $[\![\overline{P}]\!][A']$ (and $\langle \overline{P}[A], \mu\rangle$) to terminate. The base case $\overline{P}[A] = \surd$ is trivial. The proof is similar to the one of Lemma 28, so we only consider the cases that differ:

Case $\overline{P}$ is $x := e$. By Rules TsAssign and TsSubC, we have $\ell \leq \gamma'(x)$, hence and $I(\alpha) \leq I(\gamma'(x))$. Hence $\{x \mid I(\alpha) \not\leq I(\gamma'(x))\} \subseteq \{x \mid I(\alpha) \not\leq I(\gamma(x))\}$ and $\rho_\infty(\langle [\![\overline{P}]\!][A'], \mu\rangle) =^{(L,I(\alpha)),\gamma'} \rho_\infty(\langle \overline{P}[A], \mu\rangle) =^{(L,I(\alpha)),\gamma'} \mu$.

Case $\overline{P}$ is $\widetilde{x} := f(\widetilde{e})$. $\ell \leq \gamma'(x)$, we conclude similarly.

Case $\overline{P}[A']$ is A' (and $\overline{P}[A]$ is A). By Rule TsAdversary and TsSubC, $I(\gamma) \leq_{\overline{n}} I(\gamma')$ and $I(\alpha) \leq I(\ell) \leq \sqcap_{x \in n} I(\gamma'(x))$, hence $\{x \mid I(\alpha) \not\leq I(\gamma'(x))\} \subseteq \{x \mid I(\alpha) \not\leq I(\gamma(x))\}$. Moreover, $I(\Gamma) \leq_{\overline{n}} I(gamma')$ and $I(\alpha) \leq I(\ell) \leq \sqcap_{x \in n} I(\gamma'(x))$. Also, $wv(A') \cap \{x \mid \alpha \not\leq \Gamma(x)\} = \emptyset$ (and $wv(A) \cap \{x \mid \alpha \not\leq \Gamma(x)\} = \emptyset$), hence $\rho_\infty(\langle [\![\overline{P}]\!][A'], \mu\rangle) =^{(L,I(\alpha)),\gamma'} \rho_\infty(\langle \overline{P}[A], \mu\rangle) =^{(L,I(\alpha)),\gamma'} \mu$.

□

In the next lemma, we prove our invariant for the theorem. We prove that for every adversary, there exist a semi-active adversary plus a command on adversary intermediary variables such that the command have the same semantic on some part of the memory.

Lemma 34 *Let $\overline{P}$ be a semi-active command context. Let Γ be a policy. Let ℓ be a label. Let γ and γ' be two security policies such that $\gamma \vdash \overline{P} : \ell, \gamma'$. Let $\alpha \in \mathcal{L}$ be such that R is robust against α. Let $\widetilde{A}$ be an α-adversary. For every variable x, let x_{copy} be a fresh variable, with $x_{copy} \notin \mathrm{dom}(\Gamma)$. For every policy γ°, let $F(\gamma^\circ)$ be an endomorphism on memories (and, by extension, on distributions) defined on every memory μ by:*

$$\begin{array}{lll} F(\gamma^\circ)(\mu)(x) & = \mu(x_{copy}) & \text{if } C(\gamma^\circ(x)) \leq C(\alpha) \\ & = \mu(x) & \text{otherwise} \end{array}$$

Let A_{copy} be an adversary command such that $v(A_{copy}) \cap \mathrm{dom}(\Gamma) = \emptyset$. There exists a semi-active α-adversary A' (if $\gamma \vdash \overline{P} : \ell', \gamma'$ with $I(\alpha) \leq I(\ell')$, $A' = \mathrm{skip}$) and an adversary command A'_{copy} with $v(A'_{copy}) \notin \mathrm{dom}(\Gamma)$ such that for every memory μ_0 such that $[\![\overline{P}]\!][\widetilde{A}]$ terminates and μ_1 such that $F(\gamma)(\rho_\infty\langle A_{copy}, \mu_0\rangle) =_{(C(\alpha),L),\gamma} \mu_1$ and $\rho_\infty\langle A_{copy}, \mu_0\rangle =^{(L,I(\alpha)),\gamma} \mu_1$ then $\overline{P}[\widetilde{A'}]; A'_{copy}$ terminates on μ_1 and

$$\begin{array}{rll} F(\gamma')(\rho_\infty\langle \overline{P}[\widetilde{A'}]; A'_{copy}, \mu_0\rangle) & =_{(C(\alpha),L),\gamma'} & \rho_\infty\langle [\![\overline{P}]\!][\widetilde{A}], \mu_1\rangle \\ \rho_\infty\langle \overline{P}[\widetilde{A'}]; A'_{copy}, \mu_0\rangle & =^{(L,I(\alpha)),\gamma'} & \rho_\infty\langle [\![\overline{P}]\!][\widetilde{A}], \mu_1\rangle \end{array}$$

Proof: The proof is by structural induction on $\overline{P}$. The basis case (*skip*) is trivial. We first provide a command A'_{copy} and prove the equality on $=_{(C(\alpha),L),\gamma'}$ when $[\![\overline{P}]\!]$ is not a placeholder. A_{copy} can be easily swapped with commands that do not share any variable with him. If $\gamma \vdash \overline{P} : \ell', \gamma'$ with $C(\ell') \not\leq C(\alpha)$ then, by Lemma 31, $\overline{P}[\widetilde{A'}]$ contains no command variables, we can write it P, and we have $P = \overline{P}[\widetilde{A'}] = [\![\overline{P}]\!][\widetilde{A}]$. Also, choosing $A'_{copy} = A_{copy}$, by Lemma 32, we have

$$\begin{array}{ll} F(\gamma')(\rho_\infty\langle P; A'_{copy}, \mu_0\rangle) & =_{(C(\alpha),L),\gamma'} F(\gamma)(\rho_\infty\langle A_{copy}, \mu_0\rangle) \\ & =_{(C(\alpha),L),\gamma'} \mu_1 \\ & =_{(C(\alpha),L),\gamma'} \rho_\infty\langle P, \mu_1\rangle \end{array}$$

Otherwise, we now assume that $\gamma \vdash \overline{P} : \ell', \gamma'$ with $C(\ell') \leq C(\alpha)$, and without using the Rule TsSubC at top level.

Case $\overline{P}[\_]$ is $x := e$. By typing Rule TsAssign, $\vdash e : \ell'$, with $\gamma'(x) = \ell'$. We let e_{copy} be e where every variable y is replaced by y_{copy}. We have $F(\gamma)(\mu_0) =_{(C(\alpha),L),\gamma} \mu_1$, also $v(e) \subseteq \{x \mid C(\gamma(x)) \leq C(\alpha)\}$, hence $[\![e_{copy}]\!](\mu_0) = [\![e]\!](\mu_1)$ by Lemma 29. Finally, choosing $A'_{copy} = A_{copy}; x_{copy} := e_{copy}$, and since $\gamma'(x) \leq C(\alpha)$ and γ' is unchanged for every other variable, we conclude by the semantic definition Assign:

$$F(\gamma')(\rho_\infty\langle x := e; A'_{copy}, \mu_0\rangle) =_{(C(\alpha),L),\gamma'} \rho_\infty\langle x := e, \mu_1\rangle$$

Case $\overline{P}[\_]$ is if e then $P_1^-[\_]$ else $P_2^-[\_]$. By rule TCond, $\gamma \vdash P_1^-[\_] : \ell', \gamma'$ and $\vdash P_2^-[\_] : \ell'$. Let $\widetilde{A_1}$ and $\widetilde{A_2}$ be the two part of the adversary that corresponds to the two branches. By induction, there exist A'_1, A_{copy1} and A'_2, A_{copy2} such that, for $b \in \{1, 2\}$: For every memory μ_0 and μ_1 such that $F(\gamma)(\mu_0) =_{(C(\alpha),L),\gamma} \mu_1$ and $\mu_0 =^{(L,I(\alpha)),\gamma} \mu_1$ and $[\![P_b^-]\!][A_b]$ terminates on μ_1, $F(\gamma')(\rho_\infty\langle P_b^-[\widetilde{A'_b}]; A_{copyb}, \mu_0\rangle) =_{(C(\alpha),L),\gamma'} \rho_\infty\langle [\![P_b^-]\!][\widetilde{A_b}], \mu_1\rangle$.

We let $A'_{copy} = \text{if } e_{copy} \text{ then } A_{copy1} \text{ else } A_{copy2}$. $\vdash e : \ell'$, hence, similarly to the item above $[\![e_{copy}]\!](\mu_0) = [\![e]\!](\mu_1)$ by lemma 29.

- If $I(\alpha) \leq I(\gamma(\ell'))$, $A'_1 = \mathit{skip} = A'_2$, hence $v(P_b^-[\widetilde{A'_b}])$ do not contain any of the $*_{copy}$ variables, which are the only variables compared by $=_{(C(\alpha),L),\gamma'}$. Hence, we have $F(\gamma')(\rho_\infty\langle P_b^-[\widetilde{A'_b}]; A_{copyb}, \mu_0\rangle) =_{(C(\alpha),L),\gamma'} F(\gamma')(\rho_\infty\langle A_{copyb}, \mu_0\rangle)$. We have $[\![e_{copy}]\!](\mu_0)$

$= [\![e]\!](\mu_1) = b$ for some $b \in \{0,1\}$ and

$$\begin{array}{lll} F(\gamma')(\rho_\infty(\langle \text{if } e \text{ then } P_1^-[\widetilde{A'_1}] \text{ else } P_2^-[\widetilde{A'_2}]; A'_{copy}, \mu_0\rangle)) & & \\ & =_{(C(\alpha),L),\gamma'} & F(\gamma')(\rho_\infty(\langle A'_{copy}, \mu_0\rangle)) \\ & =_{(C(\alpha),L),\gamma'} & F(\gamma')(\rho_\infty(A'_{copyb}, \mu_0) \\ & =_{(C(\alpha),L),\gamma'} & F(\gamma')(\rho_\infty(\langle P_b^-[\widetilde{A'_b}]; A'_{copyb}, \mu_0\rangle)) \\ & =_{(C(\alpha),L),\gamma'} & \rho_\infty(\langle P_b^-[\widetilde{A'_b}], \mu_1\rangle) \\ & =_{(C(\alpha),L),\gamma'} & \rho_\infty(\langle \text{if } e \text{ then } P_1^-[\widetilde{A'_1}] \text{ else } P_2^-[\widetilde{A'_2}], \mu_1\rangle) \end{array}$$

We conclude.

- If $I(\alpha) \not\leq I(\gamma(\ell'))$, and since $\rho_\infty\langle A_{copy}, \mu_0\rangle =^{(L,I(\alpha)),\gamma} \mu_1$, $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by lemma 29. Thus $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) = [\![e_{copy}]\!](\mu_0) = b$ for some $b \in \{0,1\}$ and

$$\rho_\infty(\langle \text{if } e \text{ then } P_1^-[\widetilde{A'_1}] \text{ else } P_2^-[\widetilde{A'_2}]; A'_{copy}, \mu_0\rangle) = \rho_\infty(\langle P_b^-[\widetilde{A'_b}]; A'_{copyb}, \mu_0\rangle)$$
$$\rho_\infty(\langle \text{if } e \text{ then } P_1^-[\widetilde{A'_1}] \text{ else } P_2^-[\widetilde{A'_2}], \mu_1\rangle) = \rho_\infty(\langle P_b^-[\widetilde{A'_b}], \mu_1\rangle)$$

We conclude by induction.

Case P is while e do P. We conclude similarly.

Case P is $P_0^-[\_]; P_1^-[\_]$. There exists a policy γ'' such that

$$\gamma \vdash P_0^-[\_] : \ell, \gamma''$$
$$\gamma'' \vdash P_1^-[\_] : \ell, \gamma'$$

Hence, by induction, there exist A'_0, A'_{copy0} such that

$$F(\gamma'')(\rho_\infty\langle P_0^-[\widetilde{A'_0}]; A'_{copy0}, \mu_0\rangle) =_{(C(\alpha),L),\gamma''} \rho_\infty\langle [\![P_0^-]\!][\widetilde{A_0}], \mu_1\rangle$$
$$\rho_\infty\langle \overline{P}[\widetilde{A'_0}]; A'_{copy0}, \mu_0\rangle =^{(L,I(\alpha)),\gamma''} \rho_\infty\langle [\![\overline{P}]\!][\widetilde{A_0}], \mu_1\rangle$$

Also by induction, there exist A'_1, A'_{copy1}, such that for every μ'_0 and μ'_1 such that $F(\gamma'')(\rho_\infty\langle A_{copy0}, \mu'_0\rangle) =_{(C(\alpha),L),\gamma''} \mu'_1$ and $\rho_\infty\langle A_{copy0}, \mu'_0\rangle =^{(L,I(\alpha)),\gamma''} \mu'_1$

$$F(\gamma')(\rho_\infty\langle P_1^-[\widetilde{A'_1}]; A'_{copy1}, \mu'_0\rangle) =_{(C(\alpha),L),\gamma'} \rho_\infty\langle [\![P_1^-]\!][\widetilde{A_1}], \mu'_1\rangle$$
$$\rho_\infty\langle \overline{P}[\widetilde{A'_1}]; A'_{copy1}, \mu'_0\rangle =^{(L,I(\alpha)),\gamma'} \rho_\infty\langle [\![\overline{P}]\!][\widetilde{A_1}], \mu'_1\rangle$$

We conclude using Lemma 27 in a technique similar to the one used for the other non-interference theorems.

We now prove the equality on $=^{(L,I(\alpha)),\gamma'}$ when $[\![\overline{P}]\!]$ is not a placeholder. If $\gamma \vdash \overline{P} : \ell', \gamma'$ with $I(\alpha) \leq I(\ell)$, then by Lemma 33, we have

$$\begin{array}{lll} \rho_\infty\langle \overline{P}[A']; A'_{copy}, \mu_0\rangle & =^{(L,I(\alpha)),\gamma'} & \rho_\infty\langle A_{copy}, \mu_0\rangle \\ & =^{(L,I(\alpha)),\gamma'} & \mu_1 \\ & =^{(L,I(\alpha)),\gamma'} & \rho_\infty\langle [\![\overline{P}]\!][A], \mu_1\rangle \end{array}$$

Otherwise, we now assume that $\gamma \vdash \overline{P} : \ell', \gamma'$ with $I(\alpha) \not\leq I(\ell)$, and without using the Rule TsSubC at top level. Most of the cases are standard, thus not explored here.

Case $\overline{P}[\_]$ is $x := e$. By typing Rule TsAssign, $\vdash e : \ell'$, with $\gamma'(x) = \ell'$. $I(\alpha) \not\leq I(\ell)$, hence $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by Lemma 29. We conclude:

$$\rho_\infty\langle \overline{P}[\widetilde{A'}]; A'_{copy}, \mu_0\rangle =^{(L,I(\alpha)),\gamma'} \rho_\infty\langle [\![\overline{P}]\!][\widetilde{A}], \mu_1\rangle$$

We now consider the last case:

Case $\overline{P}[\_]$ is $[\_]_{d,n}$. We recall the typing Rule TsAdversary:

$$\frac{\begin{array}{c} I(\gamma') =_{\overline{n}} I(\Gamma) \sqcup I(\gamma) \\ I(\gamma) \leq R(C(\gamma)) \\ C(\gamma) =_{\overline{d}} C(\Gamma) \sqcap C(\gamma') \\ C(\gamma') =_{d} C(\Gamma) \\ i = \sqcap_{x\in n} I(\gamma'(x)) \sqcap_{x\in d} R(C(\gamma(x))) \end{array}}{\gamma \vdash [\_]_{d,n} : (C(\alpha), i), \gamma'}$$

We let A_{root} be A where every assignment to a variable x such that $x \notin n$ and $C((\Gamma \sqcap \gamma)(x)) \not\leq C(\alpha)$ is removed; and every variable x such that $C((\Gamma \sqcap \gamma)(x)) \not\leq C(\alpha)$ is replaced by its copy x_{copy}. We let $A' \doteq A_{copy}; (x_{copy} := x;)_{x\in d|C(\Gamma(x))\leq C(\alpha)} A_{root}; (x := x_{copy};)_{x\in n|C((\Gamma\sqcap\gamma)(x))\leq C(\alpha)}$ and $_{I(\alpha)\leq I(\Gamma(x))}\}$ and $A'_{copy} \doteq skip$. Let $x \in \operatorname{dom}\Gamma$.

- If $C(\gamma(x)) \leq C(\alpha)$, by hypothesis, $\rho_\infty\langle A_{copy}, \mu_0\rangle(x_{copy}) = \mu_1(x)$
- Otherwise, by robustness hypothesis on γ, $I(\alpha) \not\leq I(\gamma(x))$, hence $\rho_\infty\langle A_{copy}, \mu_0\rangle(x) = \mu_1(x)$

We then copy the values from d, hence we have

$$F(\Gamma \sqcap \gamma)(\rho_\infty\langle A_{copy}; (x_{copy} := x;)_{x\in d|C(\Gamma(x))\leq C(\alpha)}, \mu_0\rangle) = \mu_1$$

We can weaken this result in:

$$F(\Gamma \sqcap \gamma)(\rho_\infty\langle A_{copy}; (x_{copy} := x;)_{x\in d|C(\Gamma(x))\leq C(\alpha)}, \mu_0\rangle) =_{(C(\alpha),L),\Gamma\sqcap\gamma} \mu_1$$

$$F(\Gamma \sqcap \gamma)(\rho_\infty\langle A_{copy}; (x_{copy} := x;)_{x\in d|C(\Gamma(x))\leq C(\alpha)}, \mu_0\rangle) =^{(L,I(\alpha)),\gamma'} \mu_1$$

The command A and A_{root} perform the same operations on both memory (except for assignments to variables in variables x such that $x \notin n$ and $C((\Gamma \sqcap \gamma)(x)) \not\leq C(\alpha)$). Since $x \notin n$ and $x \in wv(A)$ implies $I(\alpha) \leq \gamma'(x)$, we can verify that the invariant is conserved:

$$F(\Gamma \sqcap \gamma)(\rho_\infty\langle A_{copy}; (x_{copy} := x;)_{x\in d|C(\Gamma(x))\leq C(\alpha)} A_{root}, \mu_0\rangle) =_{(C(\alpha),L),\Gamma\sqcap\gamma} \rho_\infty\langle A, \mu_1\rangle$$

$$F(\Gamma \sqcap \gamma)(\rho_\infty\langle A_{copy}; (x_{copy} := x;)_{x\in d|C(\Gamma(x))\leq C(\alpha)} A_{root}, \mu_0\rangle) =^{(L,I(\alpha)),\gamma'} \rho_\infty\langle A, \mu_1\rangle$$

The first invariant implies

$$F(\gamma')(\rho_\infty\langle \overline{P}[\widetilde{A'}]; A'_{copy}, \mu_0\rangle) =_{(C(\alpha),L),\gamma'} \rho_\infty\langle [\![\overline{P}]\!][\widetilde{A}], \mu_1\rangle$$

since $\Gamma \sqcap \gamma \leq \gamma'$. The second invariant, after copy of the endorsed variables implies

$$\rho_\infty\langle \overline{P}[\widetilde{A'}]; A'_{copy}, \mu_0\rangle =^{(L,I(\alpha)),\gamma'} \rho_\infty\langle [\![\overline{P}]\!][\widetilde{A}], \mu_1\rangle$$

If $\gamma \vdash \overline{P} : \ell', \gamma'$ with $I(\alpha) \leq I(\ell')$, $\{x \in d \mid C(\Gamma(x)) \leq C(\alpha)\} = \emptyset$ and $\{x \in n \mid I(\alpha) \leq I(\Gamma(x))\} = \emptyset$. $v(A') \notin \operatorname{dom}(\Gamma)$. We can set instead $A'_{copy} = A'$ and $A' = skip$.

□

Proof of Theorem 8 By hypothesis, there exists a level ℓ such that $\gamma_0 \vdash \overline{P} : \ell, \gamma_0$. By Lemma 34, for every α-adversary A such that $[\![\overline{P}]\!][A]$ terminates (we take $A_{copy} = skip$), there exist a semi-active α-adversary A' and an adversary command A'_{copy} with $v(A'_{copy}) \notin \operatorname{dom}(\Gamma)$ such that $[\![\overline{P}]\!][\widetilde{A}]; A'_{copy}$ terminates and for every memory μ_0 and μ_1 such that $F(\gamma_0)(\rho_\infty\langle A_{copy}, \mu_0\rangle) =_{(C(\alpha),L),\gamma_0} \mu_1$ and $\mu_0 =^{(L,I(\alpha)),\gamma_0} \mu_1$ then

$$F(\gamma_0)(\rho_\infty\langle \overline{P}[\widetilde{A'}]; A'_{copy}, \mu_0\rangle) =_{(C(\alpha),L),\gamma_0} \rho_\infty\langle [\![\overline{P}]\!][\widetilde{A}], \mu_1\rangle$$

$$\rho_\infty\langle\overline{P}[\widetilde{A'}];A'_{copy},\mu_0\rangle =^{(L,I(\alpha)),\gamma_0} \rho_\infty\langle[\![\overline{P}]\!][\widetilde{A}],\mu_1\rangle$$

We have $F(\gamma_0) = Id$. We can take $\mu_0 = \mu_1$. Also, since $\alpha \not\leq \perp_I$, the equality $=^{(L,I(\alpha)),\gamma_0}$ is an equality on all variables of $\mathrm{dom}(\Gamma)$. Furthermore, $v(A'_{copy}) \notin \mathrm{dom}(\Gamma)$, hence

$$\rho_\infty\langle\overline{P}[\widetilde{A'}],\mu\rangle =_{\mathrm{dom}(\Gamma)} \rho_\infty\langle[\![\overline{P}]\!][\widetilde{A}],\mu\rangle$$

□

4

Cryptographic Information Flow Compiler

In the chapter above, we describe how to specify (and sometimes verify) the security of programs using information flow annotations. In the following chapters, we are interested in the implementation of those programs in a system composed of multiple machines that do not necessarily trust each other and that communicate trough an untrusted network. If one machine has the required level of trust to read an write all the variables of the program, and if they are stored locally, then the implementation is trivial. However, this is generally not the case, and the implementation may need to be distributed on several of these machines ("units of trust"). For example, a program handling secrets of a client and a bank may be partly running on the client machine, an partly running on the bank machine with secure communications between the two.

Jif/Split [Zdancewic et al., 2002, Zheng et al., 2003] and Swift [Chong et al., 2009] automatically partition information flow annotated distributed programs into local code, each running at a given security level, representing the level of trust granted to each host in a distributed system. However, the partitioned code still assumes secure channels between the participants. It is sometimes possible to implement the security of these channels using protection mechanisms based on cryptography. However, their hand-writing is difficult and error-prone, and traffic analysis or active attacks may be hard to deal with. Fournet et al. [2009] automatically generate the required cryptographic mechanisms to secure communications between the participants as well as the control flow. More precisely, they compile imperative programs with security and locality annotations to distributed programs using cryptography to preserve information-flow security under standard cryptographic assumptions and against an active adversary that can control the network and the scheduling of the program.

We describe this compiler, giving its specification and outlining its algorithms; we refer to Fournet et al. [2009] for a precise description of the compiler. We slightly improve on their work by handling more source programs (e.g., programs with endorsement), and by providing more precise theorems based on the definitions of Section 3.5.

4.1 Cryptographic Information Flow Compiler

4.1.1 Source language, with locations

We consider a finite set of hosts, or locations, $\{1, 2, \ldots, i, b, c, v, \ldots, n\}$ intended to represent units of trust (principals) and of locality (runtime environments). The source language is the language of Section 6.2 extended with host annotations:

$$P ::= \ldots \mid b : P$$

The locality command $b : P$ states that command P should run on host b. Locality commands can be nested, as in $c : \{P_c; v : P_v\}$. We assume that every source program has a locality command at top level, setting an initial host. Variables are transparently shared between hosts, hence locality annotations do not affect our semantics for commands.

4.1.2 Typing locality commands rules

We extend our security policy to assign a level $\Gamma(b)$ to each host b; this level indicates which variables b can read and write. We only consider robust hosts, such that $I(b) \leq R(C(b))$. In this chapter, we use the lax type system of Figure 3.3, with the additional typing rule for localities:

$$\frac{\vdash P : \ell \qquad I(b) \leq_I I(\ell)}{\vdash (b : P) : \ (\bot, I(\ell))} \ (\text{TLocality})$$

The rule states that locality commands are public, thereby reflecting that the transfer of control between hosts can be observed by the adversary.

We illustrate CFLOW with the example of a loan application. The source code and its policy specify levels of protection, but leave the choice of cryptographic mechanisms to the compiler. The actual source and compiled programs are available online.

Example 21 (Applying for a loan: source code) *Consider a program involving two parties, a bank that offers loans, and a client that wishes to apply for a loan without disclosing private information (at least until the loan is granted). Suppose also that the bank does not want to disclose the parameters used for evaluating loan applications. In this chapter we assume a third participant, trusted to securely run the loan-evaluation code. The code of this computation is:*

b: $\{x_b := e_b\}$; c:$\{y_c := e_c\}$;
v: $\{x'_b, y'_c := f(x_b, y_c)\}$;
b: $\{print(x'_b)\}$; c: $\{print(y'_c)\}$

It involves three hosts: the client c with $\Gamma(c) = (C_c, I_c)$, the bank b with $\Gamma(b) = (C_b, I_b)$, and the trusted host v with $\Gamma(v) = (C_c \sqcup C_b, I_c \sqcap I_b)$. All variables indexed by b, c or v are private to b, c or v, respectively. For instance, $\Gamma(x_c) = (C_c, I_c)$.

The bank and the client first write their secret values (in x_b and y_c); then v computes the two results (x'_b and y'_c); finally, the bank and the client print them (locally).

To define the intended security of this program, we consider a modified version where all localities are replaced by adversary placeholders (such that the adversary interleaves with the code of the source program):

$\_$; $x_b := e_b$; $\_$; $y_c := e_c$;
$\_$; $x'_b, y'_c := f(x_b, y_c)$;
$\_$; $print(x'_b)$; $\_$; $print(y'_c)$

Then we consider that, for any given adversary level, any behavior that can be obtain with this command context plus an adversary is an authorized behavior.

For instance, an adversary at the level of the bank can read and write bank variables, hence it can learn something from the client secret trough the variable x'_b, but it cannot directly read the client secret, and vice versa.

4.1.3 Compiler transformation

The compiler inputs a command with localities P and a security policy Γ, and outputs an initialization command Q_0, used to specify initial trust assumptions, plus a series of commands $\widetilde{Q}$, which include one command Q_i for each host i that occurs in the source program. We write Γ' for the security policy of these commands. Informally, Q_i is a single command that implements and schedules all code fragments of P located on i. After type checking, the compilation proceeds in 4 passes:

1. The source program is sliced into local threads, each running on a single host.
2. The distributed control flow between threads is protected, using dynamic checks on auxiliary program-counter variables, so that the adversary cannot run high-integrity threads out of schedule.
3. Relying on a single-static-assignment transformation, each variable shared between different hosts (including the program counters) is replaced by a series of local replicas, with explicit transfers between replicas.
4. Depending on their security levels, memory transfers between replicas are cryptographically protected, by inserting encryptions to transfer instead low-confidentiality encrypted values, and inserting authentication primitives to transfer instead low-integrity values. The compiler determines which symmetric keys to use for these operations, and generates an initial key-exchange protocol to distribute them. After this pass, the only variables shared between different hosts are (1) the signature verification keys used by the initial protocol, and (2) public-untrusted variables at level $(\bot, \top)$.

(The compiler also generates untrusted code for scheduling these commands and transferring public-untrusted data.)

Example 22 (Simplified implementation) *Continuing with our example, and in preparation for Chapter 6, we give a simplified, hand-written implementation of Example 21 that illustrates the main mechanisms of CFLOW while avoiding those irrelevant here. For instance, the ordering of $x_b := e_b$ and $y_c := e_c$ is irrelevant; the ordering of $x_b := e_b$ and $x'_b, y'_c := f(x_b, y_c)$ is protected because $x'_b, y'_c := f(x_b, y_c)$ does not run unless x_b has been verified. So, instead of the globally shared and signed programs counter, we use one local anti-replay counter for each host. Communications between b and v are cryptographically protected, but we let v and c share local memory (since v will run on c's machine with a technique explained on chapter 6). Otherwise, all the new (communication) variables are public and untrusted; the only shared high-integrity variables are the public keys (k_b^+ and k_v^+), which are assumed to be known beforehand. The commands are :*

$$
\begin{array}{ll}
Q_0 \doteq & k_b^-, k_b^+ := \mathcal{G}_e(); \; k_v^-, k_v^+ := \mathcal{G}_e() \\
Q_b \doteq & \text{if } c_b{=}1 \text{ then } (* \text{ set the bank input } *) \\
& \{ \; c_b{+}{+}; \; x_b := e_b; \; x_e := \mathcal{E}(x_b, k_v^+); \; x_s := \mathcal{S}(x_e, k_b^-) \; \} \\
& \text{else if } c_b = 2 \text{ then } (* \text{ get the bank output } *) \\
& \{ \; c_b{+}{+}; \; \text{if } \mathcal{V}(x'_e, x'_s, k_v^+) \text{ then } \mathit{print}(\mathcal{D}(x'_e, k_b^-)) \; \} \\
Q_c \doteq & \text{if } c_c{=}1 \text{ then } \{ \; c_c{+}{+}; \; y_c := e_c \; \} \\
& \text{else if } c_c{=}2 \text{ then } \{ \; c_c{+}{+}; \; \mathit{print}(y'_c) \; \} \\
Q_v \doteq & \text{if } c_v{=}1 \text{ then } (* \text{ run the computation } *) \\
& \{ \; c_v{+}{+}; \; \text{if } \mathcal{V}(x_e, x_s, k_b^+) \text{ then} \\
& \quad \{ \; x_v := \mathcal{D}(x_e, k_v^-); \\
& \quad \; x'_v, y'_c := f(x_v, y_c); \\
& \quad \; x'_e := \mathcal{E}(x'_v, k_b^+); \; x'_s := \mathcal{S}(x_e, k_v^-) \; \} \; \}
\end{array}
$$

- *Q_0 initialize the keypairs used b and v.*
- *Q_b is the code that runs on the bank machine. The first time it runs, it encrypts and signs its secret for v. The second time it runs, it verifies and decrypt the result sent by v, and then print it locally.*
- *Q_c is the code that runs on the client machine. The first time it runs, it write down it secret . The second time it runs, it read the result given by v, and then print it locally.*
- *Q_v is the code that is run by the trusted participant. It runs only once. It verifies and check the value sent by the bank then it computes the result of the bank and the client; finally, it encrypt and sign the result for the bank.*

In the implementation, we assume that the adversary explicitly schedule the code of these participants. For an adversary command context A, the program considered is $Q_0; A[Q_b, Q_c, Q_v]$. For example, the adversary can schedule the code as intended: $Q_0; Q_b; Q_c; Q_v; Q_b; Q_c]$.

4.2 Cryptographic assumptions

Following the idea of section 3.5, the ideal theorem would state that for any attack on the compiled program, there exist an attack on the source program that produce the same resulting memory distribution. However, the compiler relies on cryptography and it is always possible (though unlikely or very slow) for an adversary to guess a decryption key: this is an attack that has no counterpart in the source program. In this section, we state, using games, our cryptographic assumptions, which are typically of those proven by cryptographer for standard schemes. For example, we assume that an adversary cannot reliably fake a cryptographic signature without the signing key, even with access to as many real signatures as he wants. In the section below, we use these assumptions to prove that for any adversary of the implementation program (terminating reasonably fast), there exist an adversary in the source program such that the final memories distributions coincide with an overwhelming probability.

4.2.1 Polynomial-time assumptions

We assume that all primitive operations run in polynomial time in the size of their arguments, and that all primitive probabilistic functions yield distributions that are polynomial-time samplable. As we use cryptographic primitives, we assume that they take an additional parameter included in the initial memory whose length matches a security parameter η. We overload $\Pr[\langle P, \mu\rangle; \varphi]$ to denote a probability function parametrized by η. This function is deemed negligible when, for all $c > 0$, there exists η_c such that, for all $\eta \geq \eta_c$, we have $\Pr[\langle P, \mu\rangle; \varphi] \leq \eta^{-c}$.

4.2.2 Encryptions

We consider cryptographic algorithms for asymmetric (public key) encryption. These algorithms are generally used in three steps:

- First, one of the participant generates a key pair. These two keys are long bitstrings. One of them (the decryption key) is kept secret; the other (the encryption key) is left public and sent to every other participant. This is analogous to the production of a pair key/padlock with a copy of the padlock sent to every other participant.
- Second, another participant uses the encryption key to hide a secret. It results in a bitstring that can be made public, with the assurance that it is very hard to deduce anything about the secret hidden in this bitstring without the decryption key. This is analogous to the sealing of a secret using the padlock, then sent to the other participants.

- Last, the first participant decrypts the bitstring with its private key and read the secret. This is analogous to the opening of the padlock, which gives access to the secret.

We model them in our language as probabilistic functions $\mathcal{G}_e$, $\mathcal{E}$, and $\mathcal{D}$ that meet both functional and security properties, given below.

Definition 18 (Encryption scheme) *Let publickeys, secretkeys, plaintexts, and ciphertexts be sets of polynomially-bounded bitstrings indexed by η. An asymmetric encryption scheme is a triple of algorithms $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ such that*

1. *$\mathcal{G}_e$, used for* key generation*, ranges over publickeys $\times$ secretkeys;*
2. *$\mathcal{E}$, used for* encryption*, ranges over ciphertexts;*
3. *$\mathcal{D}$, used for* decryption*, ranges over plaintexts;*
4. *for all $k_e, k_d := \mathcal{G}_e()$, if $y := \mathcal{E}(m, k_e)$ with $m \in$ plaintexts, then $\mathcal{D}(y, k_d) = m$.*

There are many different notions of security for encryption. The notion of indistinguishability against adaptive chosen ciphertext attacks (IND-CCA2), introduced by Rackoff and Simon [1991], is the strongest usually considered. To code the definition in our target language, we rely on auxiliary operations on functional lists: $\mathbf{0}$ for the empty list, $+$ for concatenation, and $\in$ for membership test.

Definition 19 (IND-CCA2 security) *Consider the probabilistic commands*

$$\begin{aligned} E &\doteq \textit{if } b = 0 \textit{ then } m := \mathcal{E}(x_0, k_e) \textit{ else } m := \mathcal{E}(x_1, k_e); \\ &\quad\; log := log + m \\ D &\doteq \textit{if } m \in log \textit{ then } x := 0 \textit{ else } x := \mathcal{D}(m, k_d) \\ CCA &\doteq b := \{0,1\}; log := \mathbf{0}; k_e, k_d := \mathcal{G}_e(); A[E, D] \end{aligned}$$

The encryption scheme $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ provides indistinguishability against adaptive chosen-plaintext and chosen-ciphertext attacks when the advantage $|\Pr[CCA; b =_0 g] - \frac{1}{2}|$ is negligible for any polynomial command context A such that $b, k_d \notin rv(A)$ and $b, k_d, k_e, \eta, log \notin wv(A)$.

The definition involves an indistinguishability game where the adversary command A attempts to guess (and write in variable g) whether the oracle encrypts x_0 or x_1. The trivial adversary $A \doteq g := \{0,1\}$ wins with probability $\frac{1}{2}$, so the security property states that any adversary that meets our hypothesis cannot do (much) better, despite its control on the usage of the key.

We recall a similar but weaker security notion for encryption, indistinguishability against chosen plaintext attacks (IND-CPA):

Definition 20 (CPA Security) *The scheme $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ provides indistinguishability against adaptive chosen-plaintexts attacks when, for the commands*

$$\begin{aligned} E &\doteq \textit{if } b = 0 \textit{ then } m := \mathcal{E}(x_0, k_e) \textit{ else } m := \mathcal{E}(x_1, k_e) \\ \mathsf{CPA} &\doteq b := \{0,1\}; k_e, k_d := \mathcal{G}_e(); A[E] \end{aligned}$$

and for any polynomials command context A such that $b, k_d \notin rv(A)$ and $b, k_d, k_e, \eta \notin wv(A)$, the advantage of the adversary defined as $|\Pr[\mathsf{CPA}; b =_0 g] - \frac{1}{2}|$ is negligible.

This notion of security does not involve the decryption algorithm. In particular, it does not cover chosen-ciphertext attacks, where, for example, the attacker try to leverage an advantage on future

descriptions from an limited access to a computer that can encrypt and decrypt with the user's key.

By definition, a scheme that is IND-CCA2 is also IND-CPA secure [Bellare et al., 1998], however, encryption schemes that are CCA are generally less efficient that those that are only CPA.

In the following example, we show that the probability that two encryptions of a CPA scheme coincide is negligible (in particular, it implies that ecryptions are probabilistic).

Example 23 *Let $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ be an IND-CPA encryption scheme. For any value v, Consider the command context:*

$$\begin{array}{rcl} A[E] & \doteq & x_0 := 0; \\ & & x_1 := v; \\ & & m' := \mathcal{E}(v, k_e); \\ & & E; \\ & & g := (m = m'); \end{array}$$

We have $|\Pr[CCA; b =_0 g] - \frac{1}{2}|$. At the end of CCA, if $b = 0$ then $g = 0$ since two encryption of different values cannot coincide; if $b = 1$ then $g = 1$ if the two encryptions of v coincide. Hence, the probability that two encryption of the same value coincide is negligible.

4.2.3 Cryptographic signatures

We consider cryptographic algorithms for asymmetric (public key) signature. These algorithms are generally used in three steps:

- First, one of the participant generates a key pair. These two keys are long bitstrings. One of them (the signing key) is kept secret; the other (the verification key) is left public and sent to every other participant. The other participants need a guarantee that the verification key they receive is the correct one. It can be hand delivered, or certified by a third party.
- Second, the first participant uses the private key to sign some data. It results in a bitstring that can be made public, with the assurance that it is very hard to deduce anything about the secret key or to forge other signatures.
- Last, any other participant can verify, with the public key, that the signature correspond to the data and that it has been made by a participant knowing the private key.

We model them in our language as probabilistic functions $\mathcal{G}_s$, $\mathcal{S}$, and $\mathcal{V}$ that meet both functional and security properties, given below.

Definition 21 (Signature scheme) *Let sigkeys, verifykeys, signedtexts, and plaintexts be sets of polynomially-bound bitstrings indexed by η.*

A signature scheme is a triple of algorithms $(\mathcal{G}_s, \mathcal{S}, \mathcal{V})$ such that

- *$\mathcal{G}_s$, used for* key generation*, ranges over sigkeys $\times$ verifykeys;*
- *$\mathcal{S}$, used for* signing*, ranges over signedtexts;*
- *$\mathcal{V}$, used for* signature verification*, ranges over $\{0, 1\}$ and is such that, for all $k_s, k_v := \mathcal{G}_s()$ and $m \in$ plaintexts, we have $\mathcal{V}(m, \mathcal{S}(m, k_s), k_v) = 1$.*

For convenience, we assume that $\mathcal{V}$ is deterministic, so that we can use test expressions $\mathcal{V}(e, e', e'')$ in conditional commands.

