Università
Ca'Foscari
Venezia

Dottorato di ricerca
in Informatica
Scuola di dottorato in Scienze e Tecnologie
Ciclo XXIII
(A.A. 2009 - 2010)

Type-based Analysis of Security APIs

SETTORE SCIENTIFICO DISCIPLINARE DI AFFERENZA: INF/01
Tesi di dottorato di Matteo Centenaro, matricola 955482

Direttore della Scuola di dottorato	**Tutore del dottorando**
Prof. Paolo Ugo	**Prof. Riccardo Focardi**

Abstract

A *Security API* is an interface between processes running at different levels of trust with the aim of assuring that a specific security policy holds. It allows an untrusted system to access the functionalities offered by a trusted secure resource assuring that no matter what sequence of the API commands are invoked, the intended security policy is satisfied.

This kind of API is often developed having in mind a target application and how it will typically use the available services. It is thus easy to miss the fact that some functionalities could be used in a malicious way to break the intended security policy. In fact, a number of attacks to existing security APIs have been found in the last years.

This thesis proposes a type-based analysis to verify the security of these critical components. Language-based analysis is, in fact, a powerful tool to formally prove security and, at the same time, helps API developers to understand the root-causes of known vulnerabilities affecting APIs and guides them in programming secure code.

A security API which slowly leaks secret data to an attacker capable to spot interferences between input parameters and a command output can be secured by a noninterference policy. The thesis extends the setting of language-based information flow security to account for cryptographic expressions (both randomized and deterministic ones) and applies the obtained results to analyse the ATM PIN verification API. A possible fix to it is also proposed and shown to be secure by typing.

A security API which, instead, directly releases a secret value as the result of a sequence of legal commands will be analysed with a type system ensuring that data secrecy is preserved at run-time. The thesis presents the case of programs implementing key management functionalities and proposes a type system to reason on the security of RSA PKCS#11 API and verify the correctness of a novel patch to it.

Sommario

Una *security API* è un 'interfaccia che regola la comunicazione tra due processi, che vengono eseguiti con diversi livelli di affidabilità, allo scopo di assicurare che sia soddisfatta una determinata proprietà di sicurezza. In questo modo, un sistema potenzialmente non fidato può fruire delle funzionalità offerte da una risorsa sicura senza mettere a rischio la sicurezza dei dati in essa contenuti: la API deve infatti impedire che una delle possibili sequenze di suoi comandi permetta di usare in maniera inappropriata la risorsa fidata. Molto spesso una API di questo tipo viene progettata e sviluppata considerando quella che sarà la sua applicazione più comune e come questa utilizzerà i servizi messi a disposizione. Una mancata visione di insieme dei comandi esposti dalla API apre la strada ad un suo possibile utilizzo malevolo atto a sovvertire la sicurezza dei dati che dovrebbe proteggere. Negli ultimi anni, infatti, sono stati scoperti numerosi attacchi contro le security API esistenti.

La tesi propone un'analisi, basata sui sistemi di tipi, per verificare la sicurezza di queste cruciali componenti. L'analisi formale dei linguaggi è, infatti, uno strumento molto potente per dimostrare che un dato programma soddisfa le proprietà di sicurezza richieste. Inoltre, fornisce anche un aiuto agli sviluppatori delle API per capire a fondo le cause delle vulnerabilità che le affliggono e li guida ad una programmazione consapevolmente sicura.

Un attaccante che è in grado di osservare differenze nei risultati ottenuti invocando la stessa funzione di una API con diversi parametri di input, può utilizzarle per derivare informazioni sui segreti custoditi nella parte fidata del sistema. Una security API soggetta a questo genere di attacchi può essere resa sicura utilizzando una proprietà di non-interferenza. La tesi estende la teoria esistente nel campo dell'*information flow security* per analizzare la sicurezza di programmi che fanno uso di primitive crittografiche (sia randomizzate che deterministiche) e applica i risultati ottenuti per studiare la API impiegata per la verifica dei PIN nella rete degli sportelli ATM (bancomat). Utilizzando il sistema di tipi proposto, è stato possibile proporre e verificare una soluzione che rende sicura tale API.

Una security API che, come risultato di una inaspettata sequenza di comandi, rivela una informazione che dovrebbe rimanere segreta, può essere invece analizzata con un sistema di tipi atto a controllare che la segretezza dei dati sia preservata durante tutto il tempo di esecuzione dei programmi. La tesi presenta il caso di programmi che offrono servizi per la gestione delle chiavi crittografiche, introducendo

un sistema di tipi in grado di ragionare sulla sicurezza dello standard RSA PKCS#11 e di verificare la correttezza di una nuova *patch* che lo rende sicuro.

Acknowledgments

Research is a team effort and I owe a great debt to several people who given me their time and input.

My very first thank goes to my supervisor, *Riccardo Focardi*, who has been supporting and motivating me during all the past three years. He introduced me to the world of research and worked very hard on each of our co-authored papers. I would also like to thank my co-authors: Graham Steel, Flaminia Luccio and Matteo Bortolozzo.

Thanks to my official reviewers Andrei Sabelfeld and Isabella Mastroeni for their valuable feedbacks and useful comments in reviewing my thesis.

Many thanks to the Computer Science Department of Ca' Foscari University and all the people working there. In particular, I would like to thank Annalisa Bossi and Antonino Salibra for their coordination of the Ph.D course.

Special thanks to all my Ph.D colleagues and friends with special mentions to Matteo Zanioli, Luca Leonardi, Alberto Carraro, Stefano Calzavara, Alvise Spanò, Gian-Luca dei Rossi, Raju Halder and Luca Rossi. A big thank goes also to "sec-group" (in particular Marco, Claudio, Lorenzo and Andrea) and to Andrea Suisani.

I am deeply grateful to my parents and my twin-sister Elisa for their unconditional support. My gratitude and my love goes to Francesca for always staying by my side.

Contents

Preface **vii**

Introduction **ix**

1 Security APIs **1**
1.1 API Attacks 3
1.2 Type-checking Security APIs 4

2 Noninterference **5**
2.1 Noninterference 6
2.2 Typing noninterference 9
2.3 Conclusion 11

3 Cryptographic Noninterference **13**
3.1 Introduction 14
3.2 A Language with Cryptography 15
3.3 Standard noninterference 18
3.4 Cryptographic noninterference 21
3.5 Hook-up properties 28
3.6 Type system 30
3.7 Conclusions 32

4 Proving Integrity by Equality **35**
4.1 Introduction 36
4.2 Secret-sensitive Noninterference 38
4.3 Hash Functions and Secrecy 39
4.4 Proving Integrity by Equality 42
4.5 Security Type System 42
4.6 Case Studies 46
4.7 Related Works 48
4.8 Conclusions 49

5 Type checking PIN Verification APIs **51**
5.1 Introduction 51
5.2 The Case Study 53
5.3 Basic Language and Security 54
5.4 Cryptographic primitives 56
5.4.1 Security with cryptography 57

5.4.2 Formal analysis of a PIN_V API attack . . . 60
5.5 Type System . . . 61
5.6 A type-checkable MAC-based API . . . 68
5.7 Conclusions . . . 70

6 Type checking PKCS#11 73
6.1 Introduction . . . 73
6.2 Language . . . 75
6.3 Type System . . . 77
6.3.1 Type Soundness . . . 81
6.4 Typed PKCS#11 . . . 84
6.4.1 RSA PKCS#11 Standard . . . 84
6.4.2 Key Diversification . . . 85
6.4.3 Secure Templates . . . 87
6.5 Conclusions . . . 89

7 Tookan: a TOOl for cryptoKi ANalysis 91
7.1 Introduction . . . 91
7.2 Model . . . 92
7.2.1 Basic Notions . . . 93
7.2.2 Modelling Real Tokens . . . 94
7.2.3 Limitations of Reverse Engineering . . . 99
7.3 Results . . . 100
7.3.1 Implemented functionality . . . 101
7.3.2 Attacks . . . 102
7.3.3 Model-checking results . . . 103
7.4 Finding Secure Configurations . . . 103
7.5 Conclusion . . . 105

Conclusions 109

A Cryptographic Noninterference - Formal Proofs 113

B Proving integrity by equality - Formal Proofs 121

C Type checking PIN Verification APIs - Formal Proofs 149
C.1 Closed key types . . . 150
C.2 Memory well-formedness . . . 150

D Type checking PKCS#11 - Formal Proofs 183
D.1 Typing values . . . 184
D.2 Formal proofs . . . 184

Bibliography 191

List of Figures

4.1 Size-aware Lattice 39

7.1 Tookan system diagram 94

List of Tables

2.1 Commands Semantics 8
2.2 Typing noninterference 10

3.1 Multi-threaded language semantics 19
3.2 Typing Multi-threaded Commands 31

4.1 Security Type System 45

5.1 The PIN verification API. 53
5.2 PIN APIs Type System - Expressions 63
5.3 PIN APIs Type System - Commands 66
5.4 The new PIN_V_M API with MAC-based integrity. 69

6.1 Commands Semantics 77
6.2 Typing expressions 80
6.3 Typing commands 81

7.1 Syntax of Meta-language for describing PKCS#11 configurations . . . 95
7.2 PKCS#11 key management subset with side conditions from the meta-language of table 7.1 96
7.3 Summary of results on devices 100

C.1 Values well-formedness 152
C.2 Command Semantics 153
C.3 Expression Semantics 153
C.4 PIN-block formats 181
C.5 Revised model of the PIN verification API with types. 182
C.6 The new PIN_T_M API with MAC-based integrity, with types. 182

Preface

The work presented in this thesis is based on some previously published papers as the result of the research carried out during the Ph.D. Program in Computer Science organized by the Department of Computer Science of Ca' Foscari University.

Chapter 3 has been first presented as joint work with Riccardo Focardi at the third *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)* in 2008 [32].

The contents of Chapter 4 appeared in the revised selected papers of *Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security Joint Workshop, ARSPA-WITS 2010* [20].

Chapter 5 is the result of a joint work with Riccardo Focardi, Flaminia Luccio and Graham Steel published in the *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS)* [21].

Contents of Chapter 6 appeared in a Computer Science Technical Report at Ca' Foscari University [19] and is going to be submitted for publication.

The contents of Chapter 7 was presented at the 17th *ACM Conference on Computer and Communications Security (CCS)* in 2010 as a joint work with Matteo Bortolozzo, Riccardo Focardi and Graham Steel.

Introduction

Computer systems are more and more used to store relevant data and need to enforce policies to protect them. Moreover, applications are growing in their complexity and it is unattainable to assume that these programs are error-free. A very important design principle is thus to identify the security critical part of a software and isolate it from the rest of the application, so that it will be possible to better check its correctness. Functionalities implemented in this sub-system can be accessed by the other components of the application through the interface exposed to them. In this way the security relevant functions would offer an API (Application Programming Interface) which programs can use in order to perform security significant tasks without worrying about respecting policies on the data. This kind of API is referred to as a *security API*, since it aims at enforcing a security policy on critical information.

It is clear that a security API, which is only implemented as a separate software component and run on the same system together with the rest of the application, could be subject to attacks aiming at subverting its protections. Moreover any malicious system administrator would be able to tamper with the API software implementation, for example, substituting it with a broken one. Special tamper-proof hardware is then used to guarantee API integrity: it implements a specific security API and assures that no one is able to access its internal components without causing an irremediable loss of data. Trusted hardware comes in different forms: in the financial applications it is often a Hardware Security Module (HSM), a PCI-card to be plugged into a computer, while in other kinds of systems, for example on authentication ones, it could present itself as a smartcard or even as a usb token. For example, think of credit-cards, public transport electronic tickets and the mobile phone Subscriber Identity Modules (SIMs): these are all trusted devices which are now ubiquitous in our day-to-day life.

The system is so divided into (at least) two sub-systems identifying a trusted component and an untrusted one. However, even if it is impossible to tamper with the trusted part as it is 'sealed' into special hardware, there is usually no proof that the provided security APIs are compliant to the intended policies. In fact, a number of attacks has been found which are able to break the security of these APIs by performing a somewhat unexpected sequence of legal calls leading to a security breach.

Designing security APIs is a really difficult task: it is necessary to think of the result of any possible API invocation sequence and see if the outcome is an attack or not. Formal analysis tools are potentially of great help in this field, assisting API developers in proving that their implementations are compliant to the desired security property. This thesis focuses on type-based analysis of security APIs: the

idea is that an API designer should code a prototype of its API and statically check if it type-checks, in that case she would be assured that the proposed implementation respects the security policy, no matter what sequence of calls are performed.

Contributions of the Thesis

The thesis first considers attacks against security APIs which are able to slowly leak secret data spotting dependency between the input parameters, chosen by an attacker, and the output given by an API command. This kind of bugs can be avoided by enforcing *noninterference*-style properties, whereby private data cannot influence values of the public ones, unless this is done in a controlled way. Security APIs often employ cryptographic primitives to protect data confidentiality and classical noninterference is too strong to be applied in this case. In fact, it forbids any flow from private to public data, contrasting, for example, the intuition that the encryption of a secret data under a secret key can be regarded as being public (indeed noninterference will regard this as insecure since the result depends on secret data). A first contribution of the thesis is the definition of new noninterference properties suitable to reason about the security of programs using cryptography.

The study of this intriguing extensions of noninterference starts from the case of symmetric key encryption. To protect data confidentiality it is common that encrypted message are randomized using an unpredictable confounder [1], otherwise information on plaintexts could be obtained comparing the corresponding ciphertexts. The capability of an intruder to observe different instances of the *same* encryption is modeled using patterns [3, 2]. Intuitively, this correspond to the attacker ability of comparing ciphertexts bit-wise as done, e.g., in traffic-analysis. Cryptographic noninterference [32] embodies these two ideas so that an intruder would not distinguish too much and a program releasing to the public a randomized encryption of a secret data under a secret key will not break the security property.

Deterministic cryptography is anyway commonly adopted in security APIs. For example, in the ATM PIN verification framework, it is possible to encrypt a user PIN in a non-randomized way: it will be encrypted together with a public identification data such as, e.g., the user account number. Combined with the fact on-line attacks are limited to very few attempts (usually 3), this enforces PIN confidentiality: given an account number, only few encrypted message can be produced. This avoids the possibility for an attacker to build a code-book of all the PINs. Limiting the wrong attempts at the ATM to just one, we can prove noninterference with deterministic encryption [21].

Determinism is also a crucial aspect of cryptographic hash functions, when employed to track data integrity. In fact, if the result of applying a deterministic operator to a given value is the same as one previously obtained by the same function, it is guaranteed (collisions apart) that the two pre-image values are the same. Technique like this one are useful, for example, to establish the integrity of a file downloaded from an untrusted site. However, the code executed inside an if-branch performing

this kind of integrity check should generally be regarded as low-integrity, since an opponent can always make the check fail by tampering with the values. Anyway, if the code is in what we informally call 'match-it-or-die' form, i.e., it always performs integrity checks at the outer level and halts whenever the check is not passed, this behaviour is harmless: for these programs, the attacker is not interested in causing a failure, as no code would be executed in such a case [21, 20].

Cryptographic hash functions can also be applied to track integrity of secret data or to commit to a secret value revealing it later on. In that case, randomization helps avoiding the above mentioned code-book attacks. However, if the entropy of the hashed value is low, an attacker might try to compute, by brute-force, the hash of all the possible values until he finds a match. This requires a non-trivial extension of the notion of patterns already used for the randomized encryption case [20].

These new noninterference properties are applied to give a type-based analysis of the ATM PIN verification API. In the last few years, several attacks have been published on this interface [11, 14, 24]. Very few of these attacks directly reveal the PIN. Instead, they involve the attacker calling the API commands repeatedly with slightly different parameters values, and using the results (which may be error codes) to deduce the value of the PIN. The machinery of information flow security, extended with the results discussed above, allows to precisely track the root-causes of these bugs and also to fix them. More precisely, the thesis focuses on the decimalization table attack [14] and shows that it is due to absence of integrity checks on the input parameters. An improved API is then proposed which assures the integrity of inputs using *Message Authentication Codes* (MACs). A type system for the analysis of this API is given and it is proved that if a program (implementing a command of the PIN verification framework) type-checks then it does not reveal any information about the PIN.

The second class of attacks to security APIs, considered in the thesis, directly leaks a secret in its output result, provided that a given sequence of commands are executed. The attack model in this case is rather different from the one discussed above: the malicious user is not looking at the outputs of the commands to learn something on the confidential data but is instead focused on finding a sequence of commands that reveal the desired information in clear. The attacker exploits an 'unusual' way of composing commands: taken alone, each API function will not leak any information on its secrets, but issuing an unexpected sequence of calls, the secrecy of the data can be broken. Noninterference would be a too strong property to require in this case. To tackle this issue, the thesis proposes the use of a type system verifying that data confidentiality is preserved at run-time.

One of the most used API for key management, RSA PCKS#11 [51], is considered as an example of interface vulnerable to these attacks. In fact, it is possible to extract the value of a sensitive key by overloading a key of different roles and executing a rather simple sequence of commands [23, 27]. The problem in this case is that cryptographic keys can be used for conflicting purposes and so the security of the operations for exporting and importing keys (in an encrypted form) are undermined.

A type system to verify the security of programs designed to perform key management tasks is introduced, proving that if they type-check then sensitive keys cannot be leaked. The type system forces programs to clearly define a unique purpose for each cryptographic key used and properly limits the functionalities of untrusted imported keys. A new patch for PKCS#11 is also presented as a novel contribution here. It is based on the usage of key diversification to ensure that no key is ever used for conflicting functions. This patch and another one [15] are shown to be secure by type-checking. We argue that this type-based analysis will help developers and hardware producers to better understand the reasons of the long known vulnerabilities affecting this security API and will be also useful to test and formally analyse new patches.

Devices implementing the PKCS#11 standard are widespread, but it was not known to what extents these vulnerabilities affected them. Moreover, a type-based analysis would require to access the source code of the firmware and/or the driver of these products: you can guess that this is no way possible. As a final result, the thesis presents a tool which is able to reverse engineer the features offered by a device and build a formal model of its functionalities [15]. This model is then analysed by a model-checker and if an attack is found it can be directly executed on the token to verify its applicability. Even if the formal techniques used to reason about the security of these devices are not based on type-theory, we have decided to give an overview of the state of the art of the security of these real devices. Moreover, this tool has recently become a commercial product of INRIA and Ca' Foscari University, a result which is worth mentioning and highlighting with a dedicated chapter.

Structure of the Thesis

The thesis gives some background notions on security APIs in Chapter 1, and on noninterference in Chapter 2. Extensions to information flow security suitable to analyse security APIs is presented in Chapter 3 and 4, then the ATM PIN verification API is analysed in Chapter 5. PKCS#11 security is the subject of Chapter 6 while Chapter 7 presents an automated tool for the analysis of real devices implementing such API.

1

Security APIs

An application programming interface (API) enables the interaction among heterogeneous components (hardware devices, software applications, libraries and so on) sharing the knowledge about the interface's vocabulary (commands) and conventions. A *security API* is an interface in between processes running with different levels of trust and which has to obey to a security policy: it allows an untrusted system to access the functionalities offered by a trusted secure resource assuring that the intended security policy is satisfied no matter what sequence of the API's commands are invoked.

Security APIs are ubiquitous these days. Think for example to a bank's ATM: it physically separates the component performing the cryptographic operations and the rest of the system (the PIN entering device, the smartcard reader, etc.); a tamper-resistant hardware security module will execute all the harmful cryptographic commands and will expose an API to the other ATM's components. The policy in this case ensures that any user PIN is always stored in an encrypted form and is never available in clear. Consider, otherwise, a tamper resistant smartcard used, for example, to store sensitive cryptographic materials such as keys and certificates. The card will offer an interface to any reader such that only an authorized user is allowed to access data stored inside the card (for example by asking for a PIN) and has also to guarantee that its keys are never extracted unencrypted. Another example is a browser JavaScript interpreter: it offers a set of commands to any web-site on the Internet visited by a user. In this case the policy requires, for example, that access to the client machine file system must be denied.

Designing security APIs is a very hard task. An API is often developed thinking about a target application and how this will legally use the available services. It is thus easy to miss the fact that some functionalities could be used in a malicious way to break the intended security policy. If it is the case then the interface is not secure and the sequence of commands leading to subvert the policy is told to be an *API attack*. The goal of an API attacker is to find out the proper parameters and calling sequence in order to mount an API attack.

The concept of security API as a research filed originated from Bond's PhD Thesis, Understanding Security APIs [13]. The idea was first introduced by Ross Anderson, Bond's supervisor, in a talk given at the Cambridge Security Protocols workshop in 2000. He described an attack he had originally presented some years before [6], which was caused by the inclusion of a specific transaction into a security API, but interestingly Anderson asked: "So how can you be sure that there isn't some chain of 17 transactions which will leak a clear key?". Over the next years a number of attacks to existing security APIs have been found and formal methods have been used to analyse these APIs.

1.1 API Attacks

A security API's policy can be broken in different ways [13]. An attacker is free to invoke each API command with parameters of his choice and so could, for example, exploit a bad choice of cryptographic algorithm or the way in which it is used, in this case we talk about cryptographic API attacks. Otherwise, an attack could simply consist of a sequence of invocations which directly leads to a result breaking the security policy without exploiting any specific cryptographic data: this is a pure API attack. Another class of API attacks is the information leakage one which spots a dependency of some API's secret data and its output (for example observing the fact that an error occurs) and consequently leak the private information by repeatedly calling the same command with slightly different parameters. A high-level example for each of the attack type follow.

Poor Key-half Binding Whenever the length of a cryptographic key is bigger than the algorithm block size, it is stored by using several blocks (as much as are needed). If the various parts composing a key are not precisely tracked, a malicious user would be able to mount an attack. Consider for example the case of the Triple-DES algorithm, usually its keys are stored into two different blocks but if the two halves of the key can be arbitrary manipulated by a user then its security is seriously undermined. In fact, if the two blocks containing the key can be set to the same value, then the key behaves exactly as a single-length DES key [12]: this is a cryptographic API attack, indeed the attacker is able to force the API to use a less-secure cryptographic algorithm.

The Wrap-Decrypt Attack PKCS#11 defines a widely adopted API for cryptographic tokens providing key management functions[51]. It offers some cryptographic functionalities and should preserve certain security properties, e.g. the values of a *sensitive* key stored on a device should never become known in the clear. Each key managed by the API has a set of attributes describing its property, e.g., signaling its valid usages. Known attacks on PKCS#11 [23, 27] are related to the operations for exporting and importing sensitive keys, called `WrapKey` and `UnwrapKey` in the API. The former performs the encryption of a key under another one and the latter performs the corresponding decrypt and import in the token. The API does not clearly separate roles for keys so that is possible to use the same key for conflicting purposes: for example, a key could have its `decrypt` and `wrap` attributes set, enabling the wrap and subsequent decrypt of a sensitive key, with the effect of leaking it outside the token as plaintext: a pure API attack. The security of this API will be the subject of Chapter 6, we will postpone all the details there.

The Decimalization Table Attack PIN codes are stored encrypted in a HSM at ATM facilities, the issuing bank when verifying the correctness of an entered PIN

will derive it from some public verification data. More precisely, it will encrypt the verification data under a PIN derivation key, truncate the result to the PIN's length and decimalize it, i.e., maps every byte to a decimal number based on a user-given mapping. Finally, the result of the comparison will say if the entered code was the right one. There is an attack [14] which exploits the fact that the function mapping the PIN to decimal number is chosen by the user invoking the API command: the attacker changes repeatedly the value of the mapping function discovering the digits of the PIN. This is an information leakage attack since no single transaction reveal the secret PIN in the clear, but the attacker uses the return values to discover information on it. The PIN verification API will be analysed in Chapter 5, all the details on this attack are discussed there.

1.2 Type-checking Security APIs

This thesis focuses on pure and information leakage API attacks. As already said above, it is not easy to build an API resistant to such attacks, it would thus be helpful to have formal tools suitable to check that a given implementation of a security API is indeed secure. Here a type-based analysis is proposed, indeed our focus is on helping API developers to understand the root-causes of known vulnerabilities affecting APIs and to aid them in developing secure code. We believe type-systems are better suite to this scope than other formal methods, such as model checkers which are instead more qualified for finding (new) attacks.

2

Noninterference

Protecting data stored in a computer system or transmitted over a public network is a relevant issue in computer security. Most of the methods employed to secure information cannot enforce end-to-end policies. Consider for example cryptography: encrypting the network communication between two peers using a fresh shared secret key prevents a malicious user to steal information by "listening" to the network traffic, anyway nothing prevents one of the parts to decrypt and reveal in the clear all the exchanged messages. Secure information flow tracks data propagation: an information *flow policy* specifies the security classes (or security levels) for data in a system and a *flow relation* declares the legal paths which the information can follow among the given security classes. To protect data confidentiality, for example, one can distinguish between secret and public data and avoid any secret information to flow to the public. More formally, an ordering on the security levels must be established and then during a program execution an information is allowed to flow from one level to another only if the former is lower or at the same level as the latter: a program is secure only if it does not contain any illegal flow. This constraint can be checked by noninterference [36], whereby a program is required to make computation on data such that any value stored in a variable of a certain security level does not depend on the value of any variable whose security level is greater or incomparable.

Information flow security can be enforced both statically at compile time and dynamically by adding specific control at run-time. This work focuses on static methods. The first static information flow certification mechanism has been presented in 1977 by Denning and Denning [30] and was implemented by instrumenting the language semantics to detect any leakage. A more interesting approach relies on a type-system [62]: variables are labeled with a security level and then if the program type-checks, it is guaranteed to not contain any illegal information flow.

This chapter gives background on language-based information flow security by first giving a noninterference property for a standard while language and then showing a type system to enforce such property.

2.1 Noninterference

Data confidentiality demands that private information are never revealed to someone which has not the clearance to access them. This thesis focus on language-based techniques thus restricting the interest to programs and their computations. A program aiming at preserving data confidentiality can access and modify secret information but must not reveal anything about such data in its public outputs: confidential data must not influence public ones so that variations on private data does not produce any observable difference in the outputs. This policy is known as *noninterference*: insecure information flows happen every time the value of a certain piece of information is influenced by the value of some data at an higher or incomparable security level. See Sabelfeld and Myers survey [53] for an overview on the literature on language-based information flow security .

Illegal flows in a language-based setting may only occur transferring information between variables in a given language. Such insecurity interferences may manifest themselves in different ways. A direct flow is an information leak that happens due to an illegal assignment, think for example of assigning the value of a secret variable to a public one. More subtly the control flow of a program can (unintentionally) reveal data, throughout a so called implicit flow:

$$\mathsf{if}\ \text{secret} = 0\ \mathsf{then}\ \text{public} := 0\ \mathsf{else}\ \text{public} := 1$$

Programs can also leak information through their termination or nontermination. Consider the following example, it is clear that whenever the program terminates the value of *secret* is zero: private information is so revealed by the fact that the computation terminates or diverges.

$$\mathsf{while}\ \text{secret}\ != 0\ \mathsf{do\ skip}$$

To certify noninterferent programs, each variable is assigned a security level, then an equivalence relation on programs' outputs (also referred to as states) is used to model the observational power of an external attacker and a program is said to respect noninterference if given two equivalent initial states it performs exactly the same observable computations on them thus leading to equivalent output states.

Security levels ℓ form a security lattice and information is allowed to flow only upwards or to the same level. In this thesis multi-level security is not considered so we will always refer to the basic case of two security levels: low for public (L) and high for secret (H) where $\mathsf{L} \sqsubseteq \mathsf{H}$, i.e., public data are allowed to flow to high variables while secret ones are forbidden to go to the low variables.

Consider a simple imperative while language like the following where expressions e are variables and binary (arithmetic and relational) operators on expressions while commands are skips (no operation), assignments, conditional branches, while loop and sequential compositions of commands.

$$\begin{array}{rcl} e & ::= & x \mid e_1\ \mathsf{op}\ e_2 \\ \mathsf{c} & ::= & \mathsf{skip} \mid x := e \mid \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 \mid \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c} \mid \mathsf{c}_1; \mathsf{c}_2 \end{array}$$

Variables x belongs to the set *Var* while *Val* is the set of values ranged over by v. Memories $\mathsf{M} : Var \to Val$ are finite maps from variables to values. We write $e \downarrow^{\mathsf{M}} v$ to note the atomic evaluation of expression e to value v in memory M. Command semantics is given by a standard structural operational semantics (Table 2.1) in terms of a small-step transition between configurations $\langle \mathsf{M}, \mathsf{c} \rangle$, pairs of memories and commands. Transitions are labeled with an *event* $\alpha \in Var \cup \{\tau\}$ indicating that an assignment to variable α (or no assignment if α is τ) has happened.

Observable behaviour A *security environment* Γ maps variables to their security levels. Users at level ℓ may only read variables whose level is lower or equal than ℓ.

Table 2.1 Commands Semantics

$$[skip] \quad \langle \mathsf{M}, \mathsf{skip} \rangle \xrightarrow{\tau} \langle \mathsf{M}, \varepsilon \rangle \qquad [assign] \quad \frac{e \downarrow^{\mathsf{M}} v}{\langle \mathsf{M}, x := e \rangle \xrightarrow{x} \langle \mathsf{M}[x \mapsto v], \varepsilon \rangle}$$

$$[seq1] \quad \frac{\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \xrightarrow{\alpha} \langle \varepsilon, \mathsf{M}_2 \rangle}{\langle \mathsf{M}_1, \mathsf{c}_1; \mathsf{c}_2 \rangle \xrightarrow{\alpha} \langle \mathsf{M}_2, \mathsf{c}_2 \rangle} \qquad [seq2] \quad \frac{\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \xrightarrow{\alpha} \langle \mathsf{M}_2, \mathsf{c}_1' \rangle}{\langle \mathsf{M}_1, \mathsf{c}_1; \mathsf{c}_2 \rangle \xrightarrow{\alpha} \langle \mathsf{M}_2, \mathsf{c}_1'; \mathsf{c}_2 \rangle}$$

$$[iff] \quad \frac{e \downarrow^{\mathsf{M}} \neq \mathsf{true}}{\langle \mathsf{M}, \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 \rangle \xrightarrow{\tau} \langle \mathsf{M}, \mathsf{c}_2 \rangle} \qquad [ift] \quad \frac{e \downarrow^{\mathsf{M}} \mathsf{true}}{\langle \mathsf{M}, \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 \rangle \xrightarrow{\tau} \langle \mathsf{M}, \mathsf{c}_1 \rangle}$$

$$[whilet] \quad \frac{e \downarrow^{\mathsf{M}} \mathsf{true}}{\langle \mathsf{M}, \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c} \rangle \xrightarrow{\tau} \langle \mathsf{M}, \mathsf{c}; \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c} \rangle}$$

$$[whilef] \quad \frac{e \downarrow^{\mathsf{M}} \neq \mathsf{true}}{\langle \mathsf{M}, \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c} \rangle \xrightarrow{\tau} \langle \varepsilon, \mathsf{M} \rangle}$$

Let $\mathsf{M}_{|\ell}$ be the projection of the memory M to level ℓ, i.e., memory M restricted to variables visible at level ℓ or below.

Definition 2.1. (Memories ℓ-equivalence)
M and M' are ℓ-equivalent, written $\mathsf{M} =_\ell \mathsf{M}'$, if $\mathsf{M}_{|\ell} = \mathsf{M}'_{|\ell}$.

Intuitively, users at level ℓ will never be able to distinguish two ℓ-equivalent memories M and M'. Similarly, users may only observe transitions assigning to variables at or below their level. Transition $\xrightarrow{\alpha}_\ell$ is defined as the least relation among configurations such that:

$$\frac{\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{x} \langle \mathsf{M}', \mathsf{c}' \rangle \quad \Gamma(x) \sqsubseteq \ell}{\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{x}_\ell \langle \mathsf{M}', \mathsf{c}' \rangle}$$

$$\frac{\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}' \rangle \quad \alpha = \tau \text{ or } \alpha = x \text{ with } \Gamma(x) \not\sqsubseteq \ell}{\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{\tau}_\ell \langle \mathsf{M}', \mathsf{c}' \rangle}$$

We write $\overset{\alpha}{\Rightarrow}_\ell$ to denote $\xrightarrow{\tau}_\ell^* \xrightarrow{\alpha}_\ell$, if $\alpha \neq \tau$, or $\xrightarrow{\tau}_\ell^*$ otherwise. Transitions $\overset{\tau}{\Rightarrow}_\ell$ are considered internal, silent reductions which are unobservable by anyone. Notice, instead, that for observable transitions $\overset{x}{\Rightarrow}_\ell$, the level of x is always at or below ℓ,

i.e., $\Gamma(x) \sqsubseteq \ell$. A configuration $\langle \mathsf{M}, \mathsf{c} \rangle$ diverges for ℓ, written $\langle \mathsf{M}, \mathsf{c} \rangle \Uparrow_\ell$, if it will never perform any ℓ-observable transition $\xrightarrow{x}_\ell$.

Secure programs Noninterference is formalized here by means of bisimulation relations. Two notions of noninterference are going to be introduced: the first one accept programs which leaks information via the termination channel while the second one is more restrictive and reject any leakage (even those using termination channels).

A termination-insensitive ℓ-bisimulation relates two programs if the observable transitions of the first program are simulated by the second one, unless the latter diverges on ℓ.

Definition 2.2. (Termination-insensitive ℓ-bisimulation)
A symmetric relation $\mathcal{R}$ on configurations is a termination-insensitive ℓ-bisimulation (ℓ-TIB) *if* $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \; \mathcal{R} \; \langle \mathsf{M}_2, \mathsf{c}_2 \rangle$ *implies* $\mathsf{M}_1 =_\ell \mathsf{M}_2$ *and whenever* $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \xrightarrow{\alpha}_\ell \langle \mathsf{M}'_1, \mathsf{c}'_1 \rangle$ *then either*

- $\langle \mathsf{M}_2, \mathsf{c}_2 \rangle \overset{\alpha}{\Rightarrow}_\ell \langle \mathsf{M}'_2, \mathsf{c}'_2 \rangle$ *and* $\langle \mathsf{M}'_1, \mathsf{c}'_1 \rangle \; \mathcal{R} \; \langle \mathsf{M}'_2, \mathsf{c}'_2 \rangle$ *or*
- $\langle \mathsf{M}_2, \mathsf{c}_2 \rangle \Uparrow_\ell$

$\langle \mathsf{M}_1, \mathsf{c}_1 \rangle$ *and* $\langle \mathsf{M}_2, \mathsf{c}_2 \rangle$ *are* termination-insensitive ℓ-bisimilar*,* $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \approx_\ell \langle \mathsf{M}_2, \mathsf{c}_2 \rangle$*, if there exists a ℓ-TIB relating them. Similarly, two commands are* termination-insensitive ℓ-bisimilar*,* $\mathsf{c}_1 \approx_\ell \mathsf{c}_2$*, if* $\forall \; \mathsf{M}_1 =_\ell \mathsf{M}_2$ *it holds* $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \approx_\ell \langle \mathsf{M}_2, \mathsf{c}_2 \rangle$.

Termination-sensitive bisimulation, instead, always requires observable actions to be simulated, enforcing a stronger noninterference policy which avoid any leakage.

Definition 2.3. (**Termination-sensitive ℓ-bisimulation**)
A symmetric relation $\mathcal{R}$ on configurations is a termination-sensitive ℓ-bisimulation (ℓ-TSB) *if* $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \; \mathcal{R} \; \langle \mathsf{M}_2, \mathsf{c}_2 \rangle$ *implies* $\mathsf{M}_1 =_\ell \mathsf{M}_2$ *and whenever* $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \xrightarrow{\alpha}_\ell \langle \mathsf{M}'_1, \mathsf{c}'_1 \rangle$ *then* $\langle \mathsf{M}_2, \mathsf{c}_2 \rangle \overset{\alpha}{\Rightarrow}_\ell \langle \mathsf{M}'_2, \mathsf{c}'_2 \rangle$, $\langle \mathsf{M}'_1, \mathsf{c}'_1 \rangle \; \mathcal{R} \; \langle \mathsf{M}'_2, \mathsf{c}'_2 \rangle$.

Configurations $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle$ *and* $\langle \mathsf{M}_2, \mathsf{c}_2 \rangle$ *are* termination-sensitive ℓ-bisimilar*, written* $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \simeq_\ell \langle \mathsf{M}_2, \mathsf{c}_2 \rangle$*, if there exists a ℓ-TSB relating them. Similarly, two commands are* termination-sensitive ℓ-bisimilar*,* $\mathsf{c}_1 \simeq_\ell \mathsf{c}_2$*, if* $\forall \; \mathsf{M}_1 =_\ell \mathsf{M}_2$, $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \simeq_\ell \langle \mathsf{M}_2, \mathsf{c}_2 \rangle$.

2.2 Typing noninterference

Noninterference can be enforced by a static analysis of programs' source code. Every expression is assigned a security level: an expression is low (L) if it does not contain any H variable while any expression can be high (H); then it is sufficient to check that programs do not assign a H-expression to a L variable (avoiding direct flows) and forbid commands with a H guard to assign in their body to L variable (preventing

implicit leaks). These requirements are straightforward to implement in a type system, this section presents a standard one that certifies termination-insensitive noninterference. The first type system to check noninterference has been proposed by Volpano, Smith and Irvine [62], and since then a lot of them appeared in the literature [53].

Expression are typed with security level ℓ using the security environment Γ to retrieve the type, i.e. the level, of each variable. The notation $\Gamma \vdash e : \ell$ means that expression e types ℓ under the security environment Γ. Commands are typed

Table 2.2 Typing noninterference

Expressions

$$(\text{var})\ \frac{\Gamma(x) = \ell}{\Gamma \vdash e : \ell} \quad (\text{exp-h})\ \Gamma \vdash e : \mathsf{H} \quad (\text{exp-l})\ \frac{\Gamma \vdash e_1 : \mathsf{L} \quad \Gamma \vdash e_2 : \mathsf{L}}{\Gamma \vdash e_1\ \mathsf{op}\ e_2 : \mathsf{L}}$$

Commands

$$(\text{skip})\ \Gamma, [\ell] \vdash \mathsf{skip} \quad (\text{assign})\ \frac{\Gamma(x) = \ell' \quad \Gamma \vdash e : \ell' \quad \ell \sqsubseteq \ell'}{\Gamma, [\ell] \vdash x := e} \quad (\text{sub})\ \frac{\Gamma, [\mathsf{H}] \vdash \mathsf{c}}{\Gamma, [\mathsf{L}] \vdash \mathsf{c}}$$

$$(\text{if})\ \frac{\Gamma \vdash e : \ell \quad \Gamma, [\ell] \vdash \mathsf{c}_1 \quad \Gamma, [\ell] \vdash \mathsf{c}_2}{\Gamma, [\ell] \vdash \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2} \quad (\text{seq})\ \frac{\Gamma, [\ell] \vdash \mathsf{c}_1 \quad \Gamma, [\ell] \vdash \mathsf{c}_2}{\Gamma, [\ell] \vdash \mathsf{c}_1; \mathsf{c}_2}$$

$$(\text{while})\ \frac{\Gamma \vdash e : \ell \quad \Gamma, [\ell] \vdash \mathsf{c}}{\Gamma, [\ell] \vdash \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}}$$

in a security context (noted $[\ell]$) as $\Gamma, [\ell] \vdash \mathsf{c}$. More precisely, a command that types whit context ℓ assigns only to variables whose security level is at least as restrictive as ℓ (see rule (assign)). The if branch and the while loop use the security context to track dependencies of the control flow from their test guard: rule (if) requires that the branches of an if command type at a security level prescribed by the guard, similarly for rule (while) in the case of loop. Implicit flows are so prevented, indeed, if the guard of branch (or loop) types H then the branches (or the loop's body) are forced to assign only to secret variables.