There are also many notions of security for signature schemes. We use a standard notion introduced by Goldwasser et al. [1988], namely existential unforgeability under chosen message attacks (CMA):

Definition 22 (CMA security) *Consider the commands*

$$\begin{aligned} S &\doteq x := \mathcal{S}(m, k_s); log := log + m; \\ CMA &\doteq k_s, k_v := \mathcal{G}_s(); log := \mathbf{0}; A[S]; \\ &\quad \text{if } m \in log \text{ then } b := 0 \text{ else } b := \mathcal{V}(m, x, k_v) \end{aligned}$$

The signature scheme $(\mathcal{G}_s, \mathcal{S}, \mathcal{V})$ is secure against forgery under adaptive chosen-message attack when $\Pr[CMA; b = 1]$ is negligible for any polynomial command context A that cannot read k_s and cannot write k_s, k_v, log, η.

The definition involves an indistinguishability game where the adversary command A, after requesting as many signatures as he wants from the signing oracle S, attempts to produce an original pair (m, x) such that x is the signature for a message m not signed by S.

4.2.4 Cryptographic hashes

We consider cryptographic algorithms for hashing. Hash functions map values of unbounded size to small values of a fixed size. It may be used, for instance, to map a program to a small hash value. This small value can be stored (or transferred) securely more easily than the program itself. Then, it is possible to know if the program has been tampered with by comparing its hash with the securely stored hash. This method is efficient, and secure if the hash of two different program are different. Yet, given the difference in size between their domain and their range, hash functions are not injective. However, good hash functions can guarantee that it is hard to find collisions. We consider an additional computational assumption on hash function [Preneel, 1993, Zheng et al., 1990].

Definition 23 (Collision-Resistant Hash Functions) *A fixed size collision resistant hash function family is a family of polynomial functions $(\mathcal{G}, \mathcal{H})_\eta$ such that $\mathcal{G}$ is probabilistic, $\mathcal{H}$ is deterministic, ranges over bitstrings of length η, and, for all polynomial commands A that do not write ν, the probability*

$$\Pr[\langle \nu := \mathcal{G}(); A,\ \mu_\perp \rangle; x_0 \neq x_1 \wedge \mathcal{H}(\nu, x_0) = \mathcal{H}(\nu, x_1)]$$

is negligible.

In the definition, a random seed $\mathcal{G}$ is first sampled, then the adversary A tries to find any collision for the resulting hash function, represented as a pair of values written to the variables x_0 and x_1. The hash function is deemed secure if this happens with negligible probability. The random seed is formally needed to prevent A from pre-computing a collision, but the parameters η and ν are otherwise usually kept implicit.

4.3 Security and completeness of the compiler

The 4 passes of the compilation presented in the section 4.1 define a program transformation

$$Q_0, \widetilde{Q} \doteq [\![P]\!]$$

such that each command Q_i has type ℓ_i^I. Next, we instantiate Definitions 16 and 17 for this transformation.

4.3.1 Source programs and their adversaries

We let source programs range over well-typed polynomial commands with locality annotations. Informally, source programs enable their active adversaries to run whenever they pass control between hosts (since the adversary controls at least the network). To each source command P, we associate the command context $\widehat{P}$ obtained from P by replacing every subcommand of the form $b : P'$ by a command context with two placeholders $\_; P'; \_$. For a given adversary level α, we let $\widetilde{A}$ range over tuples of polynomial adversary commands, with one command for each command variable. Hence, $\widehat{P}[\widetilde{A}]$ ranges over commands that interleave the code of P with adversary commands. (This is analogous to models of non-interference for concurrency, where the adversary runs between any two program steps.) Thus, we define source evaluation by

$$\langle\!\langle P, A, \mu \rangle\!\rangle \doteq \rho_\infty \langle \widehat{P}[A], \mu \rangle$$

4.3.2 Implementations and their adversaries

Implementation programs range over our compiler outputs $Q_0, \widetilde{Q}$. Once Q_0 has run, we simulate concurrency by letting the adversary explicitly schedule commands ($\widetilde{Q}$) that represent parallel threads of computation (rather than having $\widehat{P}$ schedule $\widetilde{A}$). The resulting low-level model realistically accounts for all interleaving of these threads. Implementation adversaries range over adversaries command contexts A' with one command variable for each host. Thus, we define target evaluation by

$$\langle\!\langle (Q_0, \widetilde{Q}), A', \mu' \rangle\!\rangle \doteq \rho_\infty \langle Q_0; A'[\widetilde{Q}], \mu' \rangle$$

We let π be the erasure of all variables added in Γ': $\pi(\mu') = \mu'_{|dom(\mu)}$ and we define equivalence on final memory distributions ($\rho_0 \approx \rho_1$) as computational indistinguishability : for all polynomial commands T, $|\Pr[\langle T, \rho_0 \rangle; g = 0] - \Pr[\langle T, \rho_1 \rangle, g = 0]|$ is negligible. (We use indistinguishability, similar to that of that of Section 4.2, instead of distribution equality because our compiler relies on cryptographic security assumptions.)

Although we allow endorsement and non-robust declassification, in the following, we still usually demand that our security policies be robust:

Definition 24 (Robust policies) *For a given robustness function R, $\ell \in \mathcal{L}$ is robust when $I(\ell) \leq R(C(\ell))$; Γ is robust for R when $\Gamma(x)$ is robust for all $x \in \mathrm{dom}(\Gamma)$.*

With the definitions above, and under standard security assumptions, our new compilation theorem for CFLOW is

Theorem 9 (Fournet et al. [2009]) *Let $\alpha \in \mathcal{L}$ be such that R is robust against α. Let Γ be a robust policy.*

1. *$(\llbracket \cdot \rrbracket, \pi)$ is secure;*
2. *$(\llbracket \cdot \rrbracket, \pi)$ is correct when $\alpha = (\bot_C, \top_I)$.*

The theorem demands that the source policy Γ be robust (Definition 24), so that the adversary can read any shared variable that it can write. This hypothesis stems from our decision to support endorsements in source programs. In particular, the control flow integrity enforced by pass 2 may otherwise fail to protect programs that combine endorsements and declassifications, as illustrated below.

Example 24 (Non-robust shared variables) *Consider a source program that writes a secret s with $\Gamma(s) = HH$ into a (non robust) variable x with $\Gamma(x) = HL$, then erases the content of x, and finally declassifies x by copying it to p with $\Gamma(p) = LL$:*

$$P \doteq 1{:}\{x := s\}; 2{:}\{x := 0\}; 3{:}\{p := x\}$$

Let $\alpha = LL$. With our source semantics, the command context

$$\widehat{P} \doteq X_1; x := s; X_2; x := 0; X_3; p := x; X_4$$

ensures that p finally contains either 0 or a value written by the adversary, but not the value of s.

In the implementation, however, the two local commands at hosts 1 and 2 have low integrity, so pass 2 does not guarantee their sequential execution, and an implementation adversary that schedules Q_2 before Q_1 lead Q_3 to declassify s into p.

Cryptographic protection (pass 4) is also problematic for programs that share non-robust variables, such as x in Example 24. Although their confidentiality is protected by encryption, an implementation adversary can swap their contents (by swapping their encrypted values) and similarly lead the program to declassify the wrong data.

5

Information-Flow Types for Homomorphic Encryptions

In the chapter above, we described a compiler that takes as input a program with locality an information flow annotations and outputs a distributed program secured with cryptography. In this chapter, we verify, with a type system, the correct use of cryptography mechanisms on information flow annotated programs. The approach is different since we do not try to compile the program, but only check its security. However, we consider more varied cryptographic primitives. This work help us to understand other cryptographic primitives, and may lead to an improved compiler.

5.1 Introduction

Information flow security is a well-established, high-level framework for reasoning about confidentiality and integrity, with a clear separation between security specifications and mechanisms. At a lower level, encryption provides essential mechanisms for confidentiality, with a wide range of algorithms reflecting different trade-offs between security, functionality, and efficiency. Thus, the secure usage of adequate algorithms for a particular system is far from trivial.

Even with plain encryption, the confidentiality and integrity of keys, plaintexts, and ciphertexts are interdependent: encryption with untrusted keys is clearly dangerous, and plaintexts should never be more secret than their decryption keys. Integrity also matters: attackers may swap ciphertexts, and thus cause the declassification of the wrong data after their successful decryption. Conversely, to protect against chosen-ciphertext attacks, it may be necessary to authenticate ciphertexts, even when plaintexts are untrusted.

Modern encryption schemes offer useful additional features, such as the ability to blind ciphertexts, thereby hiding dependencies between encrypted inputs and outputs, and more generally to compute homomorphically on encrypted data, for instance by multiplying ciphertexts instead of adding their plaintexts. These features are naturally explained in terms of information flows; they enable computations at a lower level of confidentiality—homomorphic operations can be delegated to an "honest but curious" principal—but they also exclude CCA2 security and require some care in the presence of active adversaries.

In this chapter, we develop an information-flow type system for cryptography, with precise typing rules to reflect the diverse functional and security features of encryption schemes. Our adversaries attempt to gain information about higher-confidentiality data, or to influence higher-integrity data, by interacting with our programs. Our main theorem states that well typed probabilistic polynomial programs using any combination of encryptions, blinding, homomorphic functions, and decryptions are such that our adversaries succeed only with a negligible probability. Depending on the relative security levels of keys, plaintexts, and ciphertexts, we propose different typing rules. Our rules are sound with regards to standard cryptographic assumptions such as CPA or CCA2;

they enable the selective re-use of keys for protecting different types of payloads, as well as blinding and homomorphic properties.

Secure Distributed Computations Our work is part of a research project on the synthesis and verification of distributed cryptographic implementations of programs from a description of their information flow security requirements. The current compiler is presented in Chapter 4. As illustrated in our programming examples, an important motivation for a new type system is to justify its efficient use of cryptography. For instance, we strive to re-use the same keys for protecting different types of data at different levels of security, so that we can reduce the overall cost of cryptographic protection. More generally, we would like to automatically generate well-typed code for performing blind computations, such as those supported by the systems of Henecka et al. [2010] and Katz and Malka [2010].

Formally, our work is based on the type system of Fournet and Rezk [2008], who introduce the notion of *computational non-interference* against active adversaries to express information flow security in probabilistic polynomial-time cryptographic systems and present a basic type system for encryption and authentication mechanisms. In comparison, their typing rules are much more restrictive, and only support CCA2 public-key encryptions with a single payload type.

Applications We illustrate our approach using programming examples based on classic encryption schemes with homomorphic properties [ElGamal, 1984, Paillier, 1999]. We also develop two challenging applications of our approach. Both applications rely on a security lattice with intermediate levels, reflecting the structure of their homomorphic operations, and enabling us to prove confidentiality properties, both for honest-but-curious servers and for compromised servers controlled by an active attacker.

- We program and typecheck a practical protocol for private search on data streams proposed by Ostrovsky and Skeith III [2005], based on a Paillier encryption of the search query. This illustrates that our types protect against both explicit and implicit information flows; for instance we crucially need to apply some blinding operations to hide information about secret loop indexes.

- We program and typecheck the bootstrapping part of Gentry's fully homomorphic encryption [Gentry, 2009]. Starting from the properties of the bootstrappable algorithms given by Gentry–being CPA and homomorphic for its own decryption and for some basic operations—we obtain an homomorphic encryption scheme for an arbitrary function. This illustrates three important features of our type system: the ability to encrypt decryption keys (which is also important for typing key establishment protocols); the use of CPA encryption despite some chosen-ciphertext attacks; and an interesting instance of homomorphism where the homomorphic function is itself a decryption.

This chapter exclusively treats public-key encryptions. We believe that their symmetric-key counterparts can be treated similarly. We also refer to Fournet and Rezk [2008] for cryptographic types for authentication primitives, such as public-key signatures. A typechecker for our system is available online at `msr-inria.inria.fr/projects/sec/cflow`.

Contents The rest of this chapter is composed as follows. Section 5.2 refines the security types of Section 3.3 with datatypes for encryptions. Section 5.3 gives typing rules for encryption schemes. Section 5.4 defines blinding schemes and gives typing rules for blinding schemes. Sections 5.5 defines homomorphic schemes and gives typing rules for homomorphic schemes. Section 5.6 states our theorem. Sections 5.7 and 5.8 illustrate our theorem with two applications.

5.2 Datatypes for encryptions and keys

In this chapter, we use the language introduced in Section 3.1. We also consider information-flow properties and labels similar to those of Chapter 3. However, we manipulate keys and ciphertexts and we try to enforce their correct usage. Hence, we need to track datatypes and not only security levels. For example, we ensure that only public keys are used for encryptions. We use the following grammar for security types:

$$\begin{array}{rll} \tau ::= t(\ell) \mid \tau * \tau \mid \mathsf{Array}\,\tau & & \text{Security types} \\ t ::= \mathsf{Data} & & \text{Payload Data types} \\ \quad \mid \mathsf{Enc}\,\tau\,K q \mid \mathsf{Ke}\,\mathsf{E}\,K \mid \mathsf{Kd}\,\mathsf{E}\,K & & \text{Encryption Data types} \end{array}$$

where $\ell \in \mathcal{L}$ is a security label, E is a set of security types, K is a key label, and q is an encryption index, as explained below. The simplest security type, e.g., for variables contaning integers, booleans, chars ... at level ℓ is $\mathsf{Data}\,(\ell)$. We have pairs at the level of the security types to keep track of tuples of values with different security types or levels, e.g. for encrypting tuples of mixed values. Pairs are also used in our proofs for eliminating cryptography. We have standard operators for pairs and functional arrays: we use $\langle e_0, e_1 \rangle$ and $(e)_i$ for constructing and projecting pairs, and use $e[e_i]$ and $\mathrm{update}(x, e, e_i)$ for reading and updating arrays. We use syntactic sugar for arrays, writing $x[e_i] := e$ instead of $x := \mathrm{update}(x, e, e_i)$, and for loops, writing $\mathsf{for}\ x := e\ \mathsf{to}\ e'\ \mathsf{do}\ P$ instead of $x := e; \text{while } x \leq e' \text{ do } \{P; x := x + 1\}$.

Encryption Datatypes

We explain our datatypes for encryptions, but defer their typing rules to the next sections. $\mathsf{Enc}\,\tau\,K$ represents an encryption of a plaintext with security type τ; $\mathsf{Ke}\,\mathsf{E}\,K$ and $\mathsf{Kd}\,\mathsf{E}\,K$ represent encryption and decryption keys, respectively, with a set E that indicates the range of security types τ for the plaintexts that may be encrypted and decrypted with these keys. This set enables us to type code that uses the same key for encrypting values of different types, which is important for efficiency. For example, $\mathsf{E} \doteq \{\mathsf{Data}\,(LH) * \mathsf{Data}\,(HL)\}$ indicates a single type of plaintexts consisting of pairs whose first elements has low-confidentiality high-integrity (e.g. a tag) and second elements have high confidentiality (e.g. a secret payload); $\mathsf{E}' \doteq \{\mathsf{Kd}\,\mathsf{E}\,K(HH), \mathsf{Data}\,(HH)\}$ indicates two types of plaintexts with different datatypes, either decryption keys for E or plain data, but with the same security level. An encryption key with type $\mathsf{Ke}\,\mathsf{E}'\,K'(LH)$ can encrypt either decryption keys for E or plain data, with security level LH. A ciphertext with type $\mathsf{Enc}\,(\mathsf{Kd}\,\mathsf{E}\,K(HH))\,K(LH)$ is an encryption of a decryption key of label K. It has itself a level LH.

Static key labels The key labels K are used to keep track of keys, grouped by their key-generation commands. For example, we latter ensure that there is only one keypair generation for a given label.

These labels are attached to the types of the generated keypairs, and propagated to the types of any derived cryptographic materials. They are used to match the usage of key pairs and to prevent key cycles.

The encryption indexes q are used to distinguish between different datatypes for encryptions, for instance when they range over different groups before and after some homomorphic operations. In all other cases, we use a single, implicit index $q = 0$.

Relations between security types and security labels We define a projection L from security types to security labels:

$$L(t(\ell)) = \ell \qquad L(\tau * \tau') = L(\tau) \sqcap L(\tau') \qquad L(\mathsf{Array}\,\tau) = L(\tau)$$

$$\text{(VAR)}\ \vdash x : \Gamma(x) \qquad \text{(TOP)}\ \frac{op : \tau_1 \ldots \tau_n \to \tau \quad \vdash e_i : \tau_i \ \text{ for } i = 1..n}{\vdash op(e_1, \ldots, e_n) : \tau} \qquad \text{(SUBE)}\ \frac{\vdash e : \tau \quad \tau \leq \tau'}{\vdash e : \tau'} \qquad \text{(TASSIGN)}\ \frac{\vdash e : \Gamma(x)}{\vdash x := e : L(x)}$$

$$\text{(TFUN)}\ \frac{\vdash e : \mathsf{Data}\,(\ell) \ \text{ for } e \in \widetilde{e} \quad \mathsf{Data}\,(\ell) \leq \Gamma(x) \ \text{ for } x \in \widetilde{x}}{\vdash \widetilde{x} := f(\widetilde{e}) : \ell} \qquad \text{(TCOND)}\ \frac{\vdash e : \mathsf{Data}\,(\ell) \quad \vdash P : \ell \quad \vdash P' : \ell}{\vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P' : \ell} \qquad \text{(TWHILE)}\ \frac{\vdash e : \mathsf{Data}\,(\ell) \quad \vdash P : \ell}{\vdash \mathsf{while}\ e\ \mathsf{do}\ P : \ell}$$

Figure 5.1: Non-cryptographic typing rules for expressions and commands with policy Γ.

We define subtyping on types as partial-ordering on their security labels, as follows:

$$\frac{\ell \leq \ell'}{t(\ell) \leq t(\ell')} \qquad \frac{\tau_0 \leq \tau_0' \quad \tau_1 \leq \tau_1'}{\tau_0 * \tau_1 \leq \tau_0' * \tau_1'} \qquad \frac{\tau \leq \tau'}{\mathsf{Array}\,(\tau) \leq \mathsf{Array}\,(\tau')}$$

For example $\mathsf{Data}\,(HH) * \mathsf{Data}\,(LH) \leq \mathsf{Data}\,(HL) * \mathsf{Data}\,(LL)$. We overload $\leq$ as follows:

$$t(\ell) \leq_\star \ell \qquad \ell \leq_\star t(\ell) \qquad \frac{\tau_0 \leq_\star \ell \quad \tau_1 \leq_\star \ell}{\tau_0 * \tau_1 \leq_\star \ell} \qquad \frac{\ell \leq_\star \tau_0 \quad \ell \leq_\star \tau_1}{\ell \leq_\star \tau_0 * \tau_1}$$

$$\frac{\tau \leq_\star \ell}{\mathsf{Array}\,(\tau) \leq_\star \ell} \qquad \frac{\ell \leq_\star \tau}{\ell \leq_\star \mathsf{Array}\,(\tau)}$$

where $\leq_\star$ is $\leq_C$, $\leq_I$ or $\leq$. For example, $\mathsf{Data}\,(HH) * \mathsf{Data}\,(LL) \leq HL$ and $\mathsf{Data}\,(HH) * \mathsf{Data}\,(LL) \leq_I L$. We also extend $\sqcup$ from labels to types:

$$t(\ell') \sqcup \ell \doteq t(\ell' \sqcup \ell) \qquad (\tau_0 * \tau_1) \sqcup \ell \doteq (\tau_0 \sqcup \ell) * (\tau_1 \sqcup \ell) \qquad \mathsf{Array}\,(\tau) \sqcup \ell \doteq \mathsf{Array}\,(\tau \sqcup \ell)$$

For example $\mathsf{Data}\,(HH) * \mathsf{Data}\,(LL) \sqcup LL = \mathsf{Data}\,(HL) * \mathsf{Data}\,(LL)$. Memory policies are now functions Γ from variables to security types. For a given policy Γ, we write $x \leq_\star \_$ and $\_ \leq_\star x$ for $\Gamma(x) \leq_\star \_$ and $\_ \leq_\star \Gamma(x)$.

Core Type System We adapt our (strict) type system of Figure 3.2 to security types in Figure 5.1; Rules TSEQ, TINERT, TSUBC, and TSKIP are inchanged from Figure 3.2. In Rule TOP, the first hypothesis refers to the type signature of *op*, defined below for generic operations, and for operations on pairs and arrays:

$$\begin{array}{rl} op & : \mathsf{Data}\,(\ell) * \cdots * \mathsf{Data}\,(\ell) \to \mathsf{Data}\,(\ell) \\ \langle\rangle & : \tau_0, \tau_1 \to (\tau_0 * \tau_1) \\ ()_i & : (\tau_0 * \tau_1) \to \tau_i \\ [] & : \mathsf{Array}\,(\tau) \\ [] & : \mathsf{Array}\,(\tau), \mathsf{Data}\,(L(\tau)) \to \tau \\ \mathrm{update} & : \mathsf{Array}\,(\tau), \tau, \mathsf{Data}\,(L(\tau)) \to \mathsf{Array}\,(\tau) \\ + & : \mathsf{Array}\,(\tau), \tau \to \mathsf{Array}\,(\tau) \\ \mathrm{size} & : \mathsf{Array}\,(\tau) \to \mathsf{Data}\,(L(\tau)) \end{array}$$

5.3 Typing Encryptions

5.3.1 Typing encryptions against Chosen-Plaintext Attacks

We now describe our cryptographic rules for typing the algorithms of a CPA encryption scheme. Figure 5.2 gives rules for typing key generation, encryption, and decryption.

$$\frac{(\mathcal{G}_e, \mathcal{E}, \mathcal{D}) \text{ is CPA} \quad \Gamma(k_e) = \mathsf{Ke}\,\mathsf{E}\,K(\ell_e) \quad \Gamma(k_d) = \mathsf{Kd}\,\mathsf{E}\,K(\ell_d) \quad \forall \tau \in \mathsf{E}, \tau \leq_C \ell_d}{\vdash k_e, k_d := \mathcal{G}_e() \ : \ell_e \sqcap \ell_d} \text{(GenE)}$$

$$\frac{\Gamma(y) = \mathsf{Enc}\,\tau K(\ell_y) \quad \vdash e : \tau \quad \tau \in \mathsf{E} \quad \vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \quad \alpha \not\leq_I k_e \ \textbf{or} \ \tau \leq_C \alpha}{\vdash y := \mathcal{E}(e, k_e) \ : \ell_y} \text{(Encrypt)}$$

$$\frac{\tau \leq \Gamma(x) \quad \tau \in \mathsf{E} \quad \vdash y : \mathsf{Enc}\,\tau K(L(x)) \quad \vdash k_d : \mathsf{Kd}\,\mathsf{E}\,K\,(\ell_d) \quad \ell_d \leq_I x \quad (\alpha \not\leq_I y \ \textbf{and} \ \alpha \not\leq_I \ell_d) \ \textbf{or} \ \ell_d \leq_C x \ \textbf{or} \ \ell_d \leq_C \alpha}{\vdash x := \mathcal{D}(y, k_d) \ : L(x)} \text{(Decrypt)}$$

Figure 5.2: Typing rules for CPA encryption with policy Γ.

GenE For key generation, the first hypothesis requires that the encryption scheme (implicitly parametrized by the key label K) be at least CPA secure. Alternatively, one may use keys with datatype Data, and type generation, encryption, and decryption using rule TFun. The next two hypotheses give matching encryption-key and decryption-key types to variables k_e and k_d, with the same key label K and the same range of plaintexts E.

The constraint $\tau \leq_C \ell_d$ for every type $\tau \in \mathsf{E}$ states that the decryption key k_d is at least as confidential as every plaintext, thereby preventing confidentiality leaks by key compromise. (The constraint appears in GenE rather than Decrypt, to ensure that all copies of the decryption key are sufficiently confidential.)

Encrypt The first three hypotheses bind types to the ciphertext y, plaintext e, and key k_e involved in the encryption; these types are related by $\tau \in \mathsf{E}$ and K. The type of y carries two security labels: ℓ_y, the level of the encryption, and $L(\tau)$ within $\mathsf{Enc}\ \tau K$, the level of the encrypted plaintext. The label ℓ_y in the typing of k_e records an ordinary flow from k_e to y: by subtyping, we must have $L(k_e) \leq \ell_y$. Conversely, there is no constraint on the flow from e to y, so our typing rule can be sound only with cryptographic assumptions:

- For confidentiality, this flow reflects that encryption is a form of declassification: e can be more confidential than y. In that case, that is $\tau \not\leq_C y$, we say that K *declassifies* $C(\tau)$.
- For integrity, this flow reflects that encryption is also a form of endorsement: intuitively, y is only a carrier for e, not in itself an observable outcome of the program; its integrity indicates that y is the result of a correct encryption, even when e itself is not trusted.

Finally, the disjunction is a robustness condition on the encryption key, requiring that its integrity be sufficient to protect the confidentiality of the plaintext. This disjunction, as well as all the other conditions on α in our type system, can be expressed with the robustness function R introduced in Section 3.4. In this case, the condition would be $k_e \leq_I R(C(\tau))$. However, we feel that this type system is easier to understand if we keep α explicit.

- The first disjunct excludes that an α-adversary may affect the encryption key (for instance overwriting it with an ill-formed key or a key he knows).
- The second disjunct states that the confidentiality protection provided by the key is nil, since an α-adversary may directly read the plaintext.

Hence, in case E contains plaintexts with different levels of secrecy, a key with relatively low integrity may still be used to encrypt a plaintext with relatively low confidentiality.

DECRYPT The first three hypotheses on the left bind types to the plaintext x, the ciphertext y, and key k_d involved in the decryption; these types are related by τ and K. The label $L(x)$ flows from y to x, as in a normal assignment. The label ℓ_d and the hypothesis $\ell_d \leq_I x$ record integrity flows from k_d to x, as in a normal assignment. Conversely, there is no constraint on the confidentiality flow from k_d to the plaintext x, so our typing rule can be sound only with cryptographic assumptions. Informally, this reflects that the decrypted plaintext yields no information on the decryption key itself, thereby enabling decryptions of plaintexts with different confidentiality levels. When $\ell_d \not\leq_C x$, we say that K *declassifies* $C(\ell_d)$.

The disjunction deals with the integrity of the ciphertext y and the decryption key k_d, with three cases: either

- both ciphertext and decryption key have high integrity; or
- the plaintext is at least as confidential as the key; or
- cryptographic protection is nil, since an α-adversary may read the decryption key then decrypt the plaintext.

The first case intuitively reflects our CPA security assumptions: it requires that the integrity of the ciphertext and the decryption key suffices to guarantee a correct decryption.

When $y \not\leq_I \alpha$ and $\ell_d \not\leq_C \alpha$, the second case covers some chosen-ciphertext attacks, which may come as a surprise: in that case, we say that K *depends on* $C(x)$ and we rely on an additional safety condition (see also Examples 27 and 28). When $\ell_d \not\leq_I \alpha$ and $\ell_d \not\leq_C \alpha$, the second case also covers compromise of the decryption key, which may then flow to the decrypted value, as discussed in Example 26.

For reference, we give two derived rule for typing encryptions and decryptions as ordinary probabilistic function on Data, rather than relying on cryptographic assumptions. These rules are correct, but do not allow any declassification.

$$\frac{\vdash e : \mathsf{Data}\,(L(y)) \qquad \vdash k_e : \mathsf{Data}\,(L(y))}{\vdash y := \mathcal{E}(e, k_e) \;:\; L(y)}\;(\text{ENCRYPT (BY TFUN)})$$

$$\frac{\vdash e : \mathsf{Data}\,(L(x)) \qquad \vdash k_d : \mathsf{Data}\,(L(x))}{\vdash x := \mathcal{D}(y, k_d) \;:\; L(x)}\;(\text{DECRYPT (BY TFUN)})$$

5.3.2 Examples

We provide a series of examples that illustrate the relative information-flow security properties of encryptions. In the examples, unless specified otherwise, we use a four-point, we assume that α is LH and k_e, k_d is a secure keypair, that is, we set $L(k_e) = LH$ and $L(k_d) = HH$ and assume that $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ is CPA. We also write for instance x_{LH} for a variable such that $L(x_{LH}) = LH$.

We first encrypt a pair of mixed integrity into a ciphertext of high integrity:

Example 25 (Higher-integrity Encryptions) *Consider the command context*

$$\begin{array}{l} k_e, k_d := \mathcal{G}_e(); \\ x := \langle x_{HH}, x_{HL} \rangle; y_{LH} := \mathcal{E}(x, k_e); \\ \_; \\ x' := \mathcal{D}(y_{LH}, k_d); x'_{HH} := (x')_0; x'_{HL} := (x')_1 \end{array}$$

Let $\Gamma(x) = \Gamma(x') = \mathsf{Data}\,(HH) * \mathsf{Data}\,(HL)$. *This command typechecks; it is safe, inasmuch as the adversary can influence the values of* x_{HL}, x'_{HL}, *and, indirectly, the value the ciphertext* y_{LH}, *but not the final value of* x'_{HH}.

Encrypting secrets with a low-integrity key clearly leads to confidentiality leaks, as the adversary may overwrite the key with its own key before the encryption. Decrypting with a low-integrity decryption key may also be problematic, especially when the plaintext is not secret, as illustrated below:

Example 26 (Low-Integrity Keys) *Consider the command context*

$$\begin{array}{l} k_e, k_d := \mathcal{G}_e(); k'_d := k_d; \\ y_{LH} := \mathcal{E}(1, k_e); \\ \text{-}; \\ \text{if } x_{HH} \text{ then } k''_d := k_d \text{ else } k''_d := k'_d; \\ x_{LL} := \mathcal{D}(y_{LH}, k''_d) \end{array}$$

We let k'_d and k''_d be low-integrity copies of the decryption key, that is $L(k'_d) = L(k''_d) = HL$. The command is not typable, and is actually unsafe, since we may fill the placeholder with the command $k'_d := 0$; thus, k''_d contains the correct key if and only if $x_{HH} \neq 0$, and the adversary can finally compare x_{LL} with 1, the correct decryption, and infer the confidential value x_{HH}.

Similarly, since we rely on CPA (for chosen plaintext) and not CCA (for chosen ciphertext), decrypting a low-integrity ciphertext may also be problematic, as illustrated below:

Example 27 (Chosen-Ciphertext Attacks) *When decrypted, low-integrity ciphertexts may leak information about their decryption keys. For a given CPA encryption scheme $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$, we derive a new scheme $(\mathcal{G}'_e, \mathcal{E}', \mathcal{D}')$ as follows:*

$$\begin{array}{rcl} \mathcal{G}'_e() & \doteq & \mathcal{G}_e() \\ \mathcal{E}'(x, k_e) & \doteq & 1|\mathcal{E}(x, k_e) \\ \mathcal{D}'(y, k_d) & \doteq & b|y' := y; \text{if } b = 0 \text{ then } k_d \text{ else } \mathcal{D}(y', k_d) \end{array}$$

(where $b|y' := y$ abbreviates decomposing y into b and y'). In the new scheme, decryption leaks its key when called on an ill-formed ciphertext (prefixed with a 0 instead of a 1).

Although this new scheme is still CPA, decryption of a low-integrity ciphertext may cause a confidentiality flow from k_d to the decrypted plaintext, letting the adversary decrypt any other value encrypted under k_e.

5.3.3 CPA versus chosen ciphertext attacks

As illustrated in Sections 5.7 and 5.8, our typing rules for CPA allow the decryption of low-integrity ciphertexts when the plaintext is secret, as long as the resulting plaintext never flows to any cryptographic declassification.

We define a notion of key dependencies, used to express the absence of key cycles as an assumption for our theorem (in Section 4.2), and we give a counter-example showing an implicit information flow for one such cycle.

Definition 25 (Key Dependencies and Free Labels) *For a given policy Γ, K and K' two key labels, and P a command context typable with Γ, we say that K* depends on *K' when either*

1. *there exists $\mathsf{Ke}\,\mathsf{E}'\,K'(\ell)$ in Γ such that $\mathsf{Kd}\,\mathsf{E}\,K(\ell)$ occurs in E'; or*
2. *K depends on c, $c \leq_C c'$, and K' declassifies c'.*

Example 28 *We illustrate a confidentiality leak when a low-integrity ciphertext is decrypted and the resulting plaintext is later declassified. For a given CPA encryption scheme $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$, we define a modified scheme $(\mathcal{G}_e, \mathcal{E}', \mathcal{D}')$ as follows:*

$$\begin{array}{rcl} \mathcal{E}'(x, k_e) & \doteq & r := random\_plaintext(); \\ & & \text{if } \mathcal{D}(\mathcal{E}(r, k_e), x) = r \text{ then } 0|x \text{ else } 1|\mathcal{E}(x, k_e) \\ \mathcal{D}'(b|y', k_d) & \doteq & \text{if } b = 0 \text{ then } k_d \text{ else } \mathcal{D}(y', k_d) \end{array}$$

In the modified scheme, the encryption of a decryption key with the corresponding encryption key leaks the decryption key. This scheme is still CPA since an adversary would have to guess the decryption key for a successful attack. However, this scheme leaks k_d when the decryption of a compromised plaintext appears in an encryption:

$$\begin{array}{l} k_e, k_d := \mathcal{G}_e(); \\ y_{LL} := \mathcal{E}(0, k_e); \\ \_; \\ x_{HL} := \mathcal{D}(y_{LL}, k_d); \\ y'_{LL} := \mathcal{E}(x_{HL}, k_e); \end{array}$$

This command has a key-dependency cycle, but it is otherwise typable, and it is unsafe: we may fill the placeholder with a command injecting an ill-formed ciphertext; thus $y'_{LL} = x_{HL} = k_d$ and the adversary can obtain the private key by reading y'_{LL}.

5.3.4 Typing CCA2 encryptions

If the scheme is CCA2 secure, we may use an additional rule for typing decryptions that takes advantage of its stronger guarantees. The rule is given below.

$$\frac{\begin{array}{l}(\textsc{Decrypt CCA2}) \\ (\mathcal{G}_e, \mathcal{E}, \mathcal{D}) \text{ is CCA2} \\ \tau \leq \Gamma(x) \qquad \tau \in \mathsf{E} \\ \vdash y : \mathsf{Enc}\, \tau K(L(x)) \\ \vdash k_d : \mathsf{Kd}\, \mathsf{E}\, K\, (\ell_d) \qquad \ell_d \leq_I x \qquad \ell_d \leq_I \alpha \textbf{ or } \ell_d \leq_C \alpha \\ \forall \tau' \in \mathsf{E}, \tau' \leq \tau \sqcup (\bot, \top) \end{array}}{\vdash x := \mathcal{D}(y, k_d) \ : L(x)}$$

Except for the CCA2 cryptographic assumption, the rule differs from rule Decrypt only on the last line of hypotheses, so Decrypt CCA2 effectively adds a fourth case for decryptions of low-integrity ciphertexts. In this new case, the adversary may be able to mix encryptions for values of type τ with those of values of any other type $\tau' \in \mathsf{E}$, so we must statically exclude some of the resulting flows between plaintexts: τ and τ' must have the same datatypes at the same levels of confidentiality, as illustrated below.

Example 29 (Ciphertext Rewriting) *Consider a command context using the same keypair for payloads at levels HH and LL:*

$$k_e, k_d := \mathcal{G}_e(); \\ y_{LL} := \mathcal{E}(x_{LL}, k_e); y'_{LL} := \mathcal{E}(x_{HH}, k_e); \_; x'_{LL} := \mathcal{D}(y_{LL}, k_d)$$

The command is not typable, and is unsafe: indeed, we may fill the placeholder with an adversary command $y_{LL} := y'_{LL}$, thereby causing the program to leak a copy of the secret x_{HH} into x'_{LL}.

5.4 Blinding Schemes: Security and Typing

We now consider blinding schemes introduced by Blaze et al. [1998] and also known as reencryption schemes. These schemes enable the re-randomization of a ciphertext, effectively providing a new ciphertext, which is difficult to link to the original one.

To precisely keep track of their information flows, we separate encryption into two stages, each with its own primitive and typing rule:

- Pre-encryption $\mathcal{P}()$ inputs a plaintext and outputs its representation as a ciphertext, but does not in itself provides confidentiality; it can be deterministic; it is typed as an ordinary operation and does not involve declassification.

- Blinding $\mathcal{B}()$ operates on ciphertexts; it hides the correlation between its input and its output, by randomly sampling another ciphertext that decrypts to the same plaintext; it is typed with a declassification, similarly to rule Encrypt.

As shown below, some standard encryption schemes can easily be decomposed into pre-encryption and blinding. This enables us for instance to blind a ciphertext without knowing its plaintext, and to perform multiple operations on ciphertexts before blinding. For conciseness, we may still write $\mathcal{E}$ instead of $\mathcal{P};\mathcal{B}$ when the two operations are executed together. We define the functional properties of blinding encryption schemes below:

Definition 26 (Blinding Scheme) *A blinding encryption scheme is a tuple $(\mathcal{G}_e, \mathcal{P}, \mathcal{B}, \mathcal{D})$ such that $(\mathcal{G}_e, \mathcal{P}; \mathcal{B}, \mathcal{D})$ is an encryption scheme and $\mathcal{B}$ is a probabilistic function such that, for all $k_e, k_d := \mathcal{G}_e()$, if v encrypts m, then $\mathcal{D}(v, k_d) = m$, where 'encrypts' is defined by*

1. *v encrypts m when $v := \mathcal{B}(\mathcal{P}(m, k_e))$ with $m \in$ plaintexts;*
2. *v' encrypts m when $v' := \mathcal{B}(v, k_e)$ and v encrypts m.*

Blinding hides whether an encrypted value is a copy of another, as shown in the example below:

Example 30 *Consider a service that, depending on a secret, either forwards or overwrites an encrypted message. We distinguish a third confidentiality level S such that $L \leq_C S \leq_C H$.*

$$
\begin{array}{l}
k_e, k_d := \mathcal{G}_e(); \\
y_{LH} := \mathcal{E}(m_{HH}, k_e); \\
\text{if } s_{SH} \text{ then } y_{SH} := \mathcal{P}(m_{SH}, k_e) \text{ else } y_{SH} := y_{LH}; \\
y'_{LH} := \mathcal{B}(y_{SH}, k_e)
\end{array}
$$

The resulting ciphertext y_{SH} is itself secret, as an adversary may otherwise compare it with y_{LH} and infer the value of s_{HH}. After blinding, however, y'_{LH} still protects the same message but can be safely treated as public, as an adversary reading y_{LH} and y'_{LH} learns nothing about m_{HH} or s_{SH}.

5.4.1 CPA for Blinding

We now describe the cryptographic assumptions necessary to use blinding schemes. A blinding scheme $(\mathcal{G}_e, \mathcal{P}, \mathcal{B}, \mathcal{D})$ is secure when for every possible ciphertext, the blinding operation hides any link between the original and the resulting ciphertext. This corresponds to a property of CPA for the scheme $(\mathcal{G}_e, \mathcal{B}, \mathcal{D})$. This property is stronger than the CPA property of the original scheme $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$, since $\mathcal{B}$ must be hiding for every ciphertext, not only those that are the direct result of the first phase of the encryption $(\mathcal{P})$. It is possible to shown this property for the blinding schemes based on ElGamal [1984] $(\mathcal{G}_e^{\mathsf{E}}, \mathcal{P}^{\mathsf{E}}, \mathcal{B}^{\mathsf{E}}, \mathcal{D}^{\mathsf{E}})$, and Paillier [1999] $(\mathcal{G}_e^{\mathsf{P}}, \mathcal{P}^{\mathsf{P}}, \mathcal{B}^{\mathsf{P}}, \mathcal{D}^{\mathsf{P}})$. The proof are similar to the proof of CPA for the original encryption shemes.

Proposition 1 $(\mathcal{G}_e^{\mathsf{E}}, \mathcal{B}^{\mathsf{E}}, \mathcal{D}^{\mathsf{E}})$ *is CPA.*

Proposition 2 $(\mathcal{G}_e^{\mathsf{P}}, \mathcal{B}^{\mathsf{P}}, \mathcal{D}^{\mathsf{P}})$ *is CPA.*

5.4.2 Typing rules for Pre-Encryption and Blinding

The typing rules for pre-encryption and blinding appear in Figure 5.3.

Pre-Encrypt This rule is similar to Encrypt, but constrains the confidentiality flow from e to y, which forbids any declassification by typing.

$$\frac{\Gamma(y) = \mathsf{Enc}\,\tau K q(\ell_y) \quad \vdash e : \tau \quad \tau \leq_C y \quad \vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \quad \tau \in \mathsf{E}}{\vdash y := \mathcal{P}(e, k_e) \ : \ell_y}\ (\textsc{Pre-Encrypt})$$

$$\frac{(\mathcal{G}_e, \mathcal{B}, \mathcal{D}) \text{ is CPA} \quad \Gamma(y) = \mathsf{Enc}\,\tau\, K q(\ell_y) \quad \tau' \leq \tau \quad \vdash z : \mathsf{Enc}\,\tau'\, K q(L(\tau)) \quad \vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \quad \tau, \tau' \in \mathsf{E} \quad \alpha \not\leq_I k_e \ \textbf{or} \ \tau \leq_C \alpha}{\vdash y := \mathcal{B}(z, k_e) \ : \ell_y}\ (\textsc{Blind})$$

$$\frac{\mathcal{F}_K(f) = f_K : \widetilde{q} \to q' \quad \Gamma(y) = \mathsf{Enc}\,\tau\, K q'(\ell_y) \quad \vdash z_i : \mathsf{Enc}\,\tau_i\, K q_i(\ell_y) \ \text{ for } z_i \in \widetilde{z} \quad \vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \quad \widetilde{\tau}, \tau \in \mathsf{E} \quad \Gamma, x : \tau \sqcup \ell_y, \widetilde{x} : \vec{\tau} \sqcup \ell_y \vdash x := f(\widetilde{x}) : L(\tau) \sqcup \ell_y}{\vdash y := f_K(\widetilde{z}, k_e) \ : \ell_y}\ (\textsc{Hom-Fun})$$

Figure 5.3: Additional typing rules for pre-encryption, blinding, and encryption homomorphisms with policy Γ.

Blind The first three hypotheses bind types to the variables k_e, z, and y. These types are related by $\tau, \tau' \in \mathsf{E}$ and K, which are also typing assumptions for encryptions with key k_e.

The label ℓ_y in the typing of k_e records the flow from k_e to y (by subtyping, we must have $k_e \leq \ell_y$).

The label $L(\tau)$ in the typing of z records the flow from z to the encrypted value in y. The hypothesis $\tau \leq \tau'$ ensures the correctness of the flow from the encrypted value in z to the encrypted value in y. Similarly to Encrypt, there is no constraint on the flow from z to y, so our typing rule may be sound only with cryptographic assumptions. When $z \not\leq_C y$, we say that K *declassifies* $C(z)$.

5.5 Homomorphic encryptions

We now consider encryption schemes with homomorphic properties: some functions on plaintexts can instead be computed on their ciphertexts, so that the command that performs the computation may run at a lower level of confidentiality. These schemes enable private remote evaluation: supposing that f_K is a function that homomorphically compute a function f, a client may delegate its evaluation to a server as follows:

1. the client encrypts the secret plaintext x into z_1;
2. the server applies f_K to the encrypted value (possibly encrypting its own secret inputs);
3. the client decrypts the result.

Programmatically, to implement $x' := f(x)$, we use a sequence of three commands sharing the variables z and z' and the encryption key k_e:

$$z := \mathcal{E}(x, k_e); z' := f_K(z, k_e); x' := \mathcal{D}(z', k_d)$$

For simplicity, we do not consider probabilistic homomorphic functions, or homomorphic functions that take non-ciphertext arguments.

As a first, trivial example, blinding can be seen as an homomorphic computation, associated with the identity function on plaintexts. The two sample schemes of Section 5.4 have more interesting homomorphic functions.

The number of consecutive homomorphic applications is typically bounded, either because "noise" is accumulated after each application and may lead to incorrect decryptions or because homomorphic applications of certain functions change the domain of the ciphertext. We use the encryption index to model these limitations.

We define the functional properties of homomorphic encryption schemes below:

Definition 27 (Homomorphic Encryption scheme) *An homomorphic encryption scheme is a tuple* $(\mathcal{G}_e, \mathcal{P}, \mathcal{B}, \mathcal{D}, \mathcal{F})$ *such that* $(\mathcal{G}_e, \mathcal{P}, \mathcal{B}, \mathcal{D})$ *is a blinding encryption scheme and* $\mathcal{F}$ *is a partial map on polynomial deterministic functions such that for all* $k_e, k_d := \mathcal{G}_e()$ *if* v *encrypts*$_q$ m*, then* $\mathcal{D}(v, k_d) = m$*, where 'encrypts' is defined by*

1. v *encrypts*$_0$ m *when* $v := \mathcal{B}(\mathcal{P}(m, k_e))$ *with* $m \in$ *plaintexts;*
2. v' *encrypts*$_q$ m *when* $v' := \mathcal{B}(v, k_e)$ *and* v *encrypts*$_q$ m*.*
3. v *encrypts*$_{q'}$ $f(m_1, \ldots, m_n)$ *when* $f \mapsto f_K : q \to q' \in \mathcal{F}$*,* $v := f_K(v_1, \ldots, v_n)$*, and each* v_i *encrypts*$_q$ m_i *for* $i \in 1..n$*.*

Also, practical encryption schemes usually support a few fixed functions f, and often put limits on the number of consecutive applications of f_K. (This is the purpose of indexes in our model.) Intuitively, these homomorphic application are not perfect; they also produce noise, which may eventually lead to incorrect decryptions. For example, Boneh et al. [2005] provides a scheme with an unlimited number of additions but only one homomorphic multiplication. In our model, this translate to: $\mathcal{F} = \{+ \mapsto +_K : 0 \to 0; * \mapsto *_K : 0 \to 1; + \mapsto +_K : 1 \to 1\}$.

Figure 5.3 includes our additional rule for homomorphic computation, Hom-Fun. This rule is parametric on the homomorphic encryption scheme, and relies on a typing assumption on the corresponding 'virtual' computation on plaintexts, using a security policy extended with fresh plaintext variables: $\Gamma, x : \tau \sqcup \ell_y, \widetilde{x} : \widetilde{\tau} \sqcup \ell_y \vdash x := f(\widetilde{x}) : L(\tau) \sqcup \ell_y$.

The first hypotheses bind types to the variables k_e, y, and $\vec{z}$ involved in the homomorphic operation; these types are related by $\vec{\tau}, \tau \in \mathsf{E}$ and K, which describe the typing assumptions for encryptions with key k_e. They also ensure that the function f is homomorphically implementable in this encryption scheme with the corresponding indexes. The last hypothesis ensures that f would be typable if applied on the decrypted ciphertext.

The label ℓ_y in the typing of k_e, y, and $\widetilde{z}$ records the flow from k_e and $\widetilde{z}$ to y.

In comparison, without cryptographic assumptions, we can also type homomorphic operations as probabilistic expressions using TFun, as follows:

$$\frac{\vdash k_e : \mathsf{Data}\,(\ell_e) \qquad \vdash \widetilde{y} : \mathsf{Data}\,(\ell_e) \qquad \vdash x : \mathsf{Data}\,(\ell_e)}{\vdash x := f_K(\widetilde{y}, k_e) \; : \ell_e} \;(\textsc{Hom-Fun}\ (\textsc{by}\ \textsc{TFun}))$$

Our typing rule ensures that if an homomorphic function types, its equivalent non-homomorphic version would also type, as illustrated in Example 31.

5.5.1 Examples

The examples below use Paillier encryption, which enables us to add plaintexts by multiplying their ciphertexts. Thus, we use $f = +$ and $f_K = *$, and for $\mathcal{F}(+) = * : 0 \mapsto 0$ we obtain a single

instance of Rule Hom-Fun:

$$\frac{\begin{array}{c}\text{(Hom-Paillier)}\\ \Gamma(y) = \mathsf{Enc}\,(\mathsf{Data}\,(\ell))\,K\,(\ell_y)\\ \vdash z_i : \mathsf{Enc}\,(\mathsf{Data}\,(\ell_i))\,K\,(\ell_y) \text{ for } i = 0,1\\ \vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \qquad \mathsf{Data}\,(\ell), \mathsf{Data}\,(\ell_0), \mathsf{Data}\,(\ell_1) \in \mathsf{E}\\ \ell_0 \sqcup \ell_1 \sqcup \ell_y \leq \ell \sqcup \ell_y\end{array}}{\vdash y := z_0 * z_1 \; : \ell_y}$$

Example 31 *To illustrate our typing rule, we compare two programs that perform the same addition, on plaintexts (on the left) and homomorphically (on the right):*

$$\begin{array}{rcl@{\qquad}rcl} P & \doteq & k_e, k_d := \mathcal{G}_e(); & P' & \doteq & k_e, k_d := \mathcal{G}_e();\\ & & z_1 := \mathcal{E}(x_1, k_e); & & & z_1 := \mathcal{E}(x_1, k_e);\\ & & x_1' := \mathcal{D}(z_1, k_d) & & & \\ & & x := x_1' + x_1'; & & & y := z_1 * z_1;\\ & & y := \mathcal{E}(x, k_e); & & & \\ & & r := \mathcal{D}(y, k_d); & & & r := \mathcal{D}(y, k_d);\end{array}$$

Suppose that we type P with a policy Γ, then, relying on the plaintext-typing assumption in rule Hom-Fun*, we can also type P' with the same policy.*

Example 32 *We show how to homomorphically multiply an encrypted value by a small integer factor.*

$$\begin{array}{l} k_e, k_d := \mathcal{G}_e();\\ y_{LH} := \mathcal{E}(x_{HH}, k_e);\\ z_{SH} := \mathcal{E}(0, k_e);\\ \mathsf{for}\; i_{SH} := 1 \;\mathsf{to}\; n_{SH} \;\mathsf{do}\; z_{SH} := z_{SH} * y_{LH};\\ z_{LH} := \mathcal{B}(z_{SH}, k_e); \_;\\ x_{HH} := \mathcal{D}(z_{LH}, k_d)\end{array}$$

By typing, we have that an honest-but-curious adversary ($\alpha = SH$) does not learn x_{HH}, and that a network adversary ($\alpha = LH$) learns neither x_{HH} nor n_{SH}. The latter guarantee crucially relies on blinding the result at the end of the loop, since otherwise the adversary may also iterate multiplications of y_{LH} and compare them to the result to guess n_{SH}.

Another classic example of homomorphic scheme is the ElGamal encryption, which enables us to multiply plaintexts by multiplying their ciphertexts; we omit similar, typable programming examples.

Homomorphic properties are usually incompatible with CCA2 security, as illustrated below for an additive scheme:

Example 33 (Homomorphic schemes cannot be CCA2) *For a given homomorphic scheme $(\mathcal{G}_e, \mathcal{P}, \mathcal{B}, \mathcal{D}, \mathcal{F})$ with $+ \in Dom(\mathcal{F})$, consider the CCA2 adversary:*

$$\begin{array}{rcl} A[E, D] & \doteq & x_0 := 1; x_1 := 2; E;\\ & & m := \mathcal{F}(+)(m, m, k_e); D;\\ & & \text{if } x = 2 \text{ then } g := 0 \text{ else } g := 1\end{array}$$

The result from the encryption oracle E and the input of the decryption oracle D are different. Hence, D is going to decrypt is input. However, since the scheme is homomorphic, the result of the decryption is going to be either 2 or 4, depending on the initial encrypted value. Our adversary always wins the CCA2 game.

5.6 Computational Soundness

The security of cryptographic programs depends on cryptographic assumptions on probabilistic games. In particular, we cannot prove non-interference of these programs, since high confidentiality values leaks to low confidentiality values during an encryption and a lucky adversary could guess the keys and decrypt the ciphertext. Hence, we define security properties for probabilistic command contexts as computational variants of noninterference, expressed as games [Fournet and Rezk, 2008]. For instance, the confidentiality version of the game states that an adversary that can only observe public variables cannot distinguish between two runs of a program with the same public initialization.

Definition 28 (Computational Non-Interference) *Let Γ be a policy, α an adversary level, and P a command context.*

P is computationally non-interferent against α-adversaries when, for both $V = V_{\mathcal{A}}^{C}$ and $V = V_{\mathcal{A}}^{I}$, and for all polynomial commands J, B_0, B_1, T containing no cryptographic variables, such that $wv(B_b) \cap V = \emptyset$ and $rv(T) \subseteq V$, $\widetilde{A}$ α-adversaries such that $P[\widetilde{A}]$ is a polynomial command, $g \notin v(J, B_0, B_1, P, \widetilde{A})$, and some variable $b \notin v(J, B_0, B_1, P, \widetilde{A}, T)$ in the command

$$CNI \doteq b := \{0,1\}; J; \text{if } b = 0 \text{ then } B_0 \text{ else } B_1; P[\widetilde{A}]$$

if we have $\Pr[CNI; \bigwedge_{x \in rv(T)} x \neq \bot] = 1$, then the advantage $|\Pr[CNI; T; b =_0 g] - \frac{1}{2}|$ is negligible.

In the game of the definition, the command $J; \text{if } b = 0 \text{ then } B_0 \text{ else } B_1$ probabilistically initializes the memory, depending on b. Definition 28, then runs the command context applied to adversaries $P[\widetilde{A}]$. Finally, T attempts to guess the value of b and set g accordingly. T cannot read cryptographic variables, which are not an observable outcome of the program. The condition $\bigwedge_{x \in rv(T)} x \neq \bot$ rules out commands T that may read uninitialized memory. Hence, the property states that the two memory distributions for $b = 0$ and $b = 1$ after running CNI cannot be separated by an adversary that reads V.

5.6.1 Assigning cryptographic schemes and lengths to types

Our cryptographic assumptions demands the length of the plaintext to be fixed (otherwise the adversary can get information about the plaintext by measuring the length of the ciphertext. Hence, each security type is given a length.

To every type τ, we associate the binary length $\lceil \tau \rceil(\eta)$ of any value of that type, assumed to be a polynomial in η. (In a more detailed model with values of variable sizes, we would include explicit marshaling, truncating, and padding operations.) We also let $\lceil plaintexts_K \rceil$ be the binary length of any value in $plaintexts_K$, also a polynomial in η.

We cannot use a single instance of each cryptographic scheme because there is no correct CCA2 encryption scheme for unbounded plaintext, and because e.g. ciphertexts for one scheme may be treated as plaintext for some other schemes. On the other hand, our model must account for potential information leakage via the length of encrypted messages.

We assume the constraints below on the relative lengths (for each η) for each scheme for

encryption or for signing.

$$\begin{aligned}
\lceil t(\ell) \rceil &= \lceil t \rceil \\
\lceil \tau * \tau' \rceil &= \lceil \tau \rceil + \lceil \tau' \rceil \\
\lceil \mathsf{Array}\, \tau \rceil &\text{ is a fixed polynomial in } \eta \text{ and } \lceil \tau \rceil \\
\lceil \mathsf{Data} \rceil &\text{ is a fixed polynomial in } \eta \\
\lceil \mathsf{Enc}\, \tau\, K \rceil &= \lceil \mathit{ciphertexts}_K \rceil \text{ with } \lceil \tau \rceil = \lceil \mathit{plaintexts}_K \rceil \\
\lceil \mathsf{Ke}\, \mathsf{E}\, K \rceil &= \lceil \mathit{publickeys}_K \rceil \\
\lceil \mathsf{Kd}\, \mathsf{E}\, K \rceil &= \lceil \mathit{secretkeys}_K \rceil
\end{aligned}$$

(One can solve these constraints given schemes parametrized by the size of their plaintexts, starting with the inner types.)

5.6.2 Safety

Besides the cryptographic assumptions, we state additional safety conditions for soundness.

Definition 29 *Let $\alpha \in \mathcal{L}$ be security label. A command context P is safe against α-adversaries when $\Gamma \vdash P$ and*

1. *Each key label is used in at most one (dynamic) key generation.*
2. *Each cryptographic variable read in P is first initialized by P.*
3. *There is no dependency cycle between key labels.*

We rely on these conditions to apply cryptographic games in the proof of type soundness, for instance to guarantee the integrity of decrypted values. They can be enforced by static analysis, for instance by collecting all relevant static occurrences of variables and forbidding encryption-key generation within loops.

Condition 1 prevents decryption-key mismatches. Condition 2 recalls our assumption on uninitialized cryptographic variables. Condition 3 recalls our assumption on key dependencies.

We also assume that each static key label is associated to a fixed scheme for encryption that meets Definitions 18, 26, or 27 and we impose constraints on the length of ciphertexts in order to prevent information leakage via the length of encrypted messages (explained above).

Relying on these conditions, we obtain our main security theorem: well-typed programs are computationally non-interferent (Definition 28).

Theorem 10 *Let $\alpha \in \mathcal{L}$ be a security label. Let Γ be a policy. Let P be a polynomial-time command context, safe against α-adversaries. P satisfies computational non-interference against α-adversaries.*

The proof of Theorem 10 is at the end of this chapter, in Subsection 5.9.

5.7 Private Search on Data Streams

We illustrate the use of Paillier encryption on a simplified version of a practical protocol developed by Ostrovsky and Skeith III [2005] for privately searching for keywords in data streams (without the Bloom filter).

The protocol has two roles: an agency P and a service S. Assume that the service issues, or processes, confidential documents such as mail orders or airline tickets, and that the agency wishes to retrieve any such document whose content matches some keywords on a secret black list. The two roles communicate using a public network. The black list is too sensitive to be given to the service. Conversely, the service may be processing a large number of documents, possibly at many different sites, and may be unwilling to pass all those documents to the agency.

We formally assume that the agency is more trusted than the service. We suppose that the documents are arrays of words, and that all words (including the keywords on the black list) appear in a public, trusted dictionary. We model the public network using variables shared between P and S. We rely on the additive property of Paillier encryption, detailed in Section 5.5. We code the protocol using three commands, explained below.

Initially, the agency generates a keypair and encodes the list of keywords as an array ($mask$) of encryptions indexed by the public dictionary that contains, for each word, either an encryption of 1 if the word appears in the black list or an encryption of 0 otherwise. The agency can distribute this array to the service without revealing the black list.

$$\begin{array}{rcl} P_0 & \doteq & k_e, k_d := \mathcal{G}_e(); \\ & & \mathsf{for}\ i := 0\ \mathsf{to}\ size(words) - 1\ \mathsf{do} \\ & & \quad \mathsf{if}\ words[i] \in keywords\ \mathsf{then}\ wb := 1\ \mathsf{else}\ wb := 0; \\ & & \quad mask[i] := \mathcal{E}(wb, k_e) \end{array}$$

Then, for each document d, the service homomorphically computes the number of matching keywords, as the sum of the 0 and 1s encrypted in $mask$ for all indexes of the words ($d[j]$) present in the document, by multiplying those encryptions in e. Moreover, the service homomorphically multiplies e and the value of the document d (seen as a large integer) in z—the loop computes e^d, and may be efficiently replaced with a fast exponentiation. In particular, if they were no matches, z is just an encryption of 0. Finally, the service blinds both e and z before sending them to the agency—this step is necessary to declassify these encryptions without leaking information on the document.

$$\begin{array}{rcl} S & \doteq & e := 1; \\ & & \mathsf{for}\ j := 0\ \mathsf{to}\ size(d) - 1\ \mathsf{do} \\ & & \quad e := e * mask[words^{-1}(d[j])]; \\ & & e' := \mathcal{B}(e, k_e); \\ & & z := 1; \\ & & \mathsf{for}\ j := 0\ \mathsf{to}\ d - 1\ \mathsf{do}\ z := z * e; \\ & & z' := \mathcal{B}(z, k_e) \end{array}$$

Last, for each pair e' and z', the agency decrypts the value containing the number of matches (in n) and, if n is different from 0, retrieves the value of the document by decrypting the product and dividing by n.

$$\begin{array}{rcl} P & \doteq & n := \mathcal{D}(e', k_d); \\ & & \mathsf{if}\ n > 0\ \mathsf{then} \\ & & \quad nd := \mathcal{D}(z', k_d); log := log + (nd/n) \end{array}$$

Adding an outer loop for the documents processed by the service, the whole protocol is modeled as the command

$$\begin{array}{rcl} Q & \doteq & P_0; \\ & & \mathsf{for}\ l := 0\ \mathsf{to}\ nb\_docs\ \mathsf{do} \\ & & \quad d := docs[l]; S; P; \_ \end{array}$$

As discussed by Ostrovsky and Skeith III, this algorithm is reasonably efficient for the service, as it requires just a multiplication and a table lookup for each word in each document. Relying on Bloom filters, the authors of the protocol and Danezis and Diaz [2007] develop more advanced

variants, requiring less bandwidth and guaranteeing additional privacy properties for the service, but the overall structure of the protocol remains unchanged.

We now specify the security of the protocol as a policy Γ that maps the variable of Q to labels. We distinguish three levels of integrity: H for the agency, S for the service, and L for untrusted data, with $H <_I S <_I L$.

- k_e and *keywords* have high integrity (H);
- all other variables have service integrity (S).

We also distinguish four levels of confidentiality: H and P for the agency, S for the service, and L for public data. With $L <_C P <_C H$ and $L <_C S <_C H$.

- *words*, *mask*, e', z', i, *nb_docs*, and k_e are public (at level L);
- *docs*, d, e, z, j, l are readable by the service (level S) and the agency;
- *keywords*, *kd*, *bw* are readable only by the agency (level P);
- *log*, n, and *nd* are also readable by the agency, but moreover they depend on low-integrity decryptions, and thus should not flow to any further encryption with key k_e (level H).

We prove the security properties of the protocol as instances of computational non-interference, established by typing our code with different choices of adversary level α.

Theorem 11 $\Gamma \vdash Q$ *when* α *ranges over SH, SS, and LS, hence* Q *is computationally non interferent against these adversaries.*

- The case $\alpha = SS$ corresponds to a powerful adversary that entirely controls the service but still learns nothing about the blacklist: it cannot distinguish between any two runs of the protocol with different values of *keywords*.
- If the service is honest but curious ($\alpha = \mathcal{L}$), the protocol also completes with the correct result.
- If the adversary controls only the network ($\alpha = LL$), it learns nothing either about the documents contents (*docs*).

Danezis and Diaz [2007] develop more advanced variants of the protocol, requiring less bandwidth, such that S sends back only a small number of linear combinations of values n' and z', under the assumption that n is almost always 0. We can program and type such variants in our setting, with the same theorems; on the other hand, our type discipline does not capture the additional privacy guarantee from the service's viewpoint, e.g. that the volume of information passed to the agency is low as comparison with the stream of documents.

5.8 Bootstrapping Homomorphic Encryptions

Gentry [2009] proposes a first fully homomorphic encryption (FHE) scheme, that is, a scheme that supports arbitrary computations on encrypted data, thereby solving a long-standing cryptographic problem [Goldwasser and Micali, 1982]. Others, e.g. Smart and Vercauteren [2010], develop more efficient constructions towards practical schemes.

These constructions are based on *bootstrappable encryption schemes*, equipped with homomorphic functions for the scheme's own decryption function as well as some more basic functions;

these basic operations may represent e.g. logical gates. (Designing a bootstrappable encryption scheme is in itself very challenging, as it severely constrains the decryption algorithm, but is not our concern here.)

The bootstrap relies on a series of keys. Whenever the noise accumulated as the result of homomorphic evaluation needs to be canceled, the intermediate result (encrypted under the current key) is encrypted using the next encryption key, then homomorphically decrypted using an encryption of the current decryption key under the next encryption key. This bootstrapping yields an homomorphic scheme for any function as long as the computation can progress using other homomorphic functions between encryptions and decryptions to the next key. (Following Gentry's terminology, the resulting scheme is a *leveled* FHE: the key length still depends on the max size of the circuit for evaluating f; some additional key-cycle assumption is required to get a fixed-length key.)

Next, we assume the properties of the base algorithms given by Gentry—being CPA and homomorphic for its own decryption plus basic operations f_i that suffice to evaluate some arbitrary function f. We then show that these properties suffice to program and verify by typing the bootstrapping part of Gentry's construction, leading to homomorphic encryption for this arbitrary function f. Typing of Gentry's bootstrapping relies on typing rules for CPA with multiple keys and types of encrypted values, and on instances of rule Hom-Fun with $f = \mathcal{D}$.

We assume given a bootstrappable scheme using a set of homomorphic functions for decrypting and for computing: $\mathcal{F} = \{\mathcal{D} \mapsto \mathcal{D}_K : 0 \to 1; f^i \mapsto f^i_K : 1 \to 2 \text{ for } i = 1..n\}$.

We are now ready to program the bootstrapping for f. Both parties agree on the function to compute, f, and its decomposition into successive computation steps $(f_i)_{i=1..n}$ such that $f = f_1; \ldots; f_n$. This may be realized by producing a circuit to evaluate f and then letting the f_i be the successive gates of the circuit. For simplicity, we assume that each f_i operates on all the bits of the encrypted values; Gentry instead uses a tuple of functions $f_i^{j=1..w}$ for each step, each f_i^j computing one bit of the encrypted values. This more detailed bootstrapping can be typed similarly.

For the plaintexts, we write x_0 for the input, $x_1, \ldots x_{n-1}$ for the intermediate results, and x_n for the final result, so the high-level computation is just $x_i := f_i(x_{i-1})$ for $i = 1..n$.

The client generates all keypairs; encrypts the input using the first encryption key; encrypts each decryption key using the next encryption key; calls the server; and decrypts the result using the final key:

$$\begin{array}{lll} P_c \doteq ke_i, kd_i := \mathcal{G}_e(); & & \text{for } i = 1..n \\ \quad skd_i \quad := \mathcal{E}(kd_i, ke_{i+1}); & & \text{for } i = 1..n-1 \\ \quad y_0 \quad := \mathcal{E}(x_0, ke_1); & & \\ \quad \_; & & \text{calling the server} \\ \quad x_n \quad := \mathcal{D}(z_n, kd_n) & & \end{array}$$

The message consists of the content of the shared variables y_0, ke_1, skd_1, …, ke_{n-1}, skd_{n-1}, ke_n. The server performs the homomorphic computation for each f_i, interleaved with the re-keying.

$$\begin{array}{lll} P_s \doteq & (z_i := f^i_{K_i}(y_{i-1}, ke_i); & \\ & \quad t_i := \mathcal{E}(z_i, ke_{i+1}); & \\ & \quad y_i := \mathcal{D}_{K_{i+1}}(t_i, skd_i, ke_{i+1});) & \text{for } i=1..n-1 \\ & z_n := f^n_{K_n}(y_{n-1}, ke_n); \_ & \end{array}$$

We specify the security of the protocol as a policy Γ that maps the variable of P_c and P_s to labels. We distinguish three levels of integrity: H for the agency, S for the service, and L for untrusted data, with $H <_I S <_I L$.

- ke_i and kd_i have high integrity (H) for $i = 1..n$;
- all other variables have service integrity (S).

Since our scheme is only CPA, we need $n+1$ levels of confidentiality to separate the results of low integrity decryptions: $H_0..H_n$ for the agency and L for public data, such that $L <_C H_0 <_C \cdots <_C H_n$.

- $ke_i, skd_i, z_i, t_i, y_i$ are public for $i = 1..n$ (at level L);
- x_0 is readable only by the agency (level H_0);
- kd_i is readable only by the agency for $i = 1..n$ (level H_i);
- x_n is also readable by the agency, but moreover it depends on low-integrity decryptions, and thus should not flow to any further encryption with any of the key ke_i for $i = 1..n$ (level H_n).

Let $\tau_i = \mathsf{Data}\,(H_i S)$. We type our variables as follows, for $i = 1..n-1$.

$$\begin{array}{ll}
\Gamma(ke_i) & = \mathsf{Ke}\,\mathsf{E}_i\,K_i(LH) \\
\Gamma(kd_i) & = \mathsf{Kd}\,\mathsf{E}_i\,K_i(H_iH) \\
\Gamma(x_0) & = \tau_0 \\
\Gamma(skd_i) & = \mathsf{Enc}\,(\mathsf{Kd}\,\mathsf{E}_i\,K_i(H_{i-1}S))K_{i+1}0(LS) \\
\Gamma(y_0) & = \mathsf{Enc}\,\tau_0\,K_1 0(LS) \\
\Gamma(z_i) & = \mathsf{Enc}\,\tau_{i-1}\,K_i 2(LS) \\
\Gamma(t_i) & = \mathsf{Enc}\,(\mathsf{Enc}\,\tau_{i-1}\,K_i 2(LS))\,K_{i+1}0(LS) \\
\Gamma(y_i) & = \mathsf{Enc}\,\tau_i\,K_{i+1}1(LS) \\
\Gamma(z_n) & = \mathsf{Enc}\,\tau_{n-1}\,K_n 2(LS) \\
\Gamma(x_n) & = \tau_n
\end{array}$$

with the encrypted sets below, for $i = 2..n$:

$$\begin{array}{l}
\mathsf{E}_1 = \{\tau_0\} \\
\mathsf{E}_i = \{\tau_{i-1}, \mathsf{Kd}\,\mathsf{E}_{i-1}\,K_{i-1}(H_{i-2}S), \mathsf{Enc}\,\tau_{i-1}\,K_{i-1}(HS)\}
\end{array}$$

We verify that the commands above are typable with the resulting policy Γ:

Theorem 12 *$\Gamma \vdash P_c[P_s]$ when α ranges over LL and LS, hence $P_c[P_s]$ is computationally non-interferent against these adversaries.*

The theorem yields strong indistinguishability guarantees.

- The case $\alpha = LL$ corresponds to an "honest but curious" adversary, who can observe all server operations on ciphertexts (that is, read all intermediate values) but still learn nothing about the plaintexts.
- The case $\alpha = LS$ corresponds to an active adversary in full control of the service, who can disrupt its operations, and still learns nothing about the plaintexts.

To clarify these confidentiality guarantees, we verify by typing that our construction is in particular

CPA. We let $\mathcal{S}$ abbreviate the scheme $(\mathcal{G}_e^h, \mathcal{E}^h, \mathcal{D}^h, \{f \mapsto f_K : 0 \to 1\})$ defined as follows:

$$
\begin{array}{rcl}
ke, kd := \mathcal{G}_e^h() & \doteq & \big(ke_i, kd_i \ := \ \mathcal{G}_e();\big)_{\text{ for } i=1..n} \\
& & \big(skd_i \ := \ \mathcal{E}(kd_i, ke_{i+1});\big)_{\text{ for } i=1..n-1} \\
& & ke \ := \ (ke_1, ..., ke_n, skd_1, skd_{n-1}); \\
& & kd \ := \ (kd_1, kd_n) \\
y := \mathcal{E}^h(e, ke) & \doteq & y \ := \ \mathcal{E}(e, (ke)_1); \\
y := f_K^h(z, ke) & \doteq & y_0 \ := \ z; \\
& & \big(z_i \ := \ f_{K_i}^i(y_{i-1}, (ke)_i); \\
& & \ t_i \ := \ \mathcal{E}(z_i, (ke)_{i+1}); \\
& & \ y_i \ := \ \mathcal{D}_{K_{i+1}}(t_i, (ke)_{i+n}, (ke)_{i+1});\big)_{\text{ for } i=1..n-1} \\
& & y \ := \ f_{K_n}^n(y_{n-1}, (ke)_n) \\
x := \mathcal{D}^h(y, kd) & \doteq & \text{if} \mathit{is\_enc}(y, (kd)_1) \\
& & \text{then } x := \mathcal{D}(y, (kd)_1) \\
& & \text{else } x := \mathcal{D}(y, (kd)_2)
\end{array}
$$

Theorem 13 *$\mathcal{S}$ is a CPA homomorphic encryption scheme.*

PROOF: We consider the program

$$
\begin{array}{rcl}
P & \doteq & ke, kd := \mathcal{G}_e^h(); \\
& & \_; \\
& & \text{if } b_{CPA} = 0 \text{ then } m := \mathcal{E}^h(x_0, k_e) \text{ else } m := \mathcal{E}^h(x_0', k_e) \\
& & \_;
\end{array}
$$

with the following types: Let $\tau_i = \mathsf{Data}\,(H_i L)$. We type our encryption variables as follows, for $i = 1..n-1$.

$$
\begin{array}{rl}
\Gamma(ke_i) & = \mathsf{Ke}\,\mathsf{E}_i\,K_i(LH) \\
\Gamma(kd_i) & = \mathsf{Kd}\,\mathsf{E}_i\,K_i(H_iH) \\
\Gamma(skd_i) & = \mathsf{Enc}\,(\mathsf{Kd}\,\mathsf{E}_i\,K_i(H_{i-1}H))K_{i+1}0(LH) \\
\Gamma(ke) & = \langle()\Gamma(ke_1), \ldots, \Gamma(ke_n), \Gamma(skd_1), \ldots, \Gamma(skd_{n-1})) \\
\Gamma(kd) & = \langle()\Gamma(kd_1), \Gamma(kd_n)) \\
\Gamma(x_0) & = \tau_0 \\
\Gamma(x_0') & = \tau_0 \\
\Gamma(m) & = \mathsf{Enc}\,\tau_0\,K_1 0(LL)
\end{array}
$$

with the encrypted sets below, for $i = 2..n$:

$$
\begin{array}{rl}
\mathsf{E}_1 & = \{\tau_0\} \\
\mathsf{E}_i & = \{\tau_{i-1}, \mathsf{Kd}\,\mathsf{E}_{i-1}\,K_{i-1}(H_{i-2}H), \mathsf{Enc}\,\tau_{i-1}\,K_{i-1}(HL)\}
\end{array}
$$

In particular

- ke, kd and b_{CPA} have high integrity
- m, x_0 and x_0' have low integrity
- kd is confidential
- ke, m, x_0 and x_0' are public

We have $\Gamma \vdash P$ for $\alpha = LL$. More precisely, P is safe against LL-adversaries. By Theorem 10, it is thus computationally non-interferent against LL-adversaries.

We instantiate Definition 28 for P, $B_0 = b_{CPA} := 0$,$B_1 = b_{CPA} := 1$, $J = skip$ and $T = skip$.For every polynomial A_1 and A_2 that reads ke and m, writes g, does not read b, kd nor write b, kd and ke, we get

$$\begin{aligned} CNI^h \doteq\ & b := \{0,1\}; \text{if } b = 0 \text{ then } b_{CPA} := 0 \text{ else } b_{CPA} := 1; \\ & ke, kd := \mathcal{G}_e^h(); \\ & A_1; \\ & \text{if } b_{CPA} = 0 \text{ then } m := \mathcal{E}^h(x_0, k_e) \text{ else } m := \mathcal{E}^h(x_0', k_e) \\ & A_2 \end{aligned}$$

or, with a simple rewriting:

$$\begin{aligned} CNI^h \doteq\ & b := \{0,1\}; ke, kd := \mathcal{G}_e^h(); \\ & A_1; \\ & \text{if } b = 0 \text{ then } m := \mathcal{E}^h(x_0, k_e) \text{ else } m := \mathcal{E}^h(x_0', k_e) \\ & A_2 \end{aligned}$$

$|\Pr[CNI^h; b =_0 g] - \frac{1}{2}|$ is negligible. This corresponds to the CPA game, hence $(\mathcal{G}_e^h, \mathcal{E}^h, \mathcal{D}^h)$ is CPA. □

5.9 Proof of Theorem 10

The proof of our main security theorem relies on a series of typability-preserving program transformations that match the structure of the games used in cryptographic security assumptions (Definitions 19). These transformations eliminate the cryptographic primitives, one static key label at a time. Hence, after eliminating a keypair for encrypting and decrypting, the values that were encrypted are now stored into auxiliary high-confidentiality logs, and we are left with encryptions of zero.

The proof is organized as follows: we first modify the lattice, and introduce new typing rules in order to be able to type $P[A]$, then, we show that typable programs with no declassification are non-interferent. Last, we show that if a typable program with cryptography breaks computational non-interference then it also breaks the hypothesis on the cryptographic schemes. In the whole section, we implicitly assume that all commands are polynomial. In the following, we will abuse memory notation μ to also denote a memory distribution with a unique point μ mapping to probability 1.

5.9.1 Reduced Lattice

Without loss of generality once α is set, we use a reduced lattice that collapses all levels controlled by the adversary.

Definition 30 (Reduced lattice) *Let $(\mathcal{L}, \leq)$ be a security lattice, $\alpha \in \mathcal{L}$ a security label, we defie a reduced lattice $\mathcal{L}_r$ with integrity levels L and H, and with confidentiality levels L, M, H, and c for $c \not\leq_C C(\alpha)$ in the lattice $\mathcal{L}$. The relations in this lattice are the smallest reflexive and transitive relations verifying: $H \leq_I L$, $L \leq_C M \leq_C c \leq_C H$ and $c \leq_C c'$ for every c and c' such that $c \not\leq_C C(\alpha)$, $c' \not\leq_C C(\alpha)$, and $c \leq c'$ in the lattice $\mathcal{L}$. Let $\mathcal{H}$ be a function from $\mathcal{L}$ to $\mathcal{L}_r$ defined by :*

$$\mathcal{H}(\ell) \doteq \begin{pmatrix} (\textit{if } C(\ell) \leq_C C(\alpha) \textit{ then } L \textit{ else } \ell), \\ (\textit{if } I(\alpha) \leq_I I(\ell) \textit{ then } L \textit{ else } H) \end{pmatrix}$$

Let $\mathcal{H}(t(\ell)) = t(\mathcal{H}(\ell))$, $\mathcal{H}(\tau_0 * \tau_1) = \mathcal{H}(\tau_0) * \mathcal{H}(\tau_1)$ and $\mathcal{H}(\mathsf{Array}\,\tau) = \mathsf{Array}\,\mathcal{H}(\tau)$. For a given policy Γ, let $\mathcal{H}(\Gamma)$ be such that $\mathcal{H}(\Gamma)(x) = \mathcal{H}(\Gamma(x))$.

We show below that any typing judgment is preserved by this reduction, and that program that meets the hypotheses of Theorem 10 for an arbitrary lattice also meets them in this more abstract reduced lattice. Hence, for the rest of the proof, we assume a reduced lattice.

Lemma 35 (Reduced Lattice) *Let $(\mathcal{L}, \leq)$ be a security lattice. Let $\alpha \in \mathcal{L}$ be a security label. We have:*

1. *The reduced lattice forms a product lattice such that $\mathcal{H}$ preserves $\leq$.*
2. *If $\Gamma \vdash e : \tau$, then $\mathcal{H}(\Gamma) \vdash e : \mathcal{H}(\tau)$.*
3. *If $\Gamma \vdash P : \ell$ with α then $\mathcal{H}(\Gamma) \vdash P : \mathcal{H}(\ell)$ with LH.*
4. *If P is safe with the policy Γ, then P it is safe with the policy $\mathcal{H}(\Gamma)$.*

Proof:

Property 1. By case analysis on security levels $\ell_0, \ell_1 \in \mathcal{L}$ such that $\ell_0 \leq \ell_1$, with three cases for confidentiality:

- $C(\mathcal{H}(\ell_1)) = L$, in which case $C(\mathcal{H}(\ell_0)) = L$ and $\mathcal{H}(\ell_0) \leq_C \mathcal{H}(\ell_1)$; or
- $C(\mathcal{H}(\ell_1)) = c_1 = C(\ell_1)$ and $C(\mathcal{H}(\ell_0)) = L$, in which case $\mathcal{H}(\ell_0) \leq_C \mathcal{H}(\ell_1)$; or
- $C(\mathcal{H}(\ell_1)) = c_1 = C(\ell_1)$ and $C(\mathcal{H}(\ell_0)) = c_0 = C(\ell_0)$, in which case $\mathcal{H}(\ell_0) \leq_C \mathcal{H}(\ell_1)$;

and two cases for integrity:

- $I(\mathcal{H}(\ell_0)) = L$, in which case $C(\mathcal{H}(\ell_1)) = L$ and $\mathcal{H}(\ell_0) \leq_I \mathcal{H}(\ell_1)$; or
- $I(\mathcal{H}(\ell_0)) = H$ in which case $\mathcal{H}(\ell_0) \leq_I \mathcal{H}(\ell_1)$.