Soundness Programs that type-checks are guaranteed to satisfy noninterference (the termination-insensitive one). In order to prove such statement two standard lemmas, explicitly highlighting the principles encoded in the typing rules, are needed.

Lemma 2.1. (Simple security)
If $\Gamma \vdash e : \ell$ *then every variable* x *occurring in* e *is such that* $\Gamma(x) \sqsubseteq \ell$.

Lemma 2.2. (Confinement)
If $\Gamma, [\ell] \vdash \mathsf{c}$ *then for every variable* x *assigned to in* c *it holds* $\ell \sqsubseteq \Gamma(x)$.

Simple security is useful to prove that assignments do not break memory ℓ-equivalence while Confinement is needed when proving the fact that branches and loops do not allow for implicit flows to occur. The following soundness theorem can be easily derived.

Theorem 2.1. (Noninterference) *If* $\Gamma, [\ell] \vdash \mathsf{c}$ *then* $\mathsf{c} \approx_\ell \mathsf{c}$.

The formal proofs of the above lemmas and theorem follow similarly to the ones originally presented by Volpano, Smith and Irvince [62].

Typing termination-sensitive noninterference The more restrictive noninterference property can also be enforced using a type system. This can be achieved by some strong limitations on the while loops [61], allowing only low loop guards, or by accounting for the termination effect of a command [17, 56].

2.3 Conclusion

This chapter introduced the language-based information flow security notions necessary for the development of the rest of the thesis. Noninterference has been defined in a bisimulation form for a simple imperative while language, and a type system to enforce a termination-insensitive policy has been given.

3

Cryptographic Noninterference

Noninterference forbids any flow from private to public data, this contrasts the common feeling that the encryption of a secret data under a secret key can be regarded as being public. This chapter extends noninterference to correctly deals with symmetric cryptography: the main idea is that to protect data confidentiality every encrypted message has to be randomized using an unpredictable confounder [1], in this way for an attacker it is of any help to compare memories variable by variable (indeed two ciphertexts on two different memories can be considered to be always different) but he can instead look for the *equality pattern* of each memory and require them to be equivalent.

3.1 Introduction

Consider a multi-threaded distributed system in which threads share local memories and (multi-threaded) processes communicate over an insecure network. Due to the public nature of the network, programs will use encryption primitives to secure messages.

The analysis is carried out on the simple imperative while-language of the previous chapter extended with commands to send and receive network messages and one for spawning a new thread. Cryptographic operations are modelled as special expressions respecting the Dolev-Yao assumptions, i.e., an attacker can never decrypt a ciphertext without knowing the right decryption key. Studying information flow in this setting is challenging [9, 57]. Encrypting secret data with a high level key is expected to produce a low result that might be sent on the insecure network, like in

$$\mathsf{send}(\mathsf{enc}(h, k))$$

where h and k are, respectively, a high level datum and a high key. Since the encryption depends on high level, variations of h cause variations on the ciphertext which become observable when the message is sent over the network. This program would be thus judged as insecure by standard noninterference notions.

A new variant of noninterference which correctly deals with cryptographic messages in the Dolev-Yao model is introduced here. First of all, since information on plaintexts can be obtained by comparing the corresponding ciphertexts, randomized encryption based on unpredictable confounders, similarly to what is done in [1], is considered. The idea is that each encryption contains a fresh confounder which makes it different from every previous and future encryption. Then, memories (and networks) will be judged equivalent based on the notion of patterns [3, 2] so that it is possible to model the capability of an intruder to observe different instances of the *same* encryption. In fact, encrypting twice the same message will yield different ciphertexts, but copying the same encryption twice will result in a memory storing the very same encrypted messages as illustrated in the following example program

taken from [42]:

$$\begin{array}{l} l_1 := \mathsf{enc}(h_1, k) \\ \mathsf{if}\ h\ \mathsf{then}\ l_2 := \mathsf{enc}(h_1, k) \\ \quad\quad \mathsf{else}\ l_2 := l_1 \end{array} \tag{3.1}$$

Depending on the high level variable h, the program assigns to l_2 either a new encryption of h_1 with k or the encrypted value stored in l_1. Because of confounders, a new encryption of h_1 under k will differ from the one already stored in l_1. At the end of the program execution, if the else branch is taken, it will hold $l_1 = l_2$ otherwise $l_1 \neq l_2$: the boolean h is thus observable.

This new equivalence notion, apart from incorporating the above mentioned Dolev-Yao assumption, looks for possible *patterns* of equal encrypted messages and requires that those patterns are the same so to make it impossible for the intruder to achieve any information about secret encrypted data. From this indistinguishable patterns derives a definition of *strongly secure programs* which naturally extends the one proposed by Sabelfeld and Sands [54] for programs without cryptography.

This is, to the best of our knowledge, the first definition of noninterference in a multi-threaded distributed setting, with insecure channels and cryptography. Interestingly, only the underlying low-equivalence notion is refined, leaving the remaining part of the definition, i.e., the low-bisimulation, substantially the same. This minimal change, together with the fact that cryptography is modelled via expressions, simplifies the task of re-proving known results. In particular, we prove compositionality of secure programs and we adapt the type system of Sabelfeld and Sands [54] to our setting, proving its correctness.

Structure of the chapter The chapter is organized as follows: Section 3.2 presents the language, Section 3.3 gives the noninterference notion of *strongly secure programs* showing it is inadequate for dealing with cryptography; Section 3.4 illustrates the new noninterference notion, through many simple examples; Section 3.5 presents some results about composition of secure programs; Section 3.6 gives a type system for the proposed NI property; Section 3.7 draws some concluding remarks.

3.2 A Language with Cryptography

This section introduces an extension with explicit cryptography and commands handling the network and multi-threading functionalities of the imperative language presented in Chapter 2. The obtained language is all the way similar to the Multi-Threaded While-Language with Message Passing [52], a simple imperative language with message passing communication. Instead of assuming the presence of secure channels, as done in [52], these are all public and thus accessible by every program. Security is then achieved by explicit cryptographic operations which we model via the special expressions enc and dec. For the sake of readability, we only consider a synchronous, blocking, receive. We are confident all the results will scale to the

full language of [52], in which a non-blocking if-receive is considered, too. As in the original language, the send command is asynchronous. The new expressions and commands syntax follows:

$$e \ ::= \ \ldots \mid \mathsf{enc}(e_1, e_2) \mid \mathsf{dec}(e_1, e_2) \mid \mathsf{pair}(e_1, e_2) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e)$$

$$\mathsf{c} \ ::= \ \ldots \mid \mathsf{fork}(\mathsf{c}\vec{\mathsf{c}}) \mid \mathsf{send}(cid, e) \mid \mathsf{receive}(cid, x)$$

Commands are ranged over by c, d, while $\vec{\mathsf{c}}, \vec{\mathsf{d}}$ denote possibly empty vectors of concurrent commands $\langle\!| \mathsf{c}_1\, \mathsf{c}_2 \ldots \mathsf{c}_n |\!\rangle$, representing *multi-threaded programs*, variables x range over Var, boolean expressions (expressions using relational operators) are ranged over by b, values v range over Val and channel identifiers cid range over a fixed set CID. As in the previous chapter a security environment Γ assigns to each variable a security level ℓ (ranging over L and H). The commands $\mathsf{send}(cid, e)$ and $\mathsf{receive}(cid, x)$ are used to send and receive messages on network channel identified by cid. As already mentioned, channels are all public (in contrast to channels partition assumed in [52]), i.e., every program can access them, and are accessed with a standard First-In-First-Out (FIFO) policy. We model cryptographic operations as special expressions following the Dolev-Yao assumptions, as explained below.

Language Semantics A *configuration* $\langle \mathsf{M}, \sigma, \vec{\mathsf{c}} \rangle$ is a triple consisting of a local memory M, a network state σ and a vector of commands $\vec{\mathsf{c}}$. The network state $\sigma : CID \rightarrow LVal$ returns, for each channel identifier, the ordered list of values which are present on that channel. A program shares, over its threads, the local memory M and we suppose that executions happen on a single processor, i.e., at most one thread is active at a given point in time. Distributed programs $\vec{\mathsf{c}_1}, \ldots \vec{\mathsf{c}_n}$ have their own memories $\mathsf{M}_1, \ldots, \mathsf{M}_n$ and communicate via the network whose state is represented by σ. *Global configurations*, noted $\lhd(\mathsf{M}_1, \vec{\mathsf{c}_1}), \ldots, (\mathsf{M}_n, \vec{\mathsf{c}_n}); \sigma\rhd$, represent distributed programs.

The semantics is formalized in Table 3.1 by transitions between configurations and global configurations. In particular, $\rightarrow$ transition formalizes the deterministic execution of sequential commands and is the same as the one presented in Chapter 2 except for the new rules for send, receive and fork and the fact that the label α has been removed since it is not necessary (as it will be discussed when introducing the noninterference policy). Only these new transitions are reported below (in Table 3.1): $\mathsf{fork}(\mathsf{c}\vec{\mathsf{c}})$ dynamically generates a new vector of threads $\vec{\mathsf{c}}$ running in parallel with c; $\mathsf{send}(cid, e)$ and $\mathsf{receive}(cid, x)$ respectively sends and receives values on the channel cid which is modeled as a FIFO queue; notice also that rule [*seq2*] needs to be redefined to account for the possible generation of threads $\vec{\mathsf{d}}$ by c_1 and that have to be executed concurrently with the rest of the program. In [52], channels are modelled as unordered lists thus capturing a lower level view of distributed systems

in which the order of message delivery is not guaranteed. We can easily adapt our definitions to also deal with this different assumption.

Concurrency is modeled by transitions $\multimap$ and $\twoheadrightarrow$, the first non-deterministically picking a thread and executing it via $\rightarrow$, the second non-deterministically picking a multi-threaded program and executing it via $\multimap$. Intuitively, $\multimap$ acts as purely nondeterministic scheduler among all the threads, while $\twoheadrightarrow$ gives an interleaving semantics to the global distributed system.

Cryptography We model cryptography via special expressions. In particular, in the set of expressions EXP, we assume to have all the usual arithmetic and relational expressions plus encryption (enc) and decryption (dec) for cryptography, pair for the constructions of tuples, fst and snd to access their contents. We thus consider the following values, ranged over by v:

$$v ::= \bot \mid n \mid b \mid \{v\}_n \mid (v_1, v_2)$$

where $\bot$ is a special value representing failures, n denotes a generic atomic value, b ranges over booleans, $\{v\}_n$ represents the encryption of v using n as key and (v_1, v_2) is a pair of values. We will sometimes omit the brackets to simplify the notation, e.g., we will write $\{v_1, v_2\}_n$ for $\{(v_1, v_2)\}_n$.

Based on this set of values we can give the semantics of the special expressions mentioned above:

$$\begin{array}{rcll} \mathsf{enc}(v, n) & = & \{v, c\}_n & c \leftarrow \mathcal{C} \\ \mathsf{dec}(\{v, c\}_n, n) & = & v & \\ \mathsf{newkey} & = & k & k \leftarrow KEY \\ \mathsf{pair}(v_1, v_2) & = & (v_1, v_2) & \\ \mathsf{fst}((v_1, v_2)) & = & v_1 & \\ \mathsf{snd}((v_1, v_2)) & = & v_2 & \end{array}$$

where c is a fresh confounder, i.e., a number which is used for one encryption and never used again, and k is a fresh key. The notation $\leftarrow$ is used to represent random extraction from a set of values, namely $\mathcal{C}$ is the stream of confounders and KEY the one of encryption keys. Further details on this latter stream will be given later on. The probability of extracting the same random value is negligible, if the set is suitably large, so we actually model random extraction by requiring that extracted values are always different, e.g., $c \leftarrow \mathcal{C}$ can be thought as extracting the first element of an infinite stream of confounders and removing it from the list so that it cannot be reused. More formally we assume that two extractions $c, c' \leftarrow \mathcal{C}$ are such that $c \neq c'$. A similar solution is also adopted, e.g., in [1, 2].

Interestingly, the above functions are not defined for all the possible values. For example, decrypting with the wrong key is undefined, as is taking the first element of a value which is not a pair. We assume that all the undefined cases will fail producing a $\bot$ as result. This choice will influence command semantics, as described below.

A simpler solution would be to stop execution for the undefined cases. This would however make many programs insecure if we assume the intruder is able to observe termination (as we will). For example the following program reads a message from the network, decrypts it using a secret key k, then sends out a public value.

$$\begin{array}{l}\mathsf{receive}(cid, x)\\ y := \mathsf{dec}(x, k)\\ \mathsf{send}(cid, l)\end{array}$$

If dec stopped the execution then the last message would not be sent and the intruder could gain information about the message sent to the program. In particular, he could discover whether or not it was encrypted with the right key k. Our solution makes dec total: in case of wrong key y will be bound to $\perp$ but the last message will be sent anyway. We leave to the programmer the task of handling failures.

To guarantee a safe use of cryptography we also assume that every expression e different from the five above and every boolean expression b different from the equality test will fail when applied to ciphertexts, producing a $\perp$. This is important to (i) avoid "magic" expressions which decrypt a ciphertext without knowing the key like, e.g., $\mathsf{magicdecrypt}(\{v, c\}_n) = v$; (ii) abstract away from the bit-stream representation of ciphertexts: in our model, doing any kind of arithmetic operation on a ciphertext has an unpredictable result given that we are assuming randomized encryption. Checking equality is instead useful to observe copies of the very same encryption.

3.3 Standard noninterference

This section recalls the notion of *strongly secure* programs [54, 52] and naively try to apply it to the setting of programs which uses explicit cryptographic primitives. The goal is to illustrate why a standard non-interference property does not scale to cryptography. In doing this, we will exploit arguments similar to the ones of Askarov et al.[8, 9].

Strongly Secure Programs To judge a multi-threaded program $\vec{\mathsf{c}}$ secure we employ the notion of *strong low-bisimilarity* [54]: two strongly low-bisimilar thread pools must be of the same size and must spawn or terminate the same number of threads at each execution step, moreover each sequential move of one thread pool must be simulated by corresponding thread of the low-bisimilar pool and lead from low-equivalent states into low-equivalent states. This last requirement is really strong and makes the noninterference notion *time-sensitive*, i.e., bisimilar programs are not allowed to release information via the time channel: this obviously include also the termination channels so this policy is stronger than the one introduced in Definition 2.3.

Table 3.1 Multi-threaded language semantics

Commands

$$[seq2] \quad \frac{\langle \mathsf{M}_1, \sigma, \mathsf{c}_1 \rangle \to \langle \mathsf{M}_2, \sigma', \mathsf{c}_1' \vec{\mathsf{d}} \rangle}{\langle \mathsf{M}_1, \sigma, \mathsf{c}_1; \mathsf{c}_2 \rangle \to \langle \mathsf{M}_2, \sigma', (\mathsf{c}_1'; \mathsf{c}_2) \vec{\mathsf{d}} \rangle}$$

$$[fork] \quad \langle \mathsf{M}, \sigma, \mathsf{fork}(\mathsf{c}\vec{\mathsf{d}}) \rangle \to \langle \mathsf{M}, \sigma, \mathsf{c}\vec{\mathsf{d}} \rangle$$

$$[send] \quad \frac{e \downarrow^{\mathsf{M}} v \quad \sigma(cid) \;=\; vals}{\langle \mathsf{M}, \sigma, \mathsf{send}(cid, exp) \rangle \to \langle \mathsf{M}, \sigma[cid \mapsto v.vals], \varepsilon \rangle}$$

$$[receive] \quad \frac{\sigma(cid) \;=\; vals.v}{\langle \mathsf{M}, \sigma, \mathsf{receive}(cid, var) \rangle \to \langle \mathsf{M}[var \mapsto v], \sigma[cid \mapsto vals], \varepsilon \rangle}$$

Threads

$$[proc] \quad \frac{\langle \mathsf{M}_1, \sigma, \mathsf{c}_i \rangle \to \langle \mathsf{M}_2, \sigma', \vec{\mathsf{c}} \rangle}{\langle \mathsf{M}_1, \sigma, \langle\!| \mathsf{c}_1 \ldots \mathsf{c}_i \ldots \mathsf{c}_n |\!\rangle \rangle \twoheadrightarrow \langle \mathsf{M}_2, \sigma', \langle\!| \mathsf{c}_1 \ldots \vec{\mathsf{c}} \ldots \mathsf{c}_n |\!\rangle \rangle}$$

Distributed programs

$$[par] \quad \frac{\langle \mathsf{M}_j, \sigma, \vec{\mathsf{c}}_j \rangle \twoheadrightarrow \langle \mathsf{M}_j', \sigma', \vec{\mathsf{c}}_j' \rangle}{\lhd(\vec{\mathsf{c}}_1, \mathsf{M}_1) \ldots (\vec{\mathsf{c}}_n, \mathsf{M}_n); \sigma \rhd \twoheadrightarrow \lhd(\mathsf{M}_1, \vec{\mathsf{c}}_1) \ldots (\mathsf{M}_j', \vec{\mathsf{c}}_j') \ldots (\mathsf{M}_n, \vec{\mathsf{c}}_n); \sigma' \rhd}$$

The memory ℓ-equivalence (see Definition 2.1) is now lifted to networks in the expected way: low level users can observe every message in every network channel $cid \in CID$. A *state* $s = (\mathsf{M}, \sigma)$ is a pair composed of a memory M and a network state σ.

Definition 3.1. *Two states* $s_1 = (\mathsf{M}_1, \sigma_1)$ *and* $s_2 = (\mathsf{M}_2, \sigma_2)$ *are* low-equivalent, *noted* $s_1 =_{\mathsf{L}} s_2$*, if*

1. $\mathsf{M}_1 =_{\mathsf{L}} \mathsf{M}_2$*;*
2. $\sigma_1 =_{\mathsf{L}} \sigma_2$*:* $\forall cid \in CID, \sigma_1(cid) = \sigma_2(cid)$*;*

Strong low-bisimilarity is now formally defined using state low-equivalence to compare the result of each computational step.

Definition 3.2. *Let* $\mathcal{R}$ *be a symmetric relation on multi-threaded programs of equal size.* $\mathcal{R}$ *is a* strong low-bisimulation *if whenever*

$$\langle\!| \mathsf{c}_1 \ldots \mathsf{c}_n |\!\rangle \;\mathcal{R}\; \langle\!| \mathsf{d}_1 \ldots \mathsf{d}_n |\!\rangle$$

then $\forall s_1, s_2, i,$ *such that* $s_1 =_{\mathsf{L}} s_2$*:*

$$\langle s_1, \mathsf{c}_i \rangle \rightarrow \langle s_1', \vec{\mathsf{c}'} \rangle \text{ implies } \langle s_2, \mathsf{d}_i \rangle \rightarrow \langle s_2', \vec{\mathsf{d}'} \rangle$$
$$\text{for some } s_2', \vec{\mathsf{d}'} \text{ such that } \vec{\mathsf{c}'} \mathcal{R} \vec{\mathsf{d}'}, s_1' =_{\mathsf{L}} s_2'.$$

Strong low-bisimilarity $\cong_L$ *is defined as the union of all strong low-bisimulations.*

The intuition behind $\vec{\mathsf{c}} \cong_L \vec{\mathsf{d}}$ is that the two programs $\vec{\mathsf{c}}$ and $\vec{\mathsf{d}}$ are not distinguishable by any low level observer. In fact, every change done by one computational step of $\vec{\mathsf{c}}$ to the state is simulated by $\vec{\mathsf{d}}$ in a way that preserves state low-equivalence. Thus, if the states were indistinguishable before such a step, they will remain indistinguishable even after. Moreover, the reached programs have to be bisimilar so to guarantee that even the future steps will be indistinguishable. Notice also the universal quantification over all the possible low-equivalent states done at each step. This ensures compositionality given that any state change possibly performed by parallel threads or distributed programs is certainly "covered" by quantifying over all the possible states.

Definition 3.3. *A program* $\vec{\mathsf{c}}$ *is* secure *if* $\vec{\mathsf{c}} \cong_L \vec{\mathsf{c}}$*.*

Intuitively, a secure program which is run on equivalent states will always produce low-equivalent states, even in the presence of parallel threads and distributed programs. Thus no information on the high variables will ever be leaked.

Standard NI and Cryptography The above noninterference notion is too restrictive if naively applied to our language with cryptography. A simple assignment

$$l := \mathsf{enc}(h, k) \tag{3.2}$$

of an encrypted high level value to a low variable would be considered insecure. Notice that this kind of assignments are the one we expect to be able to do via encryption: we actually want to hide high level information via cryptography so to, e.g., safely send it on the untrusted network or simply store it in an untrusted (low) part of the local memory. To see why this simple program is judged to be insecure, consider the two following low-equivalent memories $\mathsf{M}_1 =_{\mathsf{L}} \mathsf{M}_2$:

M_1	M_2
$h : 1234$	$h : 5678$
$l : 0$	$l : 0$
$k : K$	$k : K$

Running the assignment we get

M_1'	M_2'
$h : 1234$	$h : 5678$
$l : \{1234, c_1\}_K$	$l : \{5678, c_2\}_K$
$k : K$	$k : K$

where $\mathsf{M}'_1(l) = \{1234, c_1\}_K \neq \{5678, c_2\}_K = \mathsf{M}'_2(l)$ and so $\mathsf{M}'_1 \neq_{\mathsf{L}} \mathsf{M}'_2$. We conclude that

$$l := \mathsf{enc}(h, k) \not\cong_L l := \mathsf{enc}(h, k)$$

and, consequently, such an assignment is judged as insecure. Notice that this is not just caused by the confounders c_1 and c_2. Even without confounders the two ciphertexts $\{1234\}_k$ and $\{5678\}_k$ would differ. The problem is related to the fact that $=_{\mathsf{L}}$ do no take into account that the plaintext should not be visible without knowing the decryption key. Confounders will actually help us defining a new notion of low-equivalence which is suitable for cryptography.

3.4 Cryptographic noninterference

The notion of *strongly secure programs* can be adapted to the language with cryptographic expressions. Interestingly, only the underlying notion of *low-equivalent* states $=_{\mathsf{L}}$ has to be refined while the remaining part of the definition, i.e., the low-bisimulation part, does not need to be changed. As shown later in Section 3.6, this minimal change together with the fact that cryptography is modeled via expressions, thus leaving the language unchanged, will make it very easy to rephrase an existing type system to the new setting.

Low-equivalent ciphertexts The use of confounders models the fact that encryptions will always be different, even when the encrypted messages are the same. As already discussed, this is an abstraction of randomized encryption, where encryptions are not always different but the probability that they are the same is negligible.

Intuitively, if ciphertexts are always different we can consider them to be indistinguishable and so equate them all. Of course, this is not true for values encrypted with a low level key, i.e., a key known by low level users: high level keys need to be distinguished from low level ones. A new security level K is added to the security lattice and we also require that variables labeled as K are disjoint from the other ones, i.e., variables which are not keys cannot flow to K and key variables cannot flow in the lattice: $\ell \not\sqsubseteq \mathsf{K}$ and $\mathsf{K} \not\sqsubseteq \ell (\ell : \mathsf{L}, \mathsf{H})$.

Cryptographic low-equivalence, noted $\approx_C$, is a new version of low-equivalence on values based on the notion of patterns [3, 2]. *Patterns* (*Pat*) is a set of extended values obtained by adding $\Box_c$, representing an undecryptable message with confounder c, to *Val*. The function $\mathsf{p}(v)$ takes a value and returns the corresponding pattern by replacing all the ciphertexts that cannot be decrypted with $\Box_c$. The intuition is that equal confounders correspond to equal, undecryptable, messages. In fact, messages generated using high keys are always assumed to contain a fresh confounder making it impossible to have two different messages with the same confounder. A *confounder substitution* is a bijection on confounders $\rho : \mathcal{C} \rightarrow \mathcal{C}$ used to compare

patterns up to renaming of confounders: $p\rho$ denotes the result of applying ρ as a substitution to the pattern p.

Definition 3.4. *Let* $p : Val \rightarrow Pat$ *be defined as follows:*

$$\begin{array}{rcl} \mathsf{p}(\bot) & = & \bot \\ \mathsf{p}(n) & = & n \\ \mathsf{p}(b) & = & b \\ \mathsf{p}((v_1, v_2)) & = & (\mathsf{p}(v_1), \mathsf{p}(v_2)) \\ \mathsf{p}(\{v, c\}_n) & = & \begin{cases} \{\mathsf{p}(v), c\}_n & \textit{if } n \notin KEY \\ \square_c & \textit{otherwise} \end{cases} \end{array}$$

Two values v_1 *and* v_2 *are cryptographically-low-equivalent, written* $v_1 \approx_C v_2$*, if there exists a confounder substitution* ρ *such that* $\mathsf{p}(v_1) = \mathsf{p}(v_2)\rho$*.*

The following two simple examples illustrate how the new equivalence operator $\approx_C$ equates only indistinguishable ciphertexts. Consider again Program (3.2) of Section 3.3 which, starting from low-equivalent memories, was producing the two different ciphertexts $\{1234, c_1\}_K$ and $\{5678, c_2\}_K$. Given that $K \in KEY$, we obtain that $\mathsf{p}(\{1234, c_1\}_K) = \square_{c_1}$ and $\mathsf{p}(\{5678, c_2\}_K) = \square_{c_2}$. Since confounders represent random numbers, the two patterns are indistinguishable. In fact, taking $\rho(c_2) = c_1$ it follows $\square_{c_1} = \square_{c_2}\rho$ and so $\{1234, c_1\}_K \approx_C \{5678, c_2\}_K$; if instead the key is a low level one $n \notin KEY$, $\mathsf{p}(\{1234, c_1\}_n) = \{1234, c_1\}_n \neq \{5678, c_2\}_n = \mathsf{p}(\{5678, c_2\}_n)$. Notice that it is impossible to make the two patterns equal through a substitution ρ because of the different plaintexts, thus $\{1234, c_1\}_n \not\approx_C \{5678, c_2\}_n$. The same holds also if only one of the two keys is untrusted, e.g., $\{1234, c_1\}_K \not\approx_C \{5678, c_2\}_n$, since $\square_{c_1}$ is never equal to $\{5678, c_2\}_n$. As a matter of fact, one of the two ciphertexts can be decrypted using n which tells an observer that the first ciphertext is, at least, encrypted with a different key.

Patterns of equal ciphertexts Whenever a new ciphertext is computed, the confounder guarantees that it will be different from any other ciphertext, but if a ciphertext from a variable is copied into another variable, the two will be identical. Intuitively, this correspond to the attacker ability of comparing ciphertexts bit-wise as done, e.g., in traffic-analysis: copies of the same ciphertext will always be identical and this aspect has to be considered when defining low-equivalence. Take, for example, the following program which only acts on low variables and public channels:

$$\begin{array}{l} \mathsf{if}\ (l_1 = l_2)\ \mathsf{then} \\ \quad \mathsf{send}(cid, l_3) \\ \mathsf{else} \\ \quad \mathsf{send}(cid, l_4) \end{array} \tag{3.3}$$

Depending on the equality of l_1 and l_2 it sends the value of two different low variables on channel cid. Now, consider the following states:

M_1	σ_1	M_2	σ_2
$l_1 : \{1234, c_1\}_K$	cid :	$l_1 : \{9999, c'_1\}_K$	cid :
$l_2 : \{1234, c_1\}_K$		$l_2 : \{5678, c'_2\}_K$	
$l_3 : \mathsf{true}$		$l_3 : \mathsf{true}$	
$l_4 : \mathsf{false}$		$l_4 : \mathsf{false}$	

Since $K \in KEY$, it is $\mathsf{M}_1(l_1) = \{1234, c_1\}_K \approx_C \{9999, c'_1\}_K = \mathsf{M}_2(l_1)$ and $\mathsf{M}_1(l_2) = \{1234, c_1\}_K \approx_C \{5678, c'_2\}_K = \mathsf{M}_2(l_2)$. However, running the program on these memories result in two different network states:

σ'_1	σ'_2
$cid : \mathsf{true}$	$cid : \mathsf{false}$

Thus, m_1 and m_2 should not be considered equivalent. This crucial issue is also illustrated with another simple example. Consider the two programs:

$l_1 := \mathsf{enc}(h, k)$	$l_1 := \mathsf{enc}(h, k)$
$l_2 := \mathsf{enc}(h, k)$	$l_2 := l_1$

Starting from clearly low-equivalent memories, in which, e.g., $l_1 = l_2 = 0$, they will produce the following memories

M_1	M_2
$h : 1234$	$h : 5678$
$l_1 : \{1234, c_1\}_K$	$l_1 : \{5678, c_3\}_K$
$l_2 : \{1234, c_2\}_K$	$l_2 : \{5678, c_3\}_K$
$k : K$	$k : K$

(3.4)

Notice that $\mathsf{M}_1(l_1) \approx_C \mathsf{M}_2(l_1)$ and $\mathsf{M}_1(l_2) \approx_C \mathsf{M}_2(l_2)$ but M_2 store twice the same ciphertextx in l_1 and l_2 ($l_1 = l_2$) while M_1 do not ($l_1 \neq l_2$). Thus the previously discussed low program (3.3) would distinguish M_1 from M_2.

One might argue that the two programs above are anyway distinguishable by NI, given that we quantify over all possible low-equivalent states. After the first assignment to l_1 we might, in fact, override that value and take two equivalent memories with, e.g, $l_1 = 0$. If we run the programs on these new memories we clearly obtain non-equivalent memories, given that we will have $l_2 : \{1234, c_2\}_K$ for the first program and $l_2 : 0$ for the second one. NI can thus track copies between memory cells via plain-texts and given that we require bisimilar programs to preserve low-equivalence of all the possible low-equivalent states, the two programs above would result to be non-bisimilar. However we may write a smarter program that copies l_1 to l_2 only when l_1 actually contains a ciphertext:

```
if (dec(l1, k) ≠⊥) then
    l2 := l1
else
    l2 := enc(h, k)
```

This program produces memories like (3.4) only when l_1 is actually a ciphertext encrypted with k, but we cannot track anymore this copy via plain-texts because, when l_1 is not a ciphertext, the program writes a new fresh ciphertext to l_2 which will never be the same as l_1. Notice that, whenever we observe $l_1 = l_2$, e.g., via the low program (3.3), we learn that l_1 is encrypted with K, which should be detected as a flow from high to low.

These examples show that it is necessary to build one single pattern on the whole memory, so that equal confounders in different memory cells will be observable. Indeed, the same reasoning applies to the network: equal confounders appearing either in the local memory or on the network channels will be observable, too. In order to deal with channel values, patterns are extended to deal with list of values, noted $v_1.vals$, by just letting $\mathsf{p}(v_1.vals) = \mathsf{p}(v_1).\mathsf{p}(vals)$. As expected, $vals \approx_C vals'$ if $\mathsf{p}(vals) = \mathsf{p}(vals')\rho$ for a confounder substitution ρ.

Definition 3.5. *The set of memory, network and state* patterns *are constructed as follows:*

$$\begin{array}{rcl} \mathsf{sp}(\mathsf{M}) & = & \{\ (x,\ \mathsf{p}(\mathsf{M}(x)))\mid \forall x.\ \Gamma(x)=\mathsf{L}\} \\ \mathsf{sp}(\sigma) & = & \{\ (cid,\ \mathsf{p}(\sigma(cid)))\mid \forall cid \in CID\} \\ \mathsf{sp}(\mathsf{M},\sigma) & = & \mathsf{sp}(\mathsf{M}) \cup \mathsf{sp}(\sigma) \end{array}$$

Two memories, networks or states t_1 and t_2 are cryptographically-low-equivalent, *written $t_1 =_C t_2$, if there exists a confounder substitution ρ such that $\mathsf{sp}(t_1) = \mathsf{sp}(t_2)\rho$.*

For example, memories (3.4) have state patterns

$$\mathsf{sp}(\mathsf{M}_1) = \{\ (l_1, \Box_{c_1}),\ (l_2, \Box_{c_2})\ \}$$

and

$$\mathsf{sp}(\mathsf{M}_2) = \{\ (l_1, \Box_{c_3}),\ (l_2, \Box_{c_3})\ \}$$

Notice that there does not exist a substitution ρ which makes such state patterns equal, thus $\mathsf{M}_1 \neq_C \mathsf{M}_2$.

This reflects the fact that equality of confounders is not the same in the two states: if equality is not preserved, it is in fact impossible to find a bijection ρ on confounders that make them the same.

We can prove that equivalent states are such that values of corresponding variables and channels are equivalent, too. The opposite implication does not hold and it actually motivated the definition of state patterns. We can also prove that removing (i.e., reading) the first value of one channel (item 3) and also copying it to the same low variable (item 4) do not break state equivalence.

Lemma 3.1. *If $(\mathsf{M}_1, \sigma_1) =_C (\mathsf{M}_2, \sigma_2)$ then*

1. $\mathsf{M}_1 =_C \mathsf{M}_2$ *implies* $\forall x.\ \Gamma(x) = \mathsf{L}, \mathsf{M}_1(x) \approx_C \mathsf{M}_2(x)$*;*

2. $\sigma_1 =_C \sigma_2$ *implies* $\forall cid, \sigma_1(cid) \approx_C \sigma_2(cid)$;

3. *If* $\sigma_1(cid) = vals_1.v_1$ *and* $\sigma_2(cid) = vals_2.v_2$ *then* $(\mathsf{M}_1, \sigma_1[cid \mapsto vals_1]) =_C (\mathsf{M}_2, \sigma_2[cid \mapsto vals_2])$.

4. *If* $\Gamma(x) = \mathsf{L}$, $\sigma_1(cid) = vals_1.v_1$ *and* $\sigma_2(cid) = vals_2.v_2$ *then* $(\mathsf{M}_1[x \mapsto v_1], \sigma_1[cid \mapsto vals_1]) =_C$ $(\mathsf{M}_2[x \mapsto v_2], \sigma_2[cid \mapsto vals_2])$.

Proof. The first two statements derives from the fact that we take subsets of the state patterns. On those subsets we can apply the same ρ that we used to equate states. For example, $(\mathsf{M}_1, \sigma_1) =_C (\mathsf{M}_2, \sigma_2)$ if $\mathsf{sp}(\mathsf{M}_1, \sigma_1) = \mathsf{sp}(\mathsf{M}_2, \sigma_2)\rho$. We have that $\mathsf{sp}(\mathsf{M}_1)$ and $\mathsf{sp}(\mathsf{M}_2)$ are the subsets of $\mathsf{sp}(\mathsf{M}_1, \sigma_1)$ and $\mathsf{sp}(\mathsf{M}_2, \sigma_2)$ only containing variables with their patterns. It is thus easy to see that the same ρ equates such memory patterns, i.e., $\mathsf{sp}(\mathsf{M}_1) = \mathsf{sp}(\mathsf{M}_2)\rho$. Analogously for statement three and four: removing the first value of a channel leaves the remaining patterns identical, up to ρ; the same happens when we assign such value to a low variable. □

The next example, taken from [8], shows why it is important to simultaneously observe patterns of memories and channels, as we do.

```
l := enc(h1, k)
send(ch, l)
if h then
      l := enc(h2, k)
else
      skip
```
(3.5)

An observer can deduce the value of h by comparing the value of l with the one sent on ch: they will be different only when h is true. Consider the following states just before the branch:

M_1	σ_1
h : true	
$h_1 : 1234 \quad h_2 : 5678$	
$l : \{1234, c_1\}_K$	$ch : \{1234, c_1\}_K$

M_2	σ_2
h : false	
$h_1 : 4443 \quad h_2 : 5556$	
$l : \{4443, c_1'\}_K$	$ch : \{4443, c_1'\}_K$

After the branch, M_1 is updated (yielding M_1') with the new value $\{5678, c_2\}_K$ for variable l while M_2 will not be touched ($\mathsf{M}_2 = \mathsf{M}_2'$). If we only observe memory patterns we have $\mathsf{sp}(\mathsf{M}_1') = \{\ (l, \Box_{c_2})\ \} =_C \{\ (l, \Box_{c_1'})\ \} = \mathsf{sp}(\mathsf{M}_2')$, since they are equal, up to renaming of c_1' into c_2. This is because there are no copies of l in the memory

to compare with. Similarly we have $\mathsf{sp}(\sigma_1) = \mathsf{sp}(\sigma_2)$. However, if we observe the whole state we obtain $\mathsf{sp}(\mathsf{M}'_1, \sigma_1) = \{\ (l, \Box_{c_2}),\ (ch, \Box_{c_1})\ \} \neq_C \{\ (l, \Box_{c'_1}),\ (ch, \Box_{c'_1})\ \} = \mathsf{sp}(\mathsf{M}'_2, \sigma_2)$. Notice that the equality of c'_1 in $\mathsf{sp}(\mathsf{M}'_2, \sigma_2)$ makes it impossible to rename confounders so to make patterns equal. Intuitively, the comparison with the value sent on the network allows us to deduce the value of h.

Secure programs *Strong cryptographic low-bisimilarity*, noted $\cong_C$, is defined exactly as strong low-bisimilarity of Definition 3.2, with $=_\mathsf{L}$ replaced by $=_C$. When quantifying over all the possible states, we make two assumptions:

Confounder unicity Values encrypted with high level keys and with the same confounder are exactly the same. As already discussed, this is a consequence of using a fresh confounder for each encryption. Instead, we do not assume anything on confounders that might have been chosen by the intruder, i.e., the confounders of ciphertexts encrypted with low level keys;

High level key safety High key variables, k such that $\Gamma(k) = \mathsf{K}$ can only contain high key values $K \in KEY$. On the other hand, we never allow high key values to occur unprotected in the low level memory and on the network, given that this would imply those keys are broken. This does not mean we assume keys cannot be broken: since we start from a state with no broken key and we require that, at each steps, keys are not broken, we basically check that high key values remain protected. Moreover, we will prove that this check is actually not necessary because secure programs will never break keys.

The first assumption is just a well-formedness condition that is preserved by each program execution, independently of its security:

Definition 3.6. *A state $s = (\mathsf{M}, \sigma)$ is* well-formed *if whenever $\{v, c\}_K, \{v', c'\}_{K'}$ occur in s, with $K, K' \in KEY$, then $c = c'$ implies $\{v, c\}_K = \{v', c'\}_{K'}$.*

We will always implicitly assume that states are well-formed. The second condition, instead, is important to check that high level keys are safely dealt with. In order to formalize it, given a state s, we write $s \vdash K$ to denote that $K \in values(\mathsf{sp}(s))$ where $values(p)$ is the set of all atomic values occurring in pattern p.

Definition 3.7. *A state $s = (\mathsf{M}, \sigma)$ is* key-safe *if*

1. $\forall x.\ \Gamma(x) = \mathsf{K},\ \mathsf{M}(x) \in KEY$;
2. $s \vdash n$ *implies* $n \notin KEY$;

We denote with KS the set of key-safe states.

We are now ready to give the new notion of strong cryptographic low-bisimilarity. Notice that we require key-safety only for quantified states s_1 and s_2. Indeed, we will prove that key-safety is preserved by bisimilar programs with controlled assignments to high level variables.

Definition 3.8. *Let $\mathcal{R}$ be a symmetric relation on multi-threaded programs of equal size, it is a* strong cryptographic low-bisimulation *if whenever* $\langle\!| \mathsf{c}_1 \dots \mathsf{c}_n |\!\rangle \,\mathcal{R}\, \langle\!| \mathsf{d}_1 \dots \mathsf{d}_n |\!\rangle$ *then* $\forall i$, $\forall s_1, s_2 \in KS$*, such that* $s_1 =_C s_2$*:*

$$\langle s_1, \mathsf{c}_i\rangle \rightarrow \langle s_1', \vec{\mathsf{c}'}\rangle \textit{ implies } \langle s_2, \mathsf{d}_i\rangle \rightarrow \langle s_2', \vec{\mathsf{d}'}\rangle$$
$$\textit{for some } s_2', \vec{\mathsf{d}'} \textit{ such that } \vec{\mathsf{c}'} \;\mathcal{R}\; \vec{\mathsf{d}'}, s_1' =_C s_2'.$$

Strong cryptographic low-bisimilarity $\cong_C$ *is defined as the union of all strong cryptographic low-bisimulations.*

The universal quantification over all possible cryptographically–low–equivalent states done at each step ensures compositionality. Indeed, any state change performed by a (possibly evil) concurrent thread or distributed program will be certainly "covered" by this quantification.