We conclude that $\mathcal{H}(\ell_0) \leq \mathcal{H}(\ell_1)$.

Property 2. By induction on the expression typing derivation.

Case e is x. By typing Rule Var, we have $\Gamma \vdash x : \tau$ and $\mathcal{H}(\Gamma) \vdash x : \mathcal{H}(\tau)$.

Case e is $op(\widetilde{e})$. By typing Rule TOp, we have $op : \tau_1 \dots \tau_n \to \tau$ and $\Gamma \vdash e_i : \tau_i$ for $i = 1..n$. Hence, using the corresponding signature, $op : \mathcal{H}(\tau_1) \dots \mathcal{H}(\tau_n) \to \mathcal{H}(\tau)$ and by inductive hypothesis $\mathcal{H}(\Gamma) \vdash e_i : \mathcal{H}(\tau_i)$ for $i = 1..n$ We obtain, by typing Rule TOp $\Gamma \vdash op(e_1, \dots, e_n) : \mathcal{H}(\tau)$.

Property 3. It follows from Properties 2 and 1 since all typing rules depend only positively on $\leq$.

Property 4. By analyzing the definition of safe. A command is safe if it is typable and:

1. each key label is used in at most one (dynamic) key generation;
2. each key variable read in P is first initialized by P; and
3. there is no dependency cycle between key labels.

Typing of P with the policy $\mathcal{H}(\Gamma)$ follows from Property 3. Item 1 and 2 do not depend on the policy. Item 3 depend on $\not\leq_I \alpha$ and $\not\leq_C \alpha$, which are preserved by Property 1.

□

5.9.2 Extended typing rules and adversary typing

We do not require for adversary commands to be typed, thus it can write low-integrity values without respecting their datatypes. In order to formally type the adversary, we add the following typing rules:

$$\frac{\vdash e : \tau \qquad \tau \leq_C L \qquad L \leq_I \tau'}{\vdash e : \tau'} \text{(Coerce-LL)} \qquad \frac{\vdash e : \mathsf{Data}\,(LL) \qquad L \leq_I x}{\vdash \widetilde{x} := f(\widetilde{e}) : LL} \text{(Coerce-ProbFun-LL)}$$

The Coerce-LL typing rule is only applicable to low confidentiality expressions and permits to change their data type. In order to preserve soundness, the coerced expressions must be typed at a low integrity level.

We further extend the type system with typing rules for equality between encryptions:

$$\frac{\vdash e : \mathsf{Enc}\,\tau K(cH) \quad \vdash e' : \mathsf{Enc}\,\tau K(cH)}{\vdash e = e' : \mathsf{Data}\,(L(\tau) \sqcup cH)} \text{(Coerce-equality)} \qquad \frac{\Gamma(y) = \mathsf{Enc}\,\tau\,K(\ell_y) \qquad \vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \qquad \tau \in \mathsf{E}}{\vdash y := \mathcal{E}(0, k_e) : \ell_y} \text{(Coerce-encrypt)}$$

Rule Coerce-equality permits the typing of comparisons of encryptions that the adversary cannot modify (with high integrity). This kind of comparisons are used later in the proof and the new rule deems them typable. Rule Coerce-encrypt permits the typing of encryptions of default values (e.g., a bitstring of 0 of the correct size). This kind of encryption are used latter in the proof in indistinguishability games.

Any typing judgment is preserved by this extension, hence, for the rest of the proof we use this extended type system. Moreover, any low-confidentiality expression is now typable at level LL with any datatype:

Lemma 36 *Let Γ be a policy, e an expression such that $C(x) = L$ for every $x \in rv(e)$, and τ a security type such that $\tau \leq_I L$. Then $\vdash e : \tau$.*

Proof: The proof is by structural induction on e:

- e is x. By hypothesis, $C(x) = L$. We conclude using Coerce-LL to replace $\Gamma(x)$ by τ.
- e is $op(e_1, \ldots, e_n)$. We use inductive hypotheses on $e_1 \ldots e_n$ and conclude with Rule TOp.

□

In the extended types system for the reduced lattice, any adversary command is now typable:

Lemma 37 *Let Γ be a policy, and A an adversary command. We have $\mathcal{H}(\Gamma) \vdash A : LL$.*

Proof: The proof is by structural induction on A:

Case A is $x := e$. e contains only low confidentiality variables; also, x has low integrity, hence, by Lemma 36 and Rule TAssign, we have $\vdash x := e : LL$.

Case A is $\widetilde{x} := f(\widetilde{e})$. Similarly any of the $e \in \widetilde{e}$ contains only low confidentiality variables, hence $\vdash e : \mathsf{Data}\,(LL)$ by Lemma 36. Also, any of the $x \in \widetilde{x}$ has low integrity. Hence, by Rule Coerce-ProbFun-LL, we have $\vdash \widetilde{x} := f(\widetilde{y}) : LL$.

Case A is if e then P_1 else P_2. e contains only low confidentiality variables, hence $\vdash e : LL$. Also, by induction, $\vdash P_1 : LL$ and $\vdash P_2 : LL$. We conclude with Rule TCond

Case A **is** **while** e **do** P**.** Analog to previous case using Rule TWhile.

Case A **is** $P; P'$**.** By Rule TSeq and inductive hypothesis.

Case A **is** ***skip*****.** $\vdash skip : LL$.

□

Thus, we can also type the command obtained by applying a well-typed command context to any adversary commands.

Lemma 38 *Let Γ be a policy, let $\ell \in \mathcal{L}$ be a security label, let P be a command context such that $\mathcal{H}(\Gamma) \vdash P : \ell$, and let $\widetilde{A}$ be adversary commands. Then $\mathcal{H}(\Gamma) \vdash P[\widetilde{A}] : \ell$.*

Proof: By structural induction on the typing derivation of P. For the base case P contains no $\_$, hence $P = P[\widetilde{A}]$ and $\mathcal{H}(\Gamma) \vdash P[\widetilde{A}]$, otherwise:

Case **if** e **then** P_1 **else** P_2**.** By hypothesis, $\vdash$ if e then P_1 else $P_2 : \ell$. By Rule TCond, we have $\mathcal{H}(\Gamma) \vdash P_1 : \ell$ and $\mathcal{H}(\Gamma) \vdash P_2 : \ell$. We apply the induction hypothesis on P_1 and P_2, and we get $\mathcal{H}(\Gamma) \vdash P_1[\widetilde{A}] : \ell$ and $\mathcal{H}(\Gamma) \vdash P_2[\widetilde{A}] : \ell$. We conclude with Rule Cond.

Case **while** e **do** P'**.** Analog to previous case using Rule TWhile.

Case $P_1; P_2$**.** By Rule TSeq and inductive hypothesis.

Case $\_[\widetilde{A}]$**.** P is typed at level LL by Rule TVar. Hence, $\ell = LL$, and we conclude with Lemma 37.

□

We introduce a property of safeness for such programs $P[\widetilde{A}]$ typed with the extended typing rules. This property derives from the safeness property of P. Additionnaly, it is used as an invariant in the following states of the proof, and it is parametrised by a set of keys $\mathcal{K}$ that that contains the key labels allready treated (initially $\mathcal{K}$ is empty). At the end of the proof, when $\mathcal{K}$ contains all keys labels, Item 7 ensures that our programs contains no declassifications.

Definition 31 (Safe for $\mathcal{K}$) *Let Γ be a policy. Let $\mathcal{K}$ be a set of key labels occuring in Γ. Let α be the LH security label. A command P is safe for $\mathcal{K}$ against α-adversaries when $\Gamma \vdash P$ and*

1. *Each key label is used in at most one (dynamic) key generation.*
2. *Each cryptographic variable, with high integrity, read in P is first initialized by P.*
3. *There is no dependency cycle between key labels.*
4. *Every encryption with a high integrity result and a label in $\mathcal{K}$ is an encryption of 0.*
5. *Rule Coerce-encrypt is only used with labels $K \in \mathcal{K}$ in the typing derivation of P;*
6. *Every equality typed by Coerce-equality is such that $K \in \mathcal{K}$;*
7. *For all confidentiality level c, there is no K, with $K \in \mathcal{K}$ such that K declassifies c;*
8. *No decryption with a high integrity result is typed with $K \in \mathcal{K}$.*

5.9.3 Well-formed memory

We define a property of *well-formedness* on values, memories, and distributions used as an invariant in the execution of a program for proving non-interference. Intuitively, in a well-formed memory, high integrity values of a key type with label K are correctly generated keys consistent with the other keys of label K. Similarly, high integrity values with an encrypted type of label K are encryptions of well-formed values with a key of label K.

Definition 32 (Semantic Typing) *Let Γ be a policy, $\hat{\mu}$ a function from key labels to pair of keys values, $\mathcal{K}$ a set of encryption labels, v a value , τ a type, and ℓ a level, we define semantic typing $\models v : \tau$ as follows:*

$$\models \ \bot : \tau \qquad \models v : \mathsf{Data}\,(\ell) \qquad \frac{L \leq_I \tau}{\models v : \tau} \qquad \frac{\models v_0 : \tau_0 \qquad \models v_1 : \tau_0}{\models (v_0, v_1) : (\tau_0, \tau_1)}$$

$$\frac{t(\tau) = \mathsf{Enc}\,\tau'\,Kq \qquad v \ encrypts_{(\hat{\mu}(K))_1}\ v' \qquad \models v' : \tau'}{\models v : \tau} \qquad \frac{t(\tau) = \mathsf{Enc}\,\tau'\,Kq \qquad K \in \mathcal{K}}{\models v : \tau}$$

$$\models (\hat{\mu}(K))_1 : \mathsf{Ke}\,\mathsf{E}\,K(\ell) \qquad \models (\hat{\mu}(K))_2 : \mathsf{Kd}\,\mathsf{E}\,K(\ell)$$

A memory μ is well-formed when, there exists μ such that, for every K, $\hat{\mu}(K)$ is in the range of $\mathcal{G}_e$, and for every $x \in \Gamma$, $\models \mu(x) : \Gamma(x)$.

Similarly, a distribution of memory ρ is well-formed when, for every memory μ such that $\rho(\mu) \neq 0$, μ is well-formed.

In this lemma, we prove that the evaluation of a typed expression on values of a well-formed memory result in a well-formed value.

Lemma 39 *Let e be an expression and τ a security type such that $\vdash e : \tau$. If μ is a well-formed memory then $\models [\![e]\!](\mu) : \tau$.*

Proof: The proof is by induction on the typing derivation:

Case e is x, typed by Var**.** By typing hypothesis, $\Gamma(x) \leq \tau$. Also, $\models \mu(x) : \Gamma(x)$ since μ is well-formed. Hence $\models \mu(x) : \tau$ by Rule SubE.

Case e is $\langle e_0, e_1 \rangle$, typed by TOp**.** We conclude by induction on $\vdash e_0 : \tau_0$, $\vdash e_1 : \tau_1$ and by definition 32.

Case e is $(e')_i$, typed by TOp**.** We conclude by induction on $\vdash e' : \tau'$ and by definition 32.

Case e is $e_1 = e_2$, typed by Rule Coerce-equality**.** Hence, $\tau = \mathsf{Data}\,(\ell)$ for some ℓ, and we conclude by Definition 32.

Case e is typed by Rule Coerce-LL**.** Hence, $L \leq_I \tau$, and we conclude by Definition 32.

Case e is $op(e_1, \ldots, e_n)$, typed by TOp**.** By typing, $\tau = \mathsf{Data}\,(\ell)$ for some ℓ, and we conclude by definition 32.

□

In the next lemma, we show that a safe command running on a well-formed memory yields a well-formed memory.

Lemma 40 *Let $\mathcal{K}$ be a set of encryption labels. Let P be a polynomial-time command safe for $\mathcal{K}$ such that If the cryptographic variables of μ are un-initialized then $\rho_\infty(\langle P, \mu \rangle)$ is well-formed.*

PROOF: The proof is by induction on the maximum number of reduction steps for P to terminate. Our induction hypothesis is:

- Let P' be a polynomial-time command and μ' a memory such that $\langle P, \mu\rangle$ reduces to $\langle P', \mu'\rangle$ with an non-null probability. If μ' is well-formed for $\hat{\mu}$ were the domain of $\hat{\mu}$ corresponds to the keys that have been initialized, then $\rho_\infty(\langle P, \mu'\rangle)$ is well-formed for some $\hat{\mu}$ with a domain corresponds to the keys that have been initialized in P.

Case P' is $y := \mathcal{E}(e, k_e)$. If P' is typed with Rule COERCE-ENCRYPT with a label K, then by hypothesis, $K \in \mathcal{K}$ and we conclude by definition 32. If $I(k_e) \neq H$ then $L \leq_I y$ by Rule ENCRYPT and we conclude by definition 32. Otherwise, $\mu'(k_e) = (\hat{\mu}(K))_1$ since μ' is well-formed. Hence it is a correct encryption with key $(\hat{\mu}(K))_1$. For every μ'' such that $\rho_\infty(\langle P, \mu'\rangle)(\mu'') \neq 0$, we have:

$$\mu''(z) = \mu'(z) \text{ if} z \neq y$$
$$\mu''(y)\ encrypts_{(\hat{\mu}(K))_1}\ \mu'(x)$$

and we conclude by definition 32.

Case P' is $y := f_K(\widetilde{z}, k_e)$. and must be typed using Rule HOM-FUN at some level ℓ_P, for some security types $\widetilde{\tau}, \tau \in \mathsf{E}$ such that $\ell_d \leq L(\tau)$ and

$$\Gamma(y) = \mathsf{Enc}\,(\tau)\,K\,(\ell_P)$$
$$\vdash z_i : \mathsf{Enc}\,(\tau_i)\,K\,(\ell_P)$$
$$\vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_P)$$
$$\Gamma(x \mapsto (\tau \sqcup \ell_P), \widetilde{x} \mapsto (\widetilde{\tau} \sqcup \ell_P)) \vdash x := f(\widetilde{x}) : \tau \sqcup \ell_P$$

If $I(k_e) \neq H$ or $I(z_i) \neq H$ for some i, then$L \leq_I y$, and we conclude by definition 32. Otherwise, $\mu'(k_e) = (\hat{\mu}(K))_1$ and for each i there exists v_i such that $\mu'(z_i)\ encrypts_{(\hat{\mu}(K))_1}\ v_i$ and $\models v_i : \tau_i$ since μ' is well-formed. Hence it is a correct homomorphic operation with key $(\hat{\mu}(K))_1$. For every μ'' such that $\rho_\infty(\langle P, \mu'\rangle)(\mu'') \neq 0$, we have:

$$\mu''(z) = \mu'(z) \text{ if} z \neq y$$
$$\mu''(y)\ encrypts_{(\hat{\mu}(K))_1}\ \mu'(x)$$

and we conclude by definition 32.

Case P' is $x := \mathcal{D}(y, k_d)$. If $I(k_d) \neq H$ or $I(y) \neq H$, then $L \leq_I x$ by Rule DECRYPT and we conclude by definition 32. Otherwise, $K \notin \mathcal{K}$ by hypothesis. Hence $\mu'(k_d) = (\hat{\mu}(K))_2$ and there exists v such that $\mu'(y)\ encrypts_{(\hat{\mu}(K))_1}\ v$ and $\models v : \tau$ since μ' is well-formed. Hence it is a correct decryption operation with key $(\hat{\mu}(K))_2$. $\rho_\infty(\langle P, \mu'\rangle)(\mu'\{x \mapsto v\}) = 1$ and we conclude by definition 32.

Case P' is $k_e, k_d := \mathcal{G}_e()$. P is safe, so there is only one key initialization for each key label. Hence K is not in the domain of $\hat{\mu}$. For each memory μ'' such that $\rho_\infty(\langle P, \mu'\rangle)(\mu'') \neq 0$, μ'' is well-formed with $\hat{\mu}' = \hat{\mu}\{K \mapsto (\mu''(k_e), \mu''(k_d))\}$.

Case P' is $x := e$. We conclude by Lemma 39, Rule TASSIGN, and Definition 32.

Case P' is if e then P_1' else P_2'. By semantic, either $\langle P', \mu'\rangle \rightsquigarrow_1 \langle P_1', \mu'\rangle$ and P_1' or $\langle P', \mu'\rangle \rightsquigarrow_1 \langle P_2', \mu'\rangle$ and P_2'. We conclude by induction on P_1' or P_2'.

Case P' is while e do P''. By semantic, either $\langle P', \mu'\rangle \rightsquigarrow_1 \langle \surd, \mu'\rangle$ and we conclude, or $\langle P', \mu'\rangle \rightsquigarrow_1 \langle P''; P', \mu'\rangle$ and $P''; P'$ and we conclude by induction on $P''; P'$.

Case P' is $P_0'; P_1'$. By lemma 27, and by induction on $\langle P_0', \mu'\rangle$ and $\langle P_1', \mu''\rangle$ for each of the μ'' such that $\langle P_0', \mu'\rangle(\mu'') \neq 0$.

□

5.9.4 Non-interference for programs without declassifications

We define a projection on values, memories, and distributions that erases high-level information; our definition is parametrized by a set of encryption labels, $\mathcal{K}$, used later to keep track of keys that do not depend (anymore) on cryptogramhic assumptions; it has a special case for high-integrity encryptions , which are flattened then projected.

Definition 33 *For a given Γ, a function $\hat{\mu}$ from key labels to pair of keys values, a set of encryption labels $\mathcal{K}$, a security label ℓ, and a type τ, we define the projection of value v as follows:*

$$v_{|\ell,\tau} \doteq \begin{cases} \bot & \text{if} \quad \ell \leq \tau \\ v'_{|\ell,\tau'} & \text{if} \begin{pmatrix} \ell \not\leq \tau \quad I(\tau) = H \quad I(\ell) = L \\ t(\tau) = \mathsf{Enc}\,\tau'\,Kq \quad K \notin \mathcal{K} \\ v \textit{ encrypts}_{(\hat{\mu}(K))_1} v' \end{pmatrix} \\ (v_{0|\ell,\tau_0}, v_{1|\ell,\tau_1}) & \text{if} \quad \ell \not\leq \tau \quad v = (v_0, v_1) \quad \tau = (\tau_0, \tau_1) \\ v & \text{otherwise} \end{cases}$$

Let $\mu_{|\ell}$ be the projection of memory μ defined as:

$$\mu_{|\ell}(x) \doteq (\mu(x))_{|\ell,\Gamma(x)}$$

Let $\rho_{|\ell}$ be the projection of a distribution of memories ρ defined as:

$$\rho_{|\ell}(\mu) \doteq \sum_{\mu'|\mu'_{|\ell}=\mu} \rho(\mu')$$

We write $\mu_0 \equiv_\ell \mu_1$ for two memories μ_0 and μ_1 when $\mu_{0|\ell} = \mu_{1|\ell}$. We write $\rho_0 \equiv_\ell \rho_1$ for two distributions ρ_0, ρ_1 when $\rho_{0|\ell} = \rho_{1|\ell}$.

In this lemma, we show that $\equiv_\ell$ is preserved when ℓ decreases.

Lemma 41 *Let τ and τ' be two security types such that $\tau \leq \tau'$, let v_1 and v_2 be two values, and let ℓ be a label. If $v_{1|\ell,\tau} = v_{2|\ell,\tau}$ then $v_{1|\ell,\tau'} = v_{2|\ell,\tau'}$*

Proof: By Definition 33. □

In this lemma, we show that the projection on $(LL, \tau \sqcup \ell)$ is equivalent to the projection on LL, τ if $I(\ell) = H$.

Lemma 42 *Let τ be a type, let v be a value, and let ℓ be a label such that $I(\ell) = H$. We have $v_{|\ell,\tau\sqcup\ell} = v_{|\ell,\tau}$.*

Proof: By Definition 33. □

In this lemma, we show that two values equivalent with a low datatype must be equal if the *encrypts* rule is not available.

Lemma 43 *Let $\mathcal{K}$ be a set of encryption labels, let ℓ and ℓ' be two security label, let τ be a security type such that $\ell \not\leq \ell'$ and $\tau \leq \ell'$, let v_0 and v_1 be two values such that $v_{0|\ell,\tau} = v_{1|\ell,\tau}$. Then, if either $I(\ell) = H$ or τ contains only encryption on labels in $\mathcal{K}$ then $v_0 = v_1$.*

PROOF: By Definition 33. □

In this lemma, we show that we can lift relation on memories to distributions.

Lemma 44 *Let ρ_0 and ρ_1 be two distributions and ℓ a label such that $\rho_0 \equiv_\ell \rho_1$. Let φ be a predicate on memories such that for all memories μ such that $\rho_0(\mu) \neq 0$ or $\rho_1(\mu) \neq 0$, $\varphi(\mu)$. Let P_0 and P_1 be two commands and $\mathcal{R}$ a relation between distributions preserved by weighted sum, such that for every memory μ_0 and μ_1 such that $\varphi(\mu_0)$, $\varphi(\mu_1)$ and $\mu_0 \equiv_\ell \mu_1$, we have $\rho_\infty(P_0, \mu_0)\ \mathcal{R}\ \rho_\infty(P_1, \mu_1)$. Then $(\sum_\mu \rho_0(\mu) * \rho_\infty(\langle P_0, \mu\rangle))\ \mathcal{R}\ (\sum_\mu \rho_1(\mu) * \rho_\infty(\langle P_1, \mu\rangle))$.*

PROOF: Let $(\mu_i)_{i\leq n}$ be the set of memories such that $\rho_{0|\ell}(\mu_i) \neq 0$ (and $\rho_{1|\ell}(\mu_i) \neq 0$ since $\rho_0 \equiv_\ell \rho_1$) , we let p_i be such that $\rho_{0|\ell}(\mu_i) = p_i$. By definition of the projection on distributions, for every μ_i, p_i such that $\rho_{0|\ell}(\mu_i) = p_i$, there exists $(\mu^0_{i,j})_{j\leq n^0_i}$ and $(p^0_{i,j})_{j\leq n^0_i}$ such that

- $\mu^0_{i,j|\ell} = \mu_{i|\ell}$,
- $\rho_0(\mu^0_{i,j}) = p^0_{i,j}$,
- $\sum_j p^0_{i,j} = p_i$

Similarly, there exists $(\mu^1_{i,j})_{j\leq n^1_i}$ and $(p^1_{i,j})_{j\leq n^1_i}$ such that

- $\mu^0_{i,j|\ell} = \mu^1_{i,j|\ell} = \mu_{i|\ell}$,
- $\rho_0(\mu^0_{i,j}) = p^0_{i,j}$,
- $\rho_1(\mu^1_{i,j}) = p^1_{i,j}$,
- $\sum_j p^0_{i,j} = \sum_j p^1_{i,j} = p_i$

We can split the initial sum for these $\mu_{i,j}$, and get:

$$\begin{array}{ll} \sum_\mu \rho_0(\mu) * \rho_\infty(\langle P_0, \mu\rangle) = & \sum_i \sum_j \rho_\infty(\langle P_0, \mu_{i,j}\rangle) * p^0_{i,j} \\ \sum_\mu \rho_1(\mu) * \rho_\infty(\langle P_1, \mu\rangle) = & \sum_i \sum_j \rho_\infty(\langle P_1, \mu_{i,j}\rangle) * p^1_{i,j} \end{array}$$

By hypothesis, for every couple (i, j), we have $\rho_\infty(P_0, \mu^0_{i,j})\mathcal{R}\rho_\infty(P_1, \mu^1_{i,j})$ and we conclude. □

In this lemma, we show that a command typable at level ℓ does not influence memory at a level below ℓ.

Lemma 45 (Confinement) *Let Γ be a policy, $\ell \in \mathcal{L}$ a security label, P a command, and μ a memory.*

If $\vdash P : \ell$, then $\rho_\infty(\langle P, \mu\rangle) \equiv_\ell \mu$.

PROOF: The proof is by induction on the maximum number of reduction steps for $\langle P, \mu\rangle$ to terminate. The base case $P = \surd$ is trivial. The inductive case is by analysis on P:

Case $x := e$**.** By Rules ASSIGN and TSUBC, we have $\ell \leq L(x)$. Hence, by definition 33, for every μ' such that $\rho_\infty(\langle P, \mu\rangle)(\mu') \neq 0$, we have:

$$\begin{array}{l} \mu'(z) = \mu(z) \text{ if} z \neq x \\ \mu'(x)_{|\ell,\Gamma(x)} = \bot = \mu(x)_{|\ell,\Gamma(x)} \end{array}$$

and we conclude.

Case $x := \mathcal{E}(e, k_e)$**.** By Rule TSubC and Rule Encrypt (or Coerce-encrypt), we have $\ell \leq L(x)$. Hence, by definition 33, for every μ' such that $\rho_\infty(\langle P, \mu\rangle)(\mu') \neq 0$, we have:

$$\mu'(z) = \mu(z) \text{ if} z \neq x$$
$$\mu'(x)_{|\ell,\Gamma(x)} = \bot = \mu(x)_{|\ell,\Gamma(x)}$$

and we conclude.

Case $\widetilde{x} := f(\widetilde{e})$**.** By Rule TSubC and one of the Rules TFun, GenE, Hom-Fun, Decrypt, Blind and Pre-Encrypt, we have (for all x in $\widetilde{x}$) $\ell \leq L(x)$. The conclusion follows as the two cases above.

Case if e **then** P_1 **else** P_2**.** By Rules TCond and TSubC, if $\vdash P : \ell$ then $\vdash P_1 : \ell$ and $\vdash P_2 : \ell$. Also, either $\langle P, \mu\rangle \rightsquigarrow_1 \langle P_1, \mu\rangle$ or $\langle P, \mu\rangle \rightsquigarrow_1 \langle P_2, \mu\rangle$. We conclude by induction on $\langle P_1, \mu\rangle$ and $\langle P_2, \mu\rangle$.

Case while e **do** P'**.** By Rules TWhile, TSubC, and TSeq, if $\vdash P : \ell$ then $\vdash P'$; while e do $P' : \ell$. Also, either $\langle P, \mu\rangle \rightsquigarrow_1 \langle P'$; while e do $P', \mu\rangle$ or $\langle P, \mu\rangle \rightsquigarrow_1 \langle\surd, \mu\rangle$. We conclude by induction on $\langle P'$; while e do $P', \mu\rangle$ and $\langle\surd, \mu\rangle$.

Case $P; P'$**.** By Lemma 27, we have $\rho_\infty(\langle P; P', \mu\rangle) = \sum_{\mu'} \rho_\infty(\langle P', \mu'\rangle) * \rho_\infty(\langle P, \mu\rangle)(\mu')$. By inductive hypothesis on $\langle P, \mu\rangle$, $\rho_\infty(\langle P, \mu\rangle)_{|\ell} = \mu_{|\ell}$. Hence $\rho_\infty(\langle P, \mu\rangle)(\mu') \neq 0$ if and only if $\mu'_{|\ell} = \mu_{|\ell}$. By inductive hypothesis on each of the $\langle P', \mu'\rangle$, $\rho_\infty(\langle P', \mu'\rangle)_{|\ell} = \mu'_{|\ell} = \mu_{|\ell}$. We conclude that $\rho_\infty(\langle P; P', \mu\rangle)_{|\ell} = \mu_{|\ell}$.

Case ***skip*****.** $\langle P, \mu\rangle \rightsquigarrow_1 \langle\surd, \mu\rangle$. We conclude.

□

In the next lemma, we show that the evaluation of a typed expression on two equivalent memories yield equivalent values at least for the equivalences that are relevant to the proof of CNI for integrity and confidentiality (equivalences for high integrity values, and equivalences for public values.

Lemma 46 *Let ℓ be LL or cH for some confidentiality level $c \neq L$. Let τ be a security type. Let Γ be a policy. Let $\mathcal{K}$ be a set of encryption labels. Let μ_0 and μ_1 be memories and e be an expression such that $\vdash e : \tau$ and every instance of Rule Coerce-equality with a premise $\mathsf{Enc}\,\tau K(\ell)$ is such that $K \in \mathcal{K}$.*

If $\mu_0 \equiv_\ell \mu_1$, then $[\![e]\!](\mu_0) \equiv_{|\ell,\tau} [\![e]\!](mu_1)$.

Proof: If $\ell \leq \tau$ we conclude by Definition 33. Otherwise, the proof is by induction on the typing derivation:

Case e **is** $e_0 = e_1$**, typed by Rule Coerce-equality.** We have $e_0 : \mathsf{Enc}\,\tau' K(c'H)$ $(= \tau_0)$ and $\vdash e_1 : \mathsf{Enc}\,\tau' K(c'H)$ $(= \tau_1)$. By induction, $[\![e_0]\!](\mu_0)_{|\ell,\tau_0} = [\![e_0]\!](\mu_1)_{|\ell,\tau_0}$ and $[\![e_1]\!](\mu_0)_{|\ell,\tau_1} = [\![e_1]\!](\mu_1)_{|\ell,\tau_1}$. By hypothesis $K \in \mathcal{K}$. Hence, by Lemma 43, and since $\ell \not\leq \tau$, $[\![e_0]\!](\mu_0) = [\![e_0]\!](\mu_1)$ and $[\![e_1]\!](\mu_0) = [\![e_1]\!](\mu_1)$. We conclude.

Case e **is typed at level** τ **(with** $I(\tau) = L$**) by Rule Coerce-LL.** e must be typed at level τ' with $C(\tau') = L$ before the Rule Coerce-LL. We have $\ell \leq \tau$, hence $\ell \neq LL$. Also, $\tau' \leq LH$, and $\ell \not\leq LH$. We conclude by induction and with Lemma 43.

Case e **is** x**.** By typing hypothesis, $\Gamma(x) \leq \tau$. We conclude with Lemma 41.

Case e **is** $\langle e_0, e_1\rangle$**.** We conclude by induction on $\vdash e_0 : \tau_0$, $\vdash e_1 : \tau_1$.

Case e **is** $(e')_i$**.** We conclude by induction on $\vdash e' : \tau'$.

Case e **is** $op(e_1, \ldots, e_n)$**.** We use inductive hypotheses on $\vdash e_1 : \tau, \ldots, \vdash e_n : \tau$.

□

In the next lemma, we show that a safe command running on well-formed equivalent memories yield well-formed equivalent memories.

Lemma 47 *Let ℓ be a security label, either LL, or cH for some confidentiality level $c \neq L$. Let Γ be a policy. Let $\mathcal{K}$ be a set of encryption labels. Let P be a polynomial-time command safe for $\mathcal{K}$ and μ_0 and μ_1 be two well-formed memories such that for every key label K and confidentiality level c' such that K* declassifies *c', $c \not\leq c'$*

If $\mu_0 \equiv_\ell \mu_1$ then $\rho_\infty(\langle P, \mu_0\rangle) \equiv_{|\ell} \rho_\infty(\langle P, \mu_1\rangle)$.

Proof: If $\vdash P : \ell$, we conclude with Lemma 45. Otherwise, $\vdash P : \ell_P$ with $\ell \not\leq \ell_P$ and without using Rule **??** and the proof is by induction on the number of reduction steps for P to terminate.

Case P **is** $y := \mathcal{E}(0, k_e)$**.** By typing Rule Coerce-encrypt (or Encrypt) $\vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(L(y))$. By hypothesis and Lemma 41, $\mu_0(k_e)_{|\ell,\Gamma(y)} = \mu_1(k_e)_{|\ell,\Gamma(y)}$. Since $\vdash P : \ell$, and by definition 33, $\mu_0(k_e) = \mu_1(k_e)$, and we conclude by semantic definition Fun.

Case P **is** $y := \mathcal{E}(e, k_e)$ **(with** $e \neq 0$**)** The command is typed by Rule Encrypt with some security type $\tau \in \mathsf{E}$ such that

$$\begin{aligned} \Gamma(y) &= \mathsf{Enc}\,\tau\,K(\ell_P) \\ &\vdash e : \tau \\ &\vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_P) \\ \alpha \not\leq_I k_e \ \textbf{or}\ \ \tau &\leq_C \alpha \end{aligned}$$

We distinguish between two possible cases related to the equivalence definition. In all the cases, $\Gamma(k_e) \leq_C \ell_P$ $\ell \not\leq \ell_P$, and k_e has a key type, hence $\mu_0(k_e) = \mu_1(k_e)$ by Lemma 43.

- If $I(\ell) = H$. By the non-declassification hypothesis, either there is a declassification and directly $\ell \not\leq \tau$ or $\tau \leq_C \ell_P$ which, together with $\ell \not\leq \ell_P$ implies that $\ell \not\leq \tau$. Hence, $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by Lemma 46 and 43. We conclude by the semantic definition Fun.
- Otherwise, $\ell = LL$, and $\ell \not\leq \ell_P$, hence $I(\ell_P) = H$ and $K \notin \mathcal{K}$, $I(k_e) = H$ and since μ_b is well-formed, for $b \in \{0; 1\}$, $\mu_b(k_e) = (\hat{\mu}(K))_1$, it is a correct encryption and for every μ'_0, μ'_1 such that $\rho_\infty(\langle P, \mu_b\rangle)(\mu'_b) \neq 0$ we have:

$$\begin{aligned} &\mu'_b(z) = \mu_0(z) \text{ if} z \neq y \\ &\mu'_b(y)\ \mathit{encrypts}_{K,b}\ [\![e]\!](\mu_b) \end{aligned}$$

Since $\mu_0 \equiv_\ell \mu_1$, we get

$$\begin{aligned} \mu'_{0|\ell}(z) &= \mu_{0|\ell}(z) \text{ if} z \neq y \\ &= \mu_{1|\ell}(z) \text{ if} z \neq y \\ &= \mu'_{1|\ell}(z) \text{ if} z \neq y \\ (\mu'_0(y))_{|\ell,\Gamma(y)} &= [\![e]\!](\mu_0)_{|\ell,\tau} \\ &= [\![e]\!](\mu_1)_{|\ell,\tau} \\ &= (\mu'_1(y))_{|\ell,\Gamma(y)} \end{aligned}$$

Case P **is** $y := f_K(\widetilde{z}, k_e)$**.** The command is typed by Rule Hom-Fun, with some security type $\widetilde{\tau}, \tau \in \mathsf{E}$ such that $\ell_d \leq L(\tau)$ and

$$\begin{aligned} \Gamma(y) &= \mathsf{Enc}\,(\tau)\,K\,(\ell_P) \\ \vdash z_i &: \mathsf{Enc}\,(\tau_i)\,K\,(\ell_P) \\ \vdash k_e &: \mathsf{Ke}\,\mathsf{E}\,K(\ell_P) \\ \Gamma(x \mapsto (\tau \sqcup \ell_P), \widetilde{x} \mapsto (\widetilde{\tau} \sqcup \ell_P)) &\vdash x := f(\widetilde{x}) : \tau \sqcup \ell_P \end{aligned}$$

We distinguish between two possible cases related to the equivalence definition.

- If $K \in \mathcal{K}$ or $I(\ell) = H$, $z_i \leq \ell_P$ and $\ell \not\leq \ell_P$, hence $\mu_0(z_i) = \mu_1(z_i)$ by Lemma 43. Similarly $\mu_0(k_e) = \mu_1(k_e)$. We conclude by the semantic definition Fun.
- Otherwise, $\ell = LL$ and $\ell \not\leq \ell_P$, hence $I(\ell_P) = H$, $I(z_i) = H$, and $I(k_e) = H$. Since μ_b is well-formed, for $b \in \{0;1\}$, $\mu_b(k_e) = (\hat{\mu}(K))_1$ and, for each i, there exists $w_{b,i}$ such that $\mu(z_{b,i})$ $encrypts_{(\hat{\mu}(K))_1}$ $w_{b,i}$, with $w_{0,i} \equiv_\ell^{\widetilde{\tau}} \widetilde{w}_{1,i}$. We set $\Gamma^\circ = \Gamma\{x \mapsto (\tau \sqcup \ell_P), \widetilde{x} \mapsto (\widetilde{\tau} \sqcup \ell_P)\}$, and $\mu_b^\circ = \mu_b\{\widetilde{x} \mapsto \widetilde{w}_b\}$. Using Lemma 41, we have $\mu_0^\circ \equiv_\ell \mu_1^\circ$ and by induction: $\rho_\infty(\langle x := f(\vec{x}), \mu_0^\circ\rangle)_{|\ell} = \rho_\infty(\langle x := f(\vec{x}), \mu_1^\circ\rangle)$. This implies that $(f(\widetilde{w}_0))_{|\ell,\tau\sqcup\ell_P} = (f(\vec{w}_1))_{|\ell,\tau\sqcup\ell_P}$. By Lemma 42, $(f(\widetilde{w}_0))_{|\ell,\tau} = (f(\vec{w}_1))_{|\ell,\tau}$. It is a correct homomorphic operation with key $(\hat{\mu}(K))_1$ hence, $f_K(\vec{\mu}_b(z))$ $encrypts_{(\hat{\mu}(K))_1}$ $f(\widetilde{w}_b)$.
  Since $\mu_0 \equiv_\ell \mu_1$, we get

$$\begin{aligned}
\mu'_{0|\ell}(z) &= \mu_{0|\ell}(z) \text{ if } z \neq y \\
&= \mu_{1|\ell}(z) \text{ if } z \neq y \\
&= \mu'_{1|\ell}(z) \text{ if } z \neq y \\
(\mu'_0(y))_{|\ell,\Gamma(y)} &= (f(\widetilde{w}_0))_{|\ell,\Gamma(\widetilde{\tau})} \\
&= (f(\widetilde{w}_1))_{|\ell,\Gamma(\widetilde{\tau})} \\
&= (\mu'_1(y))_{|\ell,\Gamma(y)}
\end{aligned}$$

Case P is $x := \mathcal{D}(y, k_d)$. and must be typed using Rule Decrypt for some security type $\tau \in \mathsf{E}$ such that

$$\begin{aligned}
&\tau \leq \Gamma(x) \\
&\quad \vdash y : \mathsf{Enc}\,\tau K \ell_P \\
&\quad \vdash k_d : \mathsf{Kd}\,\mathsf{E}\,K\ell_d \quad \ell_d \leq_I x \\
&\quad (\alpha \not\leq_I y \textbf{ and } \alpha \not\leq_I \ell_d) \textbf{ or } \ell_d \leq_C x \textbf{ or } \ell_d \leq_C \alpha
\end{aligned}$$

We distinguish between two possible cases related to the equivalence definition.