We can now prove key-safety preservation for programs with controlled assignments to high level key variables, called *key-safe programs*:

Definition 3.9. *A program* c *is key-safe if*

1. $\mathsf{receive}(cid, x)$ *never occurs in* c *if* $\Gamma(x) = \mathsf{K}$*;*

2. $x := e$ *with* $\Gamma(x) = \mathsf{K}$ *occurring in* c *implies* $e = y$ *and* $\Gamma(y) = \mathsf{K}$ *or* $e = \mathsf{newkey}$*.*

Proposition 3.1. *Let* c *and* d *be two key-safe programs.*
If $\forall s_1, s_2 \in KS$*, with* $s_1 =_C s_2$ *and*

$$\langle s_1, \mathsf{c}\rangle \rightarrow \langle s_1', \vec{\mathsf{c}'}\rangle \textit{ implies } \langle s_2, \mathsf{d}\rangle \rightarrow \langle s_2', \vec{\mathsf{d}'}\rangle$$
$$\textit{for some } s_2', \vec{\mathsf{d}'} \textit{ such that } s_1' =_C s_2'$$

then $s_1', s_2' \in KS$*.*

Proof. Let us assume, by contradiction, that one of s_1', s_2', let us say s_1', is not in KS. We have that either $\mathsf{M}_1'(k) \notin KEY$ for a certain k such that $\Gamma(k) = \mathsf{K}$, or $s_1' \vdash K$ with $K \in KEY$. Since $s_1 \in KS$ we have that $\forall x.\ \Gamma(x) = \mathsf{K}, \mathsf{M}_1(x) \in KEY$. The assumption on assignments ensures that k can only have been assigned to another x, for which we know $\mathsf{M}_1(x) \in KEY$, or to newkey that, by definition, returns a value in KEY which gives a contradiction. The only remaining case is $s_1' \vdash K$ with $K \in KEY$. We have two sub-cases: (i) if K has been generated with newkey it must be different from every other name in s_2' and, since it appears in $\mathsf{sp}(s_1')$ and not in $\mathsf{sp}(s_2')$, it cannot be $s_1' =_C s_2'$; (ii) K appeared in s_1 but not in $\mathsf{sp}(s_1)$ because $s_1 \in KS$. Thus we can consider a new state $s_3 = s_1\eta$ with η being the substitution $K \mapsto K'$, with $K' \leftarrow KEY$ fresh. Since $\mathsf{sp}(s_1) = \mathsf{sp}(s_3)$, then we have $s_1 =_C s_3$. We can run again c and d on equivalent states s_1 and s_3. Since K does not occur in s_3 it is impossible that it appears in s_3'. But we have that K appeared in $\mathsf{sp}(s_1')$ from which we obtain the contradiction, i.e., $s_1' \neq_C s_3'$. Intuitively, the universal quantification on equivalent states allows us to relabel broken key K to a fresh

one and observe its leakage by comparison with the relabelled state (which does not contain it). Assumption on receive command let us preserve key-safety while reading messages from the network. This follow directly form the fact that $s_1, s_2 \in KS$, so $s_1 \not\vdash K$ and $s_2 \not\vdash K$, $\forall K \in KEY$. Thus, we avoid to assign a bad value (i.e., a value not contained in KEY) to a key. □

Definition 3.10. *A program* $\vec{\mathsf{c}}$ *is* secure *if it is key-safe and* $\vec{\mathsf{c}} \approxeq_C \vec{\mathsf{c}}$*.*

3.5 Hook-up properties

Inspired by previous works [54, 52], this section investigates hook-up properties for $\approxeq_C$: it is proved that composing secure programs yields a secure program. The results are very similar to the ones of [54, 52], but the notion of *low* expressions has to be carefully adapted. A *low* expression, in the model with no cryptography, is just an expression that evaluates the same when calculated on low-equivalent states [54, 52], i.e.,

$$\forall\ \mathsf{M}_1 =_{\mathsf{L}} \mathsf{M}_2,\quad e\downarrow^{\mathsf{M}_1} = e\downarrow^{\mathsf{M}_2}$$

This cannot just be rephrased to the new equivalences

$$\forall\ \mathsf{M}_1 =_C \mathsf{M}_2,\quad e\downarrow^{\mathsf{M}_1} \approx_C e\downarrow^{\mathsf{M}_2} \tag{3.6}$$

indeed the equivalence of two values does not guarantee that if they are stored into two memories or sent on public channels the resulting states are equivalent. More precisely, it does not hold that $\mathsf{M}_1 =_C \mathsf{M}_2$, $\Gamma(x) = \mathsf{L}$ and $e\downarrow^{\mathsf{M}_1} \approx_C e\downarrow^{\mathsf{M}_2}$ implies $\mathsf{M}_1[x \mapsto e\downarrow^{\mathsf{M}_1}] \approx_C \mathsf{M}_2[x \mapsto e\downarrow^{\mathsf{M}_2}]$, and analogously for network states, as we have already illustrated in example (3.5). The following stronger requirement is instead used to identify low expressions:

Definition 3.11. *An expression e is said to be* low *if,*
$\forall x.\ \Gamma(x) = \mathsf{L}, \forall cid \in CID$, *for all states* $(\mathsf{M}_1, \sigma_1) =_C (\mathsf{M}_2, \sigma_2)$*, if we let* $v_1 = e\downarrow^{\mathsf{M}_1}$ *and* $v_2 = e\downarrow^{\mathsf{M}_2}$*, it holds that*

1. $(\mathsf{M}_1[x \mapsto v_1], \sigma_1) =_C (\mathsf{M}_2[x \mapsto v_2], \sigma_2)$*;*
2. $(\mathsf{M}_1, \sigma_1[cid \mapsto v_1.vals]) =_C (\mathsf{M}_2, \sigma_2[cid \mapsto v_2.vals])$*.*

otherwise e is high*.*

The following simple lemma, shows that the previously proposed definition of low expressions (3.6) is implied by our new definition.

Lemma 3.2. *If e is* low *then* $\forall \mathsf{M}_1 =_C \mathsf{M}_2$*,* $e\downarrow^{\mathsf{M}_1} \approx_C e\downarrow^{\mathsf{M}_2}$*.*

Proof. This fact is a direct consequence of Lemma 3.1, item 1. Taking a σ with empty channels, we also have that $(\mathsf{M}_1, \sigma) =_C (\mathsf{M}_2, \sigma)$. By definition of low expressions, we obtain $(\mathsf{M}_1[x \mapsto e\downarrow^{\mathsf{M}_1}], \sigma) =_C (\mathsf{M}_2[x \mapsto e\downarrow^{\mathsf{M}_2}], \sigma)$ and, by Lemma 3.1, item 1, $e\downarrow^{\mathsf{M}_1} \approx_C e\downarrow^{\mathsf{M}_2}$. □

This lemma is useful for proving a deterministic behaviour in case of low boolean guards. Notice the equality instead of equivalence:

Corollary 3.1. *If b is low, then $b \downarrow^{\mathsf{M}_1} = b \downarrow^{\mathsf{M}_2}$.*

Proof. Trivial, by Lemma 3.2 and by the fact that $\mathsf{p}(b) = b$ (Definition 3.4). □

Secure contexts [54, 52] are extended taking into account direct assignment to high level key variables and a careful use of the receive command.

Definition 3.12. *A* secure context *is a context built with secure programs. Let $[\vec{\bullet}]$ and $[\bullet]$ be holes for, respectively, a command vector and singleton command. Secure contexts are defined as follows*

$$\begin{array}{rl} \mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] ::= & \mathsf{skip} \mid x := e\ (\Gamma(x) = \mathsf{H}) \mid x := Exp_L\ (\Gamma(x) = \mathsf{L}) \\ & \mid [\bullet_1]; [\bullet_2] \mid k := k'\ (\Gamma(k) = \Gamma(k') = \mathsf{K}) \mid k := \mathsf{newkey}\ (\Gamma(k) = \mathsf{K}) \\ & \mid \mathsf{if}\ b_\mathsf{L}\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2] \mid \mathsf{while}\ b_\mathsf{L}\ \mathsf{do}\ [\bullet_1] \\ & \mid \mathsf{fork}([\bullet_1][\vec{\bullet_2}]) \mid \mathsf{send}(cid, Exp_L) \\ & \mid \mathsf{receive}(cid, x)\ (\Gamma(x) \neq \mathsf{K}) \mid \langle\!\langle\, [\vec{\bullet_1}][\vec{\bullet_2}]\, \rangle\!\rangle \end{array}$$

where b_L and Exp_L denotes low expressions.

The next result proves that $\approxeq_C$ is preserved by secure contexts.

Theorem 3.1. *If $\vec{c_1} \approxeq_C \vec{c_1'}$, $\vec{c_2} \approxeq_C \vec{c_2'}$ then*

1. $\mathbb{C}[\vec{c_1}, \vec{c_2}] \approxeq_C \mathbb{C}[\vec{c_1'}, \vec{c_2'}]$;
2. *Let $\mathbb{D}[\vec{\bullet_1}, \vec{\bullet_2}] = \mathsf{if}\ b\ \mathsf{then}\ \vec{\bullet_1}\ \mathsf{else}\ \vec{\bullet_2}$, with b* high*. Then, $\vec{c_1} \approxeq_C \vec{c_2}$ implies $\mathbb{D}[\vec{c_1}, \vec{c_2}] \approxeq_C \mathbb{D}[\vec{c_1'}, \vec{c_2'}]$.*

Proof outline. (Full proof is in Appendix A.) The Theorem is proved inductively on the structure of contexts, by exploiting equivalences $\vec{c_1} \approxeq_C \vec{c_1'}$ and $\vec{c_2} \approxeq_C \vec{c_2'}$ and, for statement 2, $\vec{c_1} \approxeq_C \vec{c_2}$. It can be conducted as in [54], except for assignments, message exchange, branches and while loops. For assignments and message exchanges, we have to prove that equivalence of states will be preserved by executing the command. To this aim, we directly exploit the requirements on low expression given in Definition 3.11. In fact, such a definition states that assigning the result of an expression to a low variable or sending such a result on the network leave the states equivalent. For the reception of a message we also exploit Lemma 3.1, item 3 and 4, stating that the removal of a value from a channel and the assignment of that value to a low variable does not break state equivalence. As far as low branches (and while loops) are concerned, we have to prove that they always branch in the same way on equivalent states. This is guaranteed by Corollary 3.1, stating that the result of evaluating b on the two equivalent states is always the same. □

The next Hook-up Corollary proves that secure programs placed in secure contexts are still secure. For high branches, as expected, this happens when the two branches are equivalent.

Corollary 3.2. *Let $\vec{c_1}$, $\vec{c_2}$ be secure programs. Then*

1. $\mathbb{C}[\vec{c_1}, \vec{c_2}]$ *is secure;*
2. *Let* $\mathbb{D}[\vec{\bullet_1}, \vec{\bullet_2}] = \mathsf{if}\ b\ \mathsf{then}\ \vec{\bullet_1}\ \mathsf{else}\ \vec{\bullet_2}$, *with* b high. *Then,* $\vec{c_1} \approxeq_C \vec{c_2}$ *implies that* $\mathbb{D}[\vec{c_1}, \vec{c_2}]$ *is secure.*

Proof. Since $\vec{c_1}$ and $\vec{c_2}$ are secure we have $\vec{c_1} \approxeq_C \vec{c_1}$ and $\vec{c_2} \approxeq_C \vec{c_2}$. By Theorem 3.1 it must be that $\mathbb{C}[\vec{c_1}, \vec{c_2}] \approxeq_C \mathbb{C}[\vec{c_1}, \vec{c_2}]$ meaning that $\mathbb{C}[\vec{c_1}, \vec{c_2}]$ is secure and the same holds for $\mathbb{D}[\vec{c_1}, \vec{c_2}]$. □

3.6 Type system

The type system presented by Sabelfeld and Mantel in [52], which is an extension of [54], can be easily adapted to the setting of the current chapter. It transforms, if possible, a given program $\vec{c}$ into a new one $\vec{c'}$ which is the timing-leak free version of the original program, exploiting Agat's approach [5]. In particular, branches of conditional of different lengths are padded using skip commands, when necessary.

The typing rules for commands have the form

$$\vec{c} \hookrightarrow \vec{c'} : \vec{Sl}$$

where $\vec{c}$ is a program, $\vec{c'}$ is its transformation and $\vec{Sl}$ is the type of $\vec{c'}$. The type of a program is its *low slice*: a copy of a secure command where assignment to high and key variables have been replaced by skip. A slice models the time behaviour of $\vec{c'}$ as observable by an attacker running concurrently with it [54].

Our message passing commands send and receive are typed as the low, insecure, channels of [52]. The only real extension to previous type systems are the rules for typing expressions, including enc and dec.

Expressions Types for expressions are levels ℓ: L, H, for public and secret data, and K, for high level keys. Only encrypting with a secure key k will provide security guarantees. However, we admit encryption with untrusted (L) values so to allow encrypted communication between the trusted processes and the hostile environment that otherwise could only communicate via plain-texts.

Judgments have the form $e : \ell$. Typing rules for expressions are as the one of Chapter 2 (see Table 2.2) with the new rules handling newkey and encryption:

$$\text{(newkey)}\ \ \mathsf{newkey} : \mathsf{K} \qquad \text{(enc-sec)}\ \frac{x : \mathsf{K} \quad e : \mathsf{H}}{\mathsf{enc}(e, x) : \mathsf{L}}$$

Table 3.2 Typing Multi-threaded Commands

$$(Skip)\quad \mathsf{skip} \hookrightarrow \mathsf{skip} : \mathsf{skip} \qquad (Assign_{low})\quad \frac{e:\ \mathsf{L}}{l := e \hookrightarrow l := e : l := e}$$

$$(Assign_{high})\quad h := e \hookrightarrow h := e : \mathsf{skip} \qquad (Assign_{key})\quad \frac{e:\ \mathsf{K}}{k := e \hookrightarrow k := e : \mathsf{skip}}$$

$$(Seq)\quad \frac{\mathsf{c}_1 \hookrightarrow \mathsf{c}_1' : Sl_1\ \ \mathsf{c}_2 \hookrightarrow \mathsf{c}_2' : Sl_2}{\mathsf{c}_1; \mathsf{c}_1 \hookrightarrow \mathsf{c}_1'; \mathsf{c}_2' : Sl_1; Sl_2} \qquad (While)\quad \frac{b:\ \mathsf{L}\ \mathsf{c} \hookrightarrow \mathsf{c}' : Sl}{\mathsf{while}\ b\ \mathsf{do}\ \mathsf{c} \hookrightarrow \mathsf{while}\ b\ \mathsf{do}\ \mathsf{c}' : \mathsf{while}\ b\ \mathsf{do}\ Sl}$$

$$(Fork)\quad \frac{\mathsf{c}_1 \hookrightarrow \mathsf{c}_1' : Sl_1\ \ \vec{\mathsf{c}_2} \hookrightarrow \vec{\mathsf{c}_2'} : \vec{Sl_2}}{\mathsf{fork}(\mathsf{c}_1\vec{\mathsf{c}_2}) \hookrightarrow \mathsf{fork}(\mathsf{c}_1'\vec{\mathsf{c}_2'}) : \mathsf{fork}(Sl_1\vec{Sl_2})} \qquad (Par)\quad \frac{\mathsf{c}_1 \hookrightarrow \mathsf{c}_1' : Sl_1\ \ldots \mathsf{c}_n \hookrightarrow C_n' : Sl_n}{\langle\!|\ \mathsf{c}_1 \ldots \mathsf{c}_n\ |\!\rangle \hookrightarrow \langle\!|\ \mathsf{c}_1' \ldots \mathsf{c}_n'\ |\!\rangle : \langle\!|\ Sl_1 \ldots Sl_n\ |\!\rangle}$$

$$(If_{low})\quad \frac{b:\ \mathsf{L}\quad \mathsf{c}_1 \hookrightarrow \mathsf{c}_1' : Sl_1\quad \mathsf{c}_2 \hookrightarrow \mathsf{c}_2' : Sl_2}{\mathsf{if}\ b\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ \mathsf{c}_1'\ \mathsf{else}\ \mathsf{c}_2' : \mathsf{if}\ b\ \mathsf{then}\ Sl_1\ \mathsf{else}\ Sl_2}$$

$$(If_{high})\quad \frac{b:\ \mathsf{H}\quad \mathsf{c}_1 \hookrightarrow \mathsf{c}_1' : Sl_1\quad \mathsf{c}_2 \hookrightarrow \mathsf{c}_2' : Sl_2\quad al(Sl_1) = al(Sl_2) = \mathsf{false}}{\mathsf{if}\ b\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ \mathsf{c}_1'; Sl_2\ \mathsf{else}\ Sl_1; \mathsf{c}_2' : \mathsf{skip}; Sl_1; Sl_2}$$

$$(Send)\quad \frac{e:\ \mathsf{L}}{\mathsf{send}(cid, e) \hookrightarrow \mathsf{send}(cid, e) : \mathsf{send}(cid, e)}$$

$$(Receive)\quad \frac{\Gamma(x) \neq \mathsf{K}}{\mathsf{receive}(cid, x) \hookrightarrow \mathsf{receive}(cid, x) : \mathsf{receive}(cid, \hat{x})}$$

The (newkey) rule states that newkey returns a high level key. The (enc-sec) rule checks that a proper key is used when a secret text is ciphered: it demands to use a high level key. Note that the dec expression has not a dedicated rule: it will be typed using either rule (exp-l) or (exp-h) of Table 2.2, so apart from decryption with a L "key" dec will be always typed H.

Commands Typing and transformation rules are presented in Table 3.2. Intuitively, the command skip is typed by itself; to prevent explicit flows, rule $(Assign_{low})$ requires the expression to be typed as L; typing an assignment to a secret variable will be done using skip as its low slice, this is because we want that the slice has no occurrences of H variables $(Assign_{high})$. Rule $(Assign_{key})$ requires the expression to be typed as K and uses skip as low slice; Rules (Seq), $(While)$, $(Fork)$, (Par), (If_{low})

are as expected and do nothing interesting. Let $al(C)$ be a boolean function on command returning true whenever occurs a syntactic assignment to a low variable or a receive command of the form receive(cid, x) with $\Gamma(x) = \mathsf{L}$. Rule (If_{high}) asks that $(al(Sl_1) = al(Sl_2) = \mathsf{false})$, neither low assignment nor receive that reads message to a low variable occurs, to prevent indirect insecure flows [54]. It also aims to make the two branches of the conditional bisimilar, in fact the transformed command is composed by the same conditional with branches modified to contains the original sub-command and the low slice of the other branch. Rules (*Send*) and (*Receive*) are taken from [52] for the case of low (public) channel. We additionally require that a key cannot be read directly from a channel. Low slice cannot use secret variables so (*Receive*) let the command read the message from the network (removing it from *cid*) but do not update the variable x if it is high. This is obtained using the notation $\hat{x}$ which is defined as follow: $\hat{x} = _$ if $\Gamma(x) = \mathsf{H}$, $\hat{x} = x$ if $\Gamma(x) = \mathsf{L}$ [52].

Soundness In order to apply Theorem 3.1 and Corollary 3.2 to well-typed programs, expressions that type L must be shown to be low expressions:

Lemma 3.3. (Expression equivalence)
If $e : \mathsf{L}$ *then* e *is low according to Definition 3.11*

Proof. Proof is in Appendix A. □

It can be finally proved that well-typed programs are secure.

Theorem 3.2. ***(Correctness)***
If $\vec{\mathsf{c}} \hookrightarrow \vec{\mathsf{c}'} : \vec{Sl}$ *then* $\vec{\mathsf{c}'} \approxeq_C \vec{Sl}$.

Proof. Proof is in the Appendix. □

Theorem 3.3. ***(Program Noninterference)***
If $\vec{\mathsf{c}} \hookrightarrow \vec{\mathsf{c}'} : \vec{Sl}$*, then* $\vec{\mathsf{c}'}$ *is secure, i.e.,* $\vec{\mathsf{c}'}$ *is key-safe and* $\vec{\mathsf{c}'} \approxeq_C \vec{\mathsf{c}'}$.

Proof. The proposed type system accepts a program as valid only if it is a key-safe program: rules $(Assign_{key})$ and (*Receive*) implement the requirements of Definition 3.9. By Theorem 3.2 we know that $\vec{\mathsf{c}'} \approxeq_C \vec{Sl}$. By symmetry and transitivity of $\approxeq_C$ we get $\vec{\mathsf{c}'} \approxeq_C \vec{\mathsf{c}'}$. □

3.7 Conclusions

A noninterference property for programs using explicit cryptographic primitives has been introduced in this chapter. The setting used to develop the security analysis consider multi-threaded programs but the same ideas can be also applied to the simpler case of sequential single-threaded processes such as the one of security APIs.

Related work. Information flow security for programs with cryptographic expressions has been studied also by Askarov, Hedin and Sabelfeld [8, 9], Smith and Alpìzar [57] and Laud [42]. All of these papers, however, propose models and properties for sequential programs without multi-threading or concurrency.

More specifically, in [9] the authors adopt the notion of possibilistic noninterference, a weaker variant of noninterference. This choice has been driven by the need of distinguishing between different encryptions and copies of the same ciphertexts. The limitation of such a notion, however, is that it does not deal with possible concurrent thread executions. Consider, for example, the following program:

$$\begin{array}{l} h := \mathsf{true} \\ \mathsf{if}\ (h)\ \mathsf{then}\ l := \mathsf{true} \\ \qquad\quad \mathsf{else}\ l := \mathsf{false} \end{array}$$

It is clear that in a single-threaded setting this code can be referred as secure: in fact using the (possibilistic) noninterference notion of [9] the program would be considered secure (even if it would be rejected by the type system). Intuitively, such a property observes the result after program termination which, independently of the initial values of h and l, is always $h : \mathsf{true}$, $l : \mathsf{true}$. Our definition rejects such a program because a thread running together with the above code could change the value stored on the secret h just before the if command, thus making the program change its execution path and reveal the new high value.

The work by Smith and Alpìzar [57] uses computational probabilistic noninterference on a language with random assignments. The language is not multi-threaded but random assignments break the determinism of sequential programs making the setting much more complicate than just single-threading. The paper focus on the computational counterpart of noninterference, that we instead do not consider here.

Another paper on this line of research by Laud [42] investigates conditions under which the model proposed in [8] is computationally sound. The author essentially proves a conjecture made in [8] about the properties required on the underlying cryptographic primitives to guarantee computational security for programs that satisfy the possibilistic noninterference property discussed above. Interestingly, at the end of the paper, Laud suggests a variant of the model of [8] based on the same definition of patterns presented in this chapter. He still employs possibilistic noninterference for a single-threaded language. For example, consider the following example taken from [42]:

$$\begin{array}{l} k := \mathsf{newkey} \\ \mathsf{if}\ (h)\ \mathsf{then} \\ \qquad l_1 := \mathsf{enc}(a, k) \\ \qquad l_2 := \mathsf{enc}(b, k) \\ \mathsf{else} \\ \qquad l_2 := \mathsf{enc}(a, k) \\ \qquad l_1 := \mathsf{enc}(b, k) \end{array}$$

Notice that the order of assignments is swapped in the two branches. Nevertheless, Laud's model accept this program as secure, given that, at the end of execution, the two low variables are assigned to two different (randomized) ciphertexts. In a multi-threaded environment, however, we can think of the intruder as a concurrent thread observing, step by step, the program execution (and possibly controlling the scheduling). It is clear that by observing which of the two variables is assigned first, the intruder can deduce the value of h. The notion of noninterference introduced above correctly rejects this program. In a previous work, Laud [41] presented a type system to check secrecy of messages in cryptographic protocols implementation. While addressing multi-threading it was not aiming at noninterference result.

Vaughan and Zdancewic [59] study interaction between cryptography and information flow using implicit primitives in a single-threaded imperative language and obtaining a noninterference result which is based on both static and dynamic checking. They implement a decentralized label model (DLM) where confidentiality and integrity requirements can be specified independently. Chothia, Duggan and Vitek [22] first investigated the combination of DLM-style policies and cryptography but without providing any noninterference result.

Finally, papers [52, 45] are also quite related, even if they do not treat explicit cryptography. In particular, the language for distributed multi-threaded programs we adopted derives from the one proposed in those papers. Differently from [52, 45] we only consider insecure channels and, consequently secret data needs to be encrypted before being sent over the network. The noninterference property we adopt is basically the same but, because of the cryptographic messages, the underlying low-equivalence notion is completely different, as already discussed.

Closing remarks We have investigated information flow security for multi-threaded distributed application in the presence of explicit cryptographic operations. The model we have adopted derives from the notion of patterns [3, 2] proposed by the authors for proving computational soundness of formal cryptography. Interestingly, we have adopted it for a completely different purpose, i.e., as an underlying model for rephrasing an existing notion of noninterference [54]. Before discovering we really needed this notion, we have tried a number of different formalizations for low-equivalence, none of them as satisfactory as the present one. Extending the notion of noninterference introduced in this chapter to security APIs is straightforward since the most relevant part is the new equivalence relation on memories which can be used in any setting.

4
Proving Integrity by Equality

In the previous chapter noninterference has been extended to programs employing explicit cryptographic primitives. To preserve data secrecy, encrypted messages have been required to be randomized using an unpredictable confounder so that each ciphertext is different from every previous and future ecnryption. The following chapter investigates how to extended noninterference to deterministic cryptography considering the case of hash functions. The security property introduced will also focus on data integrity, since deterministic cryptography plays a crucial role in enabling a safe way to establish trust on messages retriven by an insecure sink. Secure usage of hash functions is also studied with respect to the confidentiality of digests by extending secret-sensitive noninterference of Demange and Sands [29].

4.1 Introduction

Cryptographic hash functions are commonly used as *modification detection codes* (*MDCs*) [46]: a hash function takes an input message and gets back its *image* or *digest*, then the goal is to provide message integrity assurance by comparing the digest of the original message with the hash of what is thought to be the intended message. Moreover, hash functions are also commonly employed to protect data secrecy as done, e.g., in Unix password files. To provide both integrity and confidentiality, hash functions are required to respectively be *collision resistant* and *one-way* [46], meaning that it should be infeasible to exhibit two messages with the same digest and to find a message whose digest matches a given one.

A first example of everyday usage of hash functions is password-based authentication: a one-way hash of the user password is securely stored in the system and is compared with the hash of the password typed by the user at the login prompt, whenever the user wants to access her account. The following code is a simplified fragment of the Unix *su* utility used to let a system administrator perform privileged actions. The password file is modeled as an array `passwd[username]`.

```
trial = hash(t_pwd);
if (trial = passwd[root]) then
    << launch the administrator shell >>
```

The typed password `t_pwd` is given as input to the program thus we regard it as untrusted. In fact, from the program perspective there could be an enemy “out there” trying to impersonate the legitimate administrator. The same holds for `trial`, being it computed from an untrusted value. Existing type systems for noninterference would consequently consider the guard of the if-branch as tainted (its result depending on untrusted data) and require that the code in the if-then branch never modifies high-integrity variables, being its execution under the control of the enemy. Clearly the administrator shell can make any change to the system including, e.g., modifying user passwords, and this program would be consequently rejected.

One of the motivations for hashing passwords is to protect confidentiality. In fact, if the hash function is one-way, it is infeasible for an opponent to find a password whose hash matches the one stored in the password file. In practice, brute-force dictionary attacks suggest that the password file should be nevertheless kept inaccessible by non-administrators, as it is done, e.g., in the *shadow password* mechanism of Unix. However, if password entropy is 'high enough' it might be safe to let every user access the hashed passwords. Formally, this would correspond to assigning the array `passwd[]` a low-confidentiality security level. Consider now the following update of Alice's password to the new, high-confidentiality value `alice_pwd`:

```
passwd[alice] := hash(alice_pwd);
```

This assignment would be rejected by usual type systems for noninterference, as it *downgrades* the confidentiality level of the password.

Hash functions are also often used for integrity checks. We consider a software producer who wants to distribute an application on the Internet, using different mirrors in order to speed-up the downloads. A common way to assure users downloading a binary file `my_blob.bin` from mirrors of its integrity, is to provide them with a trusted digest `swdigest` of the original program. The browser would then run a code similar to the following:

```
if (hash(myblob.bin) = swdigest) then
    trusted_blob.bin := my_blob.bin;
    << install trusted_blob >>
```

The idea is that the user will install the given binary only if its digest matches the one of the original program provided by the software company. In fact, if the hash function is collision resistant, it would be infeasible for an attacker to modify the downloaded program while preserving the digest. Once the check succeeds, `my_blob.bin` can be safely 'promoted' to high-integrity and installed into the system. This is modelled by assigning `my_blob.bin` to the high-integrity variable `trusted_blob.bin`. This is usually regarded as a direct integrity flaw and rejected by usual type systems for noninterference. Moreover, installing the application can be thought as writing into a high integrity area of the filesystem and, as for the root shell above, would be forbidden in a branch with a low-integrity guard.

We have discussed how typical examples of programs using cryptographic hash functions break standard notions of noninterference, even if they are intuitively secure. In this chapter, we study how to extend noninterference notions so that such kinds of program can be type-checked and proved secure. We model hash functions symbolically: the hash of a value v is simply $\mathsf{h}(v)$. We do not assume any deconstructor allowing to recover v from $\mathsf{h}(v)$ thus modelling the fact h is one-way, and we also assume $\mathsf{h}(v) = \mathsf{h}(v')$ if and only if $v = v'$, modelling collision-resistance. As is customary in symbolic settings, what has negligible probability in the computational world becomes here impossible.

We focus on what we informally call 'match-it-or-die' programs which, like the above examples, always perform integrity checks at the outer level and fail whenever the check is not passed. For these programs, the attacker is not interested in causing a failure, as no code would be executed in such a case. This enables us to type check programs that assign to high-integrity variables even in a low-integrity if branch, as in the Unix *su* example. We then observe that assignments such as `trusted_blob.bin := my_blob.bin` are safe under the check `hash(myblob.bin) = swdigest`. In fact, being `swdigest` high-integrity, matching it with the low-integrity value `hash(myblob.bin)` guarantees that `myblob.bin` has not been tampered with. This allows us to type check programs like the application downloading example.

Moreover, we investigate the confidentiality requirements for using hash functions to preserve data secrecy. We first observe that if the entropy of the hashed value is low an attacker might try to compute, by brute-force, the hash of all the possible values until he finds a match. We thus select, as our starting point, a recent noninterference variant called *secret-sensitive* noninterference [29] which distinguishes small and big secrets and allows us to treat their corresponding digests accordingly. If a secret is big, meaning that it is infeasible to guess its actual value, then the brute force attack above is also infeasible. We show that it is safe to downgrade the hash of a big secret, assuming some control over what secret is actually hashed. In fact, two hashes of the same big secret are always identical and the opponent might deduce some information by observing equality patterns of digests. This requires a non-trivial extension of the notion of memory equivalence so to suitably deal with such equality patterns.

Finally, we give a security type system to statically enforce that programs guarantees the proposed noninterference notions.

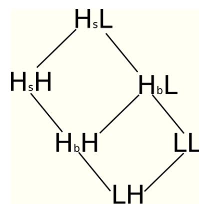
Structure of the chapter In Section 4.2 we give the background on secret-sensitive noninterference [29]; Section 4.3 extends the noninterference notions so to correctly deal with hash functions. Integrity check by equality is analyzed in Section 4.4. The security type system enforcing noninterference is given in Section 4.5 while Section 4.7 discusses related works. The chapter closes with some final remarks and ideas for future work in Section 4.8.

4.2 Secret-sensitive Noninterference

Secret-sensitive noninterference [29], by Demange and Sands, is a variant of noninterference which distinguishes small, guessable secrets from big, unguessable ones. As discussed in the introduction, this distinction will be useful to discipline the downgrading of digests of secret values, as we will see in Section 4.3.

Size-aware security lattice Secrets are partitioned into big ($\mathsf{H_b}$) and

Figure 4.1 Size-aware Lattice



small ($\mathsf{H_s}$) ones. Preorder $\sqsubseteq_C$ among confidentiality levels is defined as $\mathsf{L} \sqsubseteq_C \mathsf{H_b} \sqsubseteq_C \mathsf{H_s}$, meaning that public, low data can be regarded as secret and, as discussed above, small secrets need to be treated more carefully than big ones. We extend this size-aware confidentiality lattice by composing it with the basic two-level integrity lattice in which $\mathsf{H} \sqsubseteq_I \mathsf{L}$. Notice that integrity levels are counter-variant: low-integrity, tainted values have to be used more carefully than high-integrity, untainted ones. The product of these two lattices is depicted in Figure 4.1. We will write $\ell = \ell_C \ell_I$ to range over the product lattice elements. The ordering between the new security levels ℓ is denoted by $\sqsubseteq$ and is defined as the component-wise application of $\sqsubseteq_C$ and $\sqsubseteq_I$.

Secure programs The main idea of [29] is that for unguessable secrets, brute-force attacks will terminate only with negligible probability. Intuitively, this allows for adopting a termination-insensitive equivalence notion when comparing program behaviour. Guessable secrets, instead, can be leaked by brute-force using 'termination-channels', and for those values it is necessary to distinguish between terminating and nonterminating executions.

A secure program will preserve small secrets from being leaked via the termination channel while will be more liberal with respect to the big ones. This is achieved by requiring termination-sensitive bisimilarity (see Definition 2.3 on Chapter 2) whenever the inspected memories are the same at level $\mathsf{H_bL}$, meaning they only differ on small secrets. Notice that, as usual, the attacker is assumed to be at level LL.

Definition 4.1. (**Secret-sensitive NI**)
A command c *satisfies* secret-sensitive NI *if* $\forall\ \mathsf{M}_1 =_{\mathsf{LL}} \mathsf{M}_2$ *it holds*

1. $\langle \mathsf{M}_1, \mathsf{c} \rangle \approx_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c} \rangle$ *and*

2. $\mathsf{M}_1 =_{\mathsf{H_bL}} \mathsf{M}_2$ *implies* $\langle \mathsf{M}_1, \mathsf{c} \rangle \simeq_{\mathsf{H_bL}} \langle \mathsf{M}_2, \mathsf{c} \rangle$.

4.3 Hash Functions and Secrecy

This section extends secret-sensitive NI to program using hash functions. As already observed, hash functions could be subject to brute-force attacks, unless the hashed messages are big enough to make exhaustive search infeasible. The idea is to take advantage of the two distinct secret levels $\mathsf{H_b}$ and $\mathsf{H_s}$ protecting digests of small secrets while treating more liberally the digests of big secrets.

Hash expressions The language of Chapter 2 is augmented with a new hash expression whose semantics is defined in terms of a special constructor h. Formally, $\mathsf{hash}(e) \downarrow^{\mathsf{M}} \mathsf{h}(v)$ if $e \downarrow^{\mathsf{M}} v$ with $v \in \mathit{Val}$. We then partition Val into the sets of small and big values Val_s, Val_b, ranged over by v_s and v_b. We define the sets of small and big digests as $\mathit{Val}^d_\delta = \{\mathsf{h}(v) \mid v \in \mathit{Val}_\delta\}$, with $\delta \in \{s, b\}$. As discussed in the introduction, this simple modelling of hash functions is coherent with the assumption of being one-way (no deconstructor expressions) and collision-resistant (digests of different values *never* collide).

Memories ℓ-equivalence Lifting the notion of memory equivalence when dealing with digests requires to carefully handle equality patterns. In fact, in our symbolic model, equal digests will correspond to equal hashed messages.

Consider the program $x := \mathsf{hash}(y)$, where x is a public variable and y is a secret one. It must be considered secure only if y is a big secret variable, indeed leaking the digest of a small secret is equivalent to directly reveal the secret since an attacker could perform a brute-force attack on the hash.

The equivalence notion between memories needs, however, to be relaxed in order to capture the fact that big secrets are, in practice, random unpredictable values. We illustrate considering again $x := \mathsf{hash}(y)$ and assuming y to be a big secret variable. We let $\mathsf{M}_1(x) = 0 = \mathsf{M}_2(x)$, $\mathsf{M}_1(y) = v_b \neq v'_b = \mathsf{M}_2(y)$, then it holds $\mathsf{M}_1 =_{\mathsf{LL}} \mathsf{M}_2$. Executing the above code the resulting memories differ on the value stored in the public variable x: $\mathsf{M}'_1(x) = \mathsf{h}(v_b) \neq \mathsf{h}(v'_b) = \mathsf{M}'_2(x)$. It follows that $\mathsf{M}'_1 \neq_{\mathsf{LL}} \mathsf{M}'_2$ and the program does not respect noninterference so it would be rejected as insecure. However, being v_b and v'_b two big random numbers we can never expect they are equal and the only opportunity for the attacker is to see if they correspond to other big values in the same memory. Requiring the equality of big secrets and digests across memories is too strong.

This boils down to the idea of *patterns*, already employed in the previous chapter for cryptographic primitives. We illustrate through an example. Consider program $z := \mathsf{hash}(x); w := \mathsf{hash}(y)$ where z and w are public variables and x and y are big secrets. Consider the following memories:

$$\begin{array}{c|c} \mathsf{M}_1 & \mathsf{M}_2 \\ \hline x : v_b & x : v_b \\ y : v'_b & y : v_b \\ z : 0 & z : 0 \\ w : 0 & w : 0 \end{array} \tag{4.1}$$

executing the above code would make public two different digests in M_1 and the very same digests in M_2. The attacker is able to learn that the first memory stores two different secrets values while the second does not. In summary, we do not require the equality of big secrets and digests across memory but only that the equality-patterns are the same.

As the last example shows, in order to safely downgrade digests of big secrets we need to control how big secrets are stored in the memories. We do this by projecting out from memories big secret values which are either stored in big secret variables or hashed and observable from ℓ. This is done by the following function r, taking as parameters the value v and the level ℓ_v of a variable.

$$\mathsf{r}_\ell(v, \ell_v) = \begin{cases} v & \text{if } v \in Val_b \text{ and } \ell_v = \mathsf{H_b}\ell_I \\ v' & \text{if } v = \mathsf{h}(v'), v' \in Val_b \text{ and } \ell_v \sqsubseteq \ell \\ 0 & \text{otherwise} \end{cases}$$

A *big-secret projection* $\mathsf{r}_\ell(\mathsf{M})$ is defined as $\mathsf{r}_\ell(\mathsf{M})(x) = \mathsf{r}_\ell(\mathsf{M}(x), \Gamma(x))$, for all $x \in Dom(\mathsf{M})$. Two memories will be *comparable* if their big-secret projections can be matched renaming big values , i.e., if two big values are the same in one projection then it will also be the case that they are equal in the other one.

Definition 4.2 (**Comparable memories**).
Two memories M_1 *and* M_2 *are* ℓ-comparable*, noted* $\mathsf{M}_1 \bowtie_\ell \mathsf{M}_2$*, if there exists a bijection* $\mu : Val_b \to Val_b$ *such that* $\mathsf{r}_\ell(\mathsf{M}_1) = \mathsf{r}_\ell(\mathsf{M}_2)\mu$.

Example 1. *The two memories (4.1) are not comparable if observed at level* LL*, i.e.,* $\mathsf{M}_1 \not\bowtie_{\mathsf{LL}} \mathsf{M}_2$*. In fact,* $\mathsf{r}_\ell(\mathsf{M}_1)(x) = v_b \neq v_b' = \mathsf{r}_\ell(\mathsf{M}_1)(y)$ *while* $\mathsf{r}_\ell(\mathsf{M}_2)(x) = v_b = v_b = \mathsf{r}_\ell(\mathsf{M}_2)(y)$*. Thus there exists no bijection* μ *such that* $\mathsf{r}_\ell(\mathsf{M}_1) = \mathsf{r}_\ell(\mathsf{M}_2)\mu$*, since* μ *cannot map* v_b *to both* v_b *and* v_b'.

Two memories are ℓ-equivalent if they are ℓ-comparable and their observable big digests expose the same equality patterns. A *digest substitution* ρ is a bijection on digests of big values: $\rho : \; Val_b^d \to Val_b^d$.

Definition 4.3. (**Memory ℓ-equivalence with hash functions**)
Two memories, M_1 *and* M_2*, are* ℓ-equivalent*, written* $\mathsf{M}_1 =^h_\ell \mathsf{M}_2$*, if* $\mathsf{M}_1 \bowtie_\ell \mathsf{M}_2$ *and there exists a digest substitution* ρ *such that* $\mathsf{M}_{1|\ell} = \mathsf{M}_{2|\ell}\ \rho$.

Secure programs The bisimulation definitions given in Chapter 2 are left unchanged except for the relation used to compare memories which is now $=^h_\ell$ in place of $=_\ell$. Secret-sensitive NI is thus rephrased as follows.

Definition 4.4. (**Secret-sensitive NI with hash functions**)
A command c *satisfies* secret-sensitive NI *if* $\forall\ \mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}_2$ *it holds*

1. $\langle \mathsf{M}_1, \mathsf{c}\rangle \approx_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}\rangle$ *and*

2. $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ *implies* $\langle \mathsf{M}_1, \mathsf{c}\rangle \simeq_{\mathsf{H_bL}} \langle \mathsf{M}_2, \mathsf{c}\rangle$.

4.4 Proving Integrity by Equality

Consider the comparison (by equality test) of a high integrity value and an untrusted one: if the test succeeds we are guaranteed that the compared, low-integrity, value has not been tampered with.

Integrity can be checked via noninterference by placing the observer at level $\mathsf{H_sH}$. This amounts to quantifying over all the values in low-integrity variables and observing any interference they possibly cause on high integrity variables.

Definition 4.5. (**Integrity NI**)
A program c *satisfies* integrity NI *if for all* $\mathsf{M}_1, \mathsf{M}_2$ *such that* $\mathsf{M}_1 =_{\mathsf{H_sH}} \mathsf{M}_2$ *it holds* $\langle \mathsf{M}_1, \mathsf{c} \rangle \approx_{\mathsf{H_sH}} \langle \mathsf{M}_2, \mathsf{c} \rangle$.

Note that $\approx_{\mathsf{H_sH}}$ in the definition above refers to the termination-insensitive bisimulation introduced in Chapter 2 and not the one using the new equivalence operator of Definition 4.4.

Consider the program `if (x = y) then` c_1 `else` c_2 where x is a low-integrity variable and y is a high-integrity one. If c_1 and c_2 modify high-integrity variables this program will be rejected. In fact, an opponent manipulating the low-integrity variable x may force the program executing one of the two branches and gain control on the fact high integrity variables are updated via c_1 or c_2.

Consider now the case of the simplified *su* utility discussed in the introduction. Similarly to what we have seen above, an attacker might insert a wrong administrator password making the check fail. However, the program is in what we have called match-it-or-die form: the else branch is empty and nothing is executed after the if-then command. In the definition we have given, we obtain that the program diverges and the termination-insensitive notion of Integrity NI would consider the program secure.

A special case of integrity test is the one which involves the comparison between the on-the-fly hash of a low-integrity message and a trusted variable. Upon success, integrity of the untrusted data will be proved and it will be possible to assign it to a high-integrity variable. Consider the following program where y and z are trusted variables while x is a tainted one.

```
if (hash(x) = y) then
   z := x;
```

As in the software distribution example of the introduction, the assignment will be executed only if the contents of the variable x has been checked to be high-integrity, being its digest equal to the high integrity digest y, and is thus safe.

4.5 Security Type System

This section presents a security type system to statically analyze programs using a hash function and which derive integrity by equality test.

The proposed solution is based on the type system by Demange and Sands [29]. We only report typing rules for expressions and integrity check commands. All the remaining rules are as in [29].

We distinguish among four different types of values: small, big and their respective digests. Value types VT are S (small), B (big), $\mathsf{S}^{\#}$ (hash of a small) and $\mathsf{B}^{\#}$ (hash of a big) and are ranged over by vt. These value types are populated by the respective values:

$$\text{(v-small)}\ \frac{v \in Val_s}{\vdash v : \mathsf{S}} \quad \text{(v-big)}\ \frac{v \in Val_b}{\vdash v : \mathsf{B}} \quad \text{(v-hashs)}\ \frac{v \in Val_s^d}{\vdash v : \mathsf{S}^{\#}} \quad \text{(v-hashb)}\ \frac{v \in Val_b^d}{\vdash v : \mathsf{B}^{\#}}$$

Security types are of the form $\tau = \lambda\ell$, where $\lambda \in \{\mathsf{P}, \mathsf{D}\}$ distinguish between plain values and digests, while ℓ is the associated security level. A *security type environment* Δ is a mapping from variable to their security types. Given $\tau = \lambda\ell$ the two functions T and L give respectively its variable type and security level, i.e., $\mathsf{T}(\tau) = \lambda$ and $\mathsf{L}(\tau) = \ell$.

A subtype relation is defined over security types, it is meant to preserve λ, as we do not want to mix plain values with digests. Moreover plain big secret types are preserved by being removed from the relation: $\tau_1 \leq \tau_2$ if $\mathsf{T}(\tau_1) = \mathsf{T}(\tau_2) = \mathsf{D}$ and $\mathsf{L}(\tau_1) \sqsubseteq \mathsf{L}(\tau_2)$ or $\mathsf{T}(\tau_1) = \mathsf{T}(\tau_2) = \mathsf{P}$, $\mathsf{L}(\tau_i) \neq \mathsf{H_b}\ell_I$ and $\mathsf{L}(\tau_1) \sqsubseteq \mathsf{L}(\tau_2)$.

To prove that the type system enforces the security properties stated above it must be that plain big secret variables really store big values. To guarantee that small values are never assigned to big variables a conservative approach will be taken: every expression which involves an operator and returns a plain value lifts the confidentiality level of its result to $\mathsf{H_s}$, whenever it would be $\mathsf{H_b}$. The following function on security levels performs this upgrade:

$$\ell^{\sqcup} = \begin{cases} \mathsf{H_s}\ell_I & \text{if } \ell = \mathsf{H_b}\ell_I \\ \ell & \text{otherwise} \end{cases}$$

A variable x respects its security type $\tau = \lambda\ell$ with respect to a memory M if $\lambda = \mathsf{P}$ and $\vdash \mathsf{M}(x) : \mathsf{S}$ or $\vdash \mathsf{M}(x) : \mathsf{B}$ and similarly if $\lambda = \mathsf{D}$ then $\vdash \mathsf{M}(x) : \mathsf{S}^{\#}$ or $\vdash \mathsf{M}(x) : \mathsf{B}^{\#}$. A memory will be said to be well-formed if it respects the type of its variables, more precisely the expected properties are:

1. All variable respects their security types
2. Public variables do not store plain big values
3. Plain big secret variables only store big values.

The rules impose conditions also for the low-integrity variables. Note, indeed, that this work is not interested in spotting type flaws thus such statements do not affect the security result that is being proved here. These requirements are straightforward to formalize.

Definition 4.6. *A memory* M *is well-formed with respect to a security type environment* Δ *if*

1. $\Delta(x) = \mathsf{PH_s}\ell_I$ *implies* $\vdash \mathsf{M}(x) : vt$ *with* $vt \in \{\mathsf{S}, \mathsf{B}\}$
2. $\Delta(x) = \mathsf{PH_b}\ell_I$ *implies* $\vdash \mathsf{M}(x) : \mathsf{B}$
3. $\Delta(x) = \mathsf{PL}\ell_I$ *implies* $\vdash \mathsf{M}(x) : \mathsf{S}$
4. $\Delta(x) = \mathsf{D}\ell$ *implies* $\vdash \mathsf{M}(x) : vt$ *with* $vt \in \{\mathsf{S}^\#, \mathsf{B}^\#\}$

Expression typing rules are depicted in Table 4.1. Rules (var) and (sub) are standard. Rule (eq) types the equality test of two expressions requiring that they type the same τ and judging the result as a plain small value, being it a boolean. Rule (op) let any operator to be applied only to plain expressions, since in our symbolic model of the hash function no operation is defined on digests except for the equality test. These two rules use the $\ell^\sqcup$ function to promote the confidentiality level of their result to $\mathsf{H_s}$ whenever necessary, as already discussed above.

Hashes are typed either by (hash-b) or (hash-s). Rule (hash-b) performs a controlled declassification, the idea is that since the message is a plain big value its secrecy is not broken by releasing its digest. Indeed it can be proved that this does not break noninterference. The latter typing rule does nothing special and just preserve the security level of its argument.

The type system has to enforce a termination-insensitive noninterference for big secrets and termination-sensitive for small ones. This latter requirement can be achieved by some strong limitations on the while loops [61] or by accounting for the termination effect of a command [17, 56]. Demange and Sands type system [29] is built upon the work of Boudol and Castellani [17] following the latter approach.

A command type is a triple (w, t, f) where w and t are security levels and f is a termination flag ranging over $\downarrow$ and $\uparrow$ which respectively note that the command always terminates or that it could not terminate. A program is considered to be always terminating if it does not contain any while loop. The two flags are ordered as $\downarrow \sqsubseteq \uparrow$. A type judgement of the form $\Delta \vdash \mathsf{c} : (w, t, f)$ means that c does not assign to variables whose security level is lower than w (w is the writing effect of c), observing the termination of c gives information on variables at most at level t (t is the termination effect of c) and the termination behaviour is described by f.

Rules (int-test) and (int-hash) are new contributions of this work and implement the integrity verification tests discussed in Section 4.4. The former one let a trusted computation happens if the integrity of a tainted variable is proved by an equality test with an untainted one. The latter is pretty similar but is specific for the hash case and asserts that the intended original message, stored in the untrusted variable used to compute the on-the-fly digest, can be assigned to a trusted variable, whenever the check succeeds.

The else-branch in both cases must be the special command FAIL. It is a silent diverging while loop of the form $\mathsf{while\ true\ do\ skip}$. Requiring that each integrity

Table 4.1 Security Type System

Expressions

$$(\text{var})\ \frac{\Delta(x)=\tau}{\Delta\vdash x:\tau} \quad (\text{sub})\ \frac{\Delta\vdash e:\tau' \quad \tau'\leq\tau}{\Delta\vdash e:\tau} \quad (\text{eq})\ \frac{\Delta\vdash e:\tau \quad \Delta\vdash e_2:\tau \quad \mathsf{L}(\tau)=\ell}{\Delta\vdash e_1=e_2:\mathsf{P}\ell^{\sqcup}}$$

$$(\text{hash-b})\ \frac{\Delta\vdash x:\mathsf{PH_bH}}{\Delta\vdash \mathsf{hash}(x):\mathsf{DLH}} \quad (\text{hash-s})\ \frac{\Delta\vdash x:\mathsf{P}\ell \quad \ell\neq\mathsf{H_bH}}{\Delta\vdash \mathsf{hash}(x):\mathsf{D}\ell}$$

$$(\text{op})\ \frac{\Delta\vdash e_1:\mathsf{P}\ell \quad \Delta\vdash e_2:\mathsf{P}\ell}{\Delta\vdash e_1\ \mathsf{op}\ e_2:\mathsf{P}\ell^{\sqcup}}$$

Commands

$$(\text{skip})\quad \Delta\vdash\mathsf{skip}:(\mathsf{H_sL},\mathsf{LH},\downarrow) \quad (\text{assign})\ \frac{\Delta(x)=\tau \quad \Delta\vdash e:\tau}{\Delta\vdash x:=e:(\mathsf{L}(\tau),\mathsf{LH},\downarrow)}$$

$$(\text{if})\ \frac{\Delta\vdash e:\tau \quad \Delta\vdash \mathsf{c}_i:(w_i,t_i,f_i) \quad \mathsf{L}(\tau)\sqsubseteq w_i}{\Delta\vdash \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2:(w_1\sqcap w_2, t_1\sqcup t_2\sqcup\mathsf{L}(\tau), f_1\sqcup f_2)}$$

$$(\text{while})\ \frac{\Delta\vdash e:\tau \quad \Delta\vdash\mathsf{c}:(w,t,f) \quad \mathsf{L}(\tau)\sqsubseteq w \quad t=\mathsf{H_s}\ell_I\Rightarrow t\sqsubseteq w}{\Delta\vdash\mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}:(w,t\sqcup\mathsf{L}(\tau),\uparrow)}$$

$$(\text{seq-1})\ \frac{\Delta\vdash\mathsf{c}_i:(w_i,t_i,f_i) \quad t_1\sqsubseteq\mathsf{H_bL}\ \text{or}\ f_1=\downarrow}{\Delta\vdash\mathsf{c}_1;\mathsf{c}_2:(w_1\sqcap w_2,t_1\sqcup t_2,f_1\sqcup f_2)}$$

$$(\text{seq-2})\ \frac{\Delta\vdash\mathsf{c}_i:(w_i,t_i,f_i) \quad t_1=\mathsf{H_s}\ell_I \quad f_1=\uparrow \quad t_1\sqsubseteq w_2}{\Delta\vdash\mathsf{c}_1;\mathsf{c}_2:(w_1\sqcap w_2,t_1\sqcup t_2,\uparrow)}$$

$$(\text{int-test})\ \frac{\begin{array}{c}\Delta\vdash x:\tau \quad \Delta\vdash y:\tau' \quad \mathsf{T}(\tau)=\mathsf{T}(\tau')\\ \mathsf{L}(\tau)=\ell_C\mathsf{L} \quad \mathsf{L}(\tau')=\ell_C\mathsf{H} \qquad \ell_C\sqsubseteq_C\mathsf{H_b} \quad \Delta\vdash\mathsf{c}:(\ell_C\mathsf{H},t,f)\end{array}}{\Delta\vdash\mathsf{if}\ x=y\ \mathsf{then}\ \mathsf{c}\ \mathsf{else}\ \mathsf{FAIL}:(\ell_C\mathsf{H},t\sqcup\ell_C\mathsf{L},\uparrow)}$$

$$(\text{int-hash})\ \frac{\begin{array}{c}\Delta\vdash x:\mathsf{P}\ell_C\mathsf{L} \quad \Delta\vdash y:\mathsf{D}\ell_C\mathsf{H}\\ \Delta(z)=\mathsf{P}\ell_C\mathsf{H} \qquad \ell_C\sqsubseteq_C\mathsf{H_b} \quad \Delta\vdash\mathsf{c}:(\ell_C\mathsf{H},t,f)\end{array}}{\Delta\vdash\mathsf{if}\ \mathsf{hash}(x)=y\ \mathsf{then}\ z:=x;\mathsf{c}\ \mathsf{else}\ \mathsf{FAIL}:(\ell_C\mathsf{H},t\sqcup\ell_C\mathsf{L},\uparrow)}$$

test command executes such a program in case its guard condition is not satisfied assure that all the typed programs are in the 'match-it-or-die' form. In fact, upon failure no observable actions will be ever executed which is equivalent to say that, from an attacker point of view, no code would be run.

The confidentiality level of the variables involved in (int-test) and (int-hash) guard is constrained to be at most $\mathsf{H_b}$ to avoid brute-force attacks to small secrets. The command in the if-branch (c) is typed with a writing effect which has an high-integrity level, thus letting it to have write clearance to high-integrity variables, and the same confidentiality level as the two expressions compared in the guard. The termination effect of the overall command is constrained by the one of c and by $\ell_C\mathsf{L}$ since the test contains variables which are at most at that security level. Note that the two integrity tests would be potentially non terminating due to the FAIL branch.

Results The proposed type system enforces the security properties given in Section 4.3 and Section 4.4: If a program type checks then it is both secret-sensitive and integrity noninterferent.

Theorem 4.1. (**SSNI by typing**)
If $\Delta \vdash \mathsf{c} : (w, t, f)$ *then* c *satisfies Secret-sensitive NI.*

Theorem 4.2. (**Integrity NI by typing**)
If $\Delta \vdash \mathsf{c} : (w, t, f)$ *then* c *satisfies integrity NI.*

Formal proofs are in Appendix B.

4.6 Case Studies

In this section the case studies presented in the introduction are shown to type check and a new example will be introduced. Note that some syntactic sugars which were given introducing the examples has been removed here in order to show fully typed codes.

A simplified *su* command The first example is a simplified version of the *su* Unix utility. Let `root_shell` be a command which requires an high-integrity level to be computed, i.e., $\Delta \vdash \texttt{root_shell} : (\ell_C\mathsf{H}, t, f)$. The user entered password `t_pwd` will be deemed to be secret and low-integrity, i.e., $\Delta(\texttt{t_pwd}) = \mathsf{PH_bL}$. `root_passwd`, instead, stores the digest of the administrator password and it will be a high-integrity data since it is supposed to be stored in a write-protected file, let $\Delta(\texttt{root_passwd}) = \mathsf{D}\ell_C\mathsf{H}$. Note that the array notation have been replaced by a single variable, this does not affect the aim of the example which is proving the security of the Unix implementation of the password-based authentication mechanism.

```
trial := hash(t_pwd);
```

```
if (trial = root_passwd) then
    root_shell;
else
    FAIL;
```

The password is supposed to be strong thus it has been typed as a big secret. If such an assumption is removed, the code does not type, indeed the above program could be used to mount a brute-force attack on the password. The type system prevents such fact by requiring that the confidentiality level of the guard is at most a big secret in rule (int-test).

The confidentiality level of `root_passwd` could be either setted to L or $\mathsf{H_b}$. This models the fact that having strong passwords, they could be safely stored in a public location.

Let $\Delta(\texttt{trial}) = \mathsf{DH_bL}$, the expression $\mathsf{hash}(\texttt{pwd})$ is typed $\mathsf{DH_bL}$ by rule (hash-b) and the first assignment is then typed $(\mathsf{H_bL}, \mathsf{LH}, \downarrow)$ by (assign). The if branch is typed $(\mathsf{H_bH}, \mathsf{H_bL} \sqcup t, \uparrow)$ by (int-test): if $\ell_C = \mathsf{L}$ by sub-typing `root_passwd` will be typed $\mathsf{DH_bH}$ while if $\ell_C = \mathsf{H_b}$ nothing special is needed. The sequential composition of the two commands is then typed by (seq-1), indeed $\mathsf{LH} \sqsubseteq \mathsf{H_bL}$.

Software Distribution A software company distributes an application using different mirrors on the Internet. Having downloaded the program from one of the mirrors, a user will install the given binary only if its digest matches the one of the original application provided by the software company.

```
if (hash(my_blob.bin) = swdigest) then
    trusted_blob.bin := my_blob.bin;
    install := 1;
else
    FAIL;
```

Let `my_blob.bin` be the variable storing the downloaded binary, it is a low-integrity public variable, i.e., $\Delta(\texttt{my_blob.bin}) = \mathsf{PLL}$. The trusted digest given by the software company is stored in the `swdigest` variable which is a high-integrity one ($\Delta(\texttt{swdigest}) = \mathsf{DLH}$). The installation of the application is simulated by first saving the low-integrity binary in the trusted location `trusted_blob` ($\Delta(\texttt{trusted_blob}) = \mathsf{PLH}$) and then by assigning 1 to the high-integrity variable `install` ($\Delta(\texttt{install}) = \mathsf{PLH}$).

The if branch types $(\mathsf{LH}, \mathsf{LL}, \uparrow)$ by (int-hash), indeed all the requirements on variables are satisfied by letting $\ell_C = \mathsf{L}$ in the typing rule and, the assignment to `install` types $(\mathsf{LH}, \mathsf{LH}, \downarrow)$.

A new case study is now introduced, it shows a program which let a system administrator to manage the password file.

A simplified *passwd* This example presents a password update utility. It is a simplified version of the *passwd* Unix command where we require that only the administrator can perform such a task. This is due to the fact that the integrity of the password file must be preserved.

Three parameters are expected: the administrator password and the user old and new passwords.

```
root_trial := hash(t_root);
user_trial := hash(old);
if (root_trial = root_passwd) then
    if(user_trial = user_passwd) then
        user_passwd := hash(new);
    else
        FAIL;
else
    FAIL;
```

Variable `root_passwd` stores the digest of the root password while `user_passwd` the hash of the user one. These model, as in the first example, the needed portions of the password file ($\Delta(\texttt{root_passwd}) = \mathsf{DH_bH}$ and $\Delta(\texttt{user_passwd}) = \mathsf{DH_bH}$). The typed root password `t_root` is low-integrity, i.e., $\Delta(\texttt{t_root}) = \mathsf{PH_bL}$ as well as the `root_trial` variable used to store its digest ($\Delta(\texttt{root_trial}) = \mathsf{DH_bL}$). Similarly, $\Delta(\texttt{old}) = \mathsf{PH_bL}$ and $\Delta(\texttt{user_trial}) = \mathsf{DH_bL}$. The `new` variable which stores the new user password must be regarded as high-integrity ($\Delta(\texttt{new}) = \mathsf{PH_bH}$). This time the user input will be considered trusted since the intended user will be authenticated and only in that case the new password will be used. This is the only way to make the hash operator to type at a high-integrity level when storing the digest of the new password to `user_passwd`. In fact, there is no way to prove the integrity of a fresh new password by equality test.

The first two assignments type $(\mathsf{H_bL}, \mathsf{LH}, \downarrow)$ by (assign). The innermost if branch types $(\mathsf{H_bH}, \mathsf{H_bL}, \uparrow)$ by (int-test): variable `user_passwd` types $\mathsf{DH_bH}$ and `user_trial` $\mathsf{DH_bL}$, the assignment to `user_passwd` types $(\mathsf{H_bH}, \mathsf{LH}, \downarrow)$ by (assign) and the hash expression is typed by DLH by (hash-b) and promoted by subtyping to $\mathsf{DH_bH}$. In a similar way the main if branch is again typed by (int-test) obtaining $(\mathsf{H_bH}, \mathsf{H_bL}, \uparrow)$.

The whole program is thus typed $(\mathsf{H_bH}, \mathsf{H_bL}, \uparrow)$ by (seq-1).

4.7 Related Works

A secure usage of hash function in the setting of information flow security has been already explored by Volpano in [60]. There are, however, many difference with respect to the security properties presented above. First, Volpano does not account for data integrity and, consequently, integrity checks, which is one of the major contributions of our work. On the other side, we limit our study to a symbolic

treatment of hash functions, distinguishing between two different kind of secrets, while Volpano aims at a computational result.

The distinction between big and small secrets and the two different bisimilarity notions which have to be applied to protect them is completely inspired by Demange and Sands [29].

4.8 Conclusions

We have studied the security of programs that use hash functions in the setting of information flow security. We have shown how to prove data integrity via equality tests between a low and a high-integrity variable.

We have extended secret-sensitive noninterference to guarantee that leaks via the hash operator could not occur: the intuition is that the digest of a big enough secret s would not be subject to a brute-force attack and so releasing it to the public will not break the confidentiality of s.

A classical noninterference property has been instead used to check that secure programs do not taint high-integrity data. Equality tests to enforce data integrity have been introduced. The equality of a tainted variable with a trusted one is regarded as an evidence of the fact that the value stored in the untrusted variable is indeed untainted. This kind of integrity proof is widely adopted in real applications and this chapter gives the tools to reason about its security.

Future Work Hash functions could be used in commitment protocols. Suppose Anna challenges Bruno to solve a problem and claims she has solved it. To prove her statement Anna takes the answer and appends it to a random secret *nonce*, she then sends the hash of such message to Bruno. When the challenge finishes or when Bruno gives up, Anna has to reveal him the secret nonce thus he can check that the correct answer was sent in the first step of the process.

Formally studying this scenario in an information flow setting would be challenging. Some form of declassification would be allowed since at certain point in time the secret nonce has to be released. When the random will be downgraded then the digest could not be thought to protect Anna's answer anymore. Analyzing the security of this problem will require an interaction of a declassification mechanism, suitable to reason about the *when* dimension of downgrading [55], with the solution presented here for the secure usage of hash functions.

To guarantee that small values are never assigned to big variables we have taken the very conservative approach of forbidding expressions to return a big secret. In practice, this might be relaxed by adding some data-flow analysis in order to track values derived from big secrets. For example, the xor of two different big secrets might be considered a big secret, but the xor of two equal big secrets is 0. We intend to investigate this issue more in detail in the next future.

Memory equivalence based on patterns and the notion of *static equivalence* in process calculi seems to be strongly related. Big secrets resemble the notion of bound names which can be α-converted preserving equality patterns. We leave as a future work the intriguing comparison between the two formal notions.

5

Type checking PIN Verification APIs

This chapter presents the first of the case studies of the thesis. It considers some known attacks on the ATM PIN verification framework, based on weaknesses of the underlying security API for the tamper-resistant Hardware Security Modules used in the network. Most of these attacks do not directly reveal the PIN. Instead, they involve the attacker calling the API commands repeatedly with slightly different parameter values, and using the results (which may be error codes) to deduce the value of the PIN: these are information leakage API attacks and will be studied using the machinery of language based information flow security introduced in the previous chapters.

This API is here implemented using an imperative language with cryptographic primitives, and it will be shown how its flaws are captured by a notion of robustness that extends the one of Myers, Sabelfeld and Zdancewic [48] to the cryptographic setting. The chapater introduces a type system to assure integrity and to preserve confidentiality via randomized and non-randomized encryptions. An improved API is also proposed and it is shown type-checkable proving its security.

5.1 Introduction

In the international ATM (cash machine) network, users' personal identification numbers (PINs) have to be sent encrypted from the *PIN Entry Device* (PED) on the terminal to the issuing bank for checking. The PIN is encrypted in the PED under a key shared with the server or *switch* to which the ATM is connected. The PIN is then decrypted and re-encrypted under the key for an adjacent switch, to which it is forwarded. Eventually, the PIN reaches the issuing bank, by which time it may have been decrypted and re-encrypted several times. The issuing bank has no direct control over what happens in the intermediate switches, so to establish trust, the international standard ISO 9564 (ANSI X9.8) stipulates the use of tamper proof cryptographic *Hardware Security Modules* (HSMs). These HSMs protect the PIN encryption keys, and in the issuing banks, they also protect the *PIN Derivation Keys* (PDKs) used to derive the customer's PIN from non-secret validation data such as their *Personal Account Number* (PAN). All encryption, decryption and checking of PINs is carried out inside the HSMs, which have a carefully designed API provid-

ing functions for *translation* (i.e., decryption under one key and encryption under another one) and *verification* (i.e., PIN correctness checking). The API has to be designed so that even if an attacker obtains access to the host machine connected to the HSM, he cannot abuse the API to obtain PINs.

In the last few years, several attacks have been published on the APIs in use in these systems [11, 14, 24]. Very few of these attacks directly reveal the PIN. Instead, they involve the attacker calling the API commands repeatedly with slightly different parameter values, and using the results (which may be error codes) to deduce the value of the PIN. High-profile instances of many PINs being stolen from hacked switches has increased interest in the problem [38, 50]. PIN recovery attacks have been formally analysed, but previously the approach was to take a particular API configuration and measure its vulnerability to combinations of known attacks [58]. Other researchers have proposed improvements to the system to blunt the attacks, but these suggestions address only some attacks, and are "intended to stimulate further research" [44]. We take a step in that direction, using the techniques of language-based security [53].

Looking at the code for the current PIN processing APIs, one can immediately see that the current API functions allow an 'information flow' from the high security PIN to the low security result. However, it is a necessary feature of the verification function to reveal whether the encrypted PIN is correct or not, so some flow is inevitable. The language-based security literature has a technique for dealing with this: a 'declassification policy' [55], whereby we decide in advance that a certain flow is permitted. The problem is that an intruder can often manipulate input data in order to declassify data in a way we did not intend. Again there is a technique for this: 'robust declassification' [47], whereby we disallow 'low integrity' data, which might have been manipulated by the attacker, to affect what can be declassified. However, the functionality of our PIN verification function requires the result to depend on low-integrity data. The solution in the literature is 'endorsement' [48], where we accept that certain low integrity data is allowed to affect the result. However, in our examples, endorsing the low integrity data permits several known attacks.

From this starting point, we propose an extension to the language-based security framework for robust declassification to allow the integrity of inputs to be assured cryptographically by using *Message Authentication Codes* (MACs). We present semantics and a type system for our model, and show how it allows us to formally analyse possible improvements to PIN processing APIs. We believe our modelling of cryptographically assured integrity to be a novel contribution to language based security theory. In addition, we give new proposals for improving the PIN processing system.

Structure of the chapter This chaperter first illustrates the case study, the PIN verification command (Section 5.2). Some notions of language based security related

to declassification are reviewed in Section 5.3, then the modelling of cryptographic primitives, and in particular MACs for assuring integrity, and the flaws making PIN verification fails to be robust are discussed (Section 5.4). The type system is presented in Section 5.5 and a MAC-based improved API is type-checked in Section 5.6. Section 5.7 concludes.

5.2 The Case Study

In the introduction we have observed how PINs travelling along the network have to be decrypted and re-encrypted under a different key, using a *translation* API. Then, when the PIN reaches the issuing bank, its correspondence with the *validation data*[1] is checked via a *verification* API. We focus on this latter API, which we call PIN_V: it checks the equality of the actual *user* PIN and the *trial* PIN inserted at the ATM and returns the result of the verification or an error code. The former PIN is derived through the PIN derivation key *pdk*, from the public data *offset*, *vdata*, *dectab* (see below), while the latter comes encrypted under key k as *EPB* (Encrypted PIN block). Note that the two keys are pre-loaded in the HSM and are never exposed to the untrusted external environment. In this example we will assume only one key of each type (k and *pdk*) is used. The API, specified in Table 5.1, behaves as follows:

Table 5.1 The PIN verification API.

```
PIN_V(PAN, EPB, len, offset, vdata, dectab) {
    x1 := enc_pdk(vdata);
    x2 := left(len, x1);
    x3 := decimalize(dectab, x2);
    x4 := sum_mod10(x3, offset);
    x5 := dec_k(EPB);
    x6 := fcheck(x5);
    if (x6 == "FAIL") then return("format error");
    if (x4 == x6) then return("PIN is correct");
                  else return("PIN is wrong")}
```

-The user PIN of length *len* is obtained by encrypting validation data *vdata* with the PIN derivation key *pdk* (x_1), taking the first *len* hexadecimal digits (x_2), decimalising through *dectab* (x_3), and digit-wise summing modulo 10 the *offset* (x_4). More precisely, the outcome of the encryption x_1 is a 16 hexadecimal digit string and decimalize is a function that associates to each possible hexadecimal digit (of its second input) a decimal one as specified by its first parameter (*dectab*). The obtained decimalised value x_3 is the 'natural' PIN assigned by the issuing bank to the user. If the user wants to choose her own PIN, an *offset* is calculated by digit-wise subtracting (modulo 10) the natural PIN from the user-selected one. Thus, to get the user PIN the *offset* is summed to the natural PIN (x_4).

-The trial PIN is recovered by decrypting *EPB* with key k (x_5), and extracting the PIN by removing the random padding and checking the PIN is correctly formatted (x_6). For some PIN block formats the PAN is required for the extraction. However we give here the algorithm for extracting the PIN from an ISO1 block,

[1] The value of this parameter is up to the issuing bank. It is typically an encoding of the user PAN and possibly other 'public' data, such as the card expiration date or the customer name.

where encryption with random padding is used. In our model this amounts to projecting the first element of a pair consisting of a PIN and random padding. We will discuss other block formats and how to model them later.

-Finally, the equality of the user PIN (x_4) and the trial PIN (x_6) is returned.

Example 2. *Let len=4, offset=4732, dectab = 9753108642543210, this last parameter encoding this mapping:* $0 \to 9, 1 \to 7, 2 \to 5, 3 \to 3, 4 \to 1, 5 \to 0, 6 \to 8, 7 \to 6, 8 \to 4, 9 \to 2, A \to 5, B \to 4, C \to 3, D \to 2, E \to 1, F \to 0$. *Let also* $x_1 = \mathsf{enc}_{pdk}(vdata) = \text{A47295FDE32A48B1}$.

Then, $x_2 = \mathsf{left}(4, A47295FDE32A48B1) = \text{A472}$, $x_3 = \mathsf{decimalize}(dectab, A472) = 5165$, and $x_4 = \mathsf{sum_mod10}(5165, 4732) = 9897$. *This completes the user* PIN *recovery part. Let now* $(9897, r)$ *denote* PIN 9897 *correctly formatted and padded with a random r, as required by ISO1 (recall that we are omitting details about other* PIN *formats for the moment) and let us assume that EPB* $= \{|9897, r|\}_k$. *We thus have:* $x_5 = \mathsf{dec}_k(\{|9897, r|\}_k) = (9897, r)$, and $x_6 = \mathsf{fcheck}(\mathsf{9897}, \mathsf{r}) = 9897$. *Finally, since* x_6 *is different from "FAIL" and* $x_4 = x_6$ *the API returns "PIN is correct".* □

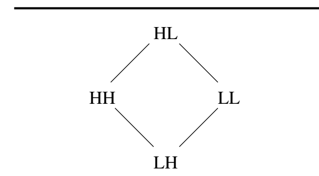
The given specification is an abstraction and a simplification of real PIN verification code, i.e., PIN_V corresponds to `Encrypted_PIN_Verify` of [39] simplified by omitting some parameters for alternative PIN extraction methods. We only model the IBM 3624 PIN calculation method with offset, but this is not limiting as the other PIN calculation methods can be similarly specified and analysed.

5.3 Basic Language and Security

The language adopted in this chapter is the same as the one presented in Chapter 2. Given that we are interested in analysing security APIs, which we assume to be executed on trusted hardware with no multi-threading, we adopt a standard big-step semantics similar to that of Volpano et al. [62].

$\langle \mathsf{M}, \mathsf{c} \rangle \Rightarrow \mathsf{M}'$ denotes the execution of a command c in a memory M, resulting in a new memory M'. For example, $\langle \mathsf{M}, x := e \rangle \Rightarrow \mathsf{M}[x \mapsto v]$ if $e \downarrow^{\mathsf{M}} v$. The complete semantics is reported in Table C.2 of Appendix C.

Security A *security environment* Γ maps each variable to a level of *confidentiality* and *integrity*. To keep the setting simple, we limit our attention to two possible levels: *high* (H) and *low* (L). For any given confidentiality (integrity) levels ℓ_1, ℓ_2, we write $\ell_1 \sqsubseteq_C \ell_2$ ($\ell_1 \sqsubseteq_I \ell_2$) to denote that ℓ_1 is as restrictive or less restrictive than ℓ_2. In particular, low-confidentiality data may be used more liberally than high-confidentiality ones, thus in this case $\mathsf{L} \sqsubseteq_C \mathsf{H}$; dually low-integrity data must be treated more carefully than high-integrity ones, giving the countervariant relation $\mathsf{H} \sqsubseteq_I \mathsf{L}$. We consider the product of the above confidentiality and integrity lattices, and we denote



with $\sqsubseteq$ the component-wise application of $\sqsubseteq_C$ and $\sqsubseteq_I$ (on the right).

Noninterference is now rephrased to this simpler big-step execution model. Two configurations are indistinguishable at level ℓ if they lead to ℓ-equivalent outputs (notice that memories are compared using the equivalence introduced in Chapter 2).

Definition 5.1 (Indistinguishability)**.** *Two configurations are* indistinguishable*, written* $\langle \mathsf{M}_1, \mathsf{c}\rangle =_\ell \langle \mathsf{M}_2, \mathsf{c}\rangle$*, if whenever* $\langle \mathsf{M}_1, \mathsf{c}\rangle \Rightarrow \mathsf{M}'_1$ *and* $\langle \mathsf{M}_2, \mathsf{c}\rangle \Rightarrow \mathsf{M}'_2$ *then* $\mathsf{M}'_1 =_\ell \mathsf{M}'_2$*.*

Noninterference requires that data from one level should never interfere with lower and incomparable levels. Intuitively, command c satisfies noninterference if, fixed a level ℓ, two indistinguishable memories remain indistinguishable even after c has been executed.

Definition 5.2 (Noninterference)**.** *A command* c *satisfies* noninterference *if* $\forall\, \ell, \mathsf{M}_1, \mathsf{M}_2$ *we have that* $\mathsf{M}_1 =_\ell \mathsf{M}_2$ *implies* $\langle \mathsf{M}_1, \mathsf{c}\rangle =_\ell \langle \mathsf{M}_2, \mathsf{c}\rangle$*.*

To see how this captures confidentiality/integrity leakages, consider the cases $\ell = \mathsf{LL}$ and $\ell = \mathsf{HH}$. The former says that high-confidentiality data cannot be leaked to low-confidentiality levels LL, LH. Dually, the latter case states that low-integrity data cannot corrupt high-integrity ones HH, LH.

Noninterference formalizes full security, with no leakage of confidential information or corruption of high-integrity data. The property proposed by Myers, Sabelfeld and Zdancewic (MSZ) in [48], called *robustness*, admits some form of *declassification* (or downgrading) of confidential data, but requires that attackers cannot influence the secret information declassified by a program c. In our case study of Section 5.2, $\mathsf{PIN_V}$ returns the correctness of the typed PIN which is a one-bit leak of information about a secret datum. Thus, the API is intended to declassify some secret information. Robustness will allow us to check that attackers cannot abuse such a declassification and gain more information than intended.

Consider a pair of memories $\mathsf{M}_1, \mathsf{M}_2$ which are not distinguishable by an intruder, i.e., $\mathsf{M}_1 =_{\mathsf{LL}} \mathsf{M}_2$. The execution of c on these memories may leak confidential information violating noninterference, i.e., $\langle \mathsf{M}_1, \mathsf{c}\rangle \neq_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}\rangle$. Robustness states that if the behaviour of the command c is not distinguishable on M_1 and M_2 then the same must happen for every pair of memories M'_1, M'_2 the attacker may obtain starting from M_1, M_2. To characterize these memories note that: (i) they are still indistinguishable by the intruder, i.e., $\mathsf{M}'_1 =_{\mathsf{LL}} \mathsf{M}'_2$, as he is deterministic and starts from indistinguishable memories; (ii) they only differ from the initial ones in the low-integrity part, i.e., $\mathsf{M}_1 =_{\mathsf{HH}} \mathsf{M}'_1$, $\mathsf{M}_2 =_{\mathsf{HH}} \mathsf{M}'_2$, given that only low-integrity variables can be modified by intruders.

As done by MSZ, we require that attackers start from terminating configurations to avoid they 'incompetently' self-corrupt their observations. This is done via a notion of *strongly indistinguishability*, written $\langle \mathsf{M}_1, \mathsf{c}\rangle \cong_\ell \langle \mathsf{M}_2, \mathsf{c}\rangle$, requiring that both

configurations terminate, i.e., $\langle \mathsf{M}_1, \mathsf{c}\rangle \Rightarrow \mathsf{M}_1'$, $\langle \mathsf{M}_2, \mathsf{c}\rangle \Rightarrow \mathsf{M}_2'$, and resulting memories are indistinguishable, i.e., $\mathsf{M}_1' =_\ell \mathsf{M}_2'$.

Definition 5.3 (Robustness)**.** *Command* c *is robust if* $\forall \mathsf{M}_1, \mathsf{M}_2, \mathsf{M}_1', \mathsf{M}_2'$ *s.t.* $\mathsf{M}_1 =_{\mathsf{LL}} \mathsf{M}_2$, $\mathsf{M}_1' =_{\mathsf{LL}} \mathsf{M}_2'$, $\mathsf{M}_1 =_{\mathsf{HH}} \mathsf{M}_1'$, $\mathsf{M}_2 =_{\mathsf{HH}} \mathsf{M}_2'$, *it holds* $\langle \mathsf{M}_1, \mathsf{c}\rangle \cong_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}\rangle$ *implies* $\langle \mathsf{M}_1', \mathsf{c}\rangle =_{\mathsf{LL}} \langle \mathsf{M}_2', \mathsf{c}\rangle$.