- If $K \in \mathcal{K}$ or $I(\ell) = H$, $y \leq \ell_P$ and $\ell \not\leq \ell_P$, hence $\mu_0(y) = \mu_1(y)$ by Lemma 43. For the key, either there is a declassification and directly $\ell \not\leq \Gamma(k_e)$ or $\Gamma(k_e) \leq_C \ell_P$ which, together with $\ell \not\leq \ell_P$ implies that $\ell \not\leq \Gamma(k_e)$. Hence, $\mu_0(k_e) = \mu_1(k_e)$ by Lemma 43. We conclude by the semantic definition Fun.
- Otherwise $K \notin \mathcal{K}$, $\ell = LL$, and $\ell \not\leq \ell_P$, hence $I(\ell_P) = H$, $I(y) = H$, and $I(k_d) = H$. Since μ_b is well-formed, for $b \in \{0;1\}$, $\mu_b(k_d) = (\hat{\mu}(K))_2$, and there exist v_b such that $\mu_b(y)$ $encrypts_{(\hat{\mu}(K))_1}$ $v_b$with $v_{0|\ell,\Gamma(x)} = v_{1|\ell,\Gamma(x)}$ (using Lemma 41 with hypothesis $(\tau \leq \Gamma(x))$. It is a correct decryption and for every μ'_0, μ'_1 such that $\rho_\infty(\langle P, \mu_b\rangle)(\mu'_b) \neq 0$ we have:

$$\begin{aligned}
\mu'_{0|\ell}(z) &= \mu_{0|\ell}(z) \text{ if } z \neq x \\
&= \mu_{1|\ell}(z) \text{ if } z \neq x \\
&= \mu'_{1|\ell}(z) \text{ if } z \neq x \\
(\mu'_0(x))_{|\ell,\Gamma(x)} &= (v_0)_{|\ell,\Gamma(x)} \\
&= (v_1)_{|\ell,\Gamma(x)} \\
&= (\mu'_1(y))_{|\ell,\Gamma(y)}
\end{aligned}$$

Case P is $x := e$. By typing Rule TAssign, $\vdash e : \Gamma(x)$, hence $[\![e]\!](\mu_0)_{|\ell,\Gamma(x)} = [\![e]\!](\mu_1)_{|\ell,\Gamma(x)}$ by Lemma 46. We conclude by the semantic definition Assign.

Case P is if e then P_1 else P_2. By Rule TCond, $t(e) = \mathsf{Data}$, also $e \leq \ell_P$, hence, by Lemma 46, $[\![e]\!](\mu_0) \equiv_\ell^{\Gamma(e)} [\![e]\!](\mu_1)$, in consequence $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) = b$ for some $b \in \{0, 1\}$, hence

$$\begin{aligned}
\rho_\infty(\langle \text{if } e \text{ then } P_1 \text{ else } P_2, \mu_0\rangle) &= \rho_\infty(\langle P_b, \mu_0\rangle) \\
\rho_\infty(\langle \text{if } e \text{ then } P_1 \text{ else } P_2, \mu_1\rangle) &= \rho_\infty(\langle P_b, \mu_1\rangle)
\end{aligned}$$

We conclude by induction on $\langle P_b, \mu_0\rangle$ and $\langle P_b, \mu_1\rangle$.

Case P is while e do P. By Rule TWhile, $t(e) = \mathsf{Data}$, also $e \leq \ell_P$ hence $\ell \not\leq e$ and, by Lemma 46, $[\![e]\!](\mu_0) \equiv_\ell^{\Gamma(e)} [\![e]\!](\mu_1)$, in consequence $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) = b$ for some b, hence either

- $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) = 0$ and

$$\rho_\infty(\langle \text{while } e \text{ do } P, \mu_0\rangle) = \mu_0$$
$$\rho_\infty(\langle \text{while } e \text{ do } P, \mu_1\rangle) = \mu_1$$

  and we conclude from hypothesis $\mu_0 \equiv_\ell \mu_1$;
- or $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) \neq 0$ and

$$\rho_\infty(\langle \text{while } e \text{ do } P, \mu_0\rangle) = \rho_\infty(\langle P; \text{while } e \text{ do } P, \mu_0\rangle)$$
$$\rho_\infty(\langle \text{while } e \text{ do } P, \mu_1\rangle) = \rho_\infty(\langle P; \text{while } e \text{ do } P, \mu_1\rangle)$$

  and we conclude by induction on $\langle P; \text{while } e \text{ do } P, \mu_0\rangle$ and $\langle P; \text{while } e \text{ do } P, \mu_1\rangle$.

Case P is $P_0; P_1$. By Lemma 27, 44, and 40, and by induction on $\langle P_0, \mu\rangle$ and $\langle P_1, \mu'\rangle$ for each of the μ' such that $\langle P_0, \mu\rangle(\mu') \neq 0$.

□

The following lemma states that, if no information is declassified from any level above c, then we can replace commands of level cL by *skip* without affecting results for memory below LL or MH.

Lemma 48 (Skipify) *Let Γ be a policy. Let $c \neq L$ be a confidentiality level. Let ℓ be LL or cH. Let μ be a well-formed memory, and $\mathcal{K}$ a set of encryption labels. Let P be a polynomial-time command safe for $\mathcal{K}$, and let $P\{skip/c\}$ be the command obtained from P by replacing every sub-command Q in P such that $\vdash Q : cL$ by skip.*

For every key label K and confidentiality level c' such that K declassifies *c', $c \not\leq c'$.*

We have

$$\rho_\infty(\langle P, \mu\rangle) \equiv_{|\ell} \rho_\infty(\langle P\{skip/c\}, \mu\rangle)$$

Proof: The proof is by induction on the number of reduction steps for $\langle P, \mu\rangle$ to terminate. If $\vdash P : cL$, $P\{skip/c\} = skip$, and by Lemma 45 $\rho_\infty(\langle P, \mu\rangle)_{|\ell} = \mu_{|\ell} = \rho_\infty(\langle P\{skip/c\}, \mu\rangle)_{|\ell}$. Otherwise

Case $x := e$. Case $\widetilde{x} := f(\widetilde{y})$. P has no strict sub-command and $\not\vdash P : cL$, hence $P = P\{skip/c\}$ and we conclude.

Case if e then P_1 else P_2. Since $\not\vdash P : cL$ by Rule TCond, we have

$$P\{skip/c\} = \text{if } e \text{ then } P_1\{skip/c\} \text{ else } P_2\{skip/c\}$$

For some $b = 0, 1$ depending on e and μ, we have both $\langle P, \mu\rangle \rightsquigarrow_1 \langle P_b, \mu\rangle$ and $\langle P\{skip/c\}, \mu\rangle \rightsquigarrow_1 \langle P_b\{skip/c\}, \mu\rangle$. We conclude by induction on $\langle P_b, \mu\rangle$.

Case while e do P. Analog to previous case using Rule TWhile.

Case $P; P'$. By Lemma 27, we have

$$\rho_\infty(\langle P; P', \mu\rangle) = \sum_{\mu'} \rho_\infty(\langle P', \mu'\rangle) * \rho_\infty(\langle P, \mu\rangle)(\mu')$$
$$\rho_\infty(\langle P; P'\{skip/c\}, \mu\rangle) = \sum_{\mu'} \rho_\infty(\langle P'\{skip/c\}, \mu'\rangle) * \rho_\infty(\langle P\{skip/c\}, \mu\rangle)(\mu')$$

By Lemma 40, $\rho_\infty(\langle P, \mu\rangle)$ and $\rho_\infty(\langle P\{skip/c\}, \mu\rangle)$ are well-formed. By inductive hypothesis on $\langle P, \mu\rangle$:

$$\rho_\infty(\langle P, \mu\rangle)_{|\ell} = \rho_\infty(\langle P\{skip/c\}, \mu\rangle)_{|\ell}$$

Then, for every well-formed memory μ', by inductive hypothesis on $\langle P', \mu'\rangle$:

$$\rho_\infty(\langle P', \mu'\rangle)_{|\ell} = \rho_\infty(\langle P'\{skip/c\}, \mu'\rangle)_{|\ell}$$

By Lemma 47, for every well-formed memory μ'', such that $\mu' \equiv_\ell \mu''$:

$$\rho_\infty(\langle P', \mu'\rangle)_{|\ell} = \rho_\infty(\langle P'\{skip/c\}, \mu''\rangle)_{|\ell}$$

Hence, by Lemma 44

$$\begin{array}{rcl} \rho_\infty(\langle P; P', \mu\rangle) = & \sum_{\mu'} \rho_\infty(\langle P', \mu'\rangle) * \rho_\infty(\langle P, \mu\rangle)(\mu') \\ = & \sum_{\mu'} \rho_\infty(\langle P'\{skip/c\}, \mu'\rangle) * \rho_\infty(\langle P\{skip/c\}, \mu\rangle)(\mu') \\ = & \rho_\infty(\langle P; P'\{skip/c\}, \mu\rangle) \end{array}$$

Case *skip*. $\vdash skip : cL$

□

The next series of lemmas deal with the commands of the game that defines computational non-interference in Definition 28. In the lemma below, we show that the two possible initialization of the CNI game yields equivalent memories.

Lemma 49 (Init) *Let Γ be a policy. Let ℓ be a security label, either LL, or with cH for some confidentiality level $c \neq L$. Let V be $\{x \mid \ell \not\leq x\}$. Let J, B_0, B_1 be polynomial commands containing no cryptographic variables, such that $wv(B_b) \cap V = \emptyset$. Let $\mu_\perp$ be an uninitialized memory. The memory distributions $\rho_\infty(\langle J; B_0, \mu_\perp\rangle)$ and $\rho_\infty(\langle J; B_1, \mu_\perp\rangle)$ are well formed and such that*

$$\rho_\infty(\langle J; B_0, \mu_\perp\rangle) \equiv_\ell \rho_\infty(\langle J; B_1, \mu_\perp\rangle)$$

Proof: By Lemma 27, we have

$$\begin{array}{l} \rho_\infty(\langle J; B_0, \mu_\perp\rangle) = \sum_\mu \rho_\infty(\langle J, \mu_\perp\rangle)(\mu) * \rho_\infty(\langle B_0, \mu\rangle) \\ \rho_\infty(\langle J; B_1, \mu_\perp\rangle) = \sum_\mu \rho_\infty(\langle J, \mu_\perp\rangle)(\mu) * \rho_\infty(\langle B_1, \mu\rangle) \end{array}$$

B_0 and B_1 only write values x such that $\ell \leq x$. Hence, for every memory μ, $\rho_\infty(\langle B_0, \mu\rangle) \equiv_\ell \rho_\infty(\langle B_1, \mu\rangle)$. We conclude that

$$\rho_\infty(\langle J; B_0, \mu_\perp\rangle) \equiv_\ell \rho_\infty(\langle J; B_1, \mu_\perp\rangle)$$

Also, J, B_0, B_1 contains no cryptographic variables, hence for every non-Data variable x, and for every memory μ such that $\rho_\infty(\langle J; B_0, \mu_\perp\rangle)(\mu) \neq 0$, $\mu(x) = \perp$. Hence, $\rho_\infty(\langle J; B_0, \mu_\perp\rangle)$ and $\rho_\infty(\langle J; B_1, \mu_\perp\rangle)$ are well formed. □

Newt, lemma, we show that the witness of the CNI game cannot distinguish between equivalent memories.

Lemma 50 (Test) *Let Γ be a policy. Let ℓ be either LL or cH for some $c \neq L$. Let V be $\{x \mid \ell \not\leq x\}$. Let T be a polynomial command with no cryptographic variables such that $rv(T) \subseteq V$. Let μ_0 and μ_1 be two memory such that $\mu_0 \equiv_\ell \mu_1$. Let g be a variable uninitialized in both μ_0 and μ_1. We have*

$$\Pr[\langle T, \mu_0\rangle; g = 0] = \Pr[\langle T, \mu_1\rangle; g = 0]$$

PROOF: T reads only non-cryptographic variables x such that $\ell \not\leq \Gamma(x)$ and $\mu_0 \equiv_\ell \mu_1$, hence T only read variables which are equal on the two memories. g is written by T, hence equal for the two executions. □

In this lemma, we show that a fully reduced command (with no declassifications left, and with all labels in $\mathcal{K}$) is CNI.

Lemma 51 (*CNI*$\backslash K$) *Let Γ be a policy, $\alpha = LL$ an adversary level, and P polynomial-time command safe for the set of all key labels.*

P is CNI.

PROOF: Let ℓ be MH if $V = V_{\mathcal{A}}^C$ and LL if $V = V_{\mathcal{A}}^I$. We verify that $V = \{x \mid \ell \not\leq x\}$. By semantics (and repeated usage of Lemma 27), if $b = 0$, $\Pr[CNI; T; b =_0 g] = \Pr[J; B_0; P; T; 0 =_0 g]$ and if $b = 1$, $\Pr[CNI; T; b =_0 g] = \Pr[J; B_1; P; T; 1 =_0 g]$. Hence

$$\Pr[CNI; T; b =_0 g] = \frac{\Pr[J; B_0; P; T; 0 =_0 g] + \Pr[J; B_1; P; T; 1 =_0 g]}{2}$$

By Lemma 27

$$\rho_\infty(\langle J; B_0; P, \mu_\perp\rangle) = \sum_\mu \rho_\infty(\langle J; B_0, \mu_\perp\rangle)(\mu) * \rho_\infty(\langle P, \mu\rangle)$$
$$\rho_\infty(\langle J; B_1; P, \mu_\perp\rangle) = \sum_\mu \rho_\infty(\langle J; B_1, \mu_\perp\rangle)(\mu) * \rho_\infty(\langle P, \mu\rangle)$$

By Lemma 49, $\rho_\infty(\langle J; B_0, \mu_\perp\rangle)$ and $\rho_\infty(\langle J; B_1, \mu_\perp\rangle)$ are well formed and

$$\rho_\infty(\langle J; B_0, \mu_\perp\rangle) \equiv_\ell \rho_\infty(\langle J; B_1, \mu_\perp\rangle)$$

Then, by Lemma 47, for every well-formed memory μ_0 and μ_1 such that $\mu_0 \equiv_\ell \mu_1$, we have $\rho_\infty(\langle P, \mu_0\rangle) = \rho_\infty(\langle P, \mu_1\rangle)$. Hence, by Lemma 44, $\sum_\mu \rho_\infty(\langle J; B_0, \mu_\perp\rangle)(\mu) * \rho_\infty(\langle P, \mu\rangle) \equiv_\ell \sum_\mu \rho_\infty(\langle J; B_1, \mu_\perp\rangle)(\mu) * \rho_\infty(\langle P, \mu\rangle)$, which gives us

$$\rho_\infty(\langle J; B_0; P, \mu_\perp\rangle) \equiv_\ell \rho_\infty(\langle J; B_1; P, \mu_\perp\rangle)$$

By Lemma 27

$$\rho_\infty(\langle J; B_0; P; T, \mu_\perp\rangle) = \sum_\mu \rho_\infty(\langle J; B_0; P, \mu_\perp\rangle)(\mu) * \rho_\infty(\langle T, \mu\rangle)$$
$$\rho_\infty(\langle J; B_1; P; T, \mu_\perp\rangle) = \sum_\mu \rho_\infty(\langle J; B_1; P, \mu_\perp\rangle)(\mu) * \rho_\infty(\langle T, \mu\rangle)$$

By Lemma 50, for every μ_0 and μ_1 two memories such that $\mu_0 \equiv_\ell \mu_1$, we have $\Pr[\langle T, \mu_0\rangle; g = 0] = \Pr[\langle T, \mu_1\rangle; g = 0]$. We define a relation $\mathcal{R}$ between distributions by:$\rho_0 \mathcal{R} \rho_1 \Leftrightarrow \sum_{\mu/\mu(g)=0} \rho_0(\mu) = \sum_{\mu/\mu(g)=0} \rho_0(\mu)$. We apply Lemma 44 with the relation $\mathcal{R}$, and we get $\Pr[\langle J; B_0; P; T, \mu_\perp\rangle; g = 0] = \Pr[\langle J; B_1; P; T, \mu_\perp\rangle; g = 0]$. Finally, we conclude:

$$\begin{aligned}
\Pr[CNI; T; b =_0 g] &= \frac{\Pr[J;B_0;P;T;0=_0 g]+\Pr[J;B_1;P;T;1=_0 g]}{2} \\
&= \frac{\Pr[J;B_0;P;T;0=_0 g]+(1-\Pr[J;B_1;P;T;0=_0 g])}{2} \\
&= \frac{\Pr[J;B_0;P;T;0=_0 g]+(1-\Pr[J;B_0;P;T;0=_0 g])}{2} \\
&= \tfrac{1}{2}
\end{aligned}$$

□

Finally, we show that every command equivalent to a CNI command is CNI.

Lemma 52 *Let P and P' be two safe commands such that, for every well-formed μ with the uninitialized crypto variables, $\rho_\infty(\langle P, \mu\rangle) \equiv_{LL} \rho_\infty(\langle P', \mu\rangle)$ and $\rho_\infty(\langle P, \mu\rangle) \equiv_{MH} \rho_\infty(\langle P', \mu\rangle)$. If P is CNI then P' is CNI.*

PROOF: Let V be $V_{\mathcal{A}}^{C}$ or $V_{\mathcal{A}}^{I}$. Let J, B_0, B_1, T be polynomial commands containing no cryptographic variables, such that $wv(B_b) \cap V = \emptyset$ and $rv(T) \subseteq V$. Let g and b be such that $g \notin v(J, B_0, B_1, P)$, and $b \notin v(J, B_0, B_1, P, T)$. Let μ be such that $\rho_\infty(J; \text{if } b = 0 \text{ then } B_0 \text{ else } B_1)(\mu) \neq 0$, μ is well-formed by Lemma 40 (and all its cryptographic variables are uninitialized). Hence $\rho_\infty(\langle P, \mu\rangle) \equiv_{LL} \rho_\infty(\langle P', \mu\rangle)$ and $\rho_\infty(\langle P, \mu\rangle) \equiv_{MH} \rho_\infty(\langle P', \mu\rangle)$. Thus, $\rho_\infty(\langle CNI, \mu_\perp\rangle) \equiv_{LL} \rho_\infty(\langle CNI', \mu_\perp\rangle)$ and $\rho_\infty(\langle CNI, \mu_\perp\rangle) \equiv_{MH} \rho_\infty(\langle CNI', \mu_\perp\rangle)$.

If $V = V_{\mathcal{A}}^{C}$, for all x such that $x \in rv(T)$, $MH \not\leq \Gamma(x)$, also, x is non cryptographic. Hence, $\rho_\infty(\langle CNI, \mu_\perp\rangle) =_{|rv(T)} \rho_\infty(\langle CNI', \mu_\perp\rangle)$. Thus $\Pr[CNI; T; b =_0 g] = \Pr[CNI'; T; b =_0 g]$ and we conclude.

Similarly, if $V = V_{\mathcal{A}}^{I}$, for all x such that $x \in rv(T)$, $LL \not\leq \Gamma(x)$ and x is non cryptographic. Hence, $\rho_\infty(\langle CNI, \mu_\perp\rangle) =_{|rv(T)} \rho_\infty(\langle CNI', \mu_\perp\rangle)$. Thus $\Pr[CNI; T; b =_0 g] = \Pr[CNI'; T; b =_0 g]$ and we conclude. □

5.9.5 From encryptions to confidentiality

In this section, when we mention typability, we refer to typability with the extended type system on the reduced lattice, including the rules for coercion.

Assume P' is a command safe for $\mathcal{K}$ against α-adversaries, with $\alpha = LL$, and typable with a policy Γ.

We will transform P' in order to eliminate, using the cryptographic hypotheses, all encryptions with a given key label K.

The main proof proceeds as follows, until all labels K are in $\mathcal{K}$:

1. We say that a label $K \notin \mathcal{K}$ is free for $\mathcal{K}$ when he does not depend on any label outside of $\mathcal{K}$. We choose a free label $K \notin \mathcal{K}$ for $\mathcal{K}$ (since P' is safe, it ensures, for example, that there is no encryption of the decryption key).

2. We move the key generation command for label K in P', and then define P by transforming commands in P' to track the value of low integrity keys, and eliminating from P' all subcommands of confidentiality level c such that K depends on c.

3. We define two commands, namely $P^\star$ and $P\backslash K$, obtained by transformation of P. Command $P^\star$ is P where cryptographic commands are substituted by corresponding cryptographic oracles. Similarly, command $P\backslash K$ is P where typable cryptographic commands are substituted by cryptographic commands that do not involve confidential data. We perform a log lookup before decryption and encrypt 0 instead of confidential data.

4. We show that $P\backslash K$ is typable and safe for $\mathcal{K} \cup \{K\}$.

5. We show, by using $P^\star$ and cryptographic hypotheses, that if an adversary breaks P then he also breaks $P\backslash K$.

6. We iterate the process for $P\backslash K$ and $\mathcal{K} \cup \{K\}$ (until there are no more free labels).

Tranformation of P' into P

Let K be a free key label for $\mathcal{K}$ such that there exists at least a key generation command $k_e, k_d := \mathcal{G}_e()$ within P', typable with rule GenE using label K, and with $k_d \not\leq_C L$. By Safety Conditions 1 and 2, K is used to type at most one dynamic key generation command.

Log variables In the transformation, we use variables named $log_{\ell\tau K}$, one for each security level ℓ, data type τ, and key label K used in the typing of P'.

Log variables $log_{\ell\tau K}$ are used in the transformed commands to log a pair of a ciphertext of security level ℓ (encrypted with a key of label K) together with data of type τ. The ciphertext is an encryption of the associated data.

For convenience, we also treat $log_{\ell\tau K}$ as a function that maps ciphertexts to data of type τ.

We represent by $(log_{\ell\tau K})_{\ell\leq L(y)}$ a tuple of log variables: one log variable for each security level $\ell \leq L(y)$, if the log variable is defined for the program being transformed. When the vector is of length one, then we just write $log_{\ell\tau K}$ to simplify the notation.

Moving key generations in front of the command We obtain P'' from P' by making some replacements in commands containing key variables of label K.

We first transform P' such that the key generation command for K is the first command. We use k_e^0 and k_d^0, for some fresh variables k_e^0 and k_d^0, such that k_e^0 and k_d^0 are never overwritten. We replace P' with

$$k_e^0, k_d^0 := \mathcal{G}_e(); (log_{\ell\tau K} := \mathbf{0};)_{for\ each\ \tau\in\mathsf{E},\ell\in\mathcal{L}}\ P''$$

where P'' is P' with every generation command $k_e, k_d := \mathcal{G}_e()$ (where k_e and k_d have label K) is replaced with $k_e := k_e^0; k_d := k_d^0$. Relying on P's single-generation for K (safety condition 1) and read-after-assign for all key variables (safety condition 2), the transform above does not affect the semantics of our command.

Flags for low integrity decryption keys We add flags for low integrity decryption keys. For every low integrity decryption key of label K (k'_d) in P'', we use a fresh variable ${k'_d}^{init}$ to track if k'_d contains the original key or contains a key that has been modified by the adversary. We then replace every key assignment $k'_d := e$ in the above command by

- $k'_d := k''_d; {k'_d}^{init} := 1$ if k''_d is a high integrity key of label K;
- $k'_d := k''_d; {k'_d}^{init} := {k''_d}^{init}$ if k''_d is a low integrity key of label K;
- $k'_d := e; {k'_d}^{init} := 0$ otherwise.

Additionally we replace $\vec{x} := e$ such that $\vec{x}$ contains k'_d by $\vec{x}' := e; {k'_d}^{init} := 0$.

We call the new command P''' .

Elimination K dependencies We eliminate K dependencies. For each security level c such that K depends on c (that is, when there is a CPA decryption of a low integrity plaintext with a low integrity key of label K, see explanation of the decryption rule), we apply Lemma 48 to eliminate all subcommands in P''' typable as cL. After elimination, we obtain an equivalent program P with no low integrity CPA decryption.

Definition of $P^\star$ and $P\backslash K$

We define variants of the program P for applying cryptographic hypotheses (either CPA or CCA2) when the decryption key is confidential and for eliminating declassifications for a given key label, respectively.

Figure 5.4 shows the definition of $P^\star$ and $P\backslash K$. In the figure, we show suitable substitutions for encryption, decryption, pre-encryption, blinding, and fully homomorphic functions commands. In the figure, for each command, we assume that $\Gamma(y) = \mathsf{Enc}\,\tau\,K(\ell_y)$.

We use $x := assoc(z, (log_{\ell\tau K})_{\ell \leq L(y)})$ as syntactic sugar for

$$\text{if } z = (log_{\ell\tau K}[0])_0 \text{ then } x := (log_{\ell\tau K}[0])_1 \text{ else if } z = (log_{\ell\tau K}[1])_0 \text{ then } x := (log_{\ell\tau K}[1])_1 \text{ else } \ldots \tag{5.1}$$

It represents the association of ciphertext z_1 with its corresponding data in the first log variable of vector $(log_{\ell\tau K})_{\ell \leq L(y)})$. If no vector has z_1 in its domain $assoc()$ returns $\perp$.

Notice that the guard in command 5.1 can be typed by Coerce-equality.

We also use operations to search in the logs.

We use if $exists(y, (log_{\ell\tau K})_{\ell \leq L(y)})$ then $\ldots$ else $\ldots$ as syntactic sugar for

$$\text{if } z = (log_{\ell\tau K}[0])_0 \text{ then } \ldots \text{ else if } z = (log_{\ell\tau K}[1])_0 \text{ then } \ldots \text{ else } \ldots \tag{5.2}$$

That is, exists represents a boolean operation that determines if ciphertext in y appears in some log variable of vector $(log_{\ell\tau K})_{\ell \leq L(y)}$.

Notice that the guard in command 5.2 can be typed by Coerce-equality.

Explanations follow for each of the substitutions in the figure.

Encryptions If the adversary writes k_e ($I(k_e) = L$), we leave this command unchanged in $P^\star$ and $P\backslash K$. Otherwise:

- In $P^\star$, we replace this encryption by a command that uses E, the command that performs oracle encryptions. (We assume that the oracle E uses variables x_0 and x_1 and that these variables are not used in P otherwise.) We then log the encryption in the corresponding log variable, to be used later if there is a decryption in $P^\star$.
- In $P\backslash K$, we replace encryption of e by an encryption of 0 and we log the encryption for a future decryption in $P\backslash K$.

Homomorphic applications If the adversary writes y, we leave this command unchanged in both $P^\star$ and $P\backslash K$. Otherwise, in both $P^\star$ and $P\backslash K$, we replace this operation by a search in the log variables for each variable z_i. If all variables z_i are previously encrypted, they should appear in one of the logs. In this case, a pair is added to the log corresponding to the y variable to be used for future decryptions.

Decryptions We present two variants for decryptions. The first one applies to the case where decryption is typed using the CPA hypothesis, the second case applies to decryption typed using CCA.

There are four different cases:

- If $I(\ell_d) = L$ and the decryption command is typed with the CPA variant (5.11) we replace the decryption by a command in $P^\star$ (resp. $P\backslash K$) to check if the low integrity key has been modified by the adversary.

  Notice that y necessarily has high integrity, since decryption of low integrity ciphertexts with low integrity keys were eliminated from P. Hence the ciphertext in y exists in a log variable since it is encrypted before by safety condition 2. If the key is not modified by the adversary, we replace the original decryption by an assignment to x of the corresponding plaintext associated to y in the log. Otherwise, if the key has been modified, we keep the decryption as is.

- If $I(\ell_d) = H$ and the decryption command is typed with the CPA variant (5.11), there is no need to check for k_d^{init} since the key cannot be modified by the adversary. Notice that since the key is of high integrity, by the decryption typing rule with CPA, y is also of high integrity. Hence by safety condition 2, y has been written before and hence it has been encrypted and stored in a log variable before being read in the decryption command.
- If $I(\ell_d) = L$ and the decryption command is typed with the CCA2 variant (5.12), we replace the decryption by a command in $P^\star$ (resp. $P\backslash K$) to check if the low integrity key has been modified by the adversary. If the key is not modified by the adversary, and if the ciphertext in y exists in the corresponding log variable then x contains the corresponding value associated to y in the log. In $P^\star$ if ciphertext y is not in the log variable, we use the decryption oracle D from CCA2. We assume here that variable m and x are fresh in $P^\star$.
- If $I(\ell_d) = H$ and the decryption command is typed with the CCA2 variant (5.12), we apply a similar transformation as in the low integrity case, except that there is no need to check whether the adversary has modified the key.

Blindings If the encryption key is of low integrity, we do not replace the command. If the encryption key is of high integrity:

- In $P^\star$, we replace this encryption by a command that uses B, the command that performs oracle blinding. Here we assume that x_0 and x_1 are fresh variables. As with simple encryption commands, we need to log the new encryption in the corresponding log. The process is more complicated than for encryptions since we need to associate the blinding to the correct plaintext (already encrypted and stored in a log variable). If the value in z does not exist in any log, we associate y with z.
- In $P\backslash K$, we replace this encryption by a blinding of 0 and again we log the encryption with its corresponding plaintext as in $P^\star$.

Types for $P\backslash K$

We show that every transformation in $P\backslash K$ is typable with $\Gamma\backslash K$, given that the original command was typable with Γ. $\Gamma\backslash K$ is an extension of Γ, featuring fresh variables $k_d'^{init}$ for every low integrity key of label K (k_d') with type $\mathsf{Data}\,(L(k_d'))$ and log variables with type $\Gamma\backslash K(log_{\ell_\tau K}) = \mathsf{Array}\,(\mathsf{Enc}\,\tau\,K(\ell) * (\tau \sqcup \ell))$ for each $\tau \in E$ and $\ell \in \mathcal{L}$ such that $\mathsf{Kd}\;E\,K(\ell')$ is in Γ.

Lemma 53 (Encryption Elimination) *Let $P\backslash K$ be the command defined above from a command P' safe for $\mathcal{K}$. Then, $P\backslash K$ is safe for $\mathcal{K} \cup \{K\}$ using $\Gamma\backslash K$.*

This lemma implies that there is no encryption of confidential decryption key under the label K.

Proof. Most of the points of the safe invariant are direct by semantics of $P\backslash K$. We prove the typing. Notice that with the modified policy $\Gamma\backslash K$, the (unique) key-generation command for K in front of $P\backslash K$ can be typed by Rule GenE.

Since $P\backslash K$ is obtained from P' and P' is a safe command typable with policy Γ and without rules for coercion, we just show for the substitution commands that they are typable with $\Gamma\backslash K$.

Encryption If $I(k_e) = H$, we show that the replacement command in $P\backslash K$ can also be typed at level ℓ_y in $\Gamma\backslash K$ using rule Encrypt without declassification.

We use the typability hypothesis of the original encryption and typing Rule Coerce-encrypt to type the encryption of 0. Using the signature for $+$ with $\tau = \mathsf{Enc}\,\tau\,K(\ell_y) * (\tau \sqcup \ell_y)$ and

$P^\star$	$P \backslash K$
$y := \mathcal{E}(e, k_e)$ typed with Rule ENCRYPT	
if $I(k_e) = L$	
$y := \mathcal{E}(e, k_e)$	$y := \mathcal{E}(e, k_e)$
if $I(k_e) = H$	
$x_0 := 0; x_1 := e; E; y := m; log_{\ell_y \tau K} := log_{\ell_y \tau K} + (y, e)$	$y := \mathcal{E}(0, k_e); log_{\ell_y \tau K} := log_{\ell_y \tau K} + (y, e)$

$y := f_K(z_1, \ldots, z_n, k_e)$ typed with Rule HOM-FUN	
if $I(y) = L$	
$y := f_K(z_1, \ldots, z_n, k_e)$	$y := f_K(z_1, \ldots, z_n, k_e)$
if $I(y) = H$	
$y := f_K(z_1, \ldots, z_n, k_e);$ $\text{if} exists(z_1, (log_{\ell\tau_1 K})_{\ell \le \ell_y}) \& \ldots \& exists(z_n, (log_{\ell\tau_n K})_{\ell \le \ell_y})$ then $log_{\ell_y \tau K} := log_{\ell_y \tau K} + (y, f(assoc(z_1, (log_{\ell\tau_1 K})_{\ell \le \ell_y}),$ $\ldots, assoc(z_n, (log_{\ell\tau_n K})_{\ell \le \ell_y}))$	$y := f_K(z_1, \ldots, z_n, k_e);$ $\text{if} exists(z_1, (log_{\ell\tau_1 K})_{\ell \le \ell_y}) \& \ldots \& exists(z_n, (log_{\ell\tau_n K})_{\ell \le \ell_y})$ then $log_{\ell_y \tau K} := log_{\ell_y \tau K} + (y, f(assoc(z_1, (log_{\ell\tau_1 K})_{\ell \le \ell_y}),$ $\ldots, assoc(z_n, (log_{\ell\tau_n K})_{\ell \le \ell_y}))$

$x := \mathcal{D}(y, k_d)$ typed with Rule DECRYPT	
if $I(k_d) = L$	
if ${k_d}^{init}$ then $x := assoc(y, (log_{\ell\tau K})_{\ell \le \ell_y})$ else $x := \mathcal{D}(y, k_d);$	if ${k_d}^{init}$ then $x := assoc(y, (log_{\ell\tau K})_{\ell \le \ell_y})$ else $x := \mathcal{D}(y, k_d);$
if $I(k_d) = H$	
$x := assoc(y, (log_{\ell\tau K})_{\ell \le \ell_y})$	$x := assoc(y, (log_{\ell\tau K})_{\ell \le \ell_y})$

$x := \mathcal{D}(y, k_d)$ typed with Rule 5.3.4 (CCA2)	
if $I(k_d) = L$	

if ${k_d}^{init}$ then
if $exists(y, (log_{\ell\tau' K})_{\ell \le \ell_y, \tau' \le \tau})$ then
$x := assoc(y, (log_{\ell\tau' K})_{\ell \le \ell_y, \tau' \le \tau})$ else
$m := y; D; x := x_{CCA}$
else $x := \mathcal{D}(y, k_d);$
	if ${k_d}^{init}$ then
	if $exists(y, (log_{\ell\tau' K})_{\ell \le \ell_y, \tau' \le \tau})$ then
	$x := assoc(y, (log_{\ell\tau' K})_{\ell \le \ell_y, \tau' \le \tau})$ else $x := \mathcal{D}(y, k_d)$
	else $x := \mathcal{D}(y, k_d);$

if $I(k_d) = H$	

if $exists(y, (log_{\ell\tau' K})_{\ell \le \ell_y, \tau' \le \tau})$ then
$x := assoc(y, (log_{\ell\tau' K})_{\ell \le \ell_y, \tau' \le \tau})$
else
$m := y; D; x := x_{CCA}$
	if $exists(y, (log_{\ell\tau' K})_{\ell \le \ell_y, \tau' \le \tau})$ then
	$x := assoc(y, (log_{\ell\tau' K})_{\ell \le \ell_y, \tau' \le \tau})$
	else
	$x := \mathcal{D}(y, k_d)$

$y := \mathcal{B}(z, k_e)$ typed with Rule BLIND	
if $I(k_e) = L$	
$y := \mathcal{B}(z, k_e)$	$y := \mathcal{B}(z, k_e)$
if $I(k_e) = H$	

$x_0 := 0; x_1 := z; B; y := m;$
if $exists(z, (log_{\ell\tau' K})_{\ell \le L(\tau), \tau' \le \tau})$ then
$log_{\ell_y \tau K} := log_{\ell_y \tau K} + (y, assoc(y, (log_{\ell\tau' K})_{\ell \le L(\tau), \tau' \le \tau}))$
else
$log_{\ell_y \tau K} := log_{\ell_y \tau K} + (y, z)$
	$y := \mathcal{E}(0, k_e);$
	if $exists(z, (log_{\ell\tau' K})_{\ell \le L(\tau), \tau' \le \tau})$ then
	$log_{\ell_y \tau K} := log_{\ell_y \tau K} + (y, assoc(y, (log_{\ell\tau' K})_{\ell \le L(\tau), \tau' \le \tau}))$
	else
	$log_{\ell_y \tau K} := log_{\ell_y \tau K} + (y, z)$

Figure 5.4: Substitutions in P, typable with $\Gamma(y) = \mathsf{Enc}\, \tau\, K(\ell_y)$, to obtain command contexts $P^\star$ and $P \backslash K$

Rule TOp, we type the concatenation operation on the log variable. Variable $log_{\ell_y \tau K}$ has the type $\mathsf{Array}\,(\mathsf{Enc}\,\tau\,K(\ell_y) * (\tau \sqcup \ell_y))$ and (y, x) has the type $\mathsf{Enc}\,\tau\,K(\ell_y) * \tau$, so the command can be typed at level ℓ_y with Rule Assign. We finally apply TSeq to type the command (5.9.5) at level ℓ_y.

Homomorphic Applications We consider the case $I(y) = H$. For every homomorphic command

$$y := f_K(z_1, \ldots, z_n, k_e)$$

typed with Rule Hom-Fun at some level ℓ_y, (for some security types $\vec{\tau}, \tau \in \mathsf{E}$ such that $\ell_d \leq L(\tau)$), we have:

$$\Gamma(y) = \mathsf{Enc}\,(\tau)\,K\,(\ell_y) \tag{5.3}$$
$$\vdash z_i : \mathsf{Enc}\,(\tau_i)\,K\,(\ell_y) \tag{5.4}$$
$$\vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y) \tag{5.5}$$
$$\Gamma(x \mapsto (\tau \sqcup \ell_y), \vec{x} \mapsto (\vec{\tau} \sqcup \ell_y)) \vdash x := f(\vec{x}) : \tau \sqcup \ell_y \tag{5.6}$$

We type the first command at level ℓ_y with Hom-Fun with the same hypotheses. For the second command:

- The conditional expression is typed with Rule Coerce-equality (see 5.2); hence, by subtyping, the guard expression has type $\mathsf{Data}\,(\ell_y)$;
- $\Gamma\backslash K \vdash f(assoc(z_1, (log_{\ell\tau_1 K})_{\ell \leq \ell_y}) : \tau \sqcup \ell_y$ from hypothesis (5.6), Rule Coerce-equality (see 5.1), and TOp;
- $log_{\ell_y \tau K}$ has type $\mathsf{Array}\,(\mathsf{Enc}\,(\tau)\,K\,(\ell_y) * (\tau \sqcup \ell_y))$.

We conclude with TCond and TSeq.

Decryptions Every decryption command

$$x := \mathcal{D}(y, k_d)$$

typed with Rule Decrypt:

$$\tau \leq \Gamma(x) \qquad \tau \in \mathsf{E} \tag{5.7}$$
$$\vdash y : \mathsf{Enc}\,\tau K(L(x)) \tag{5.8}$$
$$\vdash k_d : \mathsf{Kd}\,\mathsf{E}\,K(\ell_d) \;\; \ell_d \leq_I x \tag{5.9}$$
$$(\alpha \not\leq_I \ell_d \textbf{ and} \tag{5.10}$$
$$(\alpha \not\leq_I y \tag{5.11}$$
$$\textbf{or } (\mathcal{G}_e, \mathcal{E}, \mathcal{D}) \text{ is CCA2 } \textbf{and } \tau' \leq \tau \text{ for } \tau' \in \mathsf{E})) \tag{5.12}$$
$$\textbf{or } \ell_d \leq_C x \textbf{ or } \ell_d \leq_C \alpha \tag{5.13}$$

We check that the replacements command can also be typed with $\Gamma\backslash K$ at level $L(x)$ in $P\backslash K$, using Rule Assign instead of Decrypt. There are four different cases:

- If $I(k_d) = L$ in the CPA variant.
  - We have ${k_d}^{init} \leq_C x$ from hypothesis 5.13 and ${k_d}^{init} \leq_I x$ from hypothesis (5.9). We can type the guard at level $\mathsf{Data}\,(L(x))$.
  - In the then branch, $\vdash assoc(y, (log_{\ell\tau K})_{\ell \leq \ell_y}) : \tau' \sqcup \ell$ from Rule TOp; thanks to hypothesis (5.7) we type the command with Rule Assign at level $L(x)$.

– We type the next alternative command at level $L(x)$ with Rule DECRYPT. The original decryption command is typable with Rule DECRYPT, and by 5.9, we have that $\ell_d = L \leq_I I(x)$, then $I(x) = L$. That is, the decryption rule is applied to a low integrity result.

We conclude with TCOND.

- If $I(k_d) = H$ in the CPA variant. We type $\vdash assoc(y, (log_{\ell_\tau K})_{\ell \leq \ell_y}) : \tau' \sqcup \ell$ from Rule TOP; thanks to hypothesis (5.7) we type the command with Rule ASSIGN at level $L(x)$.
- If $I(k_d) = L$ in the CCA variant. Typing is as the low integrity case in the CPA variant, except we have an extra guard and an extra decryption command typed with Rule DECRYPT. As in the CPA case, from here we conclude that $I(x) = L$. For the extra guard,we type it from definition of logs and Rule COERCE-EQUALITY. By subtyping and from hypothesis (5.8), we type it at level $\mathsf{Data}\,(L(x))$.
- If $I(k_d) = H$ in the CCA variant. Similar to previous cases. We just need to show that the rule decrypt is applied only to low integrity results. Rule decrypt is used if the ciphertext is not in the log variables. We show by contraction that in this case, variable y is low integrity and by the decryption rule, this means that $L(x) = L$ as well. Assume that variable variable y is of high integrity. Then it can only be defined by an encryption or blinding command with a key of label K. By safety condition 2, variable y has to be defined before being read in a decryption command. If it is a high integrity variable then by the transformation applied to P', then y must appear in a log variable.

- Every blinding command

$$y := \mathcal{B}(z, k_e)$$

typed using Rule BLIND at some levels ℓ_y, with some security types $\tau, \tau' \in \mathsf{E}$ such that $\ell_d \leq L(\tau)$ and

$$\Gamma(y) = \mathsf{Enc}\,\tau\, K(\ell_y) \tag{5.14}$$
$$\tau' \leq \tau \tag{5.15}$$
$$\vdash z : \mathsf{Enc}\,\tau'\, K(L(\tau)) \;\; z \leq_I y \tag{5.16}$$
$$\vdash k_e : \mathsf{Ke}\,\mathsf{E}\,K(\ell_y)\;\tau, \tau' \in \mathsf{E} \tag{5.17}$$
$$\alpha \leq_I k_e \;\; \textbf{or} \;\; \tau \leq_C \alpha \tag{5.18}$$

In all cases, we show that the replacement command in $P \backslash K$ can also be typed at level ℓ_y in $\Gamma \backslash K$ using rule BLIND with no declassification:

For the first encryption command we use the typability hypothesis of the original blinding and typing Rule COERCE-ENCRYPT to type the encryption of 0. We type the assignments in the branches as in previous cases. We conclude with TCOND and TSEQ.

Computational Non Interference

azer

Relying on $P \backslash K$ and $P^\star$, our next lemmas show how to eliminate the use of cryptographic types for static key label K.

The following auxiliary definition and lemmas are useful to assume that the adversary cannot guess the ciphertexts of an encryption of 0.

Definition 34 (Randomly secure ciphertexts) *Consider the probabilistic commands*

$$RAND \;\doteq\; k_e, k_d := \mathcal{G}_e(); A_{RAND}; m_{RAND} := \mathcal{E}(0, k_e)$$

The encryption scheme $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ provides randomly secure ciphertext when the probability $\Pr[RAND; m_{RAND} = x_{RAND}]$ is negligible for any polynomial command context A_{RAND} such that $k_d \notin rv(A)$ and $k_d, k_e, \eta \notin wv(A)$.

Lemma 54 *If $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ is CCA2 then it provides randomly secure ciphertext.*

PROOF: Let A be a polynomial command context such that $k_d \notin rv(A)$ and $k_d, k_e, \eta \notin wv(A)$.

$$\begin{aligned} A_I[E,D] &\doteq A_{RAND}; x_0 := 0; x_1 := 1; E \\ A_{CCA}[E,D] &\doteq A_{RAND}; x_0 := 0; x_1 := 1; E; \\ &\quad \text{if } m_{RAND} = m \text{ then } g := 0 \text{ else } g := \{0,1\} \end{aligned}$$

By simple renaming, we have $\Pr[k_e, k_d := \mathcal{G}_e(); A_{CCA}, g = b_{CCA} | b_{CCA} = 0] = \Pr[k_e, k_d := \mathcal{G}_e(); A_{RAND}; m_{RAND} := \mathcal{E}(0, k_e); m_{RAND} = x_{RAND}]$. Also, $\Pr[k_e, k_d := \mathcal{G}_e(); A_{CCA}, g = b_{CCA} | b_{CCA} = 1] = frac12$. Hence, $\Pr[k_e, k_d := \mathcal{G}_e(); A_{RAND}; m_{RAND} := \mathcal{E}(0, k_e); m_{RAND} = x_{RAND}]$ is negligible, and $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ provides randomly secure ciphertext. □

Lemma 55 *If $CNI^I_\alpha(P \backslash K)$, then $CNI^I_\alpha(P)$.*

PROOF: We prove equivalence for the high-integrity memory using the invariant: if $\vdash y : \mathsf{Enc}\, \tau\, K(\ell_y)$ such that $I(\ell_y) = H$, and $\mu(y) \neq \perp$ then $\mathcal{D}(\mu(y), k_d) = assoc(\mu \backslash K(y), log_{\ell_y \tau K})$.

□

Lemma 56 (Encryption) *An opponent that breaks $CNI^C_\alpha(P)$ also breaks $CNI^C_\alpha(P \backslash K)$.*

The proof relying on CPA instead of CCA2 is the same, but there is no oracle decryption in $P^\star$.

PROOF:

In case $I(\tau) = H$, we show that the decryption table lookup always succeeds by keeping the invariant that if $\vdash y : \mathsf{Enc}\, \tau\, K(\ell_y)$ such that $I(\ell_y) = H$, and $\mu(y) \neq \perp$ then $\mathcal{D}(\mu(y), k_d) = assoc(\mu \backslash K(y), log_{\ell_y \tau K})$.

Otherwise, both branches of the decryption command return the decryption of y.

Otherwise, assume that the commands J, B_b, $\vec{A}$, T meet the hypotheses of computational non-interference against active adversaries (Definition 28) and let $G[\_, \_]$ be the polynomial command context defined by

$$\begin{aligned} G[E,D] \doteq\ & b_{CNI} := \{0,1\}; \\ & J; \text{if } b_{CNI} \text{ then } B_0 \text{ else } B_1; \\ & P^{\star\star}[\vec{A}]; T; \\ & \text{if } b_{CNI} =_0 g_{CNI} \text{ then } g_{CCA} := 1 \text{ else } g_{CCA} := 0 \end{aligned}$$

where the command $P^{\star\star}$ is obtained from $P^\star$ with the elimination of assignments of the form $k_d := k_d^0$. Notice that the decryption key k_d^0 is not used for decryptions in $P^\star$ in the cases where k_d is high integrity or if k_d is low integrity but has not been modified by the adversary. Hence the semantics of $P^\star$ and $P^{\star\star}$ is the same. Hence the variable k_d^0 does not occur in any of these commands except within D, and the variable k_e is not written by any of these commands, so G is a valid CCA2 adversary (A in Definition 19) and, by IND-CCA2 security for $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$, we know that its advantage is negligible:

$$|\Pr[CCA; g_{CCA} =_0 b_{CCA}] - \tfrac{1}{2}| \leq \epsilon(\eta) \tag{5.19}$$

We separate two cases in the runs of CCA, depending on its initial sampling of b_{CCA}:

- When $b_{CCA} =_0 1$ (that is, when E encrypts x), we have $g_{CCA} =_0 1$ after running CCA with the same probability as that of the opponent winning the game CNI^C_α for P' (command CNI): except for the auxiliary operations on the fresh CCA variables and $log_{\tau K}$, the two computations yield identical memory assignments at every step.

$$\Pr[CCA; g_{CCA} =_0 1 | b_{CCA} =_0 1] = \Pr[CNI(P'); b_{CNI} =_0 g_{CNI}] \tag{5.20}$$

To relate the two commands, compare the CCA game after specializing b_{CCA}

$$\begin{array}{l} log := \mathbf{0}; \\ k_e, k_d := \mathcal{G}_e(); \\ G[(m := \mathcal{E}(x_1, k_e); log := log + m), D] \end{array}$$

that is, by definition of G,

$$\begin{array}{l} log := \mathbf{0}; \\ k_e, k_d := \mathcal{G}_e(); \\ b_{CNI} := \{0,1\}; \\ J; \text{if } b_{CNI} =_0 0 \text{ then } B_0 \text{ else } B_1; \\ P^{\star\star}[\vec{A}]; T; \\ \text{if } b_{CNI} =_0 g_{CNI} \text{ then } g_{CCA} := 1 \text{ else } g_{CCA} := 0 \end{array}$$

where the encryption command (**??**) within $P^{\star\star}$ after inlining E and simple rewriting is

$$\begin{array}{l} y := \mathcal{E}(x, k_e); \\ log_{\tau K} := log_{\tau K} + (y, x) \\ x_0 := 0; x_1 := x; m := y; log := log + y \end{array}$$

... to the CNI game

$$\begin{array}{l} b_{CNI} := \{0,1\}; \\ J; \text{if } b_{CNI} =_0 0 \text{ then } B_0 \text{ else } B_1; \\ k_e, k_d := \mathcal{G}_e(); \\ P'[\vec{A}]; T \end{array}$$

We perform the following series of code transformations: we move the key generation forward; we insert the initializations $log_{\tau K} := \mathbf{0}$ before P'; we replace encryption and decryption commands, as defined above for $P^{\star\star}$.

As a reduction invariant, each $log_{\tau K}$ holds a list of pairs of bitstrings m, v such that $\mathcal{D}(m, k_d) = v$, and log (if we consider a CCA scheme) holds a list of bitstrings m for which there is such a pair in one of the $log_{\tau K}$. Indeed, the logs are updated only by the replaced encryption commands then E, respectively, with correct encryptions under k_e in m, and Definition 18(4) guarantees their correct decryption.

If a value m is produced by E, it is in one of the $log_{\tau K}$. If D is called on m, it means that the corresponding $log_{\tau K}$ has not been checked. Using Lemma 54, we prove that this is only possible with negligible probability. In all other cases the command that replaces the decryption also returns the decryption of y.

- When $b_{CCA} = 0$, we have $g_{CCA} =$ after running CCA with the same probability as that of the opponent winning the game CNI^C_α for $P\backslash K$ (command CCA for $P\backslash K$).

$$\Pr[CCA; g_{CCA} = 0 | b_{CCA} = 0] = \Pr[CNI(P\backslash K); b_{CNI} = g_{CNI}] + \epsilon \tag{5.21}$$

Where ϵ is negligible.

To relate the two commands, similarly compare the specialized CCA game

$$log := \mathbf{0}; k_e, k_d := \mathcal{G}_e(); G[(m := \mathcal{E}(x_0, k_e); log := log + m), D]$$

to the CNI game above. We perform the following series of code transformations: we move the key generation forward; we insert the initialization $log_K := \mathbf{0}$ before P'; we replace encryptions and decryptions commands, as defined above for $P^\star$.

As an invariant, $log_{\tau K}$ holds a list of pairs of bitstrings m, v such that $\mathcal{D}(m, k_d) = 0$, and log holds a list of bitstring m for which there is such a pair in one of the $log_{\tau K}$. Thus, D is never called on any value m produced by E. The command that replaces decryptions in $P^\star$ behaves as the command that replaces decryptions in $P\backslash K$ except when $m \in log$ but not in one of the $log_{\tau K}$ checked. We check all the logs which can flow to m .

Moreover, by induction hypothesis on $P\backslash K$ for the same opponent parameters J, B_b, $\vec{A}$, and T, this probability is $\frac{1}{2}$ up to a negligible function.

By definition, b_{CCA} is sampled from $\{0, 1\}$, so each case has probability $\frac{1}{2}$ and we have

$$\begin{aligned}\Pr[CCA; g_{CCA} = b_{CCA}] - \tfrac{1}{2} &= \tfrac{1}{2}(\Pr[CCA; g_{CCA} = b_{CCA} | b_{CCA} = 0] - \tfrac{1}{2}) \\ &\quad + \tfrac{1}{2}(\Pr[CCA; g_{CCA} = b_{CCA} | b_{CCA} = 1] - \tfrac{1}{2}) \\ &= \tfrac{1}{2}(\Pr[CNI(P); b_{CNI} = g_{CNI}] - \tfrac{1}{2}) \\ &\quad + \tfrac{1}{2}(\Pr[CNI(P\backslash K); b_{CNI} = g_{CNI}] - \tfrac{1}{2})\end{aligned}$$

$$\begin{aligned}|\Pr[CNI(P); b_{CNI} = g_{CNI}] - \tfrac{1}{2}| &\leq 2|\Pr[CCA; g_{CCA} = b_{CCA}] - \tfrac{1}{2}| \\ &\quad + |\Pr[CNI(P\backslash K); b_{CNI} = g_{CNI}] - \tfrac{1}{2}|\end{aligned}$$

The triangular inequality above bounds the advantage in the CNI game with opponents J, B_b, $\vec{A}$, and T with a sum of two functions that are negligible, by IND-CCA2 security (5.19) and induction hypothesis (5.21), respectively. □

Proof of Theorem 10 Let P be a safe command, and $\vec{A}$ an adversary. We let $P' = P[\vec{A}]$. P' is safe for $\mathcal{K} =$. Then, the proof is by induction on the number of static key labels outside of $\mathcal{K}$.

For the base case ($\mathcal{K}$ contains all key labels), we apply Lemma 51 to P', and we get computational non-interference.

For the inductive case, we select a free label K for $\mathcal{K}$. If K depends c for some c, we apply Lemma 48, and get a safe command P such that :

- For all well-formed memory μ, $\rho_\infty(\langle P, \mu\rangle) \equiv_{LL} \rho_\infty(\langle P', \mu\rangle)$.
- For all well-formed memory μ, and some c $\rho_\infty(\langle P, \mu\rangle) \equiv_{cH} \rho_\infty(\langle P', \mu\rangle)$. By Lemma 41, for all well-formed memory μ, $\rho_\infty(\langle P, \mu\rangle) \equiv_{MH} \rho_\infty(\langle P', \mu\rangle)$.
- There is no low-integrity decryption for label K.

Hence, by Lemma 52, if P is CNI then P' is CNI. Then, using Lemma 55 and 56, we show that P' is CNI if $P\backslash K$ is CNI. Also, by Lemma 53 $P\backslash K$ follows the invariant and we conclude by induction. □

6

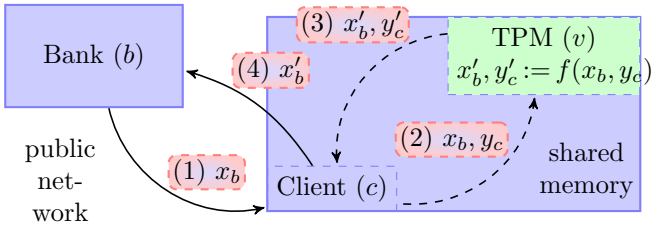
Compiling Information-Flow Security to Minimal Trusted Computing Bases

In this chapter, we use hardware mechanisms to enforce security. We consider distributed applications specified as a set of imperative program with information flow annotations (the output of the compiler of Chapter 4), one for each machine plus one for a virtual trusted participants. Then, we use hardware mechanisms (and not only cryptographic mechanisms) to simulate, with strong security guarantees, this trusted participant on an untrusted machine.

6.1 Programming with TPMs

When designing or reviewing the security of a system, a first step is to identify its trusted computing base (TCB), that is, the set of components that need to be trusted to achieve a given level of security. For general-purpose networked machines, this set is large and complex; it includes the hardware, an operating system, a runtime environment and their libraries (maybe 10^8 LOCs overall) plus drivers, applications, and dynamically loaded code. This leads to a best-effort approach to security, at odds with formal verification, which provides strong guarantees only for smaller, simpler systems.

Minimal TCBs Modern computer architectures provide hardware support for reducing TCBs and protecting privileged operations. Thus, most computers come bundled with some form of secure coprocessor with a dedicated secure instruction set—for example, most laptops now embed a Trusted Platform Module (TPM) [TCG, 2005] and many high-end processors feature a special *late launch* functionality (AMD's Secure Virtual Machine Architecture, 2005, and Intel's Trusted Execution Technology, 2009). These instructions can run a given piece of code in isolation, with strong code-based identity and privileged cryptographic operations, for instance to seal persistent state or to perform remote attestation. Such hardware mechanisms can greatly reduce the TCB of security applications, by removing the need to trust the host operating system and other applications, and thus help protect critical data and computations from malicious software. TPMs are routinely used for secure booting, e.g. BitLocker [Microsoft, 2006] guards access to the master keys for disk encryption, so that the disk content may be read only after authenticating the user and the operating system. Research papers also describe e.g. how to build secure online payment systems [Balfe and Paterson, 2008] and how to use late launches to run small pieces of application code in isolation [McCune et al., 2008, 2009]. Still, the secure instructions are remarkably seldom used in practice. We believe that the complexity of their low-level interface and the lack of programming tools are major obstacles to their mainstream adoption for writing security applications.



Example: applying for a loan We consider Example 21 introduced in Chapter 4. In this example, the program involve two distrustful parties, a bank and a client. Contrarily to Chapter 4, we assume that we do not have a trusted third party. Although the client and the bank do not trust one another, they may agree to securely run the loan-evaluation code on a TPM-enabled client machine. This simple computation is depicted above.

The bank sends its (encrypted, signed) secret input (x_b) to the client; the client forwards it to the code running the loan evaluation, together with its own input (y_c), using shared local memory; after securely booting, the TPM-protected code decrypts its input, evaluates the loan, and returns its results; finally, the client gets its result (y'_c) and may forward the (encrypted, signed) output (x'_b) to the bank if the loan is granted.

The messages passed between the bank and TPM-protected code must be cryptographically protected, so that for instance the bank input may be read and processed only by that code—not by the client or the network. Also, this code must refuse to run multiple loan evaluations for different client inputs without the bank's consent, as this may enable the extraction of the bank input.

Compiling with minimal TCBs We take as input a tuple of programs with information flow annotations, each supposed to run on and host with a given level of trust (e.g., the output of the security compiler of Chapter 4). We then implement one of the hosts on any untrusted machine using cryptography and hardware mechanisms. This result in a new set of programs, that run on hosts of the same level of trust except for one, which can run on any host with a sufficiently trusted TPM. We then show that our compilation scheme preserves information-flow security under standard cryptographic assumptions.

To this end, we first extend our imperative language with dynamic code linking, so that we can compute on core represented as data (e.g., to compute a code identity) and run this code. We then specify a subset of the TPM instructions within our core imperative programming language. Our model aims at formal simplicity while still reflecting the main security features of hardware and cryptographic specifications, at a level of detail sufficient for reasoning about information flows. We model secure instructions to manage monotonic counters; measure code; run code in isolation; cryptographically sign data using the private attestation key of the TPM; and cryptographically seal and unseal data associated with some code. We also encode the Static Root of Trust (SRTM) protocol using these commands. We then build our new security-preserving compiler using these secure instructions to simulate a trusted third party when necessary. The compiler and some implementation example can be found at `http://msr-inria.inria.fr/projects/sec/cflow`.

6.1.1 Contents

The rest of this chapter is composed as follows. Section 6.2 refines the imperative language of Section 3.1 with an higher-order primitive. Section 6.3 updates our information-flow policies and active adversaries for higher-order. Sections 6.4 and 6.5 describe and formalize secure hardware instructions and their use for securing boot sequences. Section 6.6 shows how to use TPM capabilities to implement secure virtual hosts, such as those produced by CFLOW. Section 6.7 provides experimental results and concludes.

6.2 An imperative higher-order language

One of the capabilities of secure hardware is to execute a piece of code from the memory in isolation and register its identity. In order to model this functionality, we need to be able to run code that is represented as data. Thus, we enrich the language from chapter 3 with a command to turn data into executable code, with the following grammar:

$$P ::= \cdots \mid \mathsf{link}\, e\, [\widetilde{P}]\, \ell$$

Programs P can now use this command to dynamically load, link, and run code with command link. The expression e encodes the command that is dynamically loaded. The commands in $\widetilde{P}$ are subroutines that can be called by the dynamic code in e; they are used to model callbacks (e.g., to other commands of the secure hardware, as seen below in Section 6.4). The level ℓ limits the level of the commands that can be dynamically loaded.

6.2.1 Commands as Data

We use data constructors to embed commands (and their expressions) as expressions, such as $\mathit{op\_if}(e_1,e_2,e_3)$ for conditional (if "the expression represented by e_1" then "the command represented by e_2" else "the command represented by e_3") and op_x for variable x. For instance, the command $c := c + 5$ is represented by the expression $\mathit{op\_assign}(\mathit{op}_c, (\mathit{op\_plus}(\mathit{op}_c,5)))$. To ease the writing of expressions representing commands, we let $\langle P\rangle$ be the expression that represents command P. Command expressions can also contain free variables; these variables are quoted within $\langle P\rangle$. For instance, the expression $\mathit{op\_assign}(\mathit{op}_c,(\mathit{op\_plus}(\mathit{op}_c,t)))$ is written $\langle c := c + \text{`}t\rangle$.

We give the operational semantic for the link command:

$$\frac{[\![e]\!](\mu) = \langle P\rangle \qquad \vdash P : \ell}{\langle \mathsf{link}\, e\, [\widetilde{P}]\, \ell, \mu\rangle \rightsquigarrow_1 \langle P[\widetilde{P}/\widetilde{X}], \mu\rangle} \text{(Link)}$$

The command $\mathsf{link}\, e\, [\widetilde{P}]\, \ell$ dynamically checks that the result of expression e represents a valid command at level ℓ parametrized by subcommand variables $\widetilde{X}$, and then runs that command after replacing each X_i with the command P_i. If the check fail, the program is stuck and does not terminates. The commands in $\widetilde{P}$ are not dynamically checked and may be more privileged than ℓ. We write $\mathsf{link}\, e\, \ell$ instead of $\mathsf{link}\, e\, []\, \ell$ when $\widetilde{P}$ is empty. These checks occur at link-time, before running the command, as would be the case with a high-level virtual machine that performs typchecking before loading code. In contrast, low-level protection for executable memory and data memory is usually enforced later, at runtime (e.g. by triggering memory page faults). Thus, for instance, information flows due to low-level memory error handling are outside the scope of our model. See also Askarov and Sabelfeld [2009] for a information-flow model of dynamic loading with run-time monitoring.

To illustrate this rule, consider three commands P, P' and P'' such that $\vdash P : \ell$ and $\vdash P' : \ell$, and P' contains the free command variable X. Let c and t be two variables, and μ a memory such that $\mu(t) = 7$. We have

$$\begin{array}{rcl} \langle \mathsf{link}\, \langle P\rangle\, \ell\, , \mu\rangle & \rightsquigarrow_1 & P \\ \langle \mathsf{link}\, \langle P'\rangle\, [\, P'']\ell, \mu\rangle & \rightsquigarrow_1 & P'[P''/X] \\ \langle \mathsf{link}\, \langle c := c + \text{`}t\rangle\, \ell\, , \mu\rangle & \rightsquigarrow_1 & c := c + 7 \end{array}$$

6.3 Information flow security

Next, we extend the strict and lax type systems from Section 3.3, and we prove the corresponding theorems taking into consideration the link command.

6.3.1 A strict typing rule for link

We first give a typing rule for link in our strict type system that appear in Figure 3.2 (without the additional Rule TLocality). Since code variables are now used with link (and not only as adversaries placeholders), we also give a new typing rule for codes variables:

$$\frac{\vdash e : \ell \qquad \vdash \widetilde{P} : (\bot, \top)}{\vdash \mathrm{link}\, e\, [\widetilde{P}]\, \ell : \ell}\ (\textsc{TLink Strict}) \qquad \frac{}{\vdash X : (\bot_C, \top_I)}\ (\textsc{TVar})$$

As defined in the semantic Rule Link, the command $\mathrm{link}\, e\, [\widetilde{P}]\, \ell$, when executed, will check that the expression e represents a valid command at level ℓ before running it. Accordingly, we type the link command also at level ℓ, after checking that its actual auxiliary commands ($\widetilde{P}$) have level $(\bot_C, \top_I)$, as anticipated by Rule TVar. (We considered typing auxiliary commands and command variables at other levels, but this is not needed for our present purpose.) Rule TLink Strict also checks that the expression e has level ℓ, to keep track of the implicit flow from command values to their runtime effects.

To illustrate those rules, consider two variables *secret* and x at levels $\ell_s = \Gamma(secret)$ and $\ell_x = \Gamma(x)$ such that $\ell_s \not\leq \ell_x$. We have:

1. The command $x := secret$ is not typable, since it implements a direct flow from *secret* to x.
2. The command link $\langle x := secret\rangle\ \ell$ types at level ℓ for every level ℓ, but runtime type checking fails when the command is executed. Indeed, $\langle x := secret\rangle$ is a closed expression (contrarily to the command $x := secret$), so we have $\vdash \langle x := secret\rangle : \ell$ by Rule TOp. Hence $\vdash$ link $\langle x := secret\rangle\ \ell : \ell$ by Rule TLink Strict. However, since the command $x := secret$ is not typable, the reduction Rule Link does not apply.
3. The command link $\langle x :=$ '$secret\rangle\ \ell_x$ is not typable and is not non-interferent since the state $\langle$link $\langle x :=$ '$secret\rangle\ \ell_x, \mu\rangle$ reduces to $\langle x := \mu(secret), \mu\rangle$ for every μ, and this results in an insecure flow from *secret* to x. Indeed, the expression $\langle x :=$ '$secret\rangle$ has a free variable *secret* of level ℓ_s such that $\ell_s \not\leq \ell_x$, hence the command $\langle x :=$ '$secret\rangle$ does not type at level ℓ_x and Rule TLink Strict does not apply.
4. The command link $\langle x :=$ '$secret\rangle\ \ell_s$ types at level ℓ_s but runtime type checking fails when the command is executed. Indeed, we have $\vdash \langle x :=$ '$secret\rangle : \ell_s$, hence, by Rule TLink Strict, $\vdash$ link $\langle x :=$ '$secret\rangle\ \ell_s : \ell_s$. However, irrespective of the value v of *secret* at runtime, the command $x := v$ does not type at level ℓ_s since $\ell_s \not\leq \ell_x$, and the reduction Rule Link does not apply.
5. The command link $\langle$if *secret* then $X\rangle[\ x := 0\ ]\ \ell_s$ is not typable unless $\ell_x = (\bot_C, \top_I)$, since we need $\vdash x := 0 : (\bot_C, \top_I)$ to apply Rule TLink Strict.
6. The command link $\langle$if *secret* then $X\rangle[\ x := 0\ ]\ \ell_s$ is typable at level ℓ_s if $\ell_x = (\bot_C, \top_I)$ but runtime type checking fails when the command is executed, since it implements an implicit flow from *secret* to x trough execution of the the auxiliary command. Indeed, using Rule TVar, we get $\vdash X : (\bot_C, \top_I)$, hence the command 'if s then X' is not typable and the reduction Rule Link does not apply.

We state our typing theorem of non-interference with the additional strict typing rules TLink Strict and TVar:

Theorem 14 (Non-interference with link) *Let Γ be a security policy, P a (strictly) well-typed command, $\ell \in \mathcal{L}$, and μ_0 and μ_1 two initial memories such that P terminates.*

If $\mu_0 =_\ell \mu_1$, then $\rho_\infty(\langle P, \mu_0\rangle) =_\ell \rho_\infty(\langle P, \mu_1\rangle)$. If $\mu_0 =^\ell \mu_1$, then $\rho_\infty(\langle P, \mu_0\rangle) =^\ell \rho_\infty(\langle P, \mu_1\rangle)$.

The proof of Theorem 14 is at the end of this chapter, in Subsection 6.8.1.

6.3.2 Declassifications and endorsements

Our strict type system yields non-interference, and thus excludes interesting usages of link such as, for example, privileged callbacks (i.e., callbacks that are more privileged that the code calling them). We illustrate a program using privileged callbacks below.

Example 34 (Password protection) *Consider a system that provides a subcommand that conditionally releases a secret after verifying a password, and tolerates up to three failed attempts (to protect against brute-force attacks on the password):*

$c := 0;$
$\mathsf{link}\ a[\mathsf{if}\ c < 3\ \&\&\ guess = pwd\ \mathsf{then}\ r := secret\ \mathsf{else}\ c{+}{+}]\ LL$

with $\Gamma(guess) = LL$, $\Gamma(pwd) = HH$, $\Gamma(r) = LH$, *and* $\Gamma(secret) = HH$ *(as in Example 14), using a trusted counter variable* c *with* $\Gamma(c) = LH$, *and a variable* a *with* $\Gamma(a) = LL$ *for dynamically loading code that may call this subcommand. Intuitively,* a *contains arbitrary low-level code, representing an active adversary, that cannot leak* pwd *or* secret *and cannot write* pwd, secret, *or* c. *If linking succeeds, this code can only access the secret by calling its privileged subcommand, and only the first three calls may succeed. For instance, running the command above with an initial memory where* a *is set to the command*

$guess := 0;\ \mathsf{while}\ guess < 10\ \&\&\ r = 0\ \mathsf{do}\ \{\ X;\ guess{+}{+}\ \}$

leaks secret *to* r *only if* pwd *is* 0, 1, *or* 2. *More generally, if we also assume that* pwd *is sampled at random, the probability that any adversary command learns anything about* secret *is bounded by the sum of the probabilities of the three most probable passwords (*$3/N$ *if there are* N *uniformly distributed passwords). Although this program is arguably secure, it is not typable, even in our lax type system, since the callback is more privileged than* LL.

6.3.3 A more permissive type system

For typing those unsafe programs, we introduce a new typing rule for link, which allows for implicit flows trough privileged callbacks but take them into account to compute the level of the command. The Rule TVar is left unchanged. This typing rule comes in addition of the lax rules of Figure 3.3.

$$\frac{\vdash e : \ell \qquad \vdash \widetilde{P} : \ell \qquad \ell \leq \ell'}{\vdash \mathsf{link}\, e\, [\widetilde{P}]\, \ell' : \ell} \text{ (TLink Privileged)}$$

The Rule TLink Privileged generalizes the Rule TLink Strict by allowing the caller to link e with auxiliary commands at arbitrary levels of integrity (the Rule TLink Strict already allows for arbitrary levels of confidentiality since Rule TVarsuppose that it is the case), but it records those levels in the type of the command. In particular, it allows for explicit declassifications in the auxiliary commands, as in Example 34. This endorses at link-time any calls to the auxiliary commands, since dynamic typing of the callee ignores the integrity of auxiliary command variables (Rule TVar).

As for commands without link, the lax type system enforces two fundamental properties (see Theorems 6 and 7). First, if a command has level ℓ, then it does not write variables below ℓ.

Theorem 15 (Lax Containment with link) *Let* Γ *be a policy,* $\ell \in \mathcal{L}$ *a label,* P *a command, and* μ *a memory such that* P *terminates.*

If $\vdash P : \ell$, *then* $\rho_\infty(\langle P, \mu\rangle) =^\ell \mu$ *and for any* ℓ' *such that* $\ell \not\leq \ell'$, *we have* $\rho_\infty(\langle P, \mu\rangle) =_{\ell'} \mu$.

The proof of Theorem 15 is at the end of this chapter, in Subsection 6.8.2.

Second, a command at level ℓ may declassify values of confidentiality c only if $I(\ell) \leq R(c)$.

Theorem 16 (Robust non-interference with link) *Let Γ be a policy, $\ell \in \mathcal{L}$ a label, $c \in \mathcal{L}_C$ such that $I(\ell) \not\leq R(c)$. Let P be a command and μ_0, μ_1 memories such that P terminates.*

If $\vdash P : \ell$ and $\mu_0 =^{(c,H)} \mu_1$, then $\rho_\infty(\langle P, \mu_0 \rangle) =^{(c,H)} \rho_\infty(\langle P, \mu_1 \rangle)$.

The proof of Theorem 16 is at the end of this chapter, in Subsection 6.8.3.

Active adversaries

We now update our active adversary model such that adversaries commands can use the new command link, but cannot use it to transgress its privileges. Formally, for a given $\alpha \in \mathcal{L}$, we let α-adversaries range over tuple of commands $\widetilde{A}$ that read variables in $V_{\mathcal{A}}^C$ with confidentiality level less than or equal to $C(\alpha)$, write variables in $V_{\mathcal{A}}^I$ with integrity level more than or equal to $I(\alpha)$, and use link only with an integrity level above $I(\alpha)$.

In the theorem below, we prove that those new adversaries can only write variables with an integrity level above $I(\alpha)$, and can only declassify variables with a confidentiality level below $C(\alpha)$.

Theorem 17 *Let $\alpha \in \mathcal{L}$ be an adversary level, R be a robustness function robust against α, Γ be a security policy, and A an command that reads only variables of lower confidentiality than $C(\alpha)$, write only variables of higher integrity than $I(\alpha)$, and use links only with a level above α. Let μ, μ_0, and μ_1 be three memories such that A terminates, and such that $\mu_0 =_{C(\alpha),\top_I} \mu_1$.*

Then $\rho_\infty(\langle A, \mu \rangle) =^{\top_C, I(\alpha)} \mu$ and $\rho_\infty(\langle A, \mu_0 \rangle) =^{C(\alpha),\top_I} \rho_\infty(\langle A, \mu_1 \rangle)$

The proof of Theorem 17 is at the end of this chapter, in Subsection 6.8.4.

6.4 Command semantics for secure instructions

The section above introduces an high level language primitive so we can model one particular functionality of modern processors with a TPM. We now present our model of a core subset of the security features of this modern hardware. Aiming at formal simplicity, we do not account for all the details of their hardware specification, but still intend to reflect their gist. We set an robust security level ℓ_{TPM} for the TPM, and assign that level to fixed variables that model parts of the hardware-protected memory, together with fixed commands that model secure instructions and have privileged access to these variables. (Their initialization is described at the end of the section.) Intuitively, the level ℓ_{TPM} is very high, since the only way to read variables at confidentiality $C(\ell_{TPM})$ or to write variables at integrity level $I(\ell_{TPM})$ is by breaking into the TPM, trough physical attacks. We then model software as commands linked to these privileged subcommands, thereby gaining indirect access to protected variables.

6.4.1 Monotonic counters

The TPM features a collection of monotonic counters, that is, persistent protected memory whose contents can only be read and incremented, but not reset [TCG, 2006, p 681]. Such counters are essential for protection against replays.

We model just one of these counters, using a public variable c at the integrity level of the TPM. Thus, our counter can be read by any command but it is exclusively assigned by the fixed

command INC below. In particular, c cannot be reset or decremented.

$$\text{INC} \doteq \quad c := c{+}1 \qquad\qquad \Gamma(c) = \ell^I_{TPM}$$

(Concretely, TPMs manage a few independent counters with finer access control, and the operating system is in charge of restricting increments to prevent denial of service.)

We illustrate the use of a monotonic counter below.

Example 35 *Continuing with Example 34, we may use the monotonic counter to reliably keep track of guessing attempts:*

link a[if $c < 3$ && $guess = pwd$ then $r := secret$ else INC] LL

6.4.2 Platform configuration registers

The TPM also features a collection of Platform Configuration Registers (PCR), which are cleared when the machine reboots, then selectively written by TPM commands. As detailed in Section 6.5, these registers usually contain measurements of the code running on the machine. PCRs are specialized: the first PCRs are used for static root of trust measurements as the machine boots (SRTM) [Grawrock, 2007], while PCRs 17–19 are used for dynamic root of trust measurements (DRTM) and may be selectively reset without rebooting [TCG, 2006, AMD, 2005]. PCRs are read as high-integrity implicit parameters for many other TPM commands, such as attestation and seals.

We model PCRs as variables h_i at level ℓ^I_{TPM}. For simplicity, we use just two registers, h_1 for SRTM, modified only by EXTEND_1, and h_{17} for DRTM, initialized by SKINIT and modified by EXTEND_{17}, as explained below.

EXTEND_i appends some identity to h_i (typically the identity of some code that is called) and can be used to record a delegation chain starting from a secure kernel [TCG, 2006, p 284]. To keep the size of h_i constant, the chain is implemented as a nested hash, using a cryptographic hash functions $\mathcal{H}$. We model it as

$$\text{EXTEND}_i \doteq \quad h_i := \mathcal{H}(h_i|identity) \qquad\qquad \Gamma(h_i) = \ell^I_{TPM}$$

where '|' is bitstring concatenation and *identity* is a public-untrusted variable (endorsed in the command): $\Gamma(identity) = (\bot_C, \top_I)$.

SKINIT writes a special value (v_{init}) in h_{17}, extends the identity in h_{17} with a new command passed as input, usually called a 'secure kernel', then runs that command, and finally resets h_{17} to a default value ($v_{default}$) [AMD, 2005, p 53]. We model it as an assignment to h_{17}, an EXTEND from the content of a public-untrusted variable *kernel*, followed by a link of *kernel* with subcommands parameters that pass to the new kernel the rest of the TPM interface (written $\widetilde{TPM}$), then a reset of h_{17}. We let $\widetilde{TPM} \doteq \{\text{INC},\text{EXTEND}_{17},\text{ATTEST}_{17},\text{SEAL}_{17},\text{UNSEAL}_{17}\}$ (commands that are launched with SKINIT cannot call SKINIT themselves).

$$\begin{aligned}\text{SKINIT} \doteq \quad & h_{17}{:=}v_{init}\\ & identity := kernel;\\ & \text{EXTEND}_{17};\\ & \text{link } kernel[\widetilde{TPM}]\ \ell^I_{system};\\ & h_{17} := v_{default}\end{aligned}$$

Thus, h_{17} either is at its default value or it holds the identity of the kernel that is currently running, possibly extended by a chain of hashes that records further identity information.

Concretely, the command SKINIT loads code at a privileged (kernel) level. This is reflected in our model by the link label ℓ^I_{system}. It is the responsibility of the operating system to validate the kernels passed by user commands before calling SKINIT e.g. to prevent privilege escalation.

6.4.3 Remote attestation

We now consider a command that attest the current identity stored in the TPM. Each TPM uses a fixed public-key-signature keypair, set during manufacturing and used to uniquely identify and authenticate this particular TPM. ATTEST signs an input value (typically a random challenge generated by a remote machine) and a subset of the PCRs with the private signing key [TCG, 2006]. The resulting signature guarantees that this value has been 'attested' by a command running on a machine with this TPM and these PCR values. This signature can be verified by any command that knows the verification key for the TPM, typically running on a remote machine; the verifier can then interpret the authenticated value and PCRs.

We model attestation with two variables for the TPM keypair, k^+_{TPM} of level ℓ^I_{TPM} and k^-_{TPM} of level ℓ_{TPM}, and two commands

$$\begin{aligned} \text{ATTEST}_i \doteq &\quad tag := \mathcal{S}(i|h_i|plain,\, k^-_{TPM}) \\ \text{VERIFY}_i \doteq &\quad \text{if } \mathcal{V}(i|source|plain,\, tag,\, k^+_{TPM}) \text{ then } X \end{aligned}$$

with public-untrusted inputs *source* and *plain* for the presumed value of h_i and the attested value, and output *tag* for the signature. These commands rely on public-key signing ($\mathcal{S}$) and signature-verification ($\mathcal{V}$) functions. Since the verification key is public, VERIFY need not be a privileged command; its variable X stands for the command guarded by the cryptographic verification.