This notion is a novel simplification of the one of MSZ, who allowed a malicious user to insert untrusted code at given points in the trusted code. In security APIs this is not permitted: an attacker can call a security API any number of times with different parameters but he can never inject code inside it, moreover, no intermediate result will be made public by the API. This leads to a simpler model where attackers can only act before and after each security API invocation and, thus, there is no need to make their code explicit. Memory manipulations performed by attackers are covered by considering all the pairs of indistinguishable memories which only differ in the low-integrity part with respect to the initial ones.

Example 3. *We will write* x_ℓ *to denote a variable of level* ℓ*. Consider a program* P *in which variable* x_{LL} *stores the user entered PIN,* y_{HH} *contains the real one, and* $z_{\mathsf{LL}} := (x_{\mathsf{LL}} = y_{\mathsf{HH}})$ *, i.e.,* z_{LL} *says if the entered PIN is the correct one or not. This program does not obey to noninterference since the result of the equality test depends on secret data and it is assigned to a public variable, moreover it is not robust. To see this latter fact, consider memories* M_1 *and* M_2 *such that* $\mathsf{M}_1(x_{\mathsf{LL}}) = \mathsf{M}_2(x_{\mathsf{LL}}) = 1111$*, thus* $\mathsf{M}_1 =_{\mathsf{LL}} \mathsf{M}_2$*, and* $\mathsf{M}_1(y_{\mathsf{HH}}) = 1234$ *while* $\mathsf{M}_2(y_{\mathsf{HH}}) = 5678$*. Now assume the attacker generates two memories* M_1' *and* M_2' *where* $\mathsf{M}_1'(y_{\mathsf{HH}}) = \mathsf{M}_1'(x_{\mathsf{LL}}) = \mathsf{M}_2'(x_{\mathsf{LL}}) = 1234$*, thus* $\mathsf{M}_1' =_{\mathsf{LL}} \mathsf{M}_2'$*, and* $\mathsf{M}_2'(y_{\mathsf{HH}}) = 5678$*. It clearly holds that* $\mathsf{M}_1 =_{\mathsf{HH}} \mathsf{M}_1'$ *and* $\mathsf{M}_2 =_{\mathsf{HH}} \mathsf{M}_2'$ *but the execution of* P *in the first two memories leads to indistinguishable results in* z_{LL}*,* false/false*, thus* $\langle \mathsf{M}_1, P\rangle \cong_{\mathsf{LL}} \langle \mathsf{M}_2, P\rangle$*, while for the second ones we get* true/false*, and so* $\langle \mathsf{M}_1', P\rangle \not\cong_{\mathsf{LL}} \langle \mathsf{M}_2', P\rangle$*. Intuitively, the attacker has 'guessed' one of the secret PINs and the program is revealing that his guess is correct: the attacker can tamper with the declassification mechanism via variable* x_{LL}*.* □

5.4 Cryptographic primitives

In order to model our API case-study, we now extend the language given in Chapter 2 with confounder generation, (symmetric) cryptography, Message Authentication Codes (MACs), pairing and projection. These are introduced as special expressions (as already done in Chapter 3 and 4) ranged over by e:

$$e ::= \ldots \mid \mathsf{new}() \mid \mathsf{enc}_x(e) \mid \mathsf{dec}_x(e) \mid \mathsf{mac}_x(e) \mid \mathsf{pair}(e_1, e_2) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e)$$

working on values $v ::= n \mid \{\!|v|\!\}_k \mid \langle v\rangle_k \mid (v_1, v_2)$ where $n ::= \perp \mid r \mid k \mid \ldots$ is an atomic name which can be the special value $\perp$ representing failure, a confounder $r \in C$, a cryptographic key $k \in \mathcal{K}$, or any other value used in a program, e.g.,

Booleans and integers. On these atomic values we build cryptographic values and pairs ranged over by v: more specifically, $\{\!|v|\!\}_k$ and $\langle v\rangle_k$ respectively represent the encryption and the MAC of v using k as key, and (v_1, v_2) is a pair of values. We will often omit the brackets to simplify the notation, e.g., we will write $\{\!|v_1, v_2|\!\}_k$ to indicate $\{\!|(v_1, v_2)|\!\}_k$.

Based on this set of values we can easily give the semantics of the special expressions mentioned above. For example, we have $\mathsf{enc}_x(e) \downarrow^{\mathsf{M}} \{\!|v|\!\}_k$ whenever $e \downarrow^{\mathsf{M}} v$ and $x \downarrow^{\mathsf{M}} k$. Moreover, $\mathsf{dec}_x(e') \downarrow^{\mathsf{M}} v$ if $e' \downarrow^{\mathsf{M}} \{\!|v|\!\}_k$ and $x \downarrow^{\mathsf{M}} k$; otherwise $\mathsf{dec}_x(e') \downarrow^{\mathsf{M}} \bot$, representing failure, and analogously for the other expressions. Confounder generation $\mathsf{new}() \downarrow^{\mathsf{M}} r$ extracts a 'random' value, noted $r \leftarrow C$, from a set of values C. In real cryptosystems, the probability of extracting the same random confounder is assumed to be negligible, if the set is suitably large, so we symbolically model random extraction by requiring that extracted values are always different. Thus, $r \leftarrow C$ can be thought as extracting the first element of an infinite stream of confounders and removing it from the list so that it cannot be reused. More formally, we assume that two extractions $r, r' \leftarrow C$ are such that $r \neq r'$. Moreover, similarly to [1, 2], we assume C to be disjoint from the set of atomic names used in programs. Semantics of expressions is summarized in Table C.3 of Appendix.

To guarantee a safe use of cryptography we also assume that every expression e different from enc, dec, mac, pair and every Boolean expression except the equality test: (i) always fails when applied to special values such as confounders, keys, ciphertexts, and MACs (even when occurring in pairs), producing a $\bot$ and (ii) never produces those values. This is important to avoid "magic" expressions which encrypt/decrypt/MAC messages without knowing the key like, e.g., $\mathsf{magicdecrypt}(e) \downarrow^{\mathsf{M}} v$ when $e \downarrow^{\mathsf{M}} \{\!|v|\!\}_n$. However, we permit equality checks as they allow the intruder to track if the same encryption appears twice, as occurs in traffic analysis.

5.4.1 Security with cryptography

We now rephrase the notions of noninterference and robustness in order to accommodate cryptographic primitives. In doing so, we extend the notions presented in Chapter 3 in a non-trivial way by (i) accounting for integrity primitives such as MACs and (ii) removing the assumption that cryptography is always randomized via confounders. This latter extension is motivated by the fact that our case study does not always adopt randomization in cryptographic messages, so we need to leave the programmer decide whether or not to insert confounders in encryptions. Notice that non-randomized encrypted messages are subject to traffic analysis, thus confidentiality of those messages cannot be guaranteed except in special cases that we will discuss in detail.

In order to extend the indistinguishability notion of definition 5.1 to cryptographic primitives we assume that the level of keys is known a-priori. We believe this is a fair assumption, since in practice it is fundamental to have information about a key's security before using it. Since we have only defined symmetric key

cryptography we only need *trusted* (of level HH) and *untrusted* keys (of level LL). The former are only known by the APIs while the latter can be used by the attackers. This is achieved by partitioning the set of keys $\mathcal{K}$ into $\mathcal{K}_{\mathsf{HH}}$ and $\mathcal{K}_{\mathsf{LL}}$.

Patterns and indistinguishability As the intruder cannot access (or generate, in case of MACs) cryptographic values protected by HH keys, one might state that such values are indistinguishable. However, an attacker might detect occurrences of the same cryptographic values in different parts of the memory, as occurs in some traffic analysis attacks.

Example 4. *Consider the program $z_{\mathsf{LL}} := (x_{\mathsf{LL}} = y_{\mathsf{LL}})$ which writes the result of the equality test between x_{LL} and y_{LL} into z_{LL}. Given that it only works on LL variables it can be considered as an intruder-controlled program.*

Consider the memories M_1 and M_2 (below) where the initial value of z_{LL} is not of interest, and where $k \in \mathcal{K}_{\mathsf{HH}}$.

M_1			M_2		
x_{LL}	:	$\{\|1234\|\}_k$	x_{LL}	:	$\{\|9999\|\}_k$
y_{LL}	:	$\{\|1234\|\}_k$	y_{LL}	:	$\{\|5678\|\}_k$

At first sight, one may conclude the two memories are indistinguishable as an attacker cannot distinguish $\{\|1234\|\}_k$ from $\{\|9999\|\}_k$ and $\{\|1234\|\}_k$ from $\{\|5678\|\}_k$. However, running the above intruder-program on these memories, we obtain two new memories M'_1 and M'_2 in which x_{LL} and y_{LL} remain unchanged, but $z_{\mathsf{LL}} = \mathsf{true}$ in the first one and $z_{\mathsf{LL}} = \mathsf{false}$ in the other one, i.e., M'_1 and M'_2 differ. The intruder has in fact detected the presence of two equal ciphertexts in the first memory which allows him to distinguish the initial memories M_1 and M_2. □

This ability of the attacker to find equal cryptographic values in the memories is formalized through the notion of *pattern* inspired by Abadi et al. [2, 3] and already adopted for modelling noninterference in Chapter 3. Note that we adopt patterns to obtain a realistic notion of distinguishability of ciphertexts in a symbolic model, and not to address computational soundness as is done in other works [2, 3, 4].

Patterns, ranged over by p, extend values as follows: $p ::= v \mid \Box_v$ the new symbol $\Box_v$ representing messages encrypted with a key not available at the observation level ℓ. More precisely, we define a function $\mathsf{p}_\ell(v)$ which takes a value and produces the corresponding pattern by replacing all the encrypted values v protected by keys of level $\ell' \not\sqsubseteq \ell$ with $\Box_v$, and leaving all the other values unchanged. For example, for $\{\|1234\|\}_k$ in the example above we have $\mathsf{p}_{\mathsf{LL}}(\{\|1234\|\}_k) = \Box_{\{\|1234\|\}_k}$ while $\mathsf{p}_{\mathsf{HH}}(\{\|1234\|\}_k) = \{\|1234\|\}_k$. Notice that, in $\Box_v$, v is the whole (inaccessible) encrypted value, instead of just a confounder as used in Chapter 3 and previous works [2, 3, 32, 42]. In these cases, each new encryption includes a fresh confounder which can be used as a 'representative' of the whole encrypted value. Here we cannot adopt this solution since our confounders are optional. To disregard the values of confounders, once the corresponding ciphertext has been accessed (i.e., when knowing the key), we abstract them as the constant $\perp$.

Given a bijection $\rho : \Box_v \mapsto \Box_v$, that we call *hidden values substitution*, we write $p\rho$ to denote the result of applying ρ to the pattern p, and we write $\mathsf{M}\rho$ to denote the memory in which ρ has been applied to all the patterns of M, i.e., $\mathsf{M}\rho\ (x) = \mathsf{M}(x)\ \rho$. On hidden values substitutions we always require that keys are correctly mapped, i.e, formally $\rho(\Box_{\{\!|v|\!\}_k}) = \Box_{\{\!|v'|\!\}_k}$. We write $\mathcal{K}_{\sqsubseteq \ell}$ to denote the set of keys at or below ℓ, i.e., $\cup_{\ell' \sqsubseteq \ell} \mathcal{K}_{\ell'}$.

Definition 5.4. *Let $\mathsf{p}_\ell(v) : v \mapsto p$ be recursively defined as follows:*

$$\begin{array}{rcll} \mathsf{p}_\ell(n) & = & n & n \text{ not a confounder} \\ \mathsf{p}_\ell(r) & = & \bot & \\ \mathsf{p}_\ell((v_1, v_2)) & = & (\mathsf{p}_\ell(v_1), \mathsf{p}_\ell(v_2)) & \\ \mathsf{p}_\ell(\langle v \rangle_k) & = & \langle \mathsf{p}_\ell(v) \rangle_k & \\ \mathsf{p}_\ell(\{\!|v|\!\}_k) & = & \left\{ \begin{array}{ll} \Box_{\{\!|v|\!\}_k} & \text{if } k \notin \mathcal{K}_{\sqsubseteq \ell} \\ \{\!|\mathsf{p}_\ell(v)|\!\}_k & \text{otherwise} \end{array} \right. \end{array}$$

Let $\mathsf{p}_\ell(\mathsf{M})$ denote the restriction of memory M to level ℓ in which all of the values v have been substituted by p_ℓ, formally $\mathsf{p}_\ell(\mathsf{M}) = \mathsf{p}_\ell \circ \mathsf{M}|_\ell$. Two memories M_1 and M_2 are indistinguishable at level ℓ, written $\mathsf{M}_1 \approx_\ell \mathsf{M}_2$, if there exists a hidden values substitution ρ such that $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\ \rho$.

Example 5. *Consider again the two memories M_1 and M_2 of example 4. We observed that they differ at level LL because of the presence of two equal ciphertexts in M_1. Since $k \in \mathcal{K}_{\mathsf{HH}}$ we obtain the values of x_{LL} and y_{LL} below. Now it is impossible to find a hidden values substitution ρ mapping the first memory to the second, as $\Box_{\{\!|1234|\!\}_k}$ cannot be mapped both to $\Box_{\{\!|9999|\!\}_k}$ and $\Box_{\{\!|5678|\!\}_k}$.*

$\mathsf{p}_{\mathsf{LL}}(\mathsf{M}_1)$	$\mathsf{p}_{\mathsf{LL}}(\mathsf{M}_2)$
$x_{\mathsf{LL}} : \Box_{\{\!|1234|\!\}_k}$	$x_{\mathsf{LL}} : \Box_{\{\!|9999|\!\}_k}$
$y_{\mathsf{LL}} : \Box_{\{\!|1234|\!\}_k}$	$y_{\mathsf{LL}} : \Box_{\{\!|5678|\!\}_k}$

Thus we conclude that $\mathsf{M}_1 \not\approx_{\mathsf{LL}} \mathsf{M}_2$. If, instead, $\mathsf{M}_1(y_{\mathsf{LL}})$ were, e.g., $\{\!|2222|\!\}_k$ we might use $\rho = [\Box_{\{\!|1234|\!\}_k} \mapsto \Box_{\{\!|9999|\!\}_k}, \Box_{\{\!|2222|\!\}_k} \mapsto \Box_{\{\!|5678|\!\}_k}]$ obtaining $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\rho$ and thus $\mathsf{M}_1 \approx_{\mathsf{LL}} \mathsf{M}_2$. $\Box$

Notice that we do not extend our notion of hidden values to MACs, i.e., we assume that all messages inside MACs are public, and hence equivalence between MACs can be defined on these messages. We will use encryption only for secrecy, and MACs only for authentication.

Noninterference and robustness We now reconsider the security notions of Section 5.3 in the new cryptographic setting. The idea is to keep the same definitions and substitute $=_\ell$ with $\approx_\ell$ everywhere. For example, the notion of (weak) indistinguishability of executions (definition 5.1), denoted $\langle \mathsf{M}, \mathsf{c} \rangle \approx_\ell \langle \mathsf{M}', \mathsf{c} \rangle$, is just rephrased by stating that whenever $\langle \mathsf{M}_1, \mathsf{c} \rangle \Rightarrow \mathsf{M}'_1$ and $\langle \mathsf{M}_2, \mathsf{c} \rangle \Rightarrow \mathsf{M}'_2$ then $\mathsf{M}'_1 \approx_\ell \mathsf{M}'_2$.

We need to be careful that memories do not leak cryptographic keys, i.e., that keys disclosed at level ℓ are all of that level or below. We should also discipline the quantification over all possible memories so that variables intended to contain keys really do contain keys. We prefer to postpone these aspects to Section 5.5, after types have been defined. In fact, they are easily achieved by requiring that variables of key-types are only populated by key-values, as the intuition suggests, and that memories are well-formed with respect to types.

In the next section we illustrate a known practical attack on the PIN verification API and we then show that it can be formally captured by the non-robustness of the API.

5.4.2 Formal analysis of a PIN_V API attack

In this section we show how the lack of integrity of the API parameters can be exploited to mount a real attack leaking the PIN.

Let us consider the case study of Section 5.2 and in particular the code of the PIN verification API shown in Table 5.1. Let us now concentrate on two specific parameters, the *dectab* and the *offset*, which are used to calculate the values of x_3 and x_4, respectively. A possible attack on the system works by iterating the following two steps, until the whole PIN is recovered[14]:

1. The intruder picks a decimal digit d, changes the *dectab* function so that values previously mapped to d now map to $d + 1 \bmod 10$, and then checks whether the system still returns *"PIN is correct"*. Depending on this, the intruder discovers whether or not digit d is present in the user 'natural' PIN contained in x_3, thus extracting information on the user PIN digits;

2. when a certain digit is discovered in the previous step by a *"PIN is wrong"* output, the intruder also changes the *offset* until the API returns again that the PIN is correct. This allows the intruder to locate the position of the deduced PIN digit.

Example 6. *Recall that in Example 2 we assumed* $len = 4$, $dectab = 9753108642543210$, $offset = 4732$, $x1 = \mathrm{A47295FDE32A48B1}$, $EPB = \{\!|9897, r|\!\}_k$*. As we have shown, with these parameters the API returns "PIN is correct".*

Assume now the attacker chooses the new $dectab' = 97531\underline{1}864254321\underline{1}$*, where the two 0's have been replaced by 1's. The aim is to discover whether or not 0 appears in* x_3*. If we invoke the API with* $dectab'$ *we obtain the same intermediate and final values, as* $\mathsf{decimalize}(dectab', A472) = \mathsf{decimalize}(dectab, A472) = 5165$*. This means that 0 does not appear in* x_3*.*

The attacker now proceeds by removing from the dectab the next decimal digit until the API fails: with $dectab'' = 9753\underline{2}08642543\underline{2}20$*, i.e., by replacing digit 1 with digit 2, we obtain that* $\mathsf{decimalize}(dectab'', A472) = 5265 \neq \mathsf{decimalize}(dectab, A472) =$

5165, *reflecting the presence of 1 in the original value of* x_3. *Then it follows,* $x_4 = \mathsf{sum_mod10}(5265, 4732) = 9997$ *instead of* 9897 *thus returning "PIN is wrong".*

The intruder now knows that digit 1 occurs is in x_3. *To discover its position and multiplicity, he now tries variations of the offset so to 'compensate' for the modification of the dectab. In particular, he tries to decrement each offset digit by 1. For example, testing the position of one occurrence of one digit amounts to trying the following offset variations:* $\underline{3}732, 4\underline{6}32, 47\underline{2}2, 473\underline{1}$. *Notice that, in this specific case, offset value* 4632 *makes the API return again "PIN is correct". In fact* $x_3 = \mathsf{decimalize}(dectab, A472) = 5165$ *and* $x_4 = \mathsf{sum_mod10}(5165, 4732) = 9897$ *with dectab, and* $x_3 = \mathsf{decimalize}(\underline{dectab''}, A472) = 5\underline{2}65$ *and* $x_4 = \mathsf{sum_mod10}(5\underline{2}65, 4\underline{6}32) = 9897$ *with* $\underline{dectab''}$. *Notice, in particular, that the value of* x_4 *is the same. The attacker now knows that the second digit of* x_3 *is 1. Given that the offset is public, he also calculates the second digit of the user PIN as* $1 + 7 \bmod 10 = 8$.

The above attack is based on the lack of integrity of the input data, which allows an attacker to influence the declassification mechanism as shown below.

In order to model the PIN derivation encryption of x_1, *we now adopt a small trick: we write vdata =* $\{\!|A47295FDE32A48B1|\!\}_{pdk}$ *and the encryption can now be modelled as a decryption* $x_1 := \mathsf{dec}_{pdk}(vdata)$. *The reason for this is that we have a symbolic model for encryption that does not produce any low level bit-string encrypted data. Notice also that this model is reasonable, as the high-confidentiality of the encrypted value is 'naturally' protected by the* HH *PIN derivation key.*

Consider now two memories, M *and* M_1 *that store the correct parameters of Example 2 for the PIN verification API call (parameters which are all at level* LL*), but contain dectab'' instead of dectab and have different encryption values, i.e., EPB and a different* EPB_1, *respectively. Note that* M *and* M_1 *could be built by an attacker sniffing all encryptions arriving at the verification facility.*

It holds that $\mathsf{M} \approx_{LL} \mathsf{M}_1$ *since the only unequal values are randomized and so can be re-named to make their patterns equal. If we execute* PIN_V *in* M *and* M_1 *we obtain "PIN is wrong" in both cases as for memory* M_1, *the encrypted PIN is wrong, and for memory* M, *the encrypted PIN is correct but the dectab'' will change the value of derived PIN. It follows that* $\langle \mathsf{M}, \mathsf{PIN_V} \rangle \approx_{LL} \langle \mathsf{M}_1, \mathsf{PIN_V} \rangle$.

Now suppose that the intruder replaces dectab'' with the original value dectab, and plugs this into the above memories obtaining memories M_2 *and* M_3. *Note that* $\mathsf{M}_2 \approx_{LL} \mathsf{M}_3$, *however* M_2 *will return "PIN is correct" since all the data correspond, whereas memory* M_3 *will return "PIN is wrong" because the encrypted PIN is incorrect. Thus,* $\langle \mathsf{M}_2, \mathsf{PIN_V} \rangle \not\approx_{LL} \langle \mathsf{M}_3, \mathsf{PIN_V} \rangle$, *and hence robustness does not hold. To overcome this problem, integrity of the input must be established.* □

5.5 Type System

In this section we give a type system to statically check that a program with cryptographic primitives satisfies noninterference and/or robustness. We will then use it

to type-check a MAC-based variant of the PIN verification API and a MAC-based variant of a translation API we will briefly introduce later.

We refine integrity levels by introducing the notion of *dependent domains* used to track integrity dependencies among variables. Dependent domains are denoted $\mathsf{D} : \tilde{\mathsf{D}}$ where $\mathsf{D} \in \mathcal{D}$ is a domain name. Intuitively, the values of domain $\mathsf{D} : \tilde{\mathsf{D}}$ are determined by the values in the set of domains $\tilde{\mathsf{D}}$. For example, $\mathsf{PIN} : \mathsf{PAN}$ can be read as 'the value of PIN is fixed relative to the account number PAN': when the PAN is fixed, the value of the PIN is also fixed. A domain $\mathsf{D} : \emptyset$, also written D, whose integrity does not depend on other domains is called an *integrity representative* and it can be used as a reference for checking the integrity of other domains. In fact, integrity representatives cannot be modified by programs and their values remain constant at run-time.

The integrity level associated to a dependent domain $\mathsf{D} : \tilde{\mathsf{D}}$ is written $[\mathsf{D} : \tilde{\mathsf{D}}]$, and is at a higher integrity level than H, i.e., $[\mathsf{D} : \tilde{\mathsf{D}}] \sqsubseteq_I \mathsf{H}$. In some cases, e.g., in arithmetic operations, we necessarily loose information about the precise result domain $\mathsf{D} : \tilde{\mathsf{D}}$ and we only record the fact the value is determined by domains $\tilde{\mathsf{D}}$, written $[\bullet : \tilde{\mathsf{D}}]$. In this case we know the value is determined by at most variables of domains $\tilde{\mathsf{D}}$, but we have no precise information on its domain. The obtained integrity levels are thus: $\delta_I ::= \mathsf{L} \mid \mathsf{H} \mid [\mathsf{D} : \tilde{\mathsf{D}}] \mid [\bullet : \tilde{\mathsf{D}}]$, while confidentiality levels are still $\delta_C ::= \mathsf{L} \mid \mathsf{H}$, as before. The preorder of integrity levels is extended as:

$$[\mathsf{D} : \tilde{\mathsf{D}}_1] \sqsubseteq [\bullet : \tilde{\mathsf{D}}_1] \sqsubseteq_I [\bullet : \tilde{\mathsf{D}}_2] \sqsubseteq_I \mathsf{H} \sqsubseteq_I \mathsf{L}$$

with $\tilde{\mathsf{D}}_1 \subseteq \tilde{\mathsf{D}}_2$. We will write C in place of $[\bullet]$, to denote a constant value with no specific domain. Based on these levels, we give the type syntax:

$$\tau \quad ::= \quad \delta \mid \mathsf{cK}^{\mu}_{\delta}(\tau)\ \kappa \mid \mathsf{enc}_{\delta}\ \kappa \mid \mathsf{mK}_{\delta}(\tau) \mid (\tau_1, \tau_2)$$

A variable of type δ contains generic data at level δ; types $\mathsf{cK}^{\mu}_{\delta}(\tau)\ \kappa$ and $\mathsf{mK}_{\delta}(\tau)$ respectively refer to encryption and MAC keys of level δ, working on data of type τ; κ is a label that uniquely identifies one key and is used to reconstruct types when decrypting high integrity ciphertexts; we write $\mathsf{K}(\kappa) = \mathsf{cK}^{\mu}_{\delta}(\tau)\ \kappa$ to refer to such a unique type; label μ indicates whether the ciphertext is 'randomized' via confounders ($\mu = R$) or not (μ missing); we only consider untrusted keys of level LL and trusted ones of level HC: trusted keys are not modifiable thanks to the constant integrity type C; $\mathsf{enc}_{\delta}\ \kappa$ is the type for ciphertexts at level δ, obtained using a key labelled κ, thus we will always assume that two key types with the same κ are exactly the same type; pairs are typed as (τ_1, τ_2) and the two types are required to have the same confidentiality level and, for integrity levels at or above H, even the same integrity level. Intuitively, this requirement is to avoid information leakage when projecting elements from the pairs. Dependent domains will never be observed from 'inside', since our observation level will always be in the four-point lattice, making them appear as a unique higher level.

A *security type environment* $\Delta : x \mapsto \tau$ maps variables to security types. The security environment Γ, suitably extended to the new integrity levels, can be derived

Table 5.2 PIN APIs Type System - Expressions

$$\text{(var)}\ \frac{\Delta(x)=\tau}{\Delta\vdash x:\tau}\qquad \text{(sub)}\ \frac{\Delta\vdash e:\tau'\quad \tau'\leq\tau}{\Delta\vdash e:\tau}$$

$$\text{(op)}\ \frac{\Delta\vdash e_1:\delta\quad \Delta\vdash e_2:\delta\quad \mathcal{L}_I(\delta)\neq[\mathsf{D}:\tilde{\mathsf{D}}]}{\Delta\vdash e_1\ \mathsf{op}\ e_2:\delta}$$

$$\text{(pair)}\ \frac{\Delta\vdash e_1:\tau_1\quad \Delta\vdash e_2:\tau_2}{\Delta\vdash \mathsf{pair}(e_1,e_2):(\tau_1,\tau_2)}\qquad \text{(fst/snd)}\ \frac{\Delta\vdash e:(\tau_1,\tau_2)}{\Delta\vdash \mathsf{fst}(e):\tau_1\quad \Delta\vdash \mathsf{snd}(e):\tau_2}$$

$$\text{(enc)}\ \frac{\Delta(x)=\mathsf{cK}_\delta(\tau)\ \kappa\quad \Delta\vdash e:\tau}{\Delta\vdash \mathsf{enc}_x(e):\mathsf{enc}_{\delta\sqcup\mathcal{L}(\tau)}\ \kappa}\qquad \text{(dec)}\ \frac{\Delta(x)=\mathsf{cK}_\delta(\tau)\ \kappa\quad \Delta\vdash e:\mathsf{enc}_{\delta'}\ \kappa}{\Delta\vdash \mathsf{dec}_x(e):\delta\sqcup\delta'}$$

$$\text{(enc-r)}\ \frac{\Delta(x)=\mathsf{cK}^R_{\mathsf{HC}}(\tau)\ \kappa\quad \Delta\vdash e:\tau}{\Delta\vdash \mathsf{enc}^R_x(e):\mathsf{enc}_{\mathsf{LC}\sqcup\mathcal{L}_I(\tau)}\ \kappa}$$

$$\text{(dec-}\mu\text{)}\ \frac{\Delta(x)=\mathsf{cK}^\mu_{\mathsf{HC}}(\tau)\ \kappa\quad \Delta\vdash e:\mathsf{enc}_{\delta_C\mathsf{C}\sqcup\mathcal{L}_I(\tau)}\ \kappa\quad \mathcal{L}_C(\tau)=H}{\Delta\vdash \mathsf{dec}^\mu_x(e):\tau}$$

$$\text{(enc-d)}\ \frac{\Delta(x)=\mathsf{cK}_{\mathsf{HC}}(\tau)\ \kappa\quad \Delta\vdash e:\tau\quad \mathsf{CloseDD}^{\mathsf{det}}(\tau)}{\Delta\vdash \mathsf{enc}_x(e):\mathsf{enc}_{\mathsf{LC}\sqcup\mathcal{L}_I(\tau)}\ \kappa}$$

$$\text{(mac)}\ \frac{\Delta(x)=\mathsf{mK}_{\delta_C\delta_I}(\tau)\quad \Delta\vdash e:\tau}{\Delta\vdash \mathsf{mac}_x(e):\mathsf{LL}\sqcup\mathcal{L}(\tau)}$$

from the type environment Δ by just 'extracting' from the types their security level. This is done via the following level function $\mathcal{L}:\tau\mapsto\delta$ defined as $\mathcal{L}(\delta)=\mathcal{L}(\mathsf{K}_\delta(\tau)\ \kappa)=\mathcal{L}(\mathsf{enc}_\delta\ \kappa)=\delta$ and $\mathcal{L}((\tau_1,\tau_2))=\mathcal{L}(\tau_1)\sqcup\mathcal{L}(\tau_2)$. Notice that we write $\mathsf{K}_\delta(\tau)\ \kappa$ to indifferently denote encryption and MAC key types. We will also write $\mathcal{L}_C(\tau)$ and $\mathcal{L}_I(\tau)$ to respectively extract the confidentiality and integrity level of type τ. Formally, if $\mathcal{L}(\tau)=\delta_C\delta_I$ then $\mathcal{L}_C(\tau)=\delta_C$ and $\mathcal{L}_I(\tau)=\delta_I$. From now on we will assume $\Gamma=\mathcal{L}\circ\Delta$.

The subtype preorder $\leq$ extends the security level preorder $\sqsubseteq$ on levels δ with $\mathsf{enc}_{\delta_C\delta_I}\ \kappa\leq\delta_C\mathsf{L}$. Moreover, from now on, we will implicitly identify low-integrity types at the same security level, i.e., we will not distinguish τ and τ' whenever $\mathcal{L}(\tau)=\mathcal{L}(\tau')=\delta_C\mathsf{L}$, written $\tau\equiv\tau'$. This reflects the intuitions that we do not make any assumption on what is stored into a low-integrity variable. We do not include high keys in the subtyping and we also disallow the encryption (and the MAC) of such keys: formally, in $\mathsf{K}_\delta(\tau)\ \kappa$ and (τ_1,τ_2) types $\tau,\tau_1,\tau_2\neq\mathsf{K}_{\mathsf{HC}}(\tau)\ \kappa$. We believe that transmission of high keys can be easily accounted for but we leave this extension as future work.

Closed key types

In some typing rules we will require that types transported by cryptographic keys are 'closed', meaning that they are all dependent domains and all the dependencies are satisfied, i.e., all the required representatives are present. As an example, consider $\mathsf{cK}^{\mu}_{\mathsf{HC}}(\tau)\,\kappa$ with $\tau = (\mathsf{H}[\mathsf{D}], \mathsf{H}[\mathsf{D}' : \mathsf{D}])$. Types transported by the key are all dependent domains, noted $\mathsf{DD}(\tau)$, and are closed: the set of dependencies, noted $\mathsf{Dep}(\tau)$, is $\{\mathsf{D}\}$, since $[\mathsf{D}' : \mathsf{D}]$ depends on D, and the set of representatives, noted $\mathsf{IRs}(\tau)$, is $\{\mathsf{D}\}$, because of the presence of the representative $[\mathsf{D}]$. If we instead consider $\tau' = (\mathsf{H}[\mathsf{D}], \mathsf{H}[\mathsf{D}' : \mathsf{D}], \mathsf{H}[\mathsf{D}' : \mathsf{D}''])$ we have that the set of dependencies is $\{\mathsf{D}, \mathsf{D}''\}$ and the set of representatives is $\{\mathsf{D}\}$, meaning that the type is not closed: not all the dependencies can be found in the type. We will write $\mathsf{CloseDD}(\tau)$ to denote that τ is closed, formally expressed as $\mathsf{Dep}(\tau) \subseteq \mathsf{IRs}(\tau)$, and only contains dependent domains, written $\mathsf{DD}(\tau)$. When it additionally does not transport nested randomized ciphertexts we write $\mathsf{CloseDD}^{\mathsf{det}}(\tau)$. We will describe the importance of this closure conditions when describing the typing rules. In section C.1 of Appendix C we report the formal definition of the above predicates.

Expression typing rules

Expressions are typed with judgment of the form $\Delta \vdash e : \tau$, derived from the rules in Table 5.2. The first five rules are pretty standard. The only unusual requirement is $\mathcal{L}_I(\delta) \neq [\mathsf{D} : \tilde{\mathsf{D}}]$ in rule (op). This forbids the typing of an operation with the dependent domain of the operands. This is because for an arbitrary operation, we cannot predict the precise value of the result. However, the dependency on other domains $\tilde{\mathsf{D}}$, noted $[\bullet : \tilde{\mathsf{D}}]$, is correctly preserved meaning that the result of the operation depends 'in some way' from $\tilde{\mathsf{D}}$. Recall, in fact, that $[\mathsf{D} : \tilde{\mathsf{D}}] \leq [\bullet : \tilde{\mathsf{D}}]$.

The first two rules for cryptography (enc) and (dec) correspond to the (op) rule described above: they allow one to encrypt and decrypt at a level which is the least upper bound of the levels of the key and the plaintext/ciphertext. Recall that we only consider untrusted LL keys and trusted HC ones.

The remaining rules are more interesting: rule (enc-r) is for randomized encryption: we let $\mathsf{enc}^R_x(e)$ and $\mathsf{dec}^R_x(e)$ denote, respectively, $\mathsf{enc}_x(e, \mathsf{new}())$ and $\mathsf{fst}(\mathsf{dec}_x(e))$, i.e., an encryption randomized via a fresh confounder and the corresponding decryption. The typing rule requires a trusted key HC. The integrity level is handled as before (notice that C has the effect of removing all the information on domains, as for operations), while confidentiality of the ciphertext is L, meaning that it can be assigned to low-confidentiality variables and thus accessed by an attacker. The rule intuitively states that whenever a fresh confounder is encrypted with the plaintext, the resulting ciphertext preserves secrecy, even if sent on an public/untrusted part of the memory.

Rule (dec-μ), when $\mu = R$ is the dual of (enc-r). Notice that the rule gives the correct type τ to the obtained plaintext. This can be done only if the confidentiality

of the plaintext is at least as restrictive as the one of the key used, i.e., equal to H, since H is the highest possible.

Rule (enc-d) is the most original one. It encodes a way to guarantee secrecy even without confounders, i.e., with no randomization. The idea comes from format ISO0 for the EPB, which intuitively combines the PIN with the PAN before encrypting it in order to prevent codebook-attacks. Consider, for example the ciphertext $\{\!|\mathrm{PAN}, \mathrm{PIN}|\!\}_k$. Since every account, identified by the PAN, has its own PIN, the PIN can be thought of as at level [PIN : PAN] ('the PIN is fixed relative to the PAN'). Thus equal PANs will determine equal PINs, which implies that different PINs will always be encrypted together with different PANs, producing different EPBs. This avoids, for example, allowing an attacker to build up a codebook of all the PINs. Intuitively, the PAN is a sort of confounder that is 'reused' only when its own PIN is encrypted. The rule requires $\mathsf{CloseDD}^{\mathsf{det}}(\tau)$, i.e., that each subexpression is at integrity level $[\mathsf{D_i} : \mathsf{D'_i}]$, and that dependent domains are closed, i.e., all the required integrity representatives are in τ. This is important as they play the role of confounders, as explained above. Moreover it is also required that no random ciphertexts are included in the message to encrypt. As in (enc-r) integrity is propagated and confidentiality of the ciphertext is L, meaning that it is safe to assign it to low confidentiality variables.

Rule (mac) is for the generation of MACs: it is similar to (op). The only interesting difference is that the confidentiality level of the key does not contribute to the confidentiality level of the MAC, which just takes the one of e. This reflects the fact that we only use MACs for integrity and we always assume the attacker knows the content of MACs, as formalized in definition 5.4. The reason why we force integrity to be low is related to the fact we want to forbid declassification of cryptographic values, which would greatly complicate the proof of robustness. By the way, this is not limiting as there are no good reasons to declassify what has been created to be low-confidentiality.

Typing rules for commands As in existing approaches [48] we introduce in the language a special expression $\mathsf{declassify}(e)$ for explicitly declassifying the confidentiality level of an expression e to L. This new expression has no operational import, i.e., $\mathsf{declassify}(e) \downarrow^{\mathsf{M}} v$ iff $e \downarrow^{\mathsf{M}} v$. Declassification is thus only useful in the type-system to isolate program points where downgrading of security happens, in order to control robustness.

Judgements for commands have the form $\Delta, pc \vdash \mathsf{c}$ where pc is the program counter level. It is a standard way to track what information has affected control flow up to the current program point [48]. For example, when entering a while loop, the pc is raised to be higher or equal to the level of the loop guard expression. This prevents such an expression to allow flows to lower levels. The pc is a level on the four point lattice, notice that, however, when its integrity level is high we let assignment to integrity levels below H, e.g., dependent domains, to take place: this makes sense since, as mentioned before, we never move our observation level below

Table 5.3 PIN APIs Type System - Commands

$$\text{(skip)}\ \Delta, pc \vdash \mathsf{skip} \quad \text{(seq)}\ \frac{\Delta, pc \vdash \mathsf{c}_1 \quad \Delta, pc \vdash \mathsf{c}_2}{\Delta, pc \vdash \mathsf{c}_1; \mathsf{c}_2} \quad \text{(while)}\ \frac{\Delta \vdash b : \tau \quad \Delta, \mathcal{L}(\tau) \sqcup pc \vdash \mathsf{c}}{\Delta, pc \vdash \mathsf{while}\ b\ \mathsf{do}\ \mathsf{c}}$$

$$\text{(assign)}\ \frac{\Delta(x) = \tau \quad \Delta \vdash e : \tau \quad pc \sqsubseteq \mathcal{L}(\tau) \sqcup \mathsf{LH}}{\Delta, pc \vdash x := e}$$

$$\text{(if)}\ \frac{\Delta \vdash b : \tau \quad \Delta, \mathcal{L}(\tau) \sqcup pc \vdash \mathsf{c}_1 \quad \Delta, \mathcal{L}(\tau) \sqcup pc \vdash \mathsf{c}_2}{\Delta, pc \vdash \mathsf{if}\ b\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2}$$

$$\text{(declassify)}\ \frac{\Delta(x) = \delta_C\mathsf{H} \quad \Delta \vdash e : \delta'_C\mathsf{H} \quad pc \sqsubseteq \delta_C\mathsf{H}}{\Delta, pc \vdash x := \mathsf{declassify}(e)}$$

$$\text{(if-MAC)}\ \frac{\begin{array}{c}\Delta(x) = \mathsf{mK}_{\mathsf{HC}}(\mathsf{L}[\mathsf{D}], \tau) \quad \Delta \vdash z : \mathsf{L}[\mathsf{D}] \quad \Delta \vdash e : \mathsf{LL} \quad \Delta \vdash e' : \mathsf{LL} \quad \Delta(y) = \tau \\ \mathsf{IRs}(\mathsf{L}[\mathsf{D}], \tau) = \{\mathsf{D}\} \mathsf{CloseDD}(\mathsf{L}[\mathsf{D}], \tau) \quad \Delta, pc \vdash \mathsf{c}_1 \quad \Delta, pc \vdash \mathsf{c}_2 \quad pc \sqsubseteq \mathcal{L}(\tau) \sqcup \mathsf{LH}\end{array}}{\Delta, pc \vdash \mathsf{if}\ \mathsf{mac}_x(z, e) = e'\ \mathsf{then}\ (y := e; \mathsf{c}_1)\ \mathsf{else}\ \mathsf{c}_2; \bot_{\mathsf{MAC}}}$$

LH.