Security and functionality properties for attestation

We specify the cryptographic properties of ATTEST and VERIFY by relating them to an ideal implementation that maintains a global table for all values attested so far and verifies only values present in this table.

$$\begin{aligned} \text{ATTEST}_i{}^0 \doteq &\quad tag := \mathcal{S}(i|h_i|plain,\, k^-_{TPM}); \\ &\quad log_i := log_i + (h_i|plain) \\ \text{VERIFY}_i{}^0 \doteq &\quad \text{if } exist(log_i,(source|plain)) \\ &\quad \text{then if } \mathcal{V}(i|source|plain,\, tag,\, k^+_{TPM}) \text{ then } X \end{aligned}$$

Security means that, provided the keypair is properly generated, *log* is initially empty (see initialization below), and no other part of the code accesses k^-_{TPM} or *log*, no probabilistic polynomial program can distinguish between (ATTEST$_i$,VERIFY$_i$) and (ATTEST^0_i,VERIFY^0_i), except with negligible probability. This property can be reduced to resistance against forgery attacks for the signing primitive (Definition 22).

Functionality means that verifying an attestation with matching hash and value always succeeds.

Proof outline This reduction is almost by definition, since the ATTEST$_i$ and VERIFY$_i$ command are small variants of the SIG and VERIFY commands that specify computational cryptographic resistance against forgery attacks. The specific proof step is to check that the signed payload $i|h\_i|plain$ is a non-ambiguous binary representation of the triple $(i, h_i, plain)$. □

6.4.4 Sealing

We now consider our last couple of commands, which use the identity stored in the PCR to protect data between calls to the TPM. SEAL encrypts and macs the content of a variable together with the current identity of the sender and the intended identity of the receiver [TCG, 2006, p 298]. Conversely UNSEAL decrypts the contents of a variable and verifies its mac, then verifies the identity of the sender and the current identity of the receiver [TCG, 2006, p 364] before returning

the plaintext. The TPM can handle several keys, but we only model sealing and unsealing keyed with two fixed hardware secrets $s.ke$ and $s.ka$, both at level ℓ_{TPM}.

$$
\begin{array}{rl}
\mathrm{SEAL}_i \doteq & enc := \mathcal{SE}(plain,s.ke);\\
& mac := \mathcal{M}(i|h_i|target|enc,s.ka);\\
& cipher := enc|mac;\\
& enc := 0;\ mac := 0\\
\mathrm{UNSEAL}_i \doteq & enc|mac := cipher;\\
& \text{if } \mathcal{V}_{\mathcal{M}}(i|source|h_i|enc,\ mac,\ s.ka)\\
& \text{then } plain := \mathcal{SD}(enc,s.ke)\\
& \text{else } plain := 0;\\
& enc := 0;\ mac := 0
\end{array}
$$

where $enc|mac := cipher$ is syntactic sugar for assigning to enc and mac substrings of $cipher$ at fixed indexes (since the size of mac is fixed). As illustrated in the rest of the chapter, SEAL and UNSEAL can be used to emulate a persistent, secure memory, and to communicate securely between TPM commands.

6.4.5 Security and functionality properties for seals

We specify the cryptographic properties of SEAL and UNSEAL by relating them to an ideal implementation that maintains a global table for all values sealed so far and encrypts 0s instead of the actual plaintexts. We assume that the values sealed have a fixed length so that no information leaks trough the length of the seal.

$$
\begin{array}{rl}
\mathrm{SEAL}_i^0 \doteq & enc := \mathcal{SE}(0\ldots0,s.ke);\\
& mac := \mathcal{M}(i|h_i|target|enc,s.ka);\\
& cipher := enc|mac;\\
& log_i := log_i + ((h_i|target|enc),plain)\\
& enc := 0;\ mac := 0\\
\mathrm{UNSEAL}_i^0 \doteq & \text{if } \mathcal{V}_{\mathcal{M}}(i|source|h_i|enc,\ mac,\ s.ka)\\
& \text{then if } exist(log_i,(source|h_i|enc))\\
& \text{then } plain := assoc(log_i,(source|h_i|enc))\\
& \text{else } plain := 0;\\
& enc := 0;\ mac := 0
\end{array}
$$

Security means that, provided s is generated uniformly at random and no other part of the code accesses s or log, no probabilistic polynomial program can distinguish between (SEAL, UNSEAL) and (SEAL^0,UNSEAL^0). This property can be reduced to indistinguishability against chosen plaintext attacks for encryption and resistance against forgery attacks for signing.

Functionality means that UNSEAL is a partial inverse of SEAL: unsealing a value sealed with matching source and target hashes always yields the plain sealed value.

6.4.6 Auxiliary notations

For programming convenience, we define simple macros for calling these commands.

$$
\begin{array}{rl}
\mathrm{EXTEND}_i(e) \doteq & \mathit{identity} := e;\ \mathrm{EXTEND}_i \\
x := \mathrm{SEAL}_i(e_v, e_t) \doteq & \mathit{target} := e_t;\ \mathit{plain} := e_v;\ \mathrm{SEAL}_i; \\
& x := \mathit{cipher};\ \mathit{target} := 0;\ \mathit{plain} := 0;\ \mathit{cipher} := 0 \\
x := \mathrm{UNSEAL}_i(e_c, e_s) \doteq & \mathit{source} := e_s;\ \mathit{cipher} := e_c;\ \mathrm{UNSEAL}_i; \\
& x := \mathit{plain};\ \mathit{source} := 0;\ \mathit{cipher} := 0;\ \mathit{plain} := 0 \\
x := \mathrm{ATTEST}_i(e) \doteq & \mathit{plain} := e;\ \mathrm{ATTEST}_i; \\
& x := \mathit{cipher};\ \mathit{plain} := 0;\ \mathit{cipher} := 0; \\
\mathrm{VERIFY}_i(e_s, e_p) \doteq & \mathit{source} := e_s;\ \mathit{plain} := e_p; \\
& \mathrm{VERIFY}_i(k^-_{TPM},\ \mathit{cipher},\ y)[\mathit{source} := 0;\ \mathit{plain} := 0;\ X]
\end{array}
$$

Example 36 *Continuing with our password example, we define code that seals the secret and the password to its own code identity (using the current value of h_{17} as the target identity) then store the result in a public-untrusted variable. Thus, protected by the TPM hardware keys, the secret and the password can only be retrieved by re-running this code. When re-run, the code behaves as in the password example, retrieving the password and the secret then granting access to the secret only if the password is correct after less than three attempts.*

$\mathit{kernel} := \langle$
  if $c = 0$ then $\mathit{store} := \mathrm{SEAL}(\mathit{pwd}|\mathit{secret},\ h_{17})$
  else { $\mathit{pwd}|\mathit{secret} := \mathrm{UNSEAL}(\mathit{store},\ h_{17})$;
    if $c < 4$ && $\mathit{guess} = \mathit{pwd}$ then $r := \mathit{secret}$};
  INC; $\mathit{pwd} := 0$; $\mathit{secret} := 0$ $\rangle$;
SKINIT;
$A[\widehat{TPM}$,SKINIT]

Assuming that, initially we have $c = 0$ and pwd *is sampled at random, the probability that a polynomial adversary than does not read or write ℓ_{TPM} (an adversary that controls at most the system, but not the TPM hardware) learns anything about* secret *is bounded by the sum of the probabilities of the three most probable passwords plus the (negligible) probabilities that the adversary finds a collision in the hash function or breaks the cryptography used in SEAL and UNSEAL.*

6.4.7 Initialization

In our model, the protected variables of the TPM must be correctly initialized before use. We write TPM_0 for the initialization command. Informally, this command runs once as the TPM is manufactured. It generates cryptographic keys and sets h_1, h_{17}, and c to zero. For cryptographic reasons, we also need to randomly sample $\mathcal{H}$ in a family of universal one-way hash functions; this is modeled as an implicit parameter ν for $\mathcal{H}$. Concretely, the public key of the TPM may also be certified by some authority, so that its high integrity can be dynamically verified.

$$
\begin{array}{rl}
TPM_0 \doteq & k^-_{TPM}, k^+_{TPM} := \mathcal{G}_s(); \\
& s.ke := \mathcal{G}_{\mathcal{SE}}();\ s.ka := \mathcal{G}_{\mathcal{M}}(); \\
& \nu := \mathcal{G}(); \\
& h_1 := 0;\ h_{17} := 0;\ c := 0
\end{array}
$$

6.5 Attested boot sequences

We now illustrate our model of the secure commands with a model of secure boots.

From a security viewpoint, the boot of a computer consists of running a series of increasingly complex and less trusted code, each piece of code loading and launching the next one. At each stage, if a TPM is available, this code can keep track of the boot sequence by relying on integrity measurements (SRTM), as described in Section 6.4, typically by setting or extending PCRs with the hash of each piece of code (the BIOS, the bootloader, the system kernel) just before running it. This code can also check (and attest) the boot sequence, and use it to unseal data.

A limitation of SRTM is that it provides guarantees only on the initial state of the machine, just after booting; if the measured code launches a full-fledged operating system, the PCRs do not say much about the current state of the system. In contrast, or in addition, DRTM can provide more recent and more specific guarantees, as it only identifies a small, isolated part of the system: the code that is run using the late launch functionality.

After explaining our model for memory and parallelism, we define simplified 2-stage boot sequences for machines in a distributed system, and finally relate those distributed systems to the more abstract distributed configurations that model our system in Chapter 4 (model used again in Section 6.6).

6.5.1 Shared memory

In our language, shared variables represent all mutable state, including the local memory but also the TPM secrets and registers, the network, and persistent storage. Although our operational semantics lets any command read and write any variable, we can (formally) restrict their access by using a security policy (Γ) for a lattice whose labels represent these different kinds of variables, with for instance a label representing all variables that can be read or written by software running on a given machine (This is the target architecture of the compiler of Chapter 4).

We let $i = 0..n$ range over machine identifiers, and let $\ell_{i.\,TPM}$ be the label granting unrestricted access to all its local state. We assume that the labels $\ell_{i.\,TPM}$ are pairwise incomparable, so that one machine cannot directly access the local state of another machine. Intuitively, an adversary that can read and write at level $\ell_{i.\,TPM}$ is able to tamper with the hardware of machine i (including its TPM).

Further, we let $\ell_{i.system}$ be the label granting system-level access to local memory on machine i (with weaker integrity and confidentiality than $\ell_{i.\,TPM}$). Intuitively, an adversary that can read and write at level $\ell_{i.system}$ has obtained root privileges on machine i (but still cannot tamper with its TPM state). Similarly, the system can run user-level commands with different, lower privileges, also represented as labels in the policy.

We set up naming conventions for local variables and local commands, writing $i.x$ for a variable x on machine i, and writing $i.Q$ for the command Q where all local variables are prefixed by i ($i.Q \doteq Q\{i.x/x\}$). In addition to local variables, commands running on different machines may access variables representing shared resources; for instance we represent an untrusted network as a public, untrusted variable.

6.5.2 Scheduling

As usual with models that account for concrete cryptography, our commands are sequential, and we simulate concurrent systems by letting the adversary explicitly schedule commands that represent parallel threads of computation. To this end, the commands representing machines, systems, and processes are programmed as co-routines, called by the adversary each time they are scheduled, and completing each time their thread would yield. The resulting low-level model realistically accounts for all interleaving of these threads (provided they yield often enough), and at the same time enables us to specify non-interruptible commands, for instance to reflect that a TPM command runs to completion.

6.5.3 A simple boot sequence

Conservatively, we assume that the adversary controls the scheduling for all machines, and can either let them continue their computation or trigger their reboot. Unless the machine hardware is compromised, the machine command has privileged access to its local state, and can include (in particular) an indexed version of the TPM commands specified in Section 6.4.

The variable $i.system$ has level $\ell_{i.system}$ and, by convention, contains the system loaded when the machine reboots. The adversary can effectively take control of the system if he can write in this variable, but he cannot interfere with its integrity measurement without tampering with the machine itself. We model the initial (hardware) boot on machine i as the command

$$
\begin{array}{l}
i.M \doteq \\
\quad \text{if } i.reset \text{ then} \\
\quad \{\ i.h_1 := 0;\ i.h_{17} := 0;\ (* \text{ clear the PCRs } *) \\
\quad\quad i.temp.* := 0;\ (* \text{ clear the machine volatile memory } *) \\
\quad\quad i.temp.up := i.system;\ (* \text{ endorse the system to boot } *) \\
\quad\quad i.identity := i.temp.up;\ i.\mathrm{EXTEND}_1\} \\
\quad \text{if } i.temp.up \neq 0 \text{ then} \\
\quad \{\ \text{link } i.temp.up\ [i.\widetilde{TPM}, i.\mathrm{SKINIT}]\ \ell^I_{i.system}\ \}
\end{array}
$$

The adversary controls the reset of the machine via the untrusted variable $i.reset$. The other variables introduced above have the level of the machine. All variables with prefix $i.temp.$ represent volatile memory, initialized at 0, and reset to 0 at each reboot; after booting, $i.temp.up$ contains the system command currently running on the machine, linked to its privileged TPM commands.

Next, we define the command of a minimal system initially stored in variable $i.system$ (unless overwritten by the adversary):

$$
\begin{array}{l}
i.SYSTEM \doteq \\
\quad \text{if } i.temp.n = 0 \text{ then } (* \text{ boot } *) \\
\quad \{\ i.temp.n{+}{+};\ i.\mathrm{process}[0] := i.app[0]\ \} \\
\quad \text{else if } i.next \text{ in } 0..(i.temp.n) \text{ then } (* \text{ resume } *) \\
\quad \{\ \text{link } (i.\mathrm{process}[i.next])\ [\\
\qquad i.\widetilde{TPM},\ \{\text{if } test^{\ell^I}_{i.process[i.next]}\ i.kernel \text{ then } i.\mathrm{SKINIT}\}, \\
\qquad i.FORK]\ \ell^I_{i.process[i.next]}\ \} \\
\\
i.FORK \doteq\ i.\mathrm{process}[i.temp.n] := i.app[i.temp.n];\ i.temp.n{+}{+}
\end{array}
$$

When run for the first time, the command loads a first user command. When rerun, the adversary controls which of the loaded command it resumes (with the subroutines to call the TPM and a FORK subroutine to load a new command) by setting the untrusted variable $i.next$. The user-command variables $i.app[i]$ can be given distinct security labels $\ell_{i.process[i.next]}$, to reflect adversaries that have compromised some, but not all of these commands. The other variables introduced above have level $\ell_{i.system}$.

As it links a user command (either to launch it or to resume it), the system provides privileged subcommands for the TPM and for starting new commands ($i.FORK$). The i.SKINIT command enables to run system-level commands, so its usage is restricted by defining a subcommand that first checks that the kernel has at most the privileges of the user command. To this end, it uses an auxiliary function $test$ that type checks the code in its argument, specified by $[\![test_\ell e]\!](\mu) = 1$ if and only if $[\![e]\!](\mu) = \langle P \rangle$ and $\vdash P : \ell$.

6.5.4 Application: a BitLocker-like boot sequence

To illustrate our definitions, we present a variant of our system command $i.SYSTEM$ that relies on the TPM to seal some key (which may be used, for instance, as the master key for disk encryption, as in BitLocker Microsoft [2006], or for keeping user credentials). Our code relies on the mechanisms presented for password protection, starting from Example 14.

The sealed key is stored in a new persistent, but unprotected variable $i.sealed\_key$ (e.g. written on the non-encrypted part of the disk, together with the system code). Assuming the adversary cannot tamper with the TPM, the key is released only if (1) this specific system is loaded (if $i.h_1$ holds a different value, then i.UNSEAL will fail) and (2) the user enters the right password.

$i.SYSTEM \doteq$
  if $i.temp.n = 0$ then (∗ the system is booting ∗)
  { if $i.sealed\_key = 0$ then (∗ BitLocker initialization ∗)
    { $i.temp.k := \mathcal{G}_{\mathcal{SE}}()$; (∗ generate a strong secret ∗)
      $i.temp.pwd := readline()$; (∗ set the password ∗)
      $i.sealed\_key := i.\mathrm{SEAL}_1(i.temp.k|i.temp.pwd|i.c,\ i.h_1)$ }
    else (∗ BitLocker attempts to unseal its key ∗)
    { $i.guess := readline()$; (∗ asking for the password ∗)
      $(i.temp.k|i.temp.pwd|i.c_0) := i.\mathrm{UNSEAL}_1(i.sealed\_key,\ i.h_1)$;
      if ($i.guess = i.pwd$ && $i.c < i.c_0{+}3$)
      then (∗ ok, release the key and reset anti−replay ∗)
        { $i.sealed\_key := i.\mathrm{SEAL}_1(i.temp.k|i.temp.pwd|i.c,i.h_1)$ }
      else (∗ bad password, clear the key before returning ∗)
        { i.INC; $i.temp.k := 0$ }
    $i.identity$:= ''end''; EXTEND_1;
    $i.temp.pwd := 0$;
    $i.temp.n$++; i.process[0] := $i.app$[0] }}
  else if $i.next$ in $0..i.temp.n$ then (∗ the system continues ∗)
  { link (i.process[$i.next$]) [
      $i.\widetilde{TPM}$, {$test^I_{\ell_{i.process[i.next]}}$ $kernel$; SKINIT},
      $i.FORK$] $\ell^I_{i.process[i.next]}$ }

(We leave the user-input command $readline()$ unspecified.) The other variables introduced above also have level $\ell_{i.system}$. The first time it runs (or if its sealed key has been erased), the BitLocker code generates a secret key, asks for a password, and seals them together with the current value of the TPM counter. Implicitly, SEAL also records the current system identity. Otherwise, as the machine reboots, the user is prompted to enter the password; the BitLocker code verifies it, checks the number of failed attempts, and then release the key to the system. In all cases, the password is cleared and h_1 is extended before letting the system run. (The extension changes the code identity of the system, excluding any further successful sealing/unsealing on $i.sealed\_key$ until the machine reboots.)

The probability that an adversary controlling the system (but not the machine) obtains the key is the same as in Example 36 (unless of course the user can be tricked to reveal its password). However, if the system is running and the key loaded, an adversary of level $\ell_{i.system}$ can obviously directly read the key. Concretely, the system should clear the key when it is suspended, with a trade-off between performance and security. For instance, BitLocker may be vulnerable to cold boot attacks, where one exploits the fact that the memory is not reliably erased when rebooting [Halderman et al., 2009].

6.5.5 Bootstrapping abstract distributed configurations

We now consider a model where the adversary schedule a group of machines preloaded with commands, and we compare it to the more abstract model used in Chapter 4 (used again in

Section 6.6) where the adversary directly schedule the commands.

The abstract configurations produced by our compiler consist of an initialization command, Q_0, used to specify initial trust assumptions, plus a series of commands $\widetilde{Q}$ consisting of one command Q_i for each machine involved in the source program. For simplicity, we assume that these commands use at most the SKINIT instruction of their host machine. We are interested in the security properties of commands of the form $Q_0; A[\widetilde{Q}]$ where A is an adversary that controls (at least) the "network" variables used by the commands $\widetilde{Q}$ to communicate with one another, and is also in charge of their scheduling.

The more concrete machine configurations, relying on the system code and boot sequences above, run each Q_i as a user-level command on each machine i. This gives additional power to the adversary, which may also reboot machines and, depending on its level, tamper with their system code.

We express security and functional correctness theorems for our transformation using instances of definitions 16 and 17. We set a sound and correct transformation from the abstract model to the more concrete one. Source adversaries are parametrized by α and range over adversary command contexts for Γ. The commands of the distributed programs are scheduled by the adversary after running Q_0, and can use i.SKINIT on their host machine to run code typed at level ℓ_i. Thus, we define source configurations as commands

$$\begin{array}{rl} S \doteq & Q_0; \\ & A[Q_1[\{\text{if } test_{\ell_1^I} \ kernel \text{ then } 1.\text{SKINIT}\}], \\ & \quad \ldots, \\ & \quad Q_n[\{\text{if } test_{\ell_n^I} \ kernel \text{ then } n.\text{SKINIT}\}], \\ & \quad \{\text{if } test_{\ell_1^I} \ kernel \text{ then } 1.\text{SKINIT}\}, \\ & \quad \ldots, \\ & \quad \{\text{if } test_{\ell_n^I} \ kernel \text{ then } n.\text{SKINIT}\}] \end{array}$$

with $\ell_{i.process[0]} = \bot, I(Q_i)$ and we let $\langle\!\langle \widetilde{Q}, A, \mu \rangle\!\rangle \doteq \rho_\infty \langle S, \mu \rangle$. In this case, we say that two memory distributions are equivalent when they are equal.

Implementation programs and adversary are the same as source programs and adversary. We define implementation configurations using the command

$$\begin{array}{rl} S' \doteq & Q_0; i.temp.up := 0; \\ & i.system := i.SYSTEM; i.app := Q_i; \\ & A'[1.M \ldots n.M] \\ \langle\!\langle \widetilde{Q}, A', \mu \rangle\!\rangle' \doteq & \rho_\infty \langle S', \mu \rangle \end{array}$$

Finally, we let π (the memory projection) be the erasure of all variables added for the machines (the $i.*$): $\pi(\mu') = \mu'_{|dom(\mu)}$.

Our theorem states that, for the same commands ($[\![Q_0, \widetilde{Q}]\!] = Q_0, \widetilde{Q}$), our more concrete model, accounting for machines and reboots, does not enable more attacks than those enabled against source programs. It enables us to focus on the configurations $Q_0, \widetilde{Q}$ for stating the properties of our compilers.

Theorem 18 *For every adversary parameter α, $([\![\cdot]\!], \pi)$ is secure and functionally correct.*

The proof of Theorem 18 is at the end of this chapter, in Subsection 6.8.5.

6.6 Implementing virtual hosts on TPMs

Chapter 4 shows how to compile an imperative program with shared access-controlled memory into a distributed program protected by cryptography. The resulting program runs on a series of

machines, and preserves the security properties of the source program, subject to the assumption that each local command Q_b of the distributed implementation runs on a machine with (at least) the security of its declared level ℓ_b. However, for many useful programs, it is difficult to find such machines for the most trusted parts of the computation.

This section introduces a transformation that relies on secure instructions to boot trusted virtual machines. This transformation applies to any distributed programs, including those produced by CFLOW in Chapter 4. Before giving general definitions and theorems, we illustrate the transformation on Example 22.

Example 37 (Securely booting Q_v) *The command Q_v requires a machine trusted by both the client and the bank. Assume that the bank trusts the client TPM for running Q_v and knows its public key for attestation.*

We may use the code

$$
\begin{array}{ll}
Q_0 \doteq & k_b^-, k_b^+ := \mathcal{G}_e();\ TPM_0;\ c := 0; \\
Q_b \doteq & \textbf{if } c_b{=}1 \textbf{ then} \\
& \{\ c_b{+}{+};\ x_b := e_b; \\
& \quad \textbf{if } \mathrm{VERIFY}_{17}(\mathcal{H}(\langle K_v\rangle | v_{init}, k_v^+, cert_v) \\
& \quad [\ b.k_v^+ := k_v^+; \\
& \qquad x_e := \mathcal{E}(x_b, k_v^+);\ x_s := \mathcal{S}(x_e, k_b^-)\ ]\ \} \\
& \textbf{else if } c_b{=}2 \textbf{ then} \\
& \{\ c_b{+}{+}; \\
& \quad \textbf{if } \mathcal{V}(x'_e, x'_s, k_v^+) \textbf{ then } print(\mathcal{D}(x'_e, k_b^-))\ \} \\
Q_c \doteq & \textbf{if } c_c{=}1 \textbf{ then } \{\ c_c{+}{+};\ y_c := e_c\ \} \\
& \textbf{else if } c_c{=}2 \textbf{ then } \{\ print(y'_c)\ \} \\
Q_v \doteq & kernel := \langle K_v\rangle;\ \mathrm{SKINIT}
\end{array}
$$

$$
\begin{array}{ll}
K_v \doteq & \textbf{if } c{=}0 \textbf{ then} \\
& \{\ \mathrm{INC}; \\
& \quad k_v^-, k_v^+ := \mathcal{G}_e();\ cert_v := \mathrm{ATTEST}_{17}(k_v^+); \\
& \quad key := \mathrm{SEAL}_{17}(k_v^-, h_{17})\ \} \\
& \textbf{else if } c{=}1 \textbf{ then} \\
& \{\ \mathrm{INC}; \\
& \quad k_v^- := \mathrm{UNSEAL}_{17}(key, h_{17}); \\
& \quad \textbf{if } \mathcal{V}(x_e, x_s, k_b^+) \textbf{ then} \\
& \quad \{\ x_v := \mathcal{D}(x_e, k_v^-); \\
& \qquad x'_v, y'_c := f(x_v, y_c); \\
& \qquad x'_e := \mathcal{E}(x'_v, `k_b^+);\ x'_s := \mathcal{S}(x'_e, k_v^-)\ \}\ \}
\end{array}
$$

In this transformed distributed program, the command Q_v now only calls SKINIT on a fixed code K_v. Hence, it does not need to be trusted. The first execution of this command K_v with SKINIT generates a key pair, attest it and store it securely using SEAL. Then, when the code of the bank Q_b runs, it verifies that the key he received (trough and unsecure memory) has been produced by the code K_v run with SKINIT using a TPM. Then, the program proceed as before the transformation, except that each time K_v is called, it starts by retrieving its keypair.

In contrast with the host commands of Section 4, TPM-attested host commands do not have a persistent, protected local memory to keep their trusted key pair. Instead, as we dynamically set up host v, we generate its key pair, we use remote attestation to convince the bank to encrypt its secret towards the implementation of Q_v, and we simulate the persistent local memory using seal/unseal and the TPM counter to protect against replays.

The transformation, for this particular program is secure for all adversary sets $\mathcal{A}_\alpha$, $\mathcal{A}'_\alpha$; and functionally correct for the adversary sets $\mathcal{A}_{\bot,\top}$, $\mathcal{A}'_{\bot,\top}$. A more systematic and generic transformation is presented below.

6.6.1 A general transformation

Our transformation takes as input a policy Γ, a tuple of typed commands $\widetilde{Q}$ for hosts $\widetilde{b}$, including the command of a virtual host v, and a subset of the variables $\widetilde{x}$ of Q_v (informally, $\widetilde{x}$ represents the private, trusted local state of v). It assumes the existence of a TPM on host a, at least as trusted as v and not used in the source program. It generates an implementation policy Γ' and a series of implementation commands Q'_0, $\widetilde{Q}'$, defined below.

We let $\widetilde{Q}$ range over commands such that

3. no command in $\widetilde{Q} \setminus Q_v$ accesses $\widetilde{x}$ (i.e. the variables of $\widetilde{x}$ are local to Q_v);
4. Q_v does not read $\widetilde{x}$ before initializing them;
5. $\vdash Q_b : \ell_b^I$ for $b \in \widetilde{b}$;
6. $\vdash Q_v : \ell_{TPM}^I$ (i.e. the TPM is at least as trusted as Q_v); and
7. $R(C(\ell_{TPM})) \not\leq I(\ell_b)$ for $b \in \widetilde{b}$ (i.e. no host can access the TPM private variables).

Initialization Recall that the commands $\widetilde{Q}$ (including Q_v) rely on trusted variables formally initialized in Q_0. In contrast, our implementation of Q_v uses SKINIT, so its own initialization is deferred until runtime and must be explicitly coded.

For simplicity, we assume that, for each host $b \in \widetilde{b}$, initialization is a command of the form $Q_{0,b} \doteq k_b^-, k_b^+ := \mathcal{G}_e(); \ldots$ that writes private variables and generates a single keypair, with v initialized last. We also assume that the keys written in Q_0 are not overwritten in $\widetilde{Q}$. (These assumptions hold with the CFLOW compiler, up to a reordering of hosts.)

8. $Q_0 \doteq (Q_{0,b};)_{b \neq v}; Q_{0,v};$
9. no command in $\widetilde{Q} \setminus Q_b$ accesses variables written in $Q_{0,b}$ except for k_b^+ for $b \in \widetilde{b}$ (i.e. the variables initialized in $Q_{0,b}$ are local to Q_b);
10. no command in $\widetilde{Q}$ writes k_b^+, k_b^- for $b \in \widetilde{b}$.

Implementation of Q_0 Initialization in the implementation Q'_0 is obtained from Q_0 by adding initialization for the TPM and removing initialization for v:

$Q'_0 \doteq TPM_0; (Q_{0,b}; b.k_v^+ := \bot;)_{b \neq v}$

Implementation of Q_v The command Q'_v uses SKINIT to dynamically launch a secure kernel K_v that implements Q_v:

$Q'_v \doteq kernel := \langle K_v \rangle;\ a.\text{SKINIT}$

$K_v \doteq$

```
a.INC; if a.c = c_v^0+1 then
{ v.c := a.c; Q_{v,0}; cert_v := a.ATTEST_17(k_v^+)}
else
{ k_v^-|k_v^+|v.c|v.x̃ := a.UNSEAL_17(store_v, a.h_17);
  if v.c = a.c then Q_v {v.x/x, x ∈ x̃}{'k_b^+/k_b^+, b ≠ v}};
```

$store_v := a.\mathrm{SEAL}_{17}((k_v^-|k_v^+|v.c{+}1|v.\widetilde{x}),\ a.h_{17}));$
$k_v^-,\ v.c,\ v.\widetilde{x} := 0$

The variables $v.\widetilde{x}$ (and $v.c$) are volatile for each run of Q'_v; they can be public and untrusted, but must be cleared before returning. By eliminating trusted and confidential variables, the transformation lowers the level of Q_v hence the level required to run it.

The identity of K_v is verified by the other hosts, so the command Q'_v itself need not be trusted. The other new variables k_v^+, cert_v, store_v are also (formally) public and untrusted. They need not be trusted for security, although an adversary that can overwrite them may cause the system to fail. The value c_v^0 is a constant of the program and corresponds to the initial value of the TPM monotonic counter. If the participants disagree on this constant, the adversary could run the participants with different instances of K_v. Alternatively, we could let c_v^0 be an untrusted variable, and verify in K_v that everybody has agreed to interact with the same instance.

Implementation of Q_b for $b \neq v$, $b \in \widetilde{b}$ As it runs for the first time, each host command Q'_b verifies the attested public key for v and stores it in a new local variable, $b.k_v^+$, a local trusted copy to be used instead of k_v^+ in Q_b. To this end, Q'_b recomputes the expected value of h, including the values for c_v^0 and any other keys $k_{b'}^+$ used in Q_v, and verifies its concatenation with the received public key k_v^+ using the trusted verification key of the supporting TPM.

$Q'_{b,i} \doteq$
if $b.k_v^+ = 0$
then { $a.\mathrm{VERIFY}_{17}(v.\mathcal{H}(\langle K_v\rangle|v_{init}),k_v^+,cert_v)[b.k_v^+ := k_v^+]$ }
else { $Q_{b,i}\{b.k_v^+/k_v^+\}$ }

Implementation of Γ The formal implementation of Γ is $\Gamma' = \Gamma\{v.\widetilde{x}, v.c, k_v^+, cert_v, store_v \mapsto (\bot,\top)\}\{b.k_v^+ \mapsto \ell_b, b \neq v\}$.

Security and functional correctness We express the security and functional correctness of our transformation as instances of definitions 16 and 17.

Source programs range over commands $\widetilde{Q}$ that meet conditions 3–10 above. Source adversaries are parametrized by α and range over adversary command contexts for Γ. The commands of the distributed programs are scheduled by the adversary after the execution of Q_0; formally:

$$\langle\!\langle \widetilde{Q}, A, \mu\rangle\!\rangle \doteq \rho_\infty\langle Q_0; A[\widetilde{Q}], \mu\rangle$$

Implementation programs $[\![\widetilde{Q}]\!]$ range over $\widetilde{Q}'$, as defined above. Implementation adversaries are parametrized by α and range over valid adversary command contexts for Γ'. The commands of the distributed program are scheduled by the adversary after the execution of Q'_0 but additionally, the adversary and Q_v have access to protected versions of the subroutine SKINIT_a: we let

$$\begin{aligned} S' \doteq\ & Q'_0;\\ & A'[a.\mathrm{SKINIT}^I_\alpha, \widetilde{Q}',\\ & \quad Q'_v[a.\mathrm{SKINIT}_{\ell'^I_v}]] \end{aligned}$$

$$\langle\!\langle \widetilde{Q}', A', \mu'\rangle\!\rangle' \doteq \rho_\infty(\langle S', \mu'\rangle)$$

where $a.\mathrm{SKINIT}_\ell$ runs $a.\mathrm{SKINIT}$ after testing that *kernel* contains code typed at level ℓ.

We let π be the erasure of all variables added in Γ': $\pi(\mu') = \mu'_{|dom(\mu)}$ and we define equivalence on final memory distributions ($\rho_0 \approx \rho_1$) as computational indistinguishability: for all polynomial commands T such that T does not read $\{\widetilde{x}, k_v^-, k_v^+\}$, $|\Pr[\langle T, \rho_0\rangle; g = 0] - \Pr[\langle T, \rho_1\rangle; g = 0]|$ is negligible. Relying on these definitions, our main theorem for virtual hosts on TPMs is

Theorem 19 *Let $\alpha \in \mathcal{L}$ be such that R is robust against α. Let Γ be a robust policy.*

1. *$(\llbracket\cdot\rrbracket, \pi)$ is secure;*
2. *$(\llbracket\cdot\rrbracket, \pi)$ is correct when $\alpha = (\bot_C, \top_I)$.*

The proof of Theorem 19 is at the end of this chapter, in Subsection 6.8.6.

6.7 Experimental results

Our experimental work is based on the prototype compiler of Fournet et al. [2009]. The compiler takes as input a program written in an extension of the source language of Chapter 3 with locality and security annotations. It is parametrized by a module that defines the security lattice (e.g. variants of the DLM of Myers and Liskov [1998]). It applies the translations of Chapter 4 and produces a source F# program. The compiled programs reads its initial information (the values initialized in Q_0) from a trusted configuration file.

We modified the compiler to produce straight, statically-linked C code with minimal libraries. This yields a much smaller TCB than with F#, which depends on the .NET runtime environment and a sophisticated memory manager. This also leads us to use simpler, more compact message formats. We tested it on small distributed examples (see for instance `rpc.dwl` and its compiled code).

We added compiler support for declassification and endorsement, for type checking with the lax type system of Section 3.4, and for enforcing robustness. We tested it on variants of the password example (see for instance `password_endorse.dwl`).

We implemented the virtual-host transformation of Section 6.6, using a trivial software implementation of the TPM instructions. We tested it on variants of the loan-application example (see `tpm_BC.dwl`) and a similar example involving a bank, a client, a merchant, and a virtual host that maintains a 'wallet' storing the current balance of a bank account and running a client-purchase loop for as long as this account remains in credit (see `tpm_MC.dwl`).

The code of the loan-application example is in Figure 6.1

The first line specifies our lattice. This lattice has confidentiality and integrity levels L, b, c, and H, with the relations:

$$L \leq_C b \leq_C H \qquad L \leq_C c \leq_C H \qquad H \leq_I b \leq_I L \qquad H \leq_I c \leq_I L$$

The levels b and c are independents since the bank and the client do not trust each other. The three lines after specify the roles in the program. The trusted role $TPMC$ is the virtual participant that is implemented with the TPM. The four lines after specify the types and levels of the variables of the program. Then, there is the code of the specification. We see the initialization on the bank machine and the client machine, the computation, then the print on the bank machine and client machine.

The architecture of the compiled distributed program is in Figure 6.1. The code of *Bank.c*, *Client.c*, and *Client.c* is the secure code produced following the compilation process specified in Chapter 4 and 6. The code for *sch_Bank.c*, *sch_Client.c*, and *sch_Client.c* is additional untrusted code that schedule the programs and manage the communications trough the network. These schedulers behave as honest adversaries that cooperate to run the program as intended, but the security does not depend on them. Additional libraries are used for cryptography, network communication, and secure hardware calls. *TPMLib.h* in only a trivial software implementation of the secure hardware.

We did not experiment with physical TPMs so far, but McCune et al. [2008, 2009] provides roughly the functionality we need. As a first step, we are targeting virtual machines with minimal

```
SLattice HL;

Role #HH# TPMC;
Role #bb# Bank;
Role #cc# Client;

global string(2) #bb# x;
global string(2) #cc# y;
global string(4) #bb# z;
global string(4) #cc# t;

TPMC:[
  Bank:[
    ‘print_string "\nBank secret: "
    x := ‘read_line ()
  ];
  Client:[
    ‘print_string "\nClient secret: ";
    y := ‘read_line ()
  ];
  z:="";
  t:="";
  ‘concat x 3 y 3 t;
  ‘concat x 3 y 3 z;
  Bank:[
   ‘print_string "\nBank result: ";
   ‘print_string z
  ];
  Client:[
   ‘print_string "\nClient result: ";
   ‘print_string t
  ]
]
```

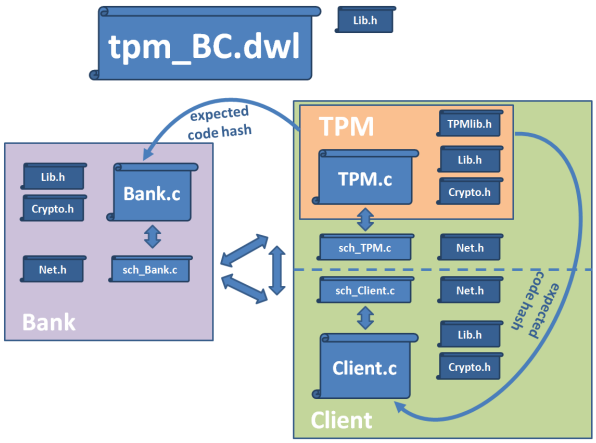


Figure 6.1: Source code and compiled architecture for the loan example

TCBs using virtual TPMs. Their main advantage is to enable direct communication via shared memory between minimal virtual hosts (with almost no libraries) and ordinary hosts (with standard libraries e.g. for networking). Although virtual TPMs are less secure, they can be much faster than physical ones, e.g. for counters and public-key cryptography. We also use a variant of HyperV with support for emancipated partitions (so that the root partition cannot directly access the memory our our virtual machines) and virtual TPM instructions. We produced small *test kernel machines* (TMKs), which enable to ‘boot’ simple stand-alone statically-linked C programs in their own partition, instead of a full-fledged system. The total amount of binary code of our TMKs is around 100K.

6.8 Proofs

6.8.1 Proof of Theorem 14

We first give an updated version of Lemma 28

Lemma 57 (Containment with link) *Let Γ be a policy, $\ell \in \mathcal{L}$ a label, P a command, and μ a memory such that P terminates.*

If $\vdash P : \ell$, then $\rho_\infty(\langle P, \mu\rangle) =^\ell \mu$ and for any ℓ' such that $\ell \not\leq \ell'$, we have $\rho_\infty(\langle P, \mu\rangle) =_{\ell'} \mu$.

Proof: The proof is by induction on the maximum number of reduction steps for $\langle P, \mu\rangle$ to terminate. The base case $P = \surd$ is trivial. The inductive case is by analysis on P. We only consider the case that differs from Lemma 28:

Case P is link $e\,[\widetilde{P'}]\,\ell$. We have $\vdash \widetilde{P'} : (\bot, \top)$ by Rule TLink Strict, and $[\![e]\!](\mu) = \langle P''\rangle$ and $\vdash P'' : \ell$ by Rule Link. Hence $\vdash P''[\widetilde{P'}] : \ell$, and we conclude by induction, and by Rule Link.