Typing rules for commands are depicted in Table 5.3. The first six rules are largely standard [48]. The pc is raised on entering if branches and while loops. Notice also that assignments are only possible at or above the pc level and, as mentioned above, at lower integrity levels if $\mathcal{L}_I(pc) = \mathsf{H}$. Rule (declassify) lets a high integrity expression to be declassified, i.e., assigned to some high-integrity variable independent of its confidentiality level, when also the program counter is at high-integrity and the assignment to the variable is legal ($pc \sqsubseteq \delta_C\mathsf{H}$). The high-integrity requirement is for guaranteeing robustness: no attacker will be able to influence declassification.

The (if-MAC) rule is peculiar of our approach: it allows the checking of a MAC with respect to an integrity representative z. The rule requires that the first parameter z is typed at level $\mathsf{L}[\mathsf{D}]$; the second parameter e and the MAC value e' are typed LL. If the MAC succeeds, variable y of type τ is bound to the result of e through an explicit assignment in the if-branch. Notice that such an assignment would be forbidden by the type-system, as it is promoting the integrity of an LL expression to an unrestricted type τ (as far as pc is high integrity). This can however be proved safe since the value returned by the LL expression matches an existing MAC, giving us enough guarantees about the integrity of the data, and allowing us to 'reconstruct' their type from the type of the MAC key.

Side conditions $\mathsf{IRs}(L[\mathsf{D}], \tau) = \{\mathsf{D}\}$ and $\mathsf{CloseDD}(\mathsf{L}[\mathsf{D}], \tau)$ ensure that the MAC contains only values which directly depends on the unique integrity representative given by variable z. The 'then' branch is typed without any particular restriction, while the 'else' one is required to end with a special failure command $\bot_{\mathsf{MAC}}$

which just aims at non-terminating the program (it may be equivalently though as a command with no semantics, which never reduces, or a diverging program as, e.g., while true do skip) so to obtain a 'match-it-or-die' program (see Chapter 4). This is needed to avoid the attacker breaks integrity and robustness by just calling an API with wrong MACs. In fact, we can assume the attacker knows which MACs pass the tests and which MACs do not (unless he is trying some brute-force/cryptanalysis attack on the MAC algorithm, that we do not account for here) and by letting the else branch fail we just disregard those obvious, uninteresting, information flows.

Security results We now give an overview of the security results, all the formal proofs can be found in Appendix C.

As mentioned above, our type-system aims at guaranteeing a form of noninterference and robustness, in the presence of cryptography and MAC-based integrity. Our results hold under some reasonable well-formedness/integrity assumptions on the memories: (*i*) variables of high level key-type really contain keys of the appropriate level, and such keys never appears elsewhere in the memory; (*ii*) values of variables or encrypted messages at integrity H, or below, must adhere to the expected type; for example, the value of a variable typed as high integrity pair is expected to be a pair; (*iii*) values for dependent domains $[\mathsf{D} : \tilde{\mathsf{D}}]$ are uniquely determined by the values of the integrity representatives $\tilde{\mathsf{D}}$, e.g., when they appear together in an encrypted message or a MAC or when they have been checked in an if-MAC statement; (*iv*) confounders are used once: there cannot be two different encrypted messages with the same confounder.

Condition (*iii*) states, for example, that if a MAC is expected (from the type of its key) to contain the PAN, of level $[\mathsf{PAN}]$ and the relative PIN, of level $[\mathsf{PIN} : \mathsf{PAN}]$, encrypted with another key, all of the possible MACs with that key will respect a function $f_{[\mathsf{PIN:PAN}]}$, pre-established for each memory. Thus, equal PANs will imply equal encrypted PINs. Intuitively, even if the attacker does not know the actual PIN, he knows that what is encrypted is the unique PIN corresponding to the PAN. The value of integrity representatives, like the PAN above, is instead instantiated 'on-the-fly', during well-formedness check, via a function g. For example, let us assume $f_{[\mathsf{PIN:PAN}]}(pan_i) = pin_i$. We have that all of these MACs are well-formed: $\langle pan_1, \{\!|pin_1|\!\}_k\rangle_{k'}$, $\langle pan_2, \{\!|pin_2|\!\}_k\rangle_{k'}$, $\ldots$, $\langle pan_m, \{\!|pin_m|\!\}_k\rangle_{k'}$, as they all respect $f_{[\mathsf{PIN:PAN}]}$. Intuitively, when we check the well-formedness of these MACs we fix a function $g(PAN)$ the exact moment we check the value in the MAC, i.e., for the first MAC we choose $g(PAN) = pan_1$, and so on. A well-formed PIN value is thus the one that matches $f_{[\mathsf{PIN:PAN}]}(g(PAN))$. Memories well-formedness, noted $\Delta \vdash_g^f \mathsf{M}$, is fully described and formalized in Section C.2 of appendix.

Program that are run on well-formed memories always return well-formed memories:

Proposition 5.1. *If* $\Delta, pc \vdash \mathsf{c}$, $\Delta \vdash_g^f \mathsf{M}$ *and* $\langle \mathsf{M}, \mathsf{c}\rangle \Rightarrow \mathsf{M}'$ *then* $\Delta \vdash_g^f \mathsf{M}'$.

From now on, we will implicitly assume that memories are well-formed. The next result states that when no declassification occurs in a program, then noninterference

holds. This might appear surprising as MAC checks seem to potentially break integrity: an attacker might manipulate one of the MAC parameters to gain control over the MAC check. In this way he can force the execution of one branch or the other, however recall that by inserting $\bot_{\mathsf{MAC}}$ at the end of the else branch we force that part of the program not to terminate. Weak indistinguishability will thus consider such an execution equivalent to any other, which means it will disregard that situation.

Next lemmas are used to prove the main results. The first one is peculiar of our extension with cryptography: if an expression is typed below the observation level ℓ, we can safely assign it to two equivalent memories and still get equivalent memories. We cannot just check the obtained values in isolation as, by traffic analysis (modelled via patterns), two apparently indistinguishable ciphertext might be distinguished once compared via equality with other ciphertexts existing in the memories.

Lemma 5.1 (Expression ℓ-equivalence)**.** *Let* $\mathsf{M}_1 \approx_\ell \mathsf{M}_2$ *and let* $\Delta \vdash e : \tau$ *and* $e \downarrow^{\mathsf{M}_i} v_i$*. If* $\mathcal{L}(\tau) \sqsubseteq \ell$ *or* $\mathcal{L}(\Delta(x)) \not\sqsubseteq \ell$ *then* $\mathsf{M}_1[x \mapsto v_i] \approx_\ell \mathsf{M}_2[x \mapsto v_i]$.

Lemma 5.2. (**Confinement**) *If* $\Delta, pc \vdash \mathsf{c}$ *then for every variable* x *assigned to in* c *and such that* $\Delta(x) = \tau$ *it holds that* $pc \sqsubseteq \mathcal{L}(\tau) \sqcup \mathsf{LH}$.

Theorem 5.1 (Noninterference)**.** *Let* c *be a program which does not contain any declassification statement. If* $\Delta, pc \vdash \mathsf{c}$ *then* c *satisfies noninterference, i.e.,* $\forall\ \ell \sqsupset \mathsf{LH}, \mathsf{M}_1, \mathsf{M}_2.\ \mathsf{M}_1 \approx_\ell \mathsf{M}_2$ *implies* $\langle \mathsf{M}_1, \mathsf{c}\rangle \approx_\ell \langle \mathsf{M}_2, \mathsf{c}\rangle$.

We can now state our final results on robustness. We will consider programs that assign declassified data to special variables assigned only once. This can be easily achieved syntactically, e.g., by using one different variable for each declassification statement (which we label for clarity), i.e., $x_1 := \mathsf{declassify_1}(e_1), \ldots, x_m := \mathsf{declassify_m}(e_m)$, and avoiding to place declassifications inside while loops. These special variables are nowhere else assigned. We call this class of programs *Clearly Declassifying* (CD). We do this to avoid, one more time, that attackers 'incompetently' hide flows by resetting variables after declassification has happened and, since our semantics is big-step, we would not detect these intermediate critical states. CD-programs, instead, keep declassified data unchanged in fixed variables up to their termination.

Theorem 5.2 (Robustness)**.** *If a CD-program* c *is such that* $\Delta, pc \vdash \mathsf{c}$ *then* c *satisfies robustness, i.e.,* $\forall \mathsf{M}_1, \mathsf{M}_2, \mathsf{M}'_1, \mathsf{M}'_2$ *such that* $\mathsf{M}_1 \approx_{\mathsf{LL}} \mathsf{M}_2$*,* $\mathsf{M}'_1 \approx_{\mathsf{LL}} \mathsf{M}'_2$ *and* $\mathsf{M}_i \approx_{\mathsf{HH}} \mathsf{M}'_i$ *it holds*

$$\langle \mathsf{M}_1, \mathsf{c}\rangle \approxeq_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}\rangle \text{ implies } \langle \mathsf{M}'_1, \mathsf{c}\rangle \approx_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}\rangle$$

5.6 A type-checkable MAC-based API

We now discuss PIN_V_M a MAC-based improvement of PIN_V, which prevents the attack of Section 5.4.2, and several others from the literature. We show PIN_V_M is type-checkable using our type system, and we also show where the original API fails to type-check.

Table 5.4 The new PIN_V_M API with MAC-based integrity.

```
PIN_V_M(PAN,EPB,len,offset,vdata,dectab,MAC){
  if (mac_ak(PAN,EPB,len,offset,vdata,dectab)==MAC)
   then EPB' := EPB; len' := len; offset' := offset;
        vdata' := vdata; dectab' := dectab;
        PIN_V(PAN,EPB',len',offset',vdata',dectab');
   else ret := "integrity violation"; ⊥_MAC}
```

We assume the intruder starts with only one EPB which contains the correct PIN. An attacker equipped with several EPBs for the same PAN, only one of which contains the correct PIN, can always violate robustness: he can try all the EPBs until he identifies the one containing the correct PIN, influencing the declassification of data. We feel our assumption is justified because an attacker with a stolen or fake card can only make three guesses at the terminal before the account is blocked. If the attacker were able to circumvent this restriction, given sufficient patience, he could make a 'brute force' attack on the PIN from the terminal which we do not expect to be able to prevent with our API.

Our fix is reported in Table 5.4. The new API checks a MAC of all the parameters at the very beginning. Intuitively, the MAC check guarantees that the parameters have not been manipulated. Some form of 'legal' manipulation is always possible: an intruder can get a different set of parameters, e.g., eavesdropped in a previous PIN verification and referring to a *different* PAN, and can call the API with these parameters. Those parameters will have a correct MAC validating their integrity. This is actually captured by our notion of dependent domains by typing all the MAC checked variables as dependent from the PAN.

We need to refine the model given in Table 5.1 for the verification API (called by our MAC checking version). As discussed in Section 5.4.2, we model PIN derivation as a *decryption*, obtaining the expected hexadecimal (secret) number from which the PIN is then derived. The only change needed in Table 5.1 is substituting $x_1 := \mathsf{enc}_{pdk}(vdata)$ with $x_1 := \mathsf{dec}_{pdk}(vdata)$. We model the extraction of the PIN from an ISO1 block as randomized decryption. The resulting function is in Table C.5 of Appendix C.

We show typing in detail: All the parameters except for the *PAN* (*EPB, len, offset, vdata, dectab, MAC*) are of type LL, since we assume the attacker can read and modify them; the *PAN* instead is used as the integrity representative and is thus typed L[PAN]. The important element is the mac key *ak* which is given type $\mathsf{mK_{HC}}(\tau)$ with $\tau = \mathsf{L[PAN]}, \mathsf{enc}_{\mathsf{L[\bullet:PAN]}}\ \kappa_{ek}, \mathsf{L[LEN:PAN]}, \mathsf{L[OFFS:PAN]}, \mathsf{enc}_{\mathsf{L[\bullet:PAN]}}\ \kappa_{pdk}, \mathsf{L[DECTAB:PAN]}$. Note that $\mathsf{IRs}(\tau) = \{PAN\}$, $\mathsf{Dep}(\tau) = \{PAN\}$, thus we have $\mathsf{Closed}(\mathsf{L}[PAN], \tau)$ and also $\mathsf{DD}(\tau)$ as τ only contains types of the form [D : PAN]. All the checked variables are typed according to the above tuple, e.g., PAN' with L[PAN], EPB' with $\mathsf{enc}_{\mathsf{L[\bullet:PAN]}}\ \kappa_{ek}$ and so on. In the code, to make it readable, we directly assign MAC checked expressions to variables. To type-check,

this assignment should be done in one variable, and than decomposed by projecting the first and second elements of pairs. The result of the API will be stored in the *ret* variable whose type is LL.

The key ek is typed as $\mathsf{cK}^R_{\mathsf{HC}}(\mathsf{H}[\mathsf{PIN} : \mathsf{PAN}])\ \kappa_{ek}$ while the PIN derivation key pdk as $\mathsf{cK}^R_{\mathsf{HC}}(\mathsf{H}[\mathsf{HEX} : \mathsf{PAN}])\ \kappa_{pdk}$. To complete the typing of the MAC we need to type the two branches. The else branch is trivial: the assignment to *ret* is legal and then it is followed by the MAC-fail command. The other one amounts to checking the original API with the new high integrity types. What happens is that x_1 is typed $\mathsf{H}[\mathsf{HEX} : \mathsf{PAN}]$ by rule (dec-μ) and $x_2, \ldots, x_4$ are typed $\mathsf{H}[\bullet : \mathsf{PAN}]$ by rule (op). x_6 is typed $\mathsf{H}[\mathsf{PIN} : \mathsf{PAN}]$ by rule (dec-μ). Thus the declassification to x_7, which is typed LH, of the result of the comparison, is type-checkable as the expression types $\mathsf{H}[\bullet : \mathsf{PAN}]$ and by subtype leads to HH. By theorem 5.2 we are guaranteed $\mathsf{PIN_V_M}$ is robust. In the original version of the API, without the MAC check, x_4 and x_6 would only be typeable with low integrity, and hence the declassification would violate robustness.

PIN translation API We conclude our results with a brief discussion of the *translation* API that we call $\mathsf{PIN_T_M}$, used to decrypt and re-encrypt a PIN under a different key. Switches may not be able to support all known PIN formats, so the translation function might need to reformat messages under different block formats. In this paper we consider only the ISO-0 and ISO-1 formats (specified in table C.4 of the appendix). ISO-0 pads the PIN with data derived from the PAN.

We specify the code of $\mathsf{PIN_T_M}$ for translating specifically from ISO-1 to ISO-0 in Table C.6 of the appendix. The API takes a PIN block EPB_I and key k. It extracts the PIN, reformats it and re-encrypts it with key k'. Decryption is as in the previous section and gives the PIN in variable x_1 with type $\mathsf{H}[\mathsf{PIN} : \mathsf{PAN}]$. The PIN is now padded with the PAN and sent encrypted with the new key, which has type $\mathsf{cK}_{\mathsf{HC}}(\mathsf{H}[\mathsf{PIN} : \mathsf{PAN}], \mathsf{H}[\mathsf{PAN}])\ \kappa_{k'}$ Encryption is thus typed via (enc-d) giving EPB_O of type $\mathsf{enc}_{\mathsf{L}[\bullet:\mathsf{PAN}]}\ \kappa_{k'}$. Recall that this is safe since the PAN is playing the role of a confounder. MAC creation is not problematic. The API type-checks and, given it does not contain any declassification, it satisfies noninterference (Theorem 5.1).

5.7 Conclusions

This chapter shown how to type check a security API subject to noninterference attacks. We have presented our extensions to information flow security types to model deterministic encryption and cryptographic assurance of integrity for robust declassification. We have shown how to apply this to PIN processing APIs. Most previous approaches to formalising cryptographic operations in information flow analysis have aimed to show how a program that is noninterfering when executed in a secure environment can be guaranteed secure when executed over an insecure network by using cryptography, see e.g., [9, 32, 34, 42, 59]. They typically use custom cryptographic schemes with strong assumptions, e.g. randomised cryptography and/or

signing of all messages. This means they are not immediately applicable to the analysis of PIN processing APIs, which have weaker assumptions on cryptography. [26] presents what seems to be the only information flow model for deterministic encryption, that shows soundness of noninterference with respect to the concrete cryptography model. However, it does not treat integrity. Gordon and Jeffreys' type system for authenticity in security protocols could be used to check correspondence assertions between the data sent from the ATM and the data checked at the API [37]. However, this would not address the problem of declassification, robustness or otherwise. Keighren et al. have outlined a framework for information flow analysis specifically for security APIs [40], though this also currently models confidentiality only. The formal analysis of security APIs has usually been carried out by Dolev-Yao style analysis of reachability properties in an abstract model of the API, e.g., [27, 43, 63]. This typically covers only confidentiality properties.

We plan in future to refine our framework on further examples from the PIN processing world and elsewhere. Focardi, Luccio and Steel have also investigated practical ways to implement our scheme in cost-effective way (see [33]).

6

Type checking PKCS#11

PKCS#11, also known as "Cryptoki", is a security API to perform key management tasks. It is known to be vulnerable to attacks [23, 27] which can directly extract in clear the value of a sensitive key, which should be protected by the API. These are pure API attacks: they are not exploiting some cryptographic issue or deriving information on the raw value of a sensitive key from the result of a command. In fact, these vulnerabilities use the possibility of having the same key to perform conflicting operations so that the API, as a result of some sequence of calls that the interface regards as being correct and legal, leaks confidential keys.

A simple imperative programming language, suitable to code programs implementing key management functionalities, is given together with a type-system checking that sensitive keys are never leaked. These offer the possibility to give secure implementations of the PKCS#11 API: a new patch based on key diversification is presented and proved secure (by typing) and also a previously proposed security fix [15, 16] is shown to be type-checkable.

6.1 Introduction

PKCS#11 defines a widely adopted API for cryptographic tokens [51]. It provides access to cryptographic functionalities and should preserve certain security properties, e.g. the values of a *sensitive* key stored on a device should never become known 'in the clear'. PKCS#11 is intended to protect its sensitive cryptographic keys even when connected to a compromised host. However, it is known to be vulnerable to various attacks that break this property [23, 27].

In a PKCS#11-based API, applications initiate a *session* with the cryptographic token, by supplying a PIN. Once a session is initiated, the application may access the *objects* stored on the token, such as keys and certificates. However, access to the objects is controlled. Objects are referenced in the API via *handles*, which can be thought of as pointers to or names for the objects. In general, the value of the handle, e.g. for a secret key, does not reveal any information about the actual value of the key. Objects have *attributes*, which may be a bitstring such as the value of a key, or a Boolean flag signalling a property of the object, e.g. whether the key may be used for encryption, or for encrypting other keys. New objects can be created by

calling a key generation command, or by 'unwrapping' an encrypted key packet. In both cases a fresh handle is returned. When a function in the token's API is called with a reference to a particular object, the token first checks that the attributes of the object allow it to be used for that function.

Known attacks on PKCS#11 are related to the operations for exporting and importing sensitive keys, called `WrapKey` and `UnwrapKey` in the API. The former performs the encryption of a key under another one and the latter performs the corresponding decrypt and import in the token. The standard does not clearly separate roles for keys so that is possible to use the same key for conflicting purposes: for example, a key could have its `decrypt` (D) and `wrap` (W) attributes set, enabling the wrap and subsequent decrypt of a sensitive key, with the effect of leaking it outside the token as plaintext:

$$
\begin{aligned}
&h_myKey := \texttt{GenerateKey}(\{D, W\});\\
&wrapped := \texttt{WrapKey}(h_mySuperSecretKey, h_myKey);\\
&leak := \texttt{Decrypt}(wrapped, h_myKey);
\end{aligned}
$$

Note, in fact, that the wrapping format adopted by PKCS#11 cannot distinguish between an arbitrary encrypted message and a wrapped cryptographic key. So when executing the decrypt command it has no way of telling that the ciphertext it is decrypting and giving out in clear contains a key.

The next chapter will show that the state of the art in PKCS#11 security tokens is rather poor: most of the commercially available devices are vulnerable to attacks or prevent them removing functionalities for secure transport of sensitive keys. The standard, however, can be patched without necessarily avoiding key wrapping [27]: the set of admissible templates has to be restricted by policies on the attributes which prevent a key to be used for conflicting operations and, a wrapping format to bind key attributes to wrapped packets must also be added [27] or the imported keys has to be used only for non-critical functions [15].

Clearly define a role for each key is the first argument of any security patch to PKCS#11. In cryptography, key diversification is a standard technique which given a key derives new ones from it. This is useful to differentiate the usage of a secret key instead of distributing and protecting many of them (one for each purpose). This chapter illustrates the first patch to PKCS#11 applying key diversification to assure that a given key is never used for conflicting operations. The token will internally diversify each sensitive key for the specific function asked by the user: notice that only the original key will be securely stored in the device, while diversified keys will be calculated at the time of the API command invocation and then thrown away.

A formal tool to reason about the security of different implementations of a PKCS#11 API would help developers and hardware producers to better understand the root-causes of the long known bugs affecting it and will also be useful to test new patches. To this aim, a type-system to verify the security of programs designed to perform key management tasks is presented here. It is used to type-check the new patch discussed above and another one proposed (and implemented in a software

emulated token) by the author and others [16, 15]. Note that the model introduced in this chapter as well as the new security patch proposed, only consider symmetric key cryptography.

Structure of the chapter A language suitable to implement programs which perform key management operations is presented in Section 6.2 and a type system to check that sensitive keys are not leaked is given in Section 6.3. It is then shown, in Section 6.4 that a program respecting the RSA PKCS#11 Standard is not secure and two security fixes are considered and proved correct. The chapter closes with some remarks on Section 6.5.

6.2 Language

This section introduces a simple imperative language suitable to implement PKCS#11 API commands.

A cryptographic token can generate values not known outside of it, this is modeled by considering an infinite stream of values G disjoint from the one (N) of values available to an external user of the API. Generated values g, ranging over G, can be thought as fresh values: $g \leftarrow G$, representing the extraction of the value g from G, takes the first element of G and removes it from the stream so that it cannot be reused, i.e., extracted values are always different. More formally, two extractions $g, g' \leftarrow G$ are such that $g \neq g'$. Such generation capability will be used to model the generation of a new key by a secure device but also to get a new handle for an object stored inside the token. With a little abuse of notation, whenever we write $g \in G$ it is meant that g is a value extracted from G.

Cryptographic keys can be also obtained by key diversification: given a key k, a new one can be derived encrypting a *tag* under k. Tags are special values distinguishing between *data* and two different kinds of *wrapping* diversified keys. Values are tags, errors, generated values (g), byte-streams (n), encrypted and decrypted messages.

$$tag ::= \mathtt{D} \mid \mathtt{W} \mid \mathtt{W}^2 \qquad \text{(tags)}$$

$$err ::= \mathtt{CKR_TEMPLATE_INCONSISTENT} \mid \mathtt{CKR_HANDLE_INVALID} \qquad \text{(errors)}$$

$$v ::= n \mid g \mid enc(v, v') \mid dec(v, v') \mid enc(tag, v) \qquad \text{(no-err values)}$$

$$z ::= v \mid err \qquad \text{(values)}$$

The properties and capabilities of keys are described by templates, ranged over by T, represented by a set of *attributes*. When a certain attribute is contained in a template T we will say that the attribute is set and unset otherwise. A key can be either sensitive or non-sensitive, a sensitive key can also be always-sensitive if it has

been generated (as a sensitive key) by a secure device. These two properties are described by the attributes S (sensitive) and A (always-sensitive). Four attributes identify the capabilities of a key: data encryption (E) and decryption (D) , wrap (W) and unwrap (U), i.e., encryption and decryption of other keys. A non-sensitive key is always assumed to be a non-always-sensitive one, i.e., $S \notin T$ implies $A \notin T$.

Expressions e are defined below: getObj retrieve the raw value of a key given its handle, checkTemplate query the template of a key stored inside the token, diversifyKey obtain a new key diversifying another one, enc and dec respectively encrypts and decrypts data or keys.

$$e_{key} ::= \mathsf{getObj}(x) \mid \mathsf{checkTemplate}(x, q) \mid \mathsf{diversifyKey}(tag, x)$$

$$e_{mem} ::= x \mid \mathsf{enc}(e, x) \mid \mathsf{dec}(e, x)$$

$$e ::= e_{key} \mid e_{mem}$$

A handle-map H maps atomic value g to pairs of key and template. A memory M maps variable to no-err values v. $e \downarrow^{\mathsf{M},\mathsf{H}} z$ denotes that the evaluation of the expression e in memory M and handle-map H leads to value z. As it is standard $\mathsf{M}(x)$ identify the value stored for variable x in memory M. The semantics of the expressions follow:

$$\mathsf{getObj}(x) \quad \downarrow^{\mathsf{M},\mathsf{H}} \begin{cases} v & \text{if } \mathsf{M}(x) = g \ \wedge \ \mathsf{H}(g) = (v, T) \\ \texttt{CKR_HANDLE_INVALID} & \text{otherwise} \end{cases}$$

$$\mathsf{checkTemplate}(x, T) \quad \downarrow^{\mathsf{M},\mathsf{H}} \begin{cases} v & \text{if } \mathsf{M}(x) = g \ \wedge \ \mathsf{H}(g) = (v, T') \ \wedge T \subseteq T' \\ \texttt{CKR_TEMPLATE_INCONSISTENT} & \text{otherwise} \end{cases}$$

checkTemplate returns the value of the key pointed to by handle stored in x if its template has all the attribute requested by the query template T set. Encryption and key diversification semantics is straightforward: let $e \downarrow^{\mathsf{M},\mathsf{H}} v$ and $\mathsf{M}(x) = v'$, then $\mathsf{enc}(e, x) \downarrow^{\mathsf{M},\mathsf{H}} enc(v, v')$ and $\mathsf{diversifyKey}(tag, x) \downarrow^{\mathsf{M},\mathsf{H}} enc(tag, v')$. The encryption mechanism modeled does not perform any integrity check on the messages so when decrypting a ciphertext, if the right key is supplied, the payload is correctly returned, i.e., $\mathsf{dec}(e, x) \downarrow^{\mathsf{M},\mathsf{H}} v$ if $e \downarrow^{\mathsf{M},\mathsf{H}} enc(v, v')$ and $\mathsf{M}(x) = v'$, otherwise $\mathsf{dec}(e, x) \downarrow^{\mathsf{M},\mathsf{H}} dec(v, v')$ (with $e \downarrow^{\mathsf{M},\mathsf{H}} v$ and $\mathsf{M}(x) = v'$).

A program c is a sequence of assignments or a failure:

$$\mathsf{c} ::= \mathsf{FAIL}[err] \mid x := e \mid x := \mathsf{genKey}(T) \mid x := \mathsf{importKey}(y, T) \mid \mathsf{c}_1; \mathsf{c}_2$$

The semantics is pretty standard and is reported in Table 6.1. Whenever an expression evaluates to an error, the program transits to FAIL reporting the error message. Key generation command (genKey) extracts a new key (g) and a new handle (g') from G and maps them in the handle-map using the given template T; the handle is also stored in the memory for future usages of the generated key. Note also that

if a sensitive key is generated, the template will also have its A attribute set: this is performed by the $\oplus$ function which returns the original template T if $S \notin T$ and $T \cup \{A\}$ otherwise. Similarly, it is possible to import a key in the handle-map.

Table 6.1 Commands Semantics

$$[assign]\quad \frac{e \downarrow^{\mathsf{M},\mathsf{H}} v}{\langle \mathsf{M}, \mathsf{H}, x := e\rangle \to \langle \mathsf{M}[x \mapsto v], \mathsf{H}, \varepsilon\rangle} \qquad [err]\quad \frac{e \downarrow^{\mathsf{M},\mathsf{H}} err}{\langle \mathsf{M}, \mathsf{H}, x := e\rangle \to \langle \mathsf{M}, \mathsf{H}, \mathsf{FAIL}[err]\rangle}$$

$$[seq1]\quad \frac{\langle \mathsf{M}_1, \mathsf{H}_1, \mathsf{c}_1\rangle \to \langle \mathsf{M}_2, \mathsf{H}_2, \varepsilon\rangle}{\langle \mathsf{M}_1, \mathsf{H}_1, \mathsf{c}_1; \mathsf{c}_2\rangle \to \langle \mathsf{M}_2, \mathsf{H}_2, \mathsf{c}_2\rangle} \qquad [seq2]\quad \frac{\langle \mathsf{M}_1, \mathsf{H}_1, \mathsf{c}_1\rangle \to \langle \mathsf{M}_2, \mathsf{H}_2, \mathsf{c}_1'\rangle}{\langle \mathsf{M}_1, \mathsf{H}_1, \mathsf{c}_1; \mathsf{c}_2\rangle \to \langle \mathsf{M}_2, \mathsf{H}_2, \mathsf{c}_1'; \mathsf{c}_2\rangle}$$

$$[genkey]\quad \frac{g, g' \leftarrow G}{\langle \mathsf{M}, \mathsf{H}, x := \mathsf{genKey}(T)\rangle \to \langle \mathsf{M}[x \mapsto g'], \mathsf{H}[g' \mapsto (g, \oplus(T))], \varepsilon\rangle}$$

$$[impkey]\quad \frac{g \leftarrow G \quad \mathsf{M}(y) = v}{\langle \mathsf{M}, \mathsf{H}, x := \mathsf{importKey}(y, T)\rangle \to \langle \mathsf{M}[x \mapsto g], \mathsf{H}[g \mapsto (v, T)], \varepsilon\rangle}$$

6.3 Type System

A secure implementation of the PKCS#11 API should never leak any sensitive key. This property can be enforced by a type system requiring that (i) every key has a specific unique purpose and (ii) keys can only be wrapped using secret (i.e. sensitive) and trusted keys.

A handle type is a template (T), reporting information about the properties set on the key when it has been generated or imported into a secure device.

Keys are divided into sensitive and non-sensitive: two confidentiality levels (ℓ_C) are needed, secret (high-confidentiality) H and public (low-confidentiality) L. Confidentiality levels are ordered by the operator $\sqsubseteq_C$ revealing that sensitive keys need to be used more carefully than non-sensitive one: $\mathsf{L} \sqsubseteq_C \mathsf{H}$. Similarly, keys are also classified by their integrity level ℓ_I: a key having its A attribute set is a high-integrity one (H) otherwise it is low-integrity (L). In fact, the always-sensitive PKCS#11 attribute cannot be set by a user when generating or unwrapping a key (see RSA PKCS#11 Standard [51], Table 15 footnotes 4 and 6), this attribute is meant to be automatically managed by the tamper resistant token whenever a key is generated as sensitive (since once set, the sensitive attribute cannot be unset anymore). The ordering relations ($\sqsubseteq_I$) on integrity levels is counter-variant: low-integrity, tainted values have to be used more carefully than high-integrity, untainted ones, i.e., $\mathsf{H} \sqsubseteq_I \mathsf{L}$.

Every key has a security level ℓ, which is the concatenation of a confidentiality level ℓ_C and an integrity one ℓ_I, $\ell = \ell_C\ell_I$. The order between security levels, denoted $\sqsubseteq$, is defined as the component-wise application of $\sqsubseteq_C$ and $\sqsubseteq_I$.

A key type could either be D^ℓ identifying a key with security level ℓ whose purpose is to encrypt and decrypt (public) data, or $\mathsf{W}^{\ell_1}[\ell_2]$ denoting a wrapping key with security level ℓ_1 used to encrypt and decrypt keys at level ℓ_2 (note that it must be that $\ell_i = \ell_C^i\ell_I^i$ and $\ell_C^2 \sqsubseteq_C \ell_C^1$; we always assume such requirement is satisfied on types). There could also be keys that mix the two functionalities, such keys are really dangerous since different kinds of data could be encrypted and when decrypting a message there is no way to be sure about the nature of its payload: well known attacks against PKCS#11 exploit such "feature". These keys will be typed with their security level ℓ and will be properly limited in their usages by the type system. Types are ranged over by τ and are defined by the following syntax:

$$\tau \ ::= \ T \mid \ell \mid \mathsf{D}^\ell \mid \mathsf{W}^{\ell_1}[\ell_2]$$

A sub-type relation is defined over non-handle types: $\ell \leq \ell'$ if $\ell \sqsubseteq \ell'$, $\mathsf{D}^\ell \leq \ell$, $\mathsf{W}^\ell[\ell'] \leq \ell$ and also $\mathsf{D}^{\ell_1} \leq \mathsf{D}^{\ell_2}$ and $\mathsf{W}^{\ell_1}[\ell] \leq \mathsf{W}^{\ell_2}[\ell]$ if $\ell_1 \sqsubseteq \ell_2$. Non-sensitive and untrusted keys (at level LL) cannot be used to perform any critical operation such as wrapping a sensitive key. Such keys will be only employed to encrypt and decrypt public messages: the sub-typing relation will so consider $\mathsf{LL} \leq \mathsf{D}^{\mathsf{HL}}$. Handles are public values available to the user of the API, they are not in any sub-type relation except for the fact that any handle could be safely considered as public byte-stream, i.e., $T \leq \mathsf{LL}$. We also assume that level LH never appears in any type, indeed there is no public trusted data that the API needs to be aware of.

Templates, used to type a handle to a key, can be automatically converted to key types: templates can be grouped into class of keys to which they refer to. Given a template T it is first necessary to understand if it defines a unique purpose for a key, note that whenever it is not possible to map a template to a unique usage (because the key has not any capability flag set or because it mixes different purposes) the special value $\bot$ is derived. In the following, $\models_W$ derives the fact that a given template refers to a wrapping/unwrapping key or not, $\models_D$ does the same for encryption/decryption keys and $\models$ finally states the key role calculated from the template.

$$\frac{W \in T \ \vee \ U \in T}{\models_W T : \mathsf{W}} \qquad \frac{W \notin T \ \wedge \ U \notin T}{\models_W T : \bot} \qquad \frac{D \in T \ \vee \ E \in T}{\models_D T : \mathsf{D}} \qquad \frac{D \notin T \ \wedge \ E \notin T}{\models_D T : \bot}$$

$$\frac{\models_D T : \mathsf{D} \quad \models_W T : \bot}{\models T : \mathsf{D}} \qquad \frac{\models_D T : \bot \quad \models_W T : \mathsf{W}}{\models T : \mathsf{W}} \qquad \frac{\models_D T : \bot \quad \models_W T : \bot}{\models T : \bot} \qquad \frac{\models_D T : \mathsf{D} \quad \models_W T : \mathsf{W}}{\models T : \bot}$$

Key types are then derived (denoted $\vdash T : \tau$) by calculating the security level induced by the template.

$$\frac{A \in T}{\models_\ell T : \mathsf{HH}} \qquad \frac{S \in T \quad A \notin T}{\models_\ell T : \mathsf{HL}} \qquad \frac{S \notin T}{\models_\ell T : \mathsf{LL}}$$

$$\frac{\models T : \perp \quad \models_\ell T : \ell}{\vdash T : \ell} \qquad \frac{\models T : \mathsf{D} \quad \models_\ell T : \ell}{\vdash T : \mathsf{D}^\ell} \qquad \frac{\models T : \mathsf{W} \quad \models_\ell T : \ell \neq \mathsf{LL}}{\models T : \mathsf{W}^\ell[\mathsf{HL}]} \qquad \frac{\models T : \mathsf{W} \quad \models_\ell T : \mathsf{LL}}{\models T : \mathsf{LL}}$$

Note that wrapping templates are mapped to a key type which can wrap any other key, this is because there is no information about the target keys on the template.

Security policy The type system is parametric with respect to a policy which governs key generation and import. More precisely, a policy P is made of two components: P_{GEN}, a set of templates which are considered legal when generating a new key, and P_{IMP} a set of templates which can be given to a handle of an unwrapped key. Unwrapped keys cannot be regarded to be always-sensitive (or trusted) and their original capabilities are unknown: it must be that $\forall T \in P_{IMP}.\ \vdash T : \mathsf{HL}$. The policy P can be defined by the API programmer to implement security patches to PKCS#11 standard: the type system enforces the fact that sensitive keys are never leaked, so if a developer can give a policy P that let his PKCS#11 implemention to type-check then the fix proposed (in terms of the policy) is a good one. By the way, the policy is also useful to encode some restrictions required by the RSA standard itself. From now on, every policy P is supposed to satisfies the following property: $\forall\ T \in P_{GEN} \cup P_{IMP},\ A \notin T$. Indeed the always-sensitive attribute must not be used when generating or unwrapping a key as already discussed above.

Expressions Expressions getObj is straightforward to type: rule (get) maps the type of the handle (i.e., a template) to its corresponding key type. A user could also supply a byte-stream as handle when trying to get a key's raw value, in that case the type system will recover to a conservative type for the result: any returned value will be typed as the top of the type hierarchy (HL) by rule (get-un).

The checkTemplate expression queries the template of the key pointed to from the handle x against T deriving some partial information about the real template of the key: rule (chk) approximates the result's type by inspecting the type of all the templates allowed by the policy P and matching the query: more precisely, it uses the function $\mathsf{topQ}(T, P)$, defined below, to calculate the least upper bound of the types corresponding to templates in the policy P satisfying the query T.

$$\mathsf{topQ}(T, P) \;=\; \bigsqcup_{\substack{T' \in P_{GEN} \cup P_{IMP} \\ T \subseteq T', \vdash T' : \tau'}} \tau'$$

Key diversification properly specializes a given key to respectively a data or a wrapping key depending on the tag issued.

Encrypting a public byte-stream message, rule (enc), always yields a public byte-stream, indeed there is no secret to protect and thus any kind of data is allowed to be

used as key except for strictly wrapping trusted keys ($\mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]$) which are intended only to wrap trusted keys. If potentially sensitive data has to be encrypted, rule (wrap) requires the use of an always-sensitive wrapping key. Note, however, that you cannot use a data key (D^{HH}) or a non-sensitive one (LL) to encrypt something which is not a byte-stream. Decryption by means of a (sensitive) data key types as a public byte-stream indeed no sensitive values are ever encrypted using such a key (see rule (dec)). If, otherwise, a (sensitive) key with no clear unique purpose is used to decrypt a message, rule (unwrap) will protect the result regarding it as a potentially sensitive untrusted key. Rules (hh-w) and (hh-u) govern the usage of the special keys intended to only wrap and unwrap other trusted keys: it is interesting to note that in such case when unwrapping a key it is possible to regard it as being trusted.

Table 6.2 Typing expressions

$$\text{(sub)}\ \frac{\Gamma \vdash_P e : \tau' \leq \tau}{\Gamma \vdash_P e : \tau} \qquad \text{(var)}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash_P x : \tau} \qquad \text{(get)}\ \frac{\Gamma \vdash_P x : T \quad \vdash T : \tau}{\Gamma \vdash_P \mathsf{getObj}(x) : \tau}$$

$$\text{(get-un)}\ \frac{\Gamma \vdash_P x : \mathsf{LL}}{\Gamma \vdash_P \mathsf{getObj}(x) : \mathsf{HL}} \qquad \text{(chk)}\ \frac{\Gamma \vdash_P x : \mathsf{LL} \quad \mathsf{topQ}(T, P) = \tau}{\Gamma \vdash_P \mathsf{checkTemplate}(x, T) : \tau}$$

$$\text{(dk)}\ \frac{\Gamma \vdash_P x : \ell}{\Gamma \vdash_P \mathsf{diversifyKey}(\mathtt{D}, x) : \mathsf{D}^{\ell}} \qquad \text{(dw-h)}\ \frac{\Gamma \vdash_P x : \ell \quad \ell \neq \mathsf{LL}}{\Gamma \vdash_P \mathsf{diversifyKey}(\mathtt{W}, x) : \mathsf{W}^{\ell}[\mathsf{HL}]}$$

$$\text{(dw-l)}\ \frac{\Gamma \vdash_P x : \mathsf{LL}}{\Gamma \vdash_P \mathsf{diversifyKey}(\mathtt{W}, x) : \mathsf{LL}} \qquad \text{(dww)}\ \frac{\Gamma \vdash_P x : \mathsf{HH}}{\Gamma \vdash_P \mathsf{diversifyKey}(\mathtt{W}^2, x) : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]}$$

$$\text{(enc)}\ \frac{\Gamma \vdash_P x : \mathsf{HL} \quad \Gamma(x) \neq \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}] \quad \Gamma \vdash_P e : \mathsf{LL}}{\Gamma \vdash_P \mathsf{enc}(e, x) : \mathsf{LL}} \qquad \text{(wrap)}\ \frac{\Gamma \vdash_P x : \mathsf{W}^{\mathsf{HH}}[\mathsf{HL}] \quad \Gamma \vdash_P e : \mathsf{HL}}{\Gamma \vdash_P \mathsf{enc}(e, x) : \mathsf{LL}}$$

$$\text{(dec)}\ \frac{\Gamma \vdash_P x : \mathsf{D}^{\mathsf{HL}} \quad \Gamma \vdash_P e : \mathsf{LL}}{\Gamma \vdash_P \mathsf{dec}(e, x) : \mathsf{LL}} \qquad \text{(unwrap)}\ \frac{\Gamma \vdash_P x : \mathsf{HL} \quad \Gamma \vdash_P e : \mathsf{LL}}{\Gamma \vdash_P \mathsf{dec}(e, x) : \mathsf{HL}}$$

$$\text{(hh-w)}\ \frac{\Gamma \vdash_P x : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}] \quad \Gamma \vdash_P e : \mathsf{HH}}{\Gamma \vdash_P \mathsf{enc}(e, x) : \mathsf{LL}} \qquad \text{(hh-u)}\ \frac{\Gamma \vdash_P x : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}] \quad \Gamma \vdash_P e : \mathsf{LL}}{\Gamma \vdash_P \mathsf{dec}(e, x) : \mathsf{HH}}$$

Commands Rules (gen), (imp-l) and (imp-h) apply the policy P as expected, note that generated sensitive keys will be marked with A flag (always-sensitive). The two distinct rules for key import consider respectively the usage of an untrusted key, in which case the template will be required to be in the P_{IMP} component of the policy, or a trusted one, permitting to import a key whose template will have its A attribute set, correctly tracking the fact that it is an high-integrity key. Rules (assign) and (seq) are standard.

Table 6.3 Typing commands

$$\text{(assign)}\quad \frac{\Gamma(x) = \tau \quad \Gamma \vdash_P e : \tau}{\Gamma \vdash_P x := e} \qquad \text{(gen)}\quad \frac{\Gamma(x) = \oplus(T) \quad T \in P_{GEN}}{\Gamma \vdash_P x := \mathsf{genKey}(T)}$$

$$\text{(fail)}\quad \Gamma \vdash_P \mathsf{FAIL}[err] \qquad \text{(seq)}\quad \frac{\Gamma \vdash_P \mathsf{c}_1 \quad \Gamma \vdash_P \mathsf{c}_2}{\Gamma \vdash_P \mathsf{c}_1; \mathsf{c}_2}$$

$$\text{(imp-l)}\quad \frac{\Gamma(x) = T \quad \Gamma \vdash_P y : \mathsf{HL} \quad T \in P_{IMP} \quad \vdash T : \mathsf{HL}}{\Gamma \vdash_P x := \mathsf{importKey}(y, T)}$$

$$\text{(imp-h)}\quad \frac{\Gamma(x) = T \quad \Gamma \vdash_P y : \mathsf{HH} \quad T \setminus \{A\} \in P_{GEN} \quad \vdash T : \mathsf{HH}}{\Gamma \vdash_P x := \mathsf{importKey}(y, T)}$$

6.3.1 Type Soundness

Sensitive values are identified observing the public portion of a memory and comparing it with the keys stored in the handle map. The function priv returns true if the given value does not appear in any byte-stream (LL) variable: $\mathsf{priv}_{\Gamma,\mathsf{M}}(v) = \forall x.\ \Gamma(x) \leq \mathsf{LL},\ \mathsf{M}(x) \neq v$. A value is sensitive if it is stored on the handle-map with only sensitive templates and it does not appear in the public memory. Let botH be a function returning, if it exists, the greatest lower bound of all the possible types (templates) assigned by H to a given value:

$$\mathsf{botH}(v, \mathsf{H}) = \prod_{\substack{(v,T) \in cod(\mathsf{H}) \\ \vdash T:\ \tau'}} \tau'$$

It is now possible to precisely describe sensitive values.

Definition 6.1. *A value v is* sensitive *according to a type environment* Γ*, a memory* M *and a handle-map* H *if* $v \in G$*,* $\mathsf{priv}_{\Gamma,\mathsf{M}}(v)$ *and* $\mathsf{botH}(v, \mathsf{H}) = \tau \neq \mathsf{LL}$ *(if* $(v, T) \in$ *cod*(H)*).*

An attacker is represented by a program using only variables whose type τ is such that $\tau \leq \mathsf{LL}$. Indeed, our model assumes that these are the only variables to which an external user of the API has access to. The idea is that attacker can be simulated by inserting attack code before and after each API command implementation: this model the fact that a malicious user can tamper with any parameter to be sent to the PKCS#11 token and also with its returned values and use these to call (the same or) other commands any number of times.

Some assumptions must be done on the initial memory and handle-map. The memory does not store any sensitive key value in its public (i.e., LL) variables, and messages encrypted with a non-sensitive key (or a byte-stream) does not contain any sensitive key in their payload. With respect to the handle-map, the templates assigned from the type environment to the handles must agree with values mapped via H. These ideas can be formalized by typing values and showing that a memory and handle-map respects the hypothesis of a given type environment. Moreover, notice that *err* values will never appear in any memory.

The notation $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v : \tau$ states that value v types τ under type-environment Γ, memory M and handle-map H. First of all, if $v \in N$ then $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v : \mathsf{LL}$. For values in G if the value is not in the public portion of the memory then it will be typed as described by the handle-map (with the greatest lower bound of all the templates under which the value appears), otherwise it will be a byte-stream. Typing of g follows:

$$\frac{v \in G \quad \mathsf{priv}_{\Gamma,\mathsf{M}}(v) \quad \mathsf{botH}(v, \mathsf{H}) = \tau \neq \mathsf{LL}}{\Gamma \vdash_{\mathsf{M},\mathsf{H}} v : \tau}$$

$$\frac{v \in G \quad (\neg(\mathsf{priv}_{\Gamma,\mathsf{M}}(v)) \vee \nexists\, \mathsf{botH}(v, \mathsf{H}) \ \vee\ \mathsf{botH}(v, \mathsf{H}) = \mathsf{LL})}{\Gamma \vdash_{\mathsf{M},\mathsf{H}} v : \mathsf{LL}}$$

Values are subject to the sub-typing relation given on types. All encrypted values must type LL but if their payload is not a public byte-stream then an always-sensitive wrapping key is required. Diversified keys type as expected based on the used tag. Decrypted messages require a public encrypted input and are typed as if the right key has been used (even if it is not the case, since such values can only be generated by wrong decryption). Rules are in Appendix D.1.

It is now possible to formalize the assumptions sketched above. First of all, always-sensitive keys must not be known outside of the tamper-resistant device: this is modeled by requiring that any always-sensitive key does not appear in public variables. Then it must be checked that a policy P is respected, i.e., every key in the handle-map should have a template allowed by P. More precisely, a non-sensitive key is required to have its template in P_{GEN} since all imported keys will be treated

as sensitive. Always-sensitive keys' templates, in the same way, are restricted only to templates in P_{GEN}, indeed these are keys generated by the secure device. Notice however that any template in the policy cannot have the attribute A set so in checking this case, this attribute will be removed. A sensitive, non-always-sensitive key's template must be allowed by P_{IMP} representing the fact that such key is an (untrusted) imported one.

Definition 6.2. (Memory and handle-map well-formedness)
A memory M *and handle-map* H *are well-formed with respect to a type environment* Γ *and policy* P*, written* $\Gamma \vdash_P \mathsf{M}, \mathsf{H}$*, if*

1. $\forall (v, T) \in cod(\mathsf{H})$, $A \in T$ *implies* $\mathsf{priv}_{\Gamma,\mathsf{M}}(v)$

2. *and for any variable* x *in the domain of* M *it holds*

    - $\Gamma(x) = \tau \neq T$ *and* $\mathsf{M}(x) = v$ *with* $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v : \tau' \leq \tau$*, or*
    - $\Gamma(x) = T$, $\mathsf{M}(x) = v$, $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v : \mathsf{LL}$, $\vdash T : \tau^T$ *and*
        - $S \notin T$ *implies* $T \in P_{GEN}$
        - $A \in T$ *implies* $T \setminus \{A\} \in P_{GEN}$
        - $S \in T$ *and* $A \notin T$ *implies* $T \in P_{IMP}$
        - *if* $v \in dom(\mathsf{H})$ *then* $\mathsf{H}(v) = (v', T)$ *with* $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v' : \tau \leq \tau^T$

The security of typed programs is proved by the following two theorems. It is first shown that well-typed programs preserve memory and handle-map well-formedness (Theorem 6.1), then it is also proved that sensitive values are never leaked (Theorem 6.2).

Theorem 6.1. *Let* $\Gamma \vdash_P \mathsf{M}, \mathsf{H}$ *and* $\Gamma \vdash_P \mathsf{c}$*. If* $\langle \mathsf{M}, \mathsf{H}, \mathsf{c} \rangle \to \langle \mathsf{M}', \mathsf{H}', \mathsf{c}' \rangle$ *then*

- *if* $\mathsf{c}' \neq \varepsilon$ *then* $\Gamma \vdash_P \mathsf{c}'$ *and*

- $\Gamma \vdash_P \mathsf{M}', \mathsf{H}'$.

Theorem 6.2. *Let* $\Gamma \vdash_P \mathsf{M}, \mathsf{H}$ *and* $\Gamma \vdash_P \mathsf{c}$*. If* $\langle \mathsf{M}, \mathsf{H}, \mathsf{c} \rangle \to \langle \mathsf{M}', \mathsf{H}', \mathsf{c}' \rangle$ *and* $v \in G$ *then*

1. $\mathsf{priv}_{\Gamma,\mathsf{M}}(v)$ *and* $\mathsf{botH}(v, \mathsf{H}) = \tau \neq \mathsf{LL}$ *implies* $\mathsf{priv}_{\Gamma,\mathsf{M}'}(v)$ *and* $\mathsf{botH}(v, \mathsf{H}') = \tau \neq \mathsf{LL}$*, and*

2. $\mathsf{priv}_{\Gamma,\mathsf{M}}(enc(tag, v))$ *and* $\mathsf{botH}(enc(tag, v), \mathsf{H}) = \tau \neq \mathsf{LL}$ *implies* $\mathsf{priv}_{\Gamma,\mathsf{M}'}(enc(tag, v))$ *and* $\mathsf{botH}(enc(tag, v), \mathsf{H}') = \tau \neq \mathsf{LL}$.

Formal proofs are in Appendix D.2.

6.4 Typed PKCS#11

PKCS#11 offers commands to encrypt and decrypt data as well as to wrap and unwrap keys. This section want to provide a way to implement such commands and type them to be sure that they are not vulnerable to attacks. Each API's command will be modeled as a program reading inputs from pre-defined variables and assigning the result to the *ret* variable.

The semantics of the commands is defined by the RSA's standard so the typing for input parameters and the result of each API function is fixed. `C_Encrypt` takes a byte-stream and a handle to a key having its E flag set and returns an encrypted byte-stream, similarly `C_Decrypt` take a byte-stream and decrypt it using the key pointed to the given handle (having its D flag set) and returns to the user the decrypted message. `C_WrapKey` takes the handle of a key to be wrapped and the one pointing to the wrapping key (this is required to have its W flag set and to be sensitive, a point missed by the standard but necessary to preserve wrapped key confidentiality) returning an encrypted byte-stream. The unwrap command (`C_UnwrapKey`) reads a byte-stream and decrypt it using the unwrapping key (a sensitive key having its U flag set) returning a handle to a key whose value corresponds to the decrypted message.

The following are the signatures of each command:

$$\texttt{C_Encrypt}(data : \mathsf{LL},\ h_key : \mathsf{LL}) : \mathsf{LL}$$

$$\texttt{C_Decrypt}(data : \mathsf{LL},\ h_key : \mathsf{LL}) : \mathsf{LL}$$

$$\texttt{C_WrapKey}(h_key : \mathsf{LL},\ h_w : \mathsf{LL}) : \mathsf{LL}$$

$$\texttt{C_UnwrapKey}(data : \mathsf{LL},\ h_w : \mathsf{LL}) : \mathsf{LL}$$

6.4.1 RSA PKCS#11 Standard

This sections shows that an implementation of PKCS#11 which exactly follows the standard, fails to type-check: this is expected since it is known to be vulnerable to attacks.

The standard impose that the always-sensitive attribute must never be used when generating or importing a key and does not add any policy for the encrypt, decrypt, wrap and unwrap attributes. The policy P encoding these requirements, allows P_{GEN} to contain any template both for sensitive and non-sensitive keys, i.e., $P_{GEN} = \{T.\ A \notin T\}$; in the same way for P_{IMP}: $P_{IMP} = P_{GEN}$. Notice however, that the model presented here does not allow for such permissive key import policy, so we suppose $P_{IMP} = \{T.\ \vdash T : \mathsf{HL}\}$: in this way we are adding more constraints than the one asked by the standard, but these are not enough to make it secure. The two commands used to mount the attacks, i.e., `C_Decrypt` and `C_WrapKey`, are

considered. In the following, to the right of the assignments the required types for the variables are reported.

$\texttt{C_Decrypt}(data : \mathsf{LL},\ h_key : \mathsf{LL}) : \mathsf{LL}$

$$\begin{array}{ll} k := \mathsf{checkTemplate}(h_key,\ \{D\}); & (\Gamma(k) = \mathsf{D}^{\mathsf{HL}}) \\ ret := \mathsf{dec}(data, k); & (\Gamma(ret) = \mathsf{LL}) \end{array}$$

The *ret* variable must be typed LL and thus the decrypt would state that $\Gamma \vdash_P k : \mathsf{D}^{\mathsf{HL}}$. But the checkTemplate expression, which queries a handle for a decryption key ($\{D\}$), will not be able to type it as needed: in fact, $\mathsf{topQ}(\{D\}, P) = \mathsf{HL}$ since it will be possible to have any kind of decryption key, also one that mixes this functionality with the wrapping one. It is thus impossible to type the above code.

Similarly, when querying for a wrapping key ($\{W\}$) the result will never be typed $\mathsf{W}^{\mathsf{HH}}[\mathsf{HL}]$ and so the following implementation of the wrap command does not type-check.

$\texttt{C_WrapKey}(h_key : \mathsf{LL},\ h_w : \mathsf{LL}) : \mathsf{LL}$

$$\begin{array}{ll} w := \mathsf{checkTemplate}(h_w,\ \{W\}) & (\Gamma(w) = \mathsf{W}^{\mathsf{HH}}[\mathsf{HL}]) \\ k := \mathsf{getObj}(h_key); & (\Gamma(k) = \mathsf{HL}) \\ ret := \mathsf{enc}(k, w); & (\Gamma(ret) = \mathsf{LL}) \end{array}$$

6.4.2 Key Diversification

This section presents a novel patch to PKCS#11. The idea is to use key diversification to avoid the same key to be used for conflicting purposes. Two different implementations for the wrap and unwrap commands are given: one using the common wrapping key which can export/import any kind of key (even a non-sensitive one) and the other one employing a wrapping key intended only for wrapping and unwrapping always-sensitive keys.

The policy P is the most liberal one, used above: $P_{GEN} = \{T.\ A \notin T\}$ and $P_{IMP} = \{T.\ \vdash T : \mathsf{HL}\}$. Encrypt and decrypt commands follow.

$\texttt{C_Encrypt}(data : \mathsf{LL},\ h_key : \mathsf{LL}) : \mathsf{LL}$

$$\begin{array}{ll} k := \mathsf{checkTemplate}(h_key,\ \{E\}) & (\Gamma(k) = \mathsf{HL}) \\ dk := \mathsf{diversifyKey}(\texttt{D}, k); & (\Gamma(dk) = \mathsf{D}^{\mathsf{HL}}) \\ ret := \mathsf{enc}(data, dk); & (\Gamma(ret) = \mathsf{LL}) \end{array}$$

$\texttt{C_Decrypt}(data : \mathsf{LL},\ h_key : \mathsf{LL}) : \mathsf{LL}$

$$\begin{array}{ll} k := \mathsf{checkTemplate}(h_key,\ \{D\}) & (\Gamma(k) = \mathsf{HL}) \\ dk := \mathsf{diversifyKey}(\texttt{D}, k); & (\Gamma(dk) = \mathsf{D}^{\mathsf{HL}}) \\ ret := \mathsf{dec}(data, dk); & (\Gamma(ret) = \mathsf{LL}) \end{array}$$

The wrap and unwrap commands below, are the ones which allow for the export and import of any kind of cryptographic key: this result in a sever restriction when

unwrapping a key since it has to be treated as an untrusted sensitive key no matter what the real type of the original key was. Note that the wrap command require an always-sensitive key to be used: this is a requirement which any patch aiming at protecting the key export functionalities has to impose, indeed exporting a key out of the token encrypted under an untrusted non-sensitive key opens up the way to straightforward attacks.

$\texttt{C_WrapKey}^{\mathsf{HL}}(h_key : \mathsf{LL},\ h_w : \mathsf{LL}) : \mathsf{LL}$

$$\begin{array}{ll} w := \mathsf{checkTemplate}(h_w,\ \{A, W\}) & (\Gamma(w) = \mathsf{HH}) \\ k := \mathsf{getObj}(h_key); & (\Gamma(k) = \mathsf{HL}) \\ dk := \mathsf{diversifyKey}(\texttt{W}, w); & (\Gamma(dk) = \mathsf{W}^{\mathsf{HH}}[\mathsf{HL}]) \\ ret := \mathsf{enc}(k, dk); & (\Gamma(ret) = \mathsf{LL}) \end{array}$$

$\texttt{C_UnwrapKey}^{\mathsf{HL}}(data : \mathsf{LL},\ h_w : \mathsf{LL}) : \mathsf{LL}$

$$\begin{array}{ll} w := \mathsf{checkTemplate}(h_w,\ \{U\}) & (\Gamma(w) = \mathsf{HL}) \\ dk := \mathsf{diversifyKey}(\texttt{W}, w); & (\Gamma(dk) = \mathsf{HL}) \\ k := \mathsf{dec}(data, dk); & (\Gamma(k) = \mathsf{HL}) \\ ret := \mathsf{importKey}(k, \{S, D, E, W, U\}); & (\Gamma(ret) = \mathsf{LL}) \end{array}$$

The following wrap and unwrap commands use always-sensitive wrapping keys employed only to wrap and unwrap other trusted keys, note that in this case also the unwrap command requires a trusted key. The wrap function will ensure that the key to be wrapped is a trusted one.

$\texttt{C_WrapKey}^{\mathsf{HH}}(h_key : \mathsf{LL},\ h_w : \mathsf{LL}) : \mathsf{LL}$

$$\begin{array}{ll} w := \mathsf{checkTemplate}(h_w,\ \{A, W\}) & (\Gamma(w) = \mathsf{HH}) \\ k := \mathsf{checkTemplate}(h_key,\ \{A\}); & (\Gamma(k) = \mathsf{HH}) \\ dk := \mathsf{diversifyKey}(\texttt{W}^2, w); & (\Gamma(dk) = \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]) \\ ret := \mathsf{enc}(k, dk); & (\Gamma(ret) = \mathsf{LL}) \end{array}$$

$\texttt{C_UnwrapKey}^{\mathsf{HH}}(data : \mathsf{LL},\ h_w : \mathsf{LL}) : \mathsf{LL}$

$$\begin{array}{ll} w := \mathsf{checkTemplate}(h_w,\ \{A, U\}) & (\Gamma(w) = \mathsf{HH}) \\ dk := \mathsf{diversifyKey}(\texttt{W}^2, w); & (\Gamma(dk) = \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]) \\ k := \mathsf{dec}(data, dk); & (\Gamma(k) = \mathsf{HH}) \\ ret := \mathsf{importKey}(k, \{A, S, D, E, W, U\}); & (\Gamma(ret) = \mathsf{LL}) \end{array}$$

Importing a wrapped key as trusted in token, allows for it to be used also to export other (trusted) keys. This is a very useful feature which could be used for example to copy all the keys of a token (implementing this patch) to another one (also using the same fix) to obtain a copy of it: it will be necessary to wrap all the keys stored inside the secure device and re-import them into the other one which will be able to perform exactly all the same commands performed by the first token.

Notice also that it would be possible to build a secure PKCS#11 module which implements both the $\texttt{C_WrapKey}^{\textsf{HL}}$ and the $\texttt{C_WrapKey}^{\textsf{HH}}$ commands (and their respectively unwraps). Then the user would be asked to choose the mode in which the token has to operate, moreover this can be changed at any time without exposing the stored keys to attacks: this is guaranteed by the fact that both the implementations type-checks and typed programs can be safely composed.

This patch is completely transparent to the user: it does not bother with some restrictive policy on the attributes of generated keys but instead it takes care of ensuring the same key is never used for encrypting and decrypting both data and other keys. It must be noted that this breaks the compatibility with other devices, indeed a key wrapped by a token implementing this patch cannot be correctly imported by one acting as described by the standard (and vice versa), the same holds for encrypted data.

Note also that for two devices to be able to communicate (exchanging encrypted data and keys) a common master key has to be shared. The token should provide a special operation allowing a user logged in as a *Security Officer*, which must be supposed to operate on an isolate and safe environment, to install such a key in all the devices which need to share data and keys.

This is, to the best of our knowledge, the only patch that ensure to keep sensitive keys secure and to let any developed application, talking with a device implementing the patch, to work as expected without raising any error.

6.4.3 Secure Templates

Secure templates is a patch proposed by the authors and others [16, 15] which is the first secure PKCS#11 configuration to appear in the literature that does not require any cryptographic mechanisms to be added to the standard. Here it is shown to be type-checkable giving a proof that it does not leak any sensitive key.

This security fix limits the set of admissible attribute combinations for keys in order to avoid that they ever assume conflicting roles at creation time. This is configurable at the level of the specific PKCS#11 operation. For example, different secure templates can be defined for different operations such as key generation and unwrapping.

More precisely, the patch allows three templates for the key generation command: a wrap and unwrap one for exporting/importing other keys, an encrypt and decrypt template for cryptographic operations, and an empty template. The unwrap command is instead allowed to set either an empty template or one which has the unwrap and encrypt attributes set and the wrap and decrypt ones unset.

These rules are encoded in a policy P which sets P_{GEN} to contain all the templates T such that $T \setminus \{S, A\} \subseteq \{D, E\}$ or $T \setminus \{S, A\} \subseteq \{W, U\}$, while the P_{IMP} component imposes that any imported key does not have its D and W attribute set: $P_{IMP} = \{T.\ W \notin T \wedge D \notin T \wedge A \notin T \wedge S \in T\}$, the requirements on the A and S attributes are due to the general rule that any template in the import component of

a policy has to be mapped to HL key type. Notice that the patch is presented here in an extended version: originally it allows the generation of sensitive keys only, we instead let non-sensitive keys to be accepted by the P_{GEN} policy component (governing key generation).

With such a policy, whenever a checkTemplate expression queries a handle for a decryption key ($\{D\}$) then the type returned is D^{HL}: if $D \in T$ then $T \in P_{GEN}$ and more precisely there are exactly two templates satisfying the query, $T_1 = \{D, E\}$ and $T_2 = \{S, D, E\}$ (meaning that there will be keys in the handle-map with template $T_2' = \{A, S, D, E\}$ since the A attribute is added by the key generation command); it holds $\vdash T_1 : \mathsf{LL}$ and $\vdash T_2' : \mathsf{D}^{\mathsf{HH}}$ from which $\mathsf{LL} \sqcup \mathsf{D}^{\mathsf{HH}} = \mathsf{D}^{\mathsf{HL}}$. Similarly, when querying for an always-sensitive wrapping key ($\{A, W\}$) the result will be typed $\mathsf{W}^{\mathsf{HH}}[\mathsf{HL}]$ since only key generated by the token can be used to wrap a key and the only template satisfying the query is $\oplus(\{S, W, U\})$.

C_Encrypt($data$: LL, h_key : LL) : LL

$$
\begin{array}{ll}
k := \mathsf{checkTemplate}(h_key,\ \{E\}) & (\Gamma(k) = \mathsf{HL}) \\
ret := \mathsf{enc}(data, k); & (\Gamma(ret) = \mathsf{LL})
\end{array}
$$

C_Decrypt($data$: LL, h_key : LL) : LL

$$
\begin{array}{ll}
k := \mathsf{checkTemplate}(h_key,\ \{D\}) & (\Gamma(k) = \mathsf{D}^{\mathsf{HL}}) \\
ret := \mathsf{dec}(data, k); & (\Gamma(ret) = \mathsf{LL})
\end{array}
$$

C_WrapKey(h_key : LL, h_w : LL) : LL

$$
\begin{array}{ll}
w := \mathsf{checkTemplate}(h_w,\ \{A, W\}) & (\Gamma(w) = \mathsf{W}^{\mathsf{HH}}[\mathsf{HL}]) \\
k := \mathsf{getObj}(h_key); & (\Gamma(k) = \mathsf{HL}) \\
ret := \mathsf{enc}(k, w); & (\Gamma(ret) = \mathsf{LL})
\end{array}
$$

C_UnwrapKey($data$: LL, h_w : LL) : LL

$$
\begin{array}{ll}
w := \mathsf{checkTemplate}(h_w,\ \{U\}) & (\Gamma(w) = \mathsf{HL}) \\
k := \mathsf{dec}(data, w); & (\Gamma(k) = \mathsf{HL}) \\
ret := \mathsf{importKey}(k, \{S, E, U\}); & (\Gamma(ret) = \mathsf{LL})
\end{array}
$$

With respect to the key diversification patch, this one pose some strong limits to the possible templates. This could be an issue if an application in use on a given system fails to obey to such requirements. Moreover, in this case compatibility with other devices is not broken (anyway using unsafe tokens would expose your keys under attacks on that devices). Also in this case for two token to be able to share wrapped keys, they have to be initialized with a common master key.

6.5 Conclusions

A type system suitable to check the security of PKCS#11 APIs has been introduced. It want to be a tool that helps developers and hardware producers to better understand the crucial bugs affecting the design and implementation of this standard.

In fact, the type system has been used to prove that `C_Decrypt` and `C_WrapKey` commands do not type if implemented as prescribed by the standard [51]. More precisely, it has been shown that the requirements on the templates of the keys used to perform such operation are not restrictive enough to avoid keys having conflicting purposes.

A new patch to the RSA standard [51], based on key diversification, has also been presented here. It lets the token to take care of separating roles for each key and free the user from keeping track of keys' templates to prevent well-known attacks [23, 27]. The type system has been used to verify the new patch as well as the secure templates one [16, 15] proving their security.

RSA introduced a new attribute, called `CKA_WRAP_WITH_TRUSTED`, to the standard starting from version 2.20. It could be helpful to prevent the attacks object of this work. The idea is that only the security officer would be able to import a trusted key (i.e., a key whose `CKA_TRUSTED` attribute is set) into a token and any sensitive key, which one would like to protect, has to have its `CKA_WRAP_WITH_TRUSTED` attribute set, meaning that it can only be wrapped under a trusted key: this recalls the idea of the special trusted key which can be diversified using the $\mathtt{W}^2$ tag in the key diversification patch, but to obtain the same result it is again necessary to restrict the keys' template to prevent them to be used for conflicting purposes. Investigating the security implications of these two attributes is indeed an interesting issue and is in the plan for an extension of the present work.

An implementation of the key diversification patch on a software emulated token is also left as a future work. As already done for the secure template patch [16, 15] the starting point would be the open-source project openCryptoki [49].

7

Tookan: a TOOl for cryptoKi ANalysis

This chapter shows how to extract sensitive cryptographic keys from a variety of commercially available tamper resistant cryptographic security tokens, exploiting vulnerabilities in their RSA PKCS#11 based APIs. The attacks are performed by Tookan, an automated tool we have developed [15], which reverse-engineers the particular token in use to deduce its functionality, constructs a model of its API for a model checker, and then executes any attack trace found by the model checker directly on the token.

Results of testing the tool on 17 commercially available tokens are presented: 9 were vulnerable to attack, while the other 8 had severely restricted functionalities. Tookan may be also used to verify patches to insecure devices, to this aim it is shown how it can prove that the secure templates patch [15], implemented in a software token simulator, is secure.

Tookan has also recently become a commercial product of a joint colloboration between INRIA and Ca' Foscari University.

7.1 Introduction

This chapter describes Tookan[1], an automated tool that reverse engineers the particular functionality offered by a device, constructs a formal model of this functionality, calls a model checker to search for possible attacks, and executes any attack trace found directly on the device. The model is based on previous work by Delaune, Kremer and Steel, [27], but enriched significantly to better match the functionality we found on real devices. We describe optimisations to the model building process that result in models which can be handled efficiently by the model checker. We also contribute a meta-language for describing PKCS#11 configurations, used by the reverse-engineering part of our tool.

The results of testing the tool on commercially available devices are disquieting: every device that offered the functionality necessary to import and export sensitive

[1] Tool for cryptoKi ANalysis

keys in an encrypted form, a standard key management operation, did so in an insecure way allowing the key value to be recovered after a few calls to the API. The tokens we tested that were not vulnerable to these attacks have very limited functionality (just asymmetric keypair generation and signing, for example).

We then show how to use our tool to verify patched tokens. We present CryptokiX, a fiXed variant of the openCryptoki [49] software token simulator, which is configurable by selectively enabling different patches. This offers a proof-of-concept that secure, fully fledged tokens can be realized in practice. At the same time, this has allowed us to test our reverse-engineering framework on (simulated) devices implementing various combinations of security patches. Among its patches, CryptokiX includes the first secure configuration to appear in the literature that does not require any new cryptographic mechanisms to be added to the standard.

Finally, we comment on the lessons for the next generation of standards for cryptographic key management such as IEEE 1619.3 and the OASIS Key Management Interoperability Protocol, currently in the draft stage.

Structure of the chapter The formal model of the API and how our tool extracts information from the token to allow us to build the model for a particular device are discussed in Section 7.2. We give our experimental results on various commercially available devices in Section 7.3. We describe how to use the tool to find secure configurations in Section 7.4. We conclude with a discussion of open problems and future key management APIs in section 7.5.

7.2 Model

To protect a key from being revealed, PKCS#11 says that the attribute sensitive must be set to true. This means that requests to view the object's key value via the API will result in an error message. Once the attribute sensitive has been set to true, it cannot be reset to false. This gives us the principal security property stated in the standard: attacks, even if they involve compromising the host machine to obtain the PIN, cannot "compromise keys marked 'sensitive', since a key that is sensitive will always remain sensitive", [51, p. 31]. Such a key may be exported outside the device if it is encrypted by another key, but only if its extractable attribute is set to true. An object with an extractable attribute set to false may not be read by the API, and additionally, once set to false, the extractable attribute cannot be set to true. Protection of the keys essentially relies on the sensitive and extractable attributes.

The vulnerabilities of this critical API has already been discussed in the previous chapter. Delaune, Kremer and Steel (DKS) proposed a Dolev-Yao style abstract model for PKCS#11 APIs, and showed how difficult it is to prevent these kinds of attacks: the commands can be restricted to prevent certain conflicting attributes from being set on the same object, but still more attacks arise [27]. The model presented here follows their approach.

The idea is to model the device as being connected to a host under the complete control of an intruder, representing a malicious piece of software. The intruder can call the commands of the API in any order he likes using any values that he knows. We abstract away details of the cryptographic algorithms in use, following the classical approach of Dolev and Yao [31]. Bitstrings are modelled as terms in an abstract algebra and the rules of the API and the abilities of an attacker are written as deduction rules in the algebra. The intruder is assumed not to be able to crack the encryption algorithm by brute-force search or similar means, thus he can only read an encrypted message if he has the correct key. We analyse security as reachability, in the model, of *attack* states, i.e. states where the intruder knows the value of a key stored on the device with the sensitive attribute set to true, or the extractable attribute set to false.

7.2.1 Basic Notions

We assume a given *signature* Σ, i.e. a finite set of *function symbols*, with an arity function $ar : \Sigma \to \mathbb{N}$, a (possibly infinite) set of *names* $\mathcal{N}$ and a (possibly infinite) set of *variables* $\mathcal{X}$. Names represent keys, data values, nonces, etc. and function symbols model cryptographic primitives, e.g. $\{\!|\, \mathsf{x} \,|\!\}_\mathsf{y}$ representing symmetric encryption of plaintext x under key y, and $\{\mathsf{x}\}_\mathsf{y}$ representing public key encryption of x under y. Function symbols of arity 0 are called *constants*. This includes the Boolean constants true ($\top$) and false ($\bot$). The set of *plain terms* $\mathcal{PT}$ is defined by the following grammar:

$$\begin{array}{rcll} t, t_i & := & x & x \in \mathcal{X} \\ & | & n & n \in \mathcal{N} \\ & | & f(t_1, \ldots, t_n) & f \in \Sigma \text{ and } ar(f) = n \end{array}$$

We also consider a finite set $\mathcal{F}$ of predicate symbols, disjoint from Σ, from which we derive a set of *facts*. The set of *facts* is defined as

$$\mathcal{FT} = \{p(t, b) \mid p \in \mathcal{F}, t \in \mathcal{PT}, b \in \{\top, \bot\}\}$$

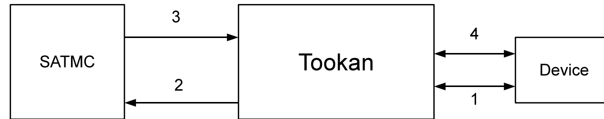
In this way, we can explicitly express the Boolean value b of an attribute p on a term t by writing $p(t, b)$. For example, to state that the key referred to by n has the wrap attribute set we write $\mathsf{wrap}(\mathsf{n}, \top)$. This is a difference in the syntax of our model compared to DKS, where attributes are expressed as literals ($\mathsf{wrap}(\mathsf{n})$ or $\neg\mathsf{wrap}(\mathsf{n})$).

The description of a system is given as a finite set of rules of the form

$$T; L \xrightarrow{\mathsf{new}\,\tilde{n}} T'; L'$$

where $T, T' \subseteq \mathcal{PT}$ are sets of plain terms $L, L' \subseteq \mathcal{F}$ are sets of facts and $\tilde{n} \subseteq \mathcal{N}$ is a set of names. The intuitive meaning of such a rule is the following. The rule can be fired if all terms in T are in the intruder knowledge and if all the facts in

Figure 7.1 Tookan system diagram



L hold in the current state. The effect of the rule is that terms in T' are added to the intruder knowledge and the valuation of the attributes is updated to satisfy L'. The new $\tilde{n}$ means that all the names in $\tilde{n}$ need to be replaced by fresh names in T' and L'. This allows us to model nonce or key generation: if the rule is executed several times, the effects are different as different names will be used each time.

Example The following rule models wrapping:

$$\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1), \mathsf{h}(\mathsf{x}_2, \mathsf{y}_2);\ \mathsf{wrap}(\mathsf{x}_1, \top), \mathsf{extract}(\mathsf{x}_2, \top) \rightarrow \{\!|\mathsf{y}_2|\!\}_{\mathsf{y}_1}$$

Intuitively, $\mathsf{h}(\mathsf{x}_1, \mathsf{y}_1)$ is a handle x_1 for key y_1 while term $\{\!|\mathsf{y}_2|\!\}_{\mathsf{y}_1}$ represents a key y_2 wrapped with y_1. Since the attribute wrap for key y_1 is set, noted as $\mathsf{unwrap}(\mathsf{x}_1, \top)$, and key y_2 is extractable, written $\mathsf{extract}(\mathsf{x}_2, \top)$, then we can wrap y_2 with y_1, creating $\{\!|\mathsf{y}_2|\!\}_{\mathsf{y}_1}$.

The semantics of the model is defined in a standard way in terms of a transition system. Each state in the model consists of a set of terms in the intruder's knowledge, and a set of global state predicates. The intruder's knowledge increases monotonically with each transition, but the global state is non-monotonic. For a formal semantics, we refer to the literature [27].

7.2.2 Modelling Real Tokens

The motivation for our work was to try to model the PKCS#11 implementations of real tokens. Our experiments on the tokens proceed following the system diagram in figure 7.1. First, Tookan extracts the capabilities of the token following a reverse engineering process (1). The results of this task are written in a meta-language for PKCS#11 models, described below. Tookan uses this information to generate a model in the above described style (2), which is given as input to the SATMC model checker [7]. Model checker output (3) is sent to Tookan for testing on the token (4).