□

Proof of Theorem 14 For the case $\mu_0 =^\ell \mu_1$, if $P \vdash \ell$ then we conclude by Lemma 57 that $\rho_\infty(\langle P, \mu_0\rangle) =^\ell \rho_\infty(\langle P, \mu_1\rangle)$. For the case $\mu_0 =_\ell \mu_1$, if there exists ℓ' such that $\vdash P : \ell'$ and $\ell' \not\leq \ell$ then we conclude with Lemma 57 that $\rho_\infty(\langle P, \mu_0\rangle) =_\ell \rho_\infty(\langle P, \mu_1\rangle)$. In all the other cases, the proof is by induction on the number of reduction steps for P to terminate. We suppose that $\vdash P : \ell'$. We only consider the case that differs from Theorem 4:

Case P is link $e\,[\widetilde{P'}]\,\ell'$. By Rule TLink Privileged, $\vdash \widetilde{P'} : (\bot, \top)$ and $\vdash e : \ell'$. For the case $\mu_0 =_\ell \mu_1$, we have $\ell' \leq \ell$, hence, by Rule SubE $\vdash e : \ell$ and $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by Lemma 29. For the case $\mu_0 =^\ell \mu_1$, we have $\ell' \not\leq \ell$, hence $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by Lemma 29. Thus, $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) = \langle P''\rangle$ for some command P'' and

$$\rho_\infty(\langle \text{link}\, e\,[\widetilde{P'}]\,\ell, \mu_0\rangle) = \rho_\infty(\langle P''[P'], \mu_0\rangle)$$
$$\rho_\infty(\langle \text{link}\, e\,[\widetilde{P'}]\,\ell, \mu_1\rangle) = \rho_\infty(\langle P''[P'], \mu_1\rangle)$$

and we conclude by induction on $\langle P''[P'], \mu_0\rangle$ and $\langle P''[P'], \mu_1\rangle$

□

6.8.2 Proof of Theorem 15

We first prove a lemma slightly more general that Theorem 15.

Lemma 58 *Let Γ be a policy, $\ell \in \mathcal{L}$ a label, P a command, $\widetilde{P'}$ a tuple of commands and μ a memory such that $P[\widetilde{P'}]$ terminates.*

If $\vdash P : \ell$ and $\vdash \widetilde{P'} : (\bot_C, I(\ell))$, then $\rho_\infty(\langle P[\widetilde{P'}], \mu\rangle) =^\ell \mu$ and for any ℓ' such that $\ell \not\leq \ell'$, we have $\rho_\infty(\langle P[\widetilde{P'}], \mu\rangle) =_{\ell'} \mu$.

Proof: The proof is by induction on the maximum number of reduction steps for $\langle P[\widetilde{P'}], \mu\rangle$ to terminate and the numbers of processes in $[\widetilde{P'}]$. The base case $P = \surd$ is trivial. The inductive case is by analysis on $P[\widetilde{P'}]$. We only consider the cases that differ from those of Lemma 6:

Case $P[\widetilde{P'}]$ is $X[\widetilde{P'}]$ By Rules TVar and TSubC, $C(\ell) = \bot_C$. X is replaced by one of process in $\widetilde{P'}$, we conclude by induction.

Case $P[\widetilde{P'}]$ is link $e\,[\widetilde{P''}]\,\ell$. We have $\vdash \widetilde{P''} : \ell$, $[\![e]\!](\mu) = \langle P''\rangle$, and $\vdash P'' : \ell$. Hence we conclude by induction, and by Rule Link.

□

Proof of Theorem 15 The proof follows from Lemma 58 □

6.8.3 Proof of Theorem 16

We first prove a lemma slightly stronger that Theorem 16.

Lemma 59 *Let Γ be a policy, $\ell \in \mathcal{L}$ a label, $c \in \mathcal{L}_C$ such that $I(\ell) \not\leq R(c)$. Let P be a command, $\widetilde{P'}$ a tuple of commands and μ_0, μ_1 memories such that $P[\widetilde{P'}]$ terminates.*

If $\vdash P : \ell$, $\vdash \widetilde{P'} : (\bot_C, I(\ell))$ and $\mu_0 =^{(c,H)} \mu_1$, then $\rho_\infty(\langle P[\widetilde{P'}], \mu_0\rangle) =^{(c,H)} \rho_\infty(\langle P[\widetilde{P'}], \mu_1\rangle)$.

Proof: If $\vdash P : (c, H)$, then, we conclude with Theorem 58 that $\rho_\infty(\langle P[\widetilde{P'}], \mu_0\rangle) =^{(c,H)} \rho_\infty(\langle P[\widetilde{P'}], \mu_1\rangle)$. Otherwise, for every ℓ' such that $\vdash P : \ell'$, we have $(c, H) \not\leq \ell'$ and the proof is by induction on the number of reduction steps for P to terminate. The proof is similar to the one of Theorem 7, so we only consider the case that differ:

Case $P[\widetilde{P'}]$ is $X[\widetilde{P'}$ By Rules TVar and TSubC, $C(\ell) = \bot_C$. X is replaced by one of process in $\widetilde{P'}$, we conclude by induction.

Case $P[\widetilde{P'}]$ is link $e\,[\widetilde{P'}]\,\ell'$. By Rule TLink Privileged, $\vdash \widetilde{P'} : \ell'$ and $\vdash e : \ell'$. We have $\mu_0 =^{\ell} \mu_1$, hence $\ell' \not\leq \ell$, hence $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ by Lemma 29. Thus, $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) = \langle P''\rangle$ for some command P'' and

$$\rho_\infty(\langle \mathsf{link}\, e\, [\widetilde{P'}]\, \ell, \mu_0\rangle) = \rho_\infty(\langle P''[P'], \mu_0\rangle)$$
$$\rho_\infty(\langle \mathsf{link}\, e\, [\widetilde{P'}]\, \ell, \mu_1\rangle) = \rho_\infty(\langle P''[P'], \mu_1\rangle)$$

and we conclude by induction on $\langle P''[P'], \mu_0\rangle$ and $\langle P''[P'], \mu_1\rangle$

□

Proof of Theorem 16 The proof follows from Lemma 59 □

6.8.4 Proof of Theorem 17

We first state a lemma for programs that call adversary commands

Lemma 60 *Let Γ be a security policy, $\ell, \alpha \in \mathcal{L}$, c a confidentiality level, R a robustness function robust against α, $\widetilde{A}$ an α-adversary, and P a command such that $\vdash P : \ell$. Let μ, μ_0, and μ_1 be three memories such that $P[\widetilde{A}]$ terminates, and such that $\mu_0 =_{C(\ell), \top_I} \mu_1$.*

If $I(\ell) \leq I(\alpha)$, then $\rho_\infty(\langle P[\widetilde{A}], \mu\rangle) =^{\top_C, I(\ell)} \mu$.

If $I(\ell) \not\leq R(c)$ and $c \not\leq C(\alpha)$, then $\rho_\infty(\langle P[\widetilde{A}], \mu_0\rangle) =^{c, \top_I} \rho_\infty(\langle P[\widetilde{A}], \mu_1\rangle)$.

Proof of Theorem 17 and Lemma 60 The proof is by induction on the maximum number of reduction steps for A (Theorem 17) and $P[\widetilde{A}]$ (Lemma 60) to terminate on memories μ, μ_0 and μ_1. We first prove Lemma 60. The base case $P = \surd$ is trivial. The inductive case is by analysis on $P[\widetilde{A}]$. The proof are similar to those of Lemma 58 and 59, so we only consider the case that differ:

Case $A[\widetilde{A}]$ **is** $X[\widetilde{A}]$ X is replaced by one of the process in A, we conclude by induction on Theorem 17.

We now prove Theorem 17 The base case $A = \surd$ is trivial. The inductive case is by analysis on A.

Case A **is is link** $e\,[\widetilde{A'}]\,\ell$**.** By Rule TLink Privileged, SubE, and definition of α-adversaries, we have $\widetilde{A''}$ are α-adversaries and $I(\alpha) \leq \ell$. For the first point, we have $[\![e]\!](\mu) = \langle P \rangle$ for some P such that $\vdash P : \alpha$ by Rule Link. We conclude by induction on Lemma 60. For the second point, by definition of α-adversaries, $v(e) \subseteq \{C(x) \mid \Gamma(x) \leq C(\alpha)\}$, hence, $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$ Thus, in both cases, $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) = \langle P \rangle$ for some command P and

$$\rho_\infty(\langle \text{link}\, e\, [\widetilde{A'}]\, \ell, \mu_0 \rangle) = \rho_\infty(\langle P''[A'], \mu_0 \rangle)$$
$$\rho_\infty(\langle \text{link}\, e\, [\widetilde{A'}]\, \ell, \mu_1 \rangle) = \rho_\infty(\langle P''[A'], \mu_1 \rangle)$$

with $\vdash P : \alpha$ by Rule Link and we conclude by induction on $\langle P[\widetilde{A'}], \mu_0 \rangle$ and $\langle P[\widetilde{A'}], \mu_1 \rangle$ with Lemma 60.

Case A **is** $x := e$**.** By definition of α-adversaries, $I(\alpha) \leq I(x)$. We conclude for the first point. By definition of α-adversaries, $v(e) \subseteq \{C(x) \mid \Gamma(x) \leq C(\alpha)\}$, hence, $[\![e]\!](\mu_0) = [\![e]\!](\mu_1)$, we conclude for the second point.

Case A **is** $\widetilde{x} := f(\widetilde{e})$ The conclusion follows as for the case above.

Case A **is if** e **then** A_1 **else** A_2**.** We have $\langle \text{if } e \text{ then } A_1 \text{ else } A_2, \mu \rangle \rightsquigarrow_1 \langle A_b, \mu \rangle$ for some $b \in \{0, 1\}$. We conclude by induction for the first case. By definition of α-adversaries, $v(e) \subseteq \{C(x) \mid \Gamma(x) \leq C(\alpha)\}$, hence, $[\![e]\!](\mu_0) = [\![e]\!](\mu_1) = b$ for some $b \in \{0, 1\}$ and

$$\rho_\infty(\langle \text{if } e \text{ then } A_1 \text{ else } A_2, \mu_0 \rangle) = \rho_\infty(\langle A_b, \mu_0 \rangle)$$
$$\rho_\infty(\langle \text{if } e \text{ then } A_1 \text{ else } A_2, \mu_1 \rangle) = \rho_\infty(\langle A_b, \mu_1 \rangle)$$

We conclude by induction on $\langle A_b, \mu_0 \rangle$ and $\langle A_b, \mu_1 \rangle$.

Case A **is while** e **do** A'**.** The conclusion follows as for the case above.

Case A **is** $A_0; A_1$ We conclude with Lemma TSeq.

□

6.8.5 Proof of Theorem 18

Proof: The theorem states that, for every implementation adversary, there exists an equivalent source adversary. In this proof, we construct a source adversary and build a simulation mapping each state of the implementation execution to a state of the source execution. The source adversary is simulating the machines and their operating system; these commands are detailed here as slight variants of $i.SYSTEM$ and $i.M$. The new variables are read and written by the adversary, hence are at level α (not necessarily the same level as in the implementation program).

$i.advSYSTEM[i.\text{SKINIT}, Q\_i] \doteq$
  if $i.temp.n = 0$ then (∗ boot ∗)
  { $i.temp.n$++; $i.\text{process}[0] := i.app[0]$ }
  else if $i.next$ in $0..(i.temp.n)$ then
  { if $I(\alpha) \leq I(\ell^I_{i.process[i.next]})$
    link $(i.\text{process}[i.next])$ [
      { if $test_{\ell^I_{i.process[i.next]}}$ $i.kernel$ then $i.\text{SKINIT}$ },
      $i.FORK$] $\ell^I_{i.process[i.next]}$

else
{ if $i.next = 0$ then Q_i
else link 0 α }}

$i.FORK \doteq i.\text{process}[i.temp.n] := i.app[i.temp.n]; i.temp.n{+}{+}$

$i.advM[i.\text{SKINIT}, Q\_i] \doteq$
if $i.reset$ then
{ $i.temp.* := 0$;
$i.temp.up := i.system$ }
if $i.temp.up \neq 0$ then
{ if $I(\alpha) \leq I(\ell^I_{i.system})$ then
link $i.temp.up$ [i.SKINIT] $\ell^I_{i.system}$
else $i.advSYSTEM[i.\text{SKINIT}, Q\_i]$}

In the proof, we will use abbreviations

$$\begin{aligned} W[A'] \doteq \quad & A'[1.advM[1.\text{SKINIT}, Q_1] \\ & \quad \ldots\ n.advM[n.\text{SKINIT}, Q_n]] \\ W'[A'] \doteq \quad & A'[1.M \ldots n.M] \end{aligned}$$

1. **Initialization, from Q_0 to A'.**

   We map the target configuration $\langle S', \mu' \rangle$ to the source configuration $\langle P, \mu' \rangle$, where

   $$\begin{aligned} P \doteq \quad & Q_0; i.temp.up := 0; \\ & i.system := i.SYSTEM; i.app := Q_i; \\ & W[A'] \end{aligned}$$

   These commands perform matching steps (keeping equal memories) until they reach the commands $W'[A']$ and $W'[A']$. Then:

2. **Running an adversary.**

   We consider $\langle W'[A'], \mu' \rangle$ (for all valid A', not only the initial one) when μ' is such respect the property $Inv(\mu')$ defined by as follows:

   - $I(\alpha) \not\leq I(\ell^I_{i.system}) \Rightarrow i.system = i.SYSTEM$
   - $I(\alpha) \not\leq I(\ell^I_{i.process[0]}) \Rightarrow (i.app[0] = Q_i \& (i.temp.n \neq 0 \Rightarrow (i.process[0] = Q_i)))$
   - $I(\alpha) \not\leq I(\ell^I_{i.process[i.next]}) \Rightarrow (i.process[i.next] = 0)$

   We map it to $\langle W[A'], \mu' \rangle$. We leave this case when the adversary runs a machine, reaching $i.M; W'[A']$ and $i.advM[i.SKINIT, Q_i]; W[A']$, respectively. Then:

3. **Running a machine $i.M$.**

   We consider $\langle i.M; W'[A'], \mu' \rangle$ when μ' respect the property $Inv(\mu')$. We map it to $\langle i.advM; W[A'], \mu' \rangle$. The invariant implies that these commands perform matching steps, keeping equal memories and $Inv(\mu')$. We leave this case either by resuming case 2 (as both host commands terminate), or if $I(\alpha) \leq I(\ell^I_{i.system})$, $\mu'(i.reset) = 0$ and $\mu'(i.temp.up) \neq 0$ reaching $i.SYSTEM; W'[A']$ and $i.advSYSTEM[i.SKINIT, Q_i]; W[A']$, respectively. Then:

4. **Running a system $i.SYSTEM$.**

   We consider $\langle i.SYSTEM; W'[A'], \mu' \rangle$ when μ' respect the property $Inv(\mu')$. We map it to $\langle i.advSYSTEM; W[A'], \mu' \rangle$. The invariant implies that these commands perform matching

steps, keeping equal memories and $Inv(\mu')$. We leave this case either by resuming case 2 (as both host commands terminate), or if $I(\alpha) \leq I(\ell^I_{i.process[0]})$ and $\mu'(i.next) = 0$ by reaching $Q_i; W'[A']$ and $Q_i; W[A']$, respectively. Then:

5. **Running Q_i.**

   We consider $\langle Q_i; W'[A'], \mu' \rangle$ when μ' respect property $Inv(\mu')$. We map it to $\langle Q_i; W[A'], \mu' \rangle$. The invariant implies that these commands perform matching steps, keeping equal memories and $Inv(\mu')$. We leave this case by resuming case 2 (as both host commands terminate).

□

6.8.6 Proof of Theorem 19

To prove the theorem, before constructing a source adversary, we rewrite our target configurations to eliminate the cryptography. The lemma below applies our cryptographic hypotheses to our target configurations. After this lemma, we only consider execution that have no hash collisions, since they probability is negligible. The next lemma anticipates the initialization of Q_v (to perform random sampling at the same time as in the source configuration).

Lemma 61 *Let log_H be a free variable. We consider*

$$
\begin{array}{rl}
EXTEND^0_{17} \doteq & \mathrm{h}_{17} := \mathcal{H}(\mathrm{h}_{17}|\mathrm{identity}); \quad \doteq \\
& \log_H := \log_H + (\mathrm{h}_{17},\mathrm{h}_{17}|\mathrm{identity})
\end{array}
$$

$$
\begin{array}{rl}
SKINIT^0 \doteq & \mathrm{identity} := kernel; \\
& EXTEND_{17}; \\
& \mathit{link}\ \mathrm{kernel}[INC, EXTEND_{17}, ATTEST^0_{17}, \\
& \qquad SEAL^0_{17}, UNSEAL^0_{17}]\ \ell^I_{system}; \\
& \mathrm{h}_{17} := 0
\end{array}
$$

$$
\begin{array}{rl}
\mathrm{S}'^0 \doteq & \log_H := \emptyset; \\
& \mathrm{Q}'_0; \\
& A'[\{\mathrm{test}_{\alpha^I}\ \mathrm{k};\ \mathrm{a}.SKINIT^0\}, \widetilde{Q}', \\
& \quad \mathrm{Q}'_v[\mathrm{test}_{\ell'^I_v}\ \mathrm{kernel};\ \mathrm{a}.SKINIT^0]]
\end{array}
$$

For all implementation memory μ', we have

$$\rho_\infty(\langle \mathrm{S}', \mu' \rangle) \approx \rho_\infty(\langle \mathrm{S}'^0, \mu' \rangle)$$

Proof: We successively apply our security hypotheses for SEAL/UNSEAL, and ATTEST. □

Lemma 62 *Consider a variant of K_v that reads the result of initialization instead of running it, defined by*

$$
\begin{array}{l}
K'_v \doteq \\
\quad \mathrm{a}.INC;\ \mathit{if}\ \mathrm{a.c} = \mathrm{c}^0_v + 1\ \mathit{then} \\
\quad \{\ \mathrm{v.c} := \mathrm{a.c}; \\
\qquad (\mathrm{x} := \mathrm{copy.x};)_{x \in wv(Q_{v,0})} \\
\qquad \mathrm{cert}_v := \mathrm{a}.ATTEST^0_{17}(k^+_v)\} \\
\quad \mathit{else} \\
\quad \{\ k^-_v|k^+_v|\mathrm{v.c}|v.\widetilde{x} := \mathrm{a}.UNSEAL^0_{17}(\mathrm{store}_v,\ \mathrm{a.h}_{17}); \\
\qquad \mathit{if}\ \mathrm{v.c} = \mathrm{a.c}\ \mathit{then}\ \mathrm{Q}_v\ \{v.x/x, x \in \widetilde{x}\}\{`k^+_b/k^+_b, b \neq v\}\}; \\
\quad \mathrm{store}_v := \mathrm{a}.SEAL^0_{17}(k^-_v|k^+_v|v.c+1|v.\widetilde{x},\ \mathrm{a.h}); \\
\quad k^-_v,\ \mathrm{v.c},\ v.\widetilde{x} := 0
\end{array}
$$

and let S'' be a variant of S'^0 that runs initialization for all hosts (using Q_0 as in S',instead of Q'_0, defined by

$$\begin{aligned} \mathrm{S}'' \doteq\ & \mathrm{Q}_0; \\ & \mathrm{a.TPM}_0 \\ & (b.k_v^+ := 0;)_{b\neq v} \\ & (\mathrm{copy}_x := \mathrm{x};)_{x\in rv(Q_{v,0})\setminus k_v^+} \\ & A'[\{\mathrm{test}_{\alpha^I}\ \mathrm{k};\ \mathrm{a}.\mathit{SKINIT}^0\},\widetilde{Q}', \\ & \quad \mathrm{Q}'_v[\mathrm{test}_{\ell'^I_v}\ \mathrm{kernel};\ \mathrm{a}.\mathit{SKINIT}^0]] \end{aligned}$$

For all implementation memory μ', we have

$$\rho_\infty(\langle S'^0, \mu'\rangle)_{|dom(\mu')\setminus\{copy.*\}} = \rho_\infty(\langle \mathrm{S}'', \mu'\rangle)_{|dom(\mu')\setminus\{copy.*\}}$$

Proof of Theorem 19 for Security In the proof, we will use abbreviations

$$\begin{aligned} W[A'] \doteq\ & A'[\{test_{\alpha^I}\ k;\ adv.\mathrm{SKINIT}^0\},\widetilde{Q}'', \\ & Q''_v[\{test_{\ell'^I_v}\ kernel;\ adv.\mathrm{SKINIT}^0\}]] \\ W'[A'] \doteq\ & A'[\{test_{\alpha^I}\ k;\ a.\mathrm{SKINIT}^0\},\widetilde{Q}', \\ & \quad Q'_v[test_{\ell'^I_v}\ kernel;\ a.\mathrm{SKINIT}^0]] \\ Q''_v \doteq\ & kernel := \langle K''_v\rangle;\ adv.\mathrm{SKINIT}^0 \\ K''_v \doteq\ & adv.\mathrm{INC};\ \mathrm{if}\ adv.c = c_v^0{+}1\ \mathrm{then} \\ & \{\ v.c := adv.c; \\ & \quad (x := copy.x;)_{x\in wv(Q_{v,0})} \\ & \quad current\_k_v^+ := copy.k_v^+; \\ & \quad cert_v := adv.\mathrm{ATTEST}_{17}^0(current\_k_v^+)\} \\ & \mathrm{else} \\ & \{\ v.c := adv.\mathrm{UNSEAL}_{17}^0(store_v,\ adv.h_{17}); \\ & \quad \mathrm{if}\ v.c = adv.c\ \mathrm{then}\ Q_v\ \}; \\ & store_v := adv.\mathrm{SEAL}_{17}^0((v.c{+}1),\ adv.h_{17})); \\ & k_v^-,\ v.c,\ v.\widetilde{x} := 0 \\ Q''_b \doteq\ & \mathrm{if}\ b.k_v^+ = 0 \\ & \mathrm{then}\ \{\ adv.\mathrm{VERIFY}_{17}^0[b.k_v^+ := current\_k_v^+] \\ & \qquad (v.\mathcal{H}(\langle K'_v\rangle|v_{init}),current\_k_v^+,cert_v)\} \\ & \mathrm{else} \begin{pmatrix} Q_{b,i} & when\ I(\alpha) \not\leq I(\ell_b) \\ Q_{b,i}\{b.k_v^+/k_v^+\} & when\ I(\alpha) \leq I(\ell_b) \end{pmatrix} \end{aligned}$$

The theorem states that for every implementation adversary, there exists an equivalent source adversary. In this proof, we first apply Lemmas 61 and 62, then we construct a source adversary and build a simulation mapping each state of the implementation execution to a state of the source execution. The proof is by case analysis on the state:

1. **Initialization, from Q_0 to A'.**

   We map the target configuration $\langle S'', \mu'\rangle$ (with S'' defined in Lemma 62) to the source configuration $\langle P, \mu\rangle$, where

$$\begin{aligned} P \doteq\ & Q_0; \\ & adv.TPM_0; \\ & (b.k_v^+ := 0;)_{b\neq v}; \\ & (copy_x := x;)_{x\in rv(Q_{v,0})\setminus k_v^+} \\ & current\_k_v^+ := k_v^+; \\ & W[A'\{k_v^+/current\_k_v^+\}] \end{aligned}$$

$$\mu \doteq \mu'_{|dom(\mu)}$$

These commands perform matching steps until reaching $W'[A']$ and $W[A]$, respectively, with memories μ' and μ such that

$$\mu_{|dom(\mu)\setminus current\_k_v^+} = \mu'_{|dom(\mu)}$$
$$\mu(current\_k_v^+) = \mu'(k_v^+)$$

Then:

2. **Running an adversary $W'[A']$.**

We consider$\langle W'[A'], \mu'\rangle$ (for all valid A', not only the initial one) when μ' respect the property $Inv(\mu')$ defined by as follows:

- for each $b \in \widetilde{b}$ such that $I(\alpha) \not\leq I(\ell_b)$, $\mu'(b.k_v^+) = \mu'(copy.k_v^+)$ or $\mu'(b.k_v^+) = 0$
- there is at most one val such that

  $exist(\mu'(log_{seal}),17|(\mathcal{H}(K_v\text{''})|\mathcal{H}(K_v\text{''})|((a.c+1)|\text{val})))$
- there is no val_c, val such that $\mu(a.c) + 1 < val_c$ and

  $$exist(\mu'(log_{seal}), 17|\mathcal{H}(\langle K_v''\rangle|v_{init})|\mathcal{H}(\langle K_v''\rangle|v_{init})|(val_c|val))$$
- for each val such that

  $$exist(\mu'(log_{attest}), (17|\mathcal{H}(K_V|val)))$$

  we have $val = k_v^+$.

We map it to $\langle P, \mu\rangle$, where

$$P \doteq W[A'\{k_v^+/current\_k_v^+\}]$$

and μ is defined by:

- $\mu(z) = \mu'(z)$ when $z \notin \{\widetilde{x}, current\_k_v^+, k_v^+, k_v^-, log_{seal}, log_{attest}\}$
- $\mu(current\_k_v^+) = \mu'(k_v^+)$
- $\mu(\widetilde{x}) = \{val/exist(\mu'(log_{seal}),$
  $17|(\mathcal{H}(\langle K_v''\rangle|v_{init})|\mathcal{H}(\langle K_v''\rangle|v_{init})|\mu'(copy.k_v^-|copy.k_v^+|a.c+1)|val))\}$ or there is no val such that $exist(\mu'(log_{seal}),$
  $17|(\mathcal{H}(\langle K_v''\rangle|v_{init})|\mathcal{H}(\langle K_v''\rangle|v_{init})|\mu'(copy.k_v^-|copy.k_v^+|a.c+1)|val))$
- $\mu(log_{attest}) = \mu'(log_{attest})\{17|\mathcal{H}(K_V''|val)/$
  $17|\mathcal{H}(K_V|val)\}$
- $\mu(log_{seal}) = \mu'(log_{seal})\{(17|\mathcal{H}(\langle K_v''\rangle|v_{init})|\mathcal{H}(\langle K_v''\rangle|v_{init})|(val_c))/$
  $(17|\mathcal{H}(K_v')|\mathcal{H}(K_v')|(val_c|val))\}$

We let g be the function such that $g(\mu') = \mu$. We leave this case when the adversary runs a host command, reaching Q_i' and Q_i'', respectively. Then:

3. **Running a host command Q_i'.**

We consider $\langle Q_i'; W'[A'], \mu'\rangle$ when μ' respect the property $Inv(\mu')$. We map it to $\langle Q_i''; W[A'\{k_v^+/current\_k_v^+\}], g(\mu')\rangle$. These commands perform matching steps, keeping $\mu = g(\mu')$ and $Inv(\mu')$. We leave this case either by resuming case 2 (as both host commands terminate), or, if we were running Q_v', by booting the secure kernel, as described next:

4. **Booting K_v.**

We consider $\langle a.SKINIT^0; W'[A'], \mu'\rangle$ when μ' respect the property $Inv(\mu')$, where $\mu'(kernel) = \langle K_v'\rangle$. We map it to $\langle adv.SKINIT^0; W[A'\{k_v^+/current\_k_v^+\}], \mu = g(\mu')\{kernel \mapsto \langle K_v''\rangle\}\rangle$.

Then:

5. **Running K_v.**

We consider $\langle K'_v; W'[A'], \mu' \rangle$ when μ' respect the property $Inv(\mu')$, and $\mu'(kernel) = \langle K'_v \rangle$. We map it to

$$\langle K''_v; W[A'\{k_v^+/current\_k_v^+\}], g(\mu')\{adv.kernel \mapsto \langle K''_v \rangle\} \rangle$$

These commands perform matching steps, keeping $\mu = g(\mu')\{adv.kernel \mapsto \langle K''_v \rangle\}$ and $Inv(\mu'\{a.c \mapsto \mu'(a.c) - 1\})$.

We leave this case either by resuming case 2 (if $\mu'(adv.c = c_v^0 + 1)$, as both commands terminate), or, after the unsealing of local variables, as described next:

6. **Starting Q_v (within K_v).**

We consider $\langle P', \mu' \rangle$ when μ' respects the property

$$Inv(\mu'\{a.c \mapsto \mu'(a.c) - 1\})$$

with either $\mu'(v.\widetilde{x}) = \{val/exist(\mu'(log_{seal}), (17|\mathcal{H}(\langle K''_v \rangle|v_{init})|\mathcal{H}(\langle K''_v \rangle|v_{init})|\mu'(copy.k_v^-)|\mu'(copy.k_v^+)|\mu'(a.c)|\mu'(val))))\}$ or $v.c \neq a.c$, and when P' is of the form:

if $v.c = a.c$ then Q_v $\{v.x/x, x \in \widetilde{x}\}\{`k_b^+/k_b^+, b \neq v\}$;
  $store_v := a.\text{SEAL}_{17}^0(k_v^-|k_v^+|v.c+1|v.\widetilde{x},\ a.h_{17})$;
$k_v^-,\ v.c,\ v.\widetilde{x} := 0$
W'$[A$'$]$

where $\mu'(adv.kernel) = \langle K'_v \rangle$.

We map it to $\langle P, \mu \rangle$, where

$$\begin{aligned} P \doteq\ & \text{if } v.c = adv.c \text{ then } Q_v; \\ & store_v := adv.\text{SEAL}_{17}^0(v.c{+}1,\ adv.h_{17})); \\ & k_v^-,\ v.c,\ v.\widetilde{x} := 0; \\ & W[A'\{k_v^+/current\_k_v^+\}] \end{aligned}$$

and $\mu = g(\mu')\{adv.kernel \mapsto \langle K''_v \rangle\}$.

We leave this case either jumping to case 8 (if $\mu'(v.c \neq adv.c)$, as both commands terminate), or, at the next instruction:

7. **Running $\mathbf{Q}_v$ (within K_v).**

We consider $\langle P', \mu' \rangle$ when μ' respect the property $Inv(\mu'\{a.c \mapsto \mu'(a.c) - 1\})$ and P' is of the form:

Q_v $\{v.x/x, x \in \widetilde{x}\}\{`k_b^+/k_b^+, b \neq v\}$;
$store_v := a.\text{SEAL}_{17}^0(k_v^-|k_v^+|v.c+1|v.\widetilde{x},a.h_{17})$;
$k_v^-,\ v.c,\ v.\widetilde{x} := 0$;
W'$[A$'$]$

where $\mu'(adv.kernel) = \langle K'_v \rangle$. We map it to $\langle P, \mu \rangle$, where

$$\begin{aligned} P \doteq\ & Q_v; \\ & store_v := adv.\text{SEAL}_{17}^0(v.c{+}1,\ adv.h_{17})); \\ & k_v^-,\ v.c,\ v.\widetilde{x} := 0; \\ & W[A'\{k_v^+/current\_k_v^+\}] \end{aligned}$$

and $\mu = g(\mu')\{adv.kernel \mapsto \langle K''_v \rangle, \widetilde{x} \mapsto \mu'(v.\widetilde{x})\}$. These commands perform matching steps, keeping $\mu = g(\mu')\{adv.kernel \mapsto \langle K''_v \rangle, \widetilde{x} \mapsto \mu'(v.\widetilde{x}), (k_v^-, k_v^+) \mapsto \mu'(k_v^-, k_v^+)\}$ and $Inv(\mu')$ until reaching the next case:

8. **Closing K_v.**

   We consider $\langle k_v^-, v.c, v.\widetilde{x} := 0; W'[A'], \mu' \rangle$ when μ' respect the property $Inv(\mu')$,and $\mu'(kernel) = \langle K'_v \rangle$. We map it to $\langle P, \mu \rangle$, where

$$P \doteq \quad k_v^-,\ v.c,\ v.\widetilde{x} := 0;\\ W[A'\{k_v^+ / current\_k_v^+\}]$$

   and $\mu = g(\mu')$. We then resume case 2 with matching commands.

The simulation terminates in a case 2, in which the invariant implies that $\pi(\mu') = \mu$. Composing with Lemma 61 and Lemma 62, we obtain that $\rho_\infty(\langle S', \mu' \rangle) \approx \rho_\infty(\langle Q_0; A[Q_1, \ldots, Q_n], \pi\mu' \rangle)$ with

$$A \doteq \quad adv.TPM_0;\\ (b.k_v^+ := 0;)_{b \neq v};\\ (copy_x := x;)_{x \in rv(Q_{v,0}) \setminus k_v^+}\\ current\_k_v^+ := k_v^+;\\ W[A'\{k_v^+ / current\_k_v^+\}]$$

hence, the program transformation is secure for all adversary parameter α. □

Bibliography

AMD. AMD64 virtualization: Secure virtual machine architecture reference manual. *AMD Publication*, 33047, 2005.

A. Askarov and A. Myers. A semantic framework for declassification and endorsement. *Programming Languages and Systems*, pages 64–84, 2010.

A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*, pages 43–59. IEEE, 2009.

S. Balfe and K. G. Paterson. e-EMV: Emulating EMV for internet payments using trusted computing technology. *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing (STC 2008)*, pages 81–92, 2008.

G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *ACM SIGPLAN Notices*, volume 44, pages 90–101. ACM, 2009.

M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *CRYPTO'98*, volume 1462 of *LNCS*, pages 26–45. Springer-Verlag, Aug. 1998.

L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. *CONCUR 2008-Concurrency Theory*, pages 418–433, 2008.

K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Cryptographically verified implementations for tls. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 459–468. ACM, 2008.

K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *22nd IEEE Computer Security Foundations Symposium (CSF'09)*, July 2009.

M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In *EUROCRYPT*, pages 127–144, 1998.

L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. *CONCUR 2010-Concurrency Theory*, pages 162–176, 2011.

D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Theory of Cryptography (TCC)*, number 3378 in LNCS, pages 325–341. Springer, Feb. 2005.

G. Boudol and I. Castellani. Flow models of distributed computations: three equivalent semantics for CCS. *Information and Computation*, 114(2):247–314, 1994.

S. Capecchi, E. Giachino, N. Yoshida, K. Lodaya, and M. Mahajan. Global escape in multiparty sessions. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, volume 8, pages 338–351. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

S. Chong and A. Myers. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop, 2006*, page 12, 2006.

S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. *Communications of the ACM*, 52(2):79–87, 2009.

R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. Leifer. Secure implementations for typed session abstractions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 170–186. IEEE, July 2007. URL http://www.msr-inria.inria.fr/projects/sec/sessions/.

G. Danezis and C. Diaz. Space-efficient private search with applications to rateless codes. In *Financial cryptography and data security: 11th international conference, FC 2007, and 1st International Workshop on Usable Security, USEC 2007*, 2007.

P. Deniélou and N. Yoshida. Buffered communication analysis in distributed multiparty sessions. *CONCUR 2010-Concurrency Theory*, pages 343–357, 2011a.

P. Deniélou and N. Yoshida. Dynamic multirole session types. *ACM SIGPLAN Notices*, 46(1): 435–446, 2011b.

D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

M. Dezani-Ciancaglini and U. de'Liguoro. Sessions and session types: An overview. *Web Services and Formal Methods*, pages 1–28, 2010.

D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, pages 10–18, 1984.

C. Fournet and J. Planul. Compiling Information-Flow Security to Minimal Trusted Computing Bases. In *20th European Symposium on Programming (ESOP'11)*, pages 216–235. Springer, Mar. 2011. ISBN 978-3-642-19717-8.

C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 323–335, Jan. 2008.

C. Fournet, G. le Guernic, and T. Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *ACM Conference on Computer and Communications Security*, pages 432–441, Nov. 2009.

C. Fournet, J. Planul, and T. Rezk. Information-flow types for homomorphic encryptions. In *ACM Conference on Computer and Communications Security*, Oct. 2011.

S. J. Gay and M. Hole. Types and subtypes for client-server interactions. In *Programming Languages and Systems, 8th European Symposium on Programming (ESOP)*, pages 74–90, 1999.

C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM symposium on Theory of computing (STOC)*, pages 169–178, 2009.

J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

S. Goldwasser and S. Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *ACM symposium on Theory of computing (STOC)*, pages 365–377, 1982.

S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attack. *SIAM Journal on Computing*, 17(2):281–308, 1988.

D. Grawrock. TCG Specification Architecture Overview, Revision 1.4, 2007.

J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *17th ACM Conference on Computer and Communications Security*, pages 451–462, Oct. 2010.

K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *Programming Languages and Systems, 7th European Symposium on Programming (ESOP)*, volume 1381, pages 22–138. Springer, 1998.

K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 273–284. ACM, 2008.

Intel. Intel Trusted Execution Technology Software Development Guide, 2009. http://www.intel.com/technology/security/.

J. Katz and L. Malka. Secure text processing with applications to private DNA matching. In *17th ACM Conference on Computer and Communications Security*, pages 485–492, Oct. 2010.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

P. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328. ACM, 2008.

J. McCune, N. Qu, Y. Li, A. Datta, V. Gligor, and A. Perrig. Efficient TCB Reduction and Attestation. *CMU-CyLab-09-003, CyLab Carnegie Mellon University*, 9, 2009.

Microsoft. Windows BitLocker drive encryption, 2006. http://technet.microsoft.com/en-us/library/cc766200.aspx.

R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. *Programming Languages and Systems*, pages 316–332, 2009.

A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *19th IEEE Symposium on Research in Security and Privacy (RSP)*, Oakland, California, May 1998.

L. Nielsen, N. Yoshida, and K. Honda. Multiparty symmetric sum types. *Arxiv preprint arXiv:1011.6436*, 2010.

R. Ostrovsky and W. E. Skeith III. Private searching on streaming data. In V. Shoup, editor, *Advances in Cryptology–CRYPTO 2005*, volume 3621 of *LNCS*, pages 223–240, 2005.

P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

O. Pernet, N. Ng, R. Hu, N. Yoshida, and Y. Kryftis. Safe parallel programming with session java. Technical report, Technical Report 14, Department of Computing, Imperial College London, 2010.

J. Planul, R. Corin, and C. Fournet. Secure enforcement for global process specifications. In *CONCUR 2009: Proceedings of the 20th International Conference on Concurrency Theory*, pages 511–526, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04080-1.

B. Preneel. Analysis and design of cryptographic hash functions. *Doct Dissertation KULeuven*, 1993.

C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *CRYPTO'91*, volume 576 of *LNCS*, pages 433–444. Springer-Verlag, 1991.

A. Sabelfeld and A. Myers. A model for delimited information release. *Software Security-Theories and Systems*, pages 174–191, 2004.

A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 255–269. IEEE, 2005.

K. Sivaramakrishnan, K. Nagaraj, L. Ziarek, and P. Eugster. Efficient session type guided distributed interaction. In *Coordination Models and Languages*, pages 152–167. Springer, 2010.

N. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. *Public Key Cryptography–PKC 2010*, pages 420–443, 2010.

G. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ACM SIGPLAN Notices*, volume 39, pages 85–96. ACM, 2004.

K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. *PARLE'94 Parallel Architectures and Languages Europe*, pages 398–413, 1994.

TCG. Client Specific TPM Interface Specification (TIS), Version 1.2. Trusted Computing Group, 2005.

P. TCG. TCG Software Stack (TSS), Version 1.2. Trusted Computing Group, 2006.

D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 156–168. IEEE, 1997.

D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2/3):167–188, 1996.

N. Yoshida, V. Vasconcelos, H. Paulino, and K. Honda. Session-based compilation framework for multicore programming. In *Formal Methods for Components and Objects*, pages 226–246. Springer, 2009.

N. Yoshida, P. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. *Foundations of Software Science and Computational Structures*, pages 128–145, 2010.

S. Zdancewic and A. Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop, 2001. Proceedings*, pages 15–23, 2001.

S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, 2002.

L. Zheng, S. Chong, A. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *15th IEEE Symposium on Security and Privacy*, 2003.

Y. Zheng, T. Matsumoto, and H. Imai. Connections among several versions of one-way hash functions. *Proc. of IEICE of Japan E*, 73, 1990.