In table 7.1 we give the syntax for the model meta-language. The language describes the functions and attributes supported by the token. It is also designed to capture the restrictions on functionality the token imposes. In table 7.2 we give our model for PKCS#11 showing how it is parametrised by the meta-model. We describe this relationship in more detail below. Note that the model we give here is slightly

Table 7.1 Syntax of Meta-language for describing PKCS#11 configurations

```
PKCS11_CONFIG       = Key_Types
                      Functions
                      Attributes
                      Attribute_Restrictions
                      Templates
                      Flags

Key_Types           = supports_symmetric_keys(BOOL);
                      supports_asymmetric_keys(BOOL);

Functions           = functions( FunctionList );

FunctionList        = nil |  Function, FunctionList

Function            = wrap | unwrap | encrypt | decrypt | create_object

Attributes          = attributes( AttributeList );

AttributeList       = nil | Attribute, AttributeList

Attribute           =  sensitive | extract | always_sensitive |
                       never_extract | wrap | unwrap | encrypt |  decrypt

Attribute_Restrictions = Sticky_On
                         Sticky_Off
                         Conflicts
                         Tied

Sticky_On           = sticky_on( AttributeList );

Sticky_Off          = sticky_off( AttributeList );

Conflicts           = conflict( AttributePairList );

Tied                = tied( AttributePairList );

AttributePairList   = nil | ( Attribute,Attribute ) , AttributePairList

Templates           = generate_templates(TemplateList);
                      create_templates(TemplateList);
                      unwrap_templates(TemplateList);

TemplateList        = nil | (Template) , TemplateList

Template            = nil | ( Attribute , BOOL ) , Template

Flags               = sensitive_prevents_read(BOOL);
                      unextractable_prevents_read(BOOL);

BOOL                = true | false
```

Table 7.2 PKCS#11 key management subset with side conditions from the meta-language of table 7.1

$$\mathsf{KeyGenerate}: \xrightarrow{\mathsf{new}\ n,k} h(n,k); \mathcal{A}(n,B) \quad (\text{with } B \in \mathcal{G})$$

$$\mathsf{KeyPairGenerate}: \xrightarrow{\mathsf{new}\ n,s} h(n,s), pub(s); \mathcal{A}(n,B) \quad (\text{with } B \in \mathcal{G})$$

$$\mathsf{Wrap\ (sym/sym)}: h(x_1,y_1), h(x_2,y_2);\ wrap(x_1,\top), extract(x_2,\top) \rightarrow \{\!|y_2|\!\}_{y_1}$$

$$\mathsf{Wrap\ (sym/asym)}: h(x_1,priv(z)), h(x_2,y_2);\ wrap(x_1,\top), extract(x_2,\top) \rightarrow \{y_2\}_{pub(z)}$$

$$\mathsf{Wrap\ (asym/sym)}: h(x_1,y_1), h(x_2,priv(z));\ wrap(x_1,\top), extract(x_2,\top) \rightarrow \{\!|priv(z)|\!\}_{y_1}$$

$$\mathsf{Unwrap\ (sym/sym)}: h(x,y_2), \{\!|y_1|\!\}_{y_2};\ unwrap(x,\top), \xrightarrow{\mathsf{new}\ n_1} h(n_1,y_1);\ \mathcal{A}(n_1,B) \quad (\text{with } B \in \mathcal{U})$$

$$\mathsf{Unwrap\ (sym/asym)}: h(x,priv(z)), \{y_1\}_{pub(z)};\ unwrap(x,\top), \xrightarrow{\mathsf{new}\ n_1} h(n_1,y_1);\ \mathcal{A}(n_1,B) \quad (\text{with } B \in \mathcal{U})$$

$$\mathsf{Unwrap\ (asym/sym)}: h(x,y_2), \{\!|priv(z)|\!\}_{y_2};\ unwrap(x,\top), \xrightarrow{\mathsf{new}\ n_1} h(n_1,priv(z));\ \mathcal{A}(n_1,B) \quad (\text{with } B \in \mathcal{U})$$

$$\mathsf{SEncrypt}: h(x_1,y_1), y_2;\ encrypt(x_1,\top) \rightarrow \{\!|y_2|\!\}_{y_1}$$

$$\mathsf{SDecrypt}: h(x_1,y_1), \{\!|y_2|\!\}_{y_1};\ decrypt(x_1,\top) \rightarrow y_2$$

$$\mathsf{AEncrypt}: h(x_1,priv(z)), y_1;\ encrypt(x_1,\top) \rightarrow \{y_1\}_{pub(z)}$$

$$\mathsf{ADecrypt}: h(x_1,priv(z)), \{y_2\}_{pub(z)};\ decrypt(x_1,\top) \rightarrow y_2$$

$$\mathsf{SetAttribute}: h(x_1,y_1);\ a(x_1,\bot), \mathcal{A}^{conf(a)}(x_1,\bot) \rightarrow\ ;\ a(x_1,\top), \mathcal{A}^{tied(a)}(x_1,\top) \quad (\text{with } a \in \mathcal{A} \setminus \mathtt{sticky_off_attributes})$$

$$\mathsf{UnsetAttribute}: h(x_1,y_1);\ a(x_1,\top) \rightarrow\ ;\ a(x_1,\bot), \mathcal{A}^{tied(a)}(x_1,\bot) \quad (\text{with } a \in \mathcal{A} \setminus \mathtt{sticky_on_attributes})$$

$$\mathsf{CreateObject}: x; \xrightarrow{\mathsf{new}\ n} h(n,x);\ \mathcal{A}(n,B) \quad (\text{with } B \in \mathcal{C})$$

$$\mathsf{GetAttribute}: h(n,x); extract(n,b_e), sensitive(n,b_s) \rightarrow x$$

$$\left(\begin{array}{l}\text{with } b_e, b_s \in \{\bot,\top\} \text{ and} \\ \mathtt{sensitive_prevents_read}(\top) \Rightarrow b_s = \bot \text{ and} \\ \mathtt{unextractable_prevents_read}(\top) \Rightarrow b_e = \top\end{array}\right)$$

Notation:
- $\mathcal{A} = \{a_1, \ldots, a_m\}$ denotes the (ordered) set of `attributes`
- $B = \{b_1, \ldots, b_m\}$ denotes a *template*, i.e. a set of Boolean values for attributes $\mathcal{A}$
- $\mathcal{A}(n,B)$ stands for $a_1(n,b_1), \ldots, a_m(n,b_m)$ while $\mathcal{A}(n,b)$ stands for $a_1(n,b), \ldots, a_m(n,b)$
- $\mathcal{B}(n,B)$, with $\mathcal{B} = \{a_{j_1}, \ldots, a_{j_k}\} \subseteq \mathcal{A}$ denotes $a_{j_1}(n,b_{j_1}), \ldots, a_{j_k}(n,b_{j_k})$, i.e., the projection of $\mathcal{A}(n,B)$ on $\mathcal{B}$
- $\mathcal{A}^{conf(a)}$ is the subset of attributes $a' \in \mathcal{A}$ conflicting with a, i.e., such that $\mathtt{conflict}(a',a)$
- $\mathcal{A}^{tied(a)}$ is the subset of attributes $a' \in \mathcal{A}$ tied to a, i.e., such that $\mathtt{tied}(a',a)$

simplified: in Tookan we construct separate sets of Attribute_Restrictions and Templates for asymmetric and symmetric keys, since many tokens impose quite different policies for these two different types. The full syntax and all the configurations derived during our experiments on real tokens can be viewed online[2].

Cryptographic Keys and Key Attributes

Tookan tests to see if a token supports the generation of asymmetric or symmetric keys, and returns the results, respectively, in the Booleans supports_symmetric_keys and supports_asymmetric_keys. By trying successive key generation commands, Tookan extracts the list of attributes in use for key objects and delivers these as the list attributes. These are used throughout the construction of the model and are noted as $\mathcal{A}$ in table 7.2. Note that as shown in the BNF in table 7.1, we restrict attention at the moment to a subset of PKCS#11 attributes. We do not consider signing and verification capabilities for example.

Functions

Tookan returns a list of functions supported, including one important function not modelled in the DKS work: CreateObject. This function allows the application to directly set the value of a new key on the device. Only the functions on the list are included in the final model.

Key Generation Templates

A major difference between our model and the DKS model is that we take into account *key templates*. In DKS, the key generation commands create a key with all its attributes unset [27, Fig. 2]. Attributes are then be enabled one by one using the SetAttribute command. In our experiments with real devices, we discovered that some tokens do not allow attributes of a key to be changed. Instead, they use a key template specifying settings for the attributes which are given to freshly generated keys. Templates are used for the import of encrypted keys (unwrapping), key creation using CreateObject and key generation. The template to be used in a specific command instance is specified as a parameter, and must come from a set of valid templates, which we label $\mathcal{G}, \mathcal{C}$ and $\mathcal{U}$ for the valid templates for key generation, creation and unwrapping respectively. Tookan can construct the set of templates in two ways: the first, by exhaustively testing the commands using templates for all possible combinations of attribute settings, which may be very time consuming, but is necessary if we aim to verify the security of a token. The second method is to construct the set of templates that should be allowed based on the reverse-engineered attribute policy (see next paragraph). This is an approximate process, but can be useful for quickly finding attacks. Indeed, in our experiments, we found that these

[2] http://secgroup.ext.dsi.unive.it/pkcs11-security

models reflected well the operation of the token, i.e. the attacks found by the model checker all executed on the tokens without any 'template invalid' errors.

Attribute Policies

Most tokens we tested attempt to impose some restrictions on the combinations of attributes that can be set on a key and how these may be changed. Some restrictions are listed as mandatory in the standard, though we found that not all tokens actually implement them. In our meta-model language, we describe four kinds of restriction that Tookan can infer from its reverse engineering process:

`Sticky_on` These are attributes that once set, may not be unset. The PKCS#11 standard lists some of these [51, Table 15]: sensitive for secret keys, for example. As shown in table 7.2, the UnsetAttribute rule is only included for attributes which are not sticky on. To test if a device treats an attribute as sticky on, Tookan attempts to create a key with the attribute on, and then calls SetAttribute to change the attribute to off.

`Sticky_off` These are attributes that once unset may not be set. In the standard, extractable is listed as such an attribute. As shown in table 7.2, the SetAttribute rule is only included for attributes which are not sticky off. To test if a device treats an attribute as sticky on, Tookan attempts to create a key with the attribute off, and then calls SetAttribute to change the attribute to on.

`Conflicts` Many tokens (appear to) disallow certain pairs of attributes to be set, either in the template or when changing attributes on a live key. For example, some tokens do not allow sensitive and extractable to be set on the same key. As shown in table 7.2, the SetAttribute rule is adjusted to prevent conflicting attributes from being set on an object or on the template. When calculating the template sets $\mathcal{C}, \mathcal{G}, \mathcal{U}$ (see above), we forbid templates which have both the conflicting attributes set. To test if a device treats an attribute pair as a conflict, Tookan attempts to generate a key with the pair of attributes set, then if no error is reported, it calls GetAttribute to check that the token really has created a key with the desired attributes set.

`Tied` Some tokens automatically set the value of some attributes based on the value of others. For example, many tokens set the value of always_sensitive based on the value of the attribute sensitive. As shown in table 7.2, the SetAttribute and UnsetAttribute rules are adjusted to account for tied attributes. The template sets $\mathcal{C}, \mathcal{G}, \mathcal{U}$ are also adjusted accordingly. To test if a device treats an attribute pair as tied, Tookan attempts to generate a key with some attribute a on and all other attributes off. It then uses GetAttribute to examine the key as it was actually created, and tests to see if any other attributes were turned on.

Respecting the Standard

Tookan checks two vital aspects of the token's behaviour: footnote 7 in table 15 of the standard specifies that certain attributes of an object may not be revealed via

a GetAttribute query if either the object's sensitive attribute is set to true, or the extractable attribute is set to false. We test these conditions independently by attempting to read the attribute giving the true value of a secret key. The results are respectively stored in `sensitive_prevents_read` and `unextractable_prevents_read`. Clearly if either of these are false for a real token, we have a vulnerability, since these are two of the critical security properties the token is supposed to provide. Nevertheless, we include them in our model since several of the tokens we tested fail to enforce these restrictions.

Optimising the Template Set

For tokens which allow a large number of different templates, the sets $\mathcal{C}, \mathcal{G}, \mathcal{U}$ can get very large, which creates a model that is very slow to search. We apply some simple optimisations to the template set that make a significant improvement to performance. Specifically, we construct a set of attributes $\mathcal{A}^+$ which only appear in the model set to true and do not appear in any conflicts. It is easy to see that if there are no rules that test this attribute is false, and it does not affect the value of any other attributes, then we need only construct templates where these attributes are set to true. Likewise, we construct a set of attributes $\mathcal{A}^-$ which only appear in the model set to false. We need not construct templates where this attribute is true.

Implementing Abstractions for Proving Security

A previous work [35] proved that, for models where attributes are static (i.e. they are all both sticky on and sticky off), it is possible to over-approximate the generation of fresh handles and keys to prove security for an unbounded number of handles and keys using a small finite model. Intuitively, the idea is to generate one key for each template, and to allocate one handle for each template. If a template is used twice, the same handle is generated, even if the key is different. Tookan has an option that builds a model following this abstraction. Since it is an over approximation, the abstract model may suggest false attacks. In this case, the user can switch back to the concrete, bounded model, where a user defined number of fresh handles and keys are used.

7.2.3 Limitations of Reverse Engineering

Our reverse engineering process is not complete: it may result in a model that is too restricted to find some attacks possible on the token, and it may suggest false attacks which cannot be executed on the token. This is because in theory, no matter what the results of our finite test sequence, the token may be running any software at all, perhaps even behaving randomly. However, if a token implements its attribute policy in the manner in which we can describe it, i.e. as a combination of sticky on, sticky off, conflict and tied attributes, then our process is complete in the sense

Table 7.3 Summary of results on devices

	Device		Supported Functionality						Attacks found					
	Company	Model	sym	asym	cobj	chan	w	ws	a1	a2	a3	a4	a5	mc
USB	Aladdin	eToken PRO	✓	✓	✓	✓	✓	✓	✓	✓				a1
	Athena	ASEKey	✓	✓	✓									
	Bull	Trustway RCI	✓	✓	✓	✓	✓	✓	✓	✓				a1
	Eutron	Crypto Id. ITSEC		✓	✓									
	Feitian	StorePass2000	✓	✓	✓	✓	✓	✓		✓	✓	✓		a3
	Feitian	ePass2000	✓	✓	✓	✓	✓	✓		✓	✓	✓		a3
	Feitian	ePass3003Auto	✓	✓	✓	✓	✓	✓		✓	✓	✓		a3
	Gemalto	Smart Enterprise Guardian		✓		✓								
	MXI Security	Stealth MXP Bio	✓	✓		✓								
	SafeNet	iKey 2032	✓	✓	✓		✓							
	Sata	DKey	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	a3
Card	ACS	ACOS5	✓	✓	✓	✓								
	Athena	ASE Smartcard	✓	✓	✓									
	Gemalto	Cyberflex V2	✓	✓	✓		✓	✓		✓				a2
	Gemalto	SafeSite Classic TPC IS V1		✓		✓								
	Gemalto	SafeSite Classic TPC IS V2	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	a3
	Siemens	CardOS V4.3 B	✓	✓	✓		✓					✓		a4
Soft	Eracom	HSM simulator	✓	✓		✓	✓	✓	✓	✓		✓		a1
	IBM	opencryptoki 2.3.1	✓	✓	✓	✓	✓	✓	✓	✓		✓		a1

	Acronym	Description
Supported functionality	sym	symmetric-key cryptography
	asym	asymmetric-key cryptography
	cobj	inserting new keys via `C_CreateObject`
	chan	changing key attributes
	w	wrapping keys
	ws	wrapping sensitive keys
Attacks	a1	wrap/decrypt attack based on symmetric keys
	a2	wrap/decrypt attack based on asymmetric keys
	a3	sensitive keys are directly readable
	a4	unextractable keys are directly readable (forbidden by the standard)
	a5	sensitive/unextractable keys can be changed into nonsensitive/extractable
	mc	first attack found by Tookan

that the model built will reflect exactly what the token can do (modulo the usual Dolev-Yao abstractions for cryptography).

In our testing, the model performed very well: the Tookan consistently found true attacks on flawed tokens, and we were unable to find 'by hand' any attacks on tokens which the model checker deemed secure. This suggests that real devices do indeed implement their attribute policies in a manner similar to our model.

7.3 Results

In this section, we report experimental results from using our tool to find attacks on commercially available devices. We acquired as many tokens as we could subject to our lab budgets, and the retail or loan availability of single tokens and cards. Tokens cost anything from 20 to 400 USD, with the global market estimated at 5 billion

USD[3]. We also tested our tool on two software simulators, intended for development purposes. Table 7.3 summarises the outcome of the analysis. For each token, we give a summary of the configuration information obtained from the token and a core subset of the attacks we found. Our testing on tokens is ongoing. Latest results can be viewed at the project website[4].

7.3.1 Implemented functionality

Columns 'sym' and 'asym' respectively indicate whether or not symmetric and asymmetric key cryptography are supported, i.e. the values of `supports_symmetric_keys` and `supports_asymmetric_keys` from the extracted configuration. We do not attempt to distinguish which particular cryptographic algorithms are supported in our analysis, since it is not relevant to the kinds of attacks we are looking for. Both kinds of cryptography are available on all the devices except three: the Eutron Crypto Identity ITSEC, Gemalto Smart Enterprise Guardian and the Gemalto SafeSite Classic TPC IS V1, which only provide asymmetric key cryptography. This last device should implement both symmetric and asymmetric cryptography according to its specification, but the one we tested could not generate and use symmetric keys. This may be a hardware issue with the specific token we possess.

Column 'cobj' refers to the possibility of inserting external, unencrypted, keys on the device via `C_CreateObject` PKCS#11 function, i.e. whether `create_object` is included in the list of functions in the extracted configuration. This is allowed by almost all of the analysed tokens. Although this command does not directly violate a security property, allowing known keys onto a device is generally a dangerous thing: an attacker might import an untrusted wrapping key from outside and ask the device to wrap a sensitive internal key with it [27].

The next column, 'chan', refers to the possibility of changing key attributes through `C_SetAttributeValue`. This functionality can easily be abused if not limited in some way. For example, it is clear (and stated in the standard) that it should never be possible to make a sensitive key nonsensitive. The behaviour of the `C_SetAttributeValue` command for a particular token is reported to the model checker via the `sticky_on` and `sticky_off` lists. A tick in this column indicates that at least one attribute was found that was not both `sticky_on` and `sticky_off`. The three Feitian devices correctly limit `C_SetAttributeValue` so that a sensitive key can never be changed into nonsensitive. However, this is of no use, since these tokens let any user directly access sensitive and unextractable keys (see attacks a3 and a4), disregarding the standard. The Sata and the Gemalto SafeSite Classic V2 devices are the only ones which allow the sensitive attribute to be unset with no limitation; this is in a perverse sense coherent, as just like the Feitian devices, they let any user access sensitive/unextractable keys. An interesting case is the Eracom

[3]InfoSecurity Magazine February 2010, `http://fanaticmedia.com/infosecurity/archive/Feb10/AuthenticationTokensstory.htm`

[4]`http://secgroup.ext.dsi.unive.it/pkcs11-security`

HSM simulator, which allows attribute change, but correctly implements the above mentioned policy, i.e., it disallows making a sensitive key nonsensitive, while also making sensitive keys unreadable: in this way, once a key is set as sensitive it will never become directly accessible. Subtler attacks on the keys are still possible by exploiting wrap/unwrap functions (see below attacks a1 and a2).

The following two columns, ‘w’ and ‘ws’, respectively indicate whether the token permits wrapping of nonsensitive and sensitive keys. It is discouraging to observe that every device providing ‘ws’, i.e., the wrapping of sensitive keys, is also vulnerable to attack. All the other devices avoid attacks at the price of removing such functionality. Forbidding the wrapping of sensitive keys is a quite limiting design choice since it compromises any proper management of sensitive keys among different devices. Wrapping sensitive keys is necessary in order to export/import those keys in a secure way. Most of these ‘limited’ tokens simply remove the whole wrapping functionality, i.e., both ‘w’ and ‘ws’. There are however two devices which allow the wrapping of nonsensitive keys only: SafeNet iKey and Siemens CardOS. Although this choice is less restrictive than removing the whole wrapping functionality, it seems difficult to think of an application where this would be a useful functionality. As we will discuss in the next section, it is indeed possible to produce a secure token configuration which allows wrapping (and unwrapping) of sensitive keys.

7.3.2 Attacks

Attack a1 is a wrap/decrypt attack: the attacker exploits a key k_2 with attributes wrap and decrypt and uses it to attack a sensitive key k_1. Using the notation from Section 7.2:

$$\begin{array}{ll} \mathsf{Wrap:} & \mathsf{h}(\mathsf{n}_2,\mathsf{k}_2),\ \mathsf{h}(\mathsf{n}_1,\mathsf{k}_1) \rightarrow \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_2} \\ \mathsf{SDecrypt:} & \mathsf{h}(\mathsf{n}_2,\mathsf{k}_2),\ \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_2} \rightarrow \mathsf{k}_1 \end{array}$$

As we have discussed above, the possibility of inserting new keys in the token (column ‘cobj’) might simplify further the attack. It is sufficient to add a known wrapping key:

$$\begin{array}{ll} \mathsf{CreateObject:} & \mathsf{k}_2 \xrightarrow{\mathsf{new}\ \mathsf{n}_2} \mathsf{h}(\mathsf{n}_2,\mathsf{k}_2) \\ \mathsf{Wrap:} & \mathsf{h}(\mathsf{n}_2,\mathsf{k}_2),\ \mathsf{h}(\mathsf{n}_1,\mathsf{k}_1) \rightarrow \{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_2} \end{array}$$

The attacker can then decrypt $\{\!|\mathsf{k}_1|\!\}_{\mathsf{k}_2}$ since he knows key k_2. SATMC discovered this variant of the attack on vulnerable tokens. We note that despite its apparent simplicity, this attack has not appeared before in the PKCS#11 security literature [23, 27].

Attack a2 is a variant of the previous ones in which the wrapping key is a public key $\mathsf{pub}(\mathsf{z})$ and the decryption key is the corresponding private key $\mathsf{priv}(\mathsf{z})$:

$$\begin{array}{ll} \mathsf{Wrap:} & \mathsf{h}(\mathsf{n}_2,\mathsf{pub}(\mathsf{z})),\ \mathsf{h}(\mathsf{n}_1,\mathsf{k}_1) \rightarrow \{\mathsf{k}_1\}_{\mathsf{pub}(\mathsf{z})} \\ \mathsf{ADecrypt:} & \mathsf{h}(\mathsf{n}_2,\mathsf{priv}(\mathsf{z})),\ \{\mathsf{k}_1\}_{\mathsf{k}_2} \rightarrow \mathsf{k}_1 \end{array}$$

In this case too, the possibility of importing key pairs simplifies even more the attacker’s task by allowing him to import a public wrapping key while knowing the

corresponding private key. Once the wrap of the sensitive key has been performed, the attacker can decrypt the obtained ciphertext using the private key.

Attack a3 is a clear flaw in the PKCS#11 implementation. It is explicitly required that the value of sensitive keys should never be communicated outside the token. In practice, when the token is asked for the value of a sensitive key, it should return some "value is sensitive" error code. Instead, we found that some of the analysed devices just return the plain key value, ignoring this basic policy. Attack a4 is similar to a3: PKCS#11 requires that keys declared to be unextractable should not be readable, even if they are nonsensitive. If they are in fact readable, this is another violation of PKCS#11 security policy.

Finally, attack a5 refers to the possibility of changing sensitive and unextractable keys respectively into nonsensitive and extractable ones. Only the Sata and Gemalto SafeSite Classic V2 tokens allow this operation. However, notice that this attack is not adding any new flaw for such devices, given that attacks a3 and a4 are already possible and sensitive or unextractable keys are already accessible.

7.3.3 Model-checking results

Column 'mc' reports which of the attacks has been automatically rediscovered via model-checking. SATMC terminates once it has found an attack, hence we report the attack that was found first. Run-times for finding the attacks vary from a couple of seconds to just over 3 minutes. We evaluate the performance of the model checker further in Section 7.5.

7.4 Finding Secure Configurations

As we noted in the last section, none of the tokens we tested are able to import and export sensitive keys in a secure fashion. In particular, all the analysed tokens are either insecure or have been drastically restricted in their functionality, e.g. by completely disabling wrap and unwrap. In this section, we present CryptokiX, a software (fiXed) implementation of a Cryptoki token, whose security is configurable by selectively enabling different patches. This offers a proof-of-concept that secure, fully fledged token can be realized in practice and, at the same time, it allows us to test the reverse-engineering framework on devices implementing various combinations of security patches. As well as providing Tookan with test data, this proof-of-concept of a secure token has also been adopted for educational purposes in a security lab class at the University of Venice, during which students are challenged to extract a sensitive key from a token which has only a subset of the patches turned on, so as to be insecure but not easy to attack [10].

Our starting point is openCryptoki [49], an open-source PKCS#11 implementation for Linux including a software token for testing. As shown in Table 7.3, the analysis of openCryptoki software token has revealed that it is subject to all the

non-trivial attacks. This is in a sense expected, as it implements the standard 'as is', i.e., with no security patches. We have thus extended openCryptoki with:

Conflicting attributes. We have seen, for example, that it is insecure to allow the same key to be used for wrapping and decrypting. In CryptokiX it is possible to specify a set of conflicting attributes.

Sticky attributes. We know that some attributes should always be sticky, such as sensitive. This is also useful when combined with the 'conflicting attributes' patch above: if wrap and decrypt are conflicting, we certainly want to avoid that the wrap attribute can be unset so as to allow the decrypt attribute to be set.

Wrapping formats. It has been shown that specifying a non-conflicting attribute policy is not sufficient for security [23, 27]. A wrapping format should also be used to correctly bind key attributes to the key. This prevents attacks where the key is unwrapped twice with conflicting attributes. Some existing devices already include such wrapping formats; an example is the Eracom ProtectServer [28].

Secure templates. The set of admissible attribute combinations for keys are limited in order to avoid that they ever assume conflicting roles at creation time. This patch has already been presented in the previous chapter.

A way to combine the first three patches with a wrapping format that binds attributes to keys in order to create a secure token has already been demonstrated [35]. In the following, we show how to check the security of the fourth patch, originally proposed by the author and others [15] using Tookan. Consider a set of templates with attributes sensitive and extractable always set. Other attributes wrap, unwrap, encrypt and decrypt are set as follows:

Key generation: we allow three possible templates:

1. wrap and unwrap, for exporting/importing other keys;
2. encrypt and decrypt, for cryptographic operations;
3. neither of the four attributes set, i.e. the default template if none of the above is specified.

Key creation/import: we allow two possible templates for any key created with `CreateObject` or imported with `Unwrap`:

1. unwrap,encrypt set and wrap,decrypt unset;
2. none of the four attributes set.

The templates for key generation are rather intuitive and correspond to a clear separation of key roles, which seems a sound basis for a secure configuration. The rationale behind the single template for key creation/import, however, is less obvious and might appear rather restrictive. The idea is to allow wrapping and unwrapping of keys while 'halving' the functionality of created/unwrapped keys: these latter keys

can only be used to unwrap other keys or to encrypt data, wrapping and decrypting under such keys are forbidden. This, in a sense, offers an asymmetric usage of imported keys: to achieve full-duplex encrypted communication two devices will each have to wrap and send a freshly generated key to the other device. Once the keys are unwrapped and imported in the other devices they can be used to encrypt outgoing data in the two directions. Notice that imported keys can never be used to wrap sensitive keys. Note also that we require that all attributes are sticky on and off, and that we assume for bootstrapping that any two devices that may at some point wish to communicate have a shared long term symmetric key installed on them at personalisation time. This need only be used once in each direction. Our solution works well for pairwise communication, where the overhead is just one extra key, but would be more cumbersome for group key sharing applications.

We analysed the developed solution by extracting the model using Tookan. A model for SATMC was constructed using the abstraction option (see section 7.2.2). Given the resulting model, SATMC terminates with no attacks in a couple of seconds, allowing us to conclude the patch is safe in our abstract model for unbounded numbers of fresh keys and handles. Note that although no sensitive keys can be extracted by an intruder, there is of course no integrity check on the wrapped keys that are imported. Indeed, without having an encryption mode with an integrity check this would seem to be impossible. This means that one cannot be sure that a key imported on to the device really corresponds to a key held securely on the intended receipient's device. This limitation would have to be taken into account when evaluating the suitability of this configuration for an application. CryptokiX is available online[5].

7.5 Conclusion

We conclude by evaluating the state of commercial security tokens, the performance of Tookan, and lessons for future key management APIs.

The state of the art in PKCS#11 security tokens seems rather poor. In our sample of 17 devices, we found 5 tokens that trivially gave up their sensitive keys in complete disregard of the standard, 3 that were vulnerable to a variety of key separation attacks, and a further smartcard that allowed unextractable keys to be read in breach of the standard. The remainder provide no functionality for secure transport of sensitive keys. We sent vulnerability reports to the manufacturers concerned at least 5 months before publication. Their responses can be viewed at the project website[6].

The tokens we have encountered so far have not provided much of a challenge for Tookan. At the start of the project, we hoped to encounter tokens that were patched in an effort to mitigate the attacks. Instead we found tokens with simple

[5] http://secgroup.ext.dsi.unive.it/cryptokix

[6] http://secgroup.ext.dsi.unive.it/pkcs11-security

flaws or minimal functionality. Attacks were found on all the vulnerable tokens, usually in just a few seconds. The potential value of the tool is perhaps best indicated by the work in section 7.4, where we implement patches on a software token simulator obtaining a fully featured software prototype of a secure (at least in our model) token, capable of wrapping and unwrapping keys. The software token can be reverse-engineered accurately by our automated framework, indicating that Tookan is ready to analyse more sophisticated devices as soon as they become available on the market. Our software token might be useful as a reference to develop such next-generation devices.

Notice, however, that if a manufacturer would like to validate a security patch to be implemented in its products, having the full accessibility to the source-code of the drivers and firmwares, it would be better to use a type-based approach like the one proposed in Chapter 6. Indeed having the possibility to validate if the code is secure appears more efficient and also time and cost preserving than building a token and use Tookan to reverse-engineer it. On the other side, a company (or a person) which would like to check if the PKCS#11 devices in its posses are not vulnerable (to the attacks considered here) will find Tookan a valuable and perfectly suited product.

In future work we will be extending our model to more cryptographic detail. We would also like to try Tookan on PKCS#11 based devices currently outside our budgets, such as Hardware Security Modules (HSMs).

Finally, there are at least two new standards which address key management currently at the draft stage: IEEE 1619.3[7] (for secure storage) and OASIS Key Management Interoperability Protocol (KMIP)[8]. Although neither is aimed at cryptographic tokens, it is clear there is a move towards better standards for key management in general. Given the apparent difficulty of constructing a secure interface based on PKCS#11, this seems a timely intervention. Our conclusions based on the research in this paper are that the new standards should:

- Specify clearly what security properties an interface complying to the standard should uphold. Our experimental evidence suggests that the security goals in PKCS#11, i.e. protection of sensitive or unextractable keys, are apparently too well hidden for some implementers to notice. A clear set of security properties would make life substantially easier for application developers as well.

- Include a format for key wrapping that securely preserves key metadata (i.e. attributes etc.). This has already been noted by recent proposals for secure interfaces [18, 25].

- Treat explicitly the problem of key roles, and give guidance to avoid conflicting roles. Again this issue has been treated by recent proposals for APIs in the academic literature [18, 25].

[7] https://siswg.net
[8] http://www.oasis-open.org/committees/kmip/

- Make provision for compliance testing, to weed out poorly implemented tokens.

Conclusions

Security APIs are Application Programming Interfaces that allow an untrusted system to access the functionalities offered by a trusted secure resource, assuring that a security policy is satisfied no matter what sequence of the API commands are invoked. Unfortunately, a number of attacks on existing security APIs have been found. This is because designing them is very hard. Formal methods have been applied in order to verify the security of these interfaces, and this thesis proposed to use a type-based analysis: an API designer should code a prototype of its API and if it type-checks, she would be assured that the proposed implementation is secure, i.e., respects the given security policy.

Two different kinds of attacks have been considered: information leakage and *pure* API attacks. The former slowly leak secret data to an attacker which will be able to spot dependency between the input parameters and the output given by an API command. The latter directly leaks a secret in its output result, provided that a given sequence of commands are executed. The attack model in this case is rather different: the malicious user is not looking at the outputs of the commands to learn something on the confidential data but will find a way in which commands can be composed so that to reveal the desired information in clear.

This work argues that information leakage attacks can be prevented enforcing noninterference-style properties. To this aim, the setting of language-based information flow security has been extended to account for randomized and deterministic cryptography. These are general results which are novel contributions to the language-based security foundational research filed. They have also opened up the possibility of a type-based analysis of the ATM PIN verification API which has highlighted the causes of its vulnerabilities so that a possible fix has been proposed.

Pure API attacks can be avoided asking that data confidentiality is always preserved at run-time, a more liberal policy than noninterference. The thesis have so presented a type system for a key management API which proves that secret keys are never leaked. It has been used to reason on the security of PKCS#11 and to prove the correctness of a novel security fix for it.

In the future, it would be interesting to use a type-based approach to investigate the security of more APIs. The typing introduced for key management APIs could be used to verify the security of two new standards currently in their draft stage: IEEE 1619.3 (`https://siswg.net`) (for secure storage) and OASIS Key Management Interoperability Protocol (KMIP) (`http://www.oasis-open.org/committees/kmip/`).

Web applications could also be considered. In fact, these are moving to offer a lot of services through HTTP-based APIs. This is true also for some of the biggest social networks, e.g., Facebook and Twitter, and for big sites like Google's ones: in these cases the privacy of the users data must not be broken by flawed APIs. Another interesting test case would also be the client-side part of the web experience: the browser. Any web site visited by a user could interact with the browser's underlying system through JavaScript, this is a source of possible attacks

to the user's computer. It would be possible to model the JavaScript interpreter as the entity exposing a security API to the browser and to verify if some desirable security policy holds. For example, it could be proved that information stored in cookies belonging to a domain are not accessible to a web site out of that domain.

Type-based analysis has proved to be a valid tool for the formal analysis of the APIs security properties, and also for helping API developers to understand the root-causes of known vulnerabilities affecting APIs and to aid them in developing secure code. We believe that, for trusted hardware manufacturers, having the possibility to type-check the code to put inside their devices is more efficient and also time and cost preserving than the possibility offered by other formal methods.

This thesis does not give a last answer to the problem of verifying the correctness of security APIs. It instead provide valuable examples to argue that a type-based analysis would be profitable in this field. In fact, type systems are rather common tools these days and can be considered well-understood by most (if not all) the programmers and so their application to the development of security APIs seems to be the natural way to go.

A

Cryptographic Noninterference - Formal Proofs

Theorem. 3.1 *If* $\vec{\mathsf{C}_1} \approxeq_C \vec{\mathsf{C}'_1}$, $\vec{\mathsf{C}_2} \approxeq_C \vec{\mathsf{C}'_2}$ *then*

1. $\mathbb{C}[\vec{\mathsf{C}_1}, \vec{\mathsf{C}_2}] \approxeq_C \mathbb{C}[\vec{\mathsf{C}'_1}, \vec{\mathsf{C}'_2}]$;

2. *Let* $\mathbb{D}[\vec{\bullet_1}, \vec{\bullet_2}] = \mathsf{if}\ b\ \mathsf{then}\ \vec{\bullet_1}\ \mathsf{else}\ \vec{\bullet_2}$, *with* b high. *Then,* $\vec{\mathsf{C}_1} \approxeq_C \vec{\mathsf{C}_2}$ *implies* $\mathbb{D}[\vec{\mathsf{C}_1}, \vec{\mathsf{C}_2}] \approxeq_C \mathbb{D}[\vec{\mathsf{C}'_1}, \vec{\mathsf{C}'_2}]$.

Proof. Proof follows by induction on the structure of $\mathbb{C}$.

For these first cases, the single-threaded ones, suppose $\mathbb{C}[\vec{\mathsf{C}_1}, \vec{\mathsf{C}_2}] = C$, $\mathbb{C}[\vec{\mathsf{C}'_1}, \vec{\mathsf{C}'_2}] = C'$ and let R be the relation $\{(C, C')\} \cup \approxeq_C$.

skip

Starting with $s_1 =_C s_2$ with both states the computation ends in one step with the same states.

$x := e_L$ ($\Gamma(x) = \mathsf{L}$)

e_L is a *low* expression so the computation of the command, in the two low-equivalent states $s_1 =_C s_2$, terminates in one step producing two new low-equivalent states. Indeed, let $v_1 = e_L \downarrow^{\mathsf{M}_1}$ and $v_2 = e_L \downarrow^{\mathsf{M}_2}$, then by definition of *low expression* (Definition 3.11) it holds that $(\mathsf{M}_1[x \mapsto v_1], \sigma_1) =_C (\mathsf{M}_2[x \mapsto v_2], \sigma_2)$.

$x := e$ ($\Gamma(x) = \mathsf{H}$)

Computing the command terminates in one step and does not have visible effects on the two low-equivalent starting states. In fact $\mathsf{sp}(s'_1) = \mathsf{sp}(s_1)$ and $\mathsf{sp}(s'_2) = \mathsf{sp}(s_2)$ since variables at level H are not considered in patterns construction. So $s'_1 =_C s'_2$.

$k := k'$, $k := \mathsf{newkey}$ ($\Gamma(k) = \Gamma(k') = \mathsf{K}$)

These cases assign to a key variable a value contained in KEY. As for the above case, computation does not have visible effect and the proof follows in the same way.

if b_L then $[\bullet_1]$ else $[\bullet_2]$

Computation starts with $\langle s_1, \mathsf{if}\ b\ \mathsf{then}\ \mathsf{C}_1\ \mathsf{else}\ \mathsf{C}_2\rangle$ and $\langle s_2, \mathsf{if}\ b\ \mathsf{then}\ \mathsf{C}'_1\ \mathsf{else}\ \mathsf{C}'_2\rangle$. Let $s_1 = (\mathsf{M}_1, \sigma_1)$ and $s_2 = (\mathsf{M}_2, \sigma_2)$. b_L will be evaluated to the same boolean value on both states, indeed it is a low expression and by Corollary 3.1 we know that $b_L \downarrow^{\mathsf{M}_1} = b = b_L \downarrow^{\mathsf{M}_2}$. Thus, in one step these configurations move to two bisimilar configurations, namely these could be either $\langle s_1, \mathsf{C}_1\rangle$, $\langle s_2, \mathsf{C}'_1\rangle$ ($\mathsf{C}_1 \approxeq_C \mathsf{C}'_1$) or $\langle s_1, \mathsf{C}_2\rangle$, $\langle s_2, \mathsf{C}'_2\rangle$ ($\mathsf{C}_2 \approxeq_C \mathsf{C}'_2$).

if b then $[\bullet_1]$ else $[\bullet_2]$ (b **is high)**

Start with $s_1 =_C s_2$. Here the value of b might differ for s_1 and s_2, but since $\mathsf{C}_i \approxeq_C \mathsf{C}_j$, $i, j : 1, 2$, one step of computation ends up in bisimilar commands and low-equivalent states (indeed no state change occurs in this step).

$\mathsf{fork}([\bullet_1][\bullet_2])$
Computation does not depend on state so this case is trivial.

$\mathsf{send}(cid, e_L)$
Let $s_1 = (\mathsf{M}_1, \sigma_1)$, $s_2 = (\mathsf{M}_2, \sigma_2)$, $v_1 = e_L \downarrow^{\mathsf{M}_1}$ and $v_2 = e_L \downarrow^{\mathsf{M}_2}$. With both states, the computation of send terminates in one step giving two new low-equivalent states, since e_L is a low expression and so it holds (by Definition 3.11) that $(\mathsf{M}_1, \sigma_1[cid \mapsto v_1.vals]) =_C (\mathsf{M}_2, \sigma_2[cid \mapsto v_2.vals])$.

$\mathsf{receive}(cid, x)$ $(\Gamma(x) \neq \mathsf{K})$
Let $(\mathsf{M}_1, \sigma_1) = s_1 =_C s_2 = (\mathsf{M}_2, \sigma_2)$. Two cases can be analyzed:

- $\Gamma(x) = \mathsf{H}$: let $\sigma_1 = vals_1.v_1$ and $\sigma_2 = vals_2.v_1$. Reception of a message on channel cid will get the following states: $s_1' = (\mathsf{M}_1, \sigma_1[cid \mapsto vals_1])$ and $s_2' = (\mathsf{M}_2, \sigma_2[cid \mapsto vals_2])$. By Lemma 3.1 item 3 we can state that $s_1' =_C s_2'$.
- $\Gamma(x) = \mathsf{L}$: in both s_1 and s_2 $\mathsf{receive}$ terminates in one computational step leading the following new states: $s_1'' = (\mathsf{M}_1[x \mapsto v_1], \sigma_1[cid \mapsto vals_1])$ and $s_2'' = (\mathsf{M}_2[x \mapsto v_2], \sigma_2[cid \mapsto vals_2])$. By Lemma 3.1 item 4 we can say $s_1'' =_C s_2''$.

Note that we cannot have the case $\Gamma(x) = \mathsf{K}$. This let us preserve key-safe property on states as stated by Proposition 3.1.

We now take care of multi-threaded cases:

$[\bullet_1]; [\bullet_2]$
In this case C_1 and C_1' can create new threads, so we need to construct R properly:

$$R = \{((\widetilde{\mathsf{C}_1}; \mathsf{C}_2)\vec{\mathsf{D}_1},\ (\widetilde{\mathsf{C}_1'}; \mathsf{C}_2')\vec{\mathsf{D}_1'}\ ) \mid \widetilde{\mathsf{C}_1} \approxeq_C \widetilde{\mathsf{C}_1'},\ \vec{\mathsf{D}_1} \approxeq_C \vec{\mathsf{D}_1'}\} \cup \approxeq_C$$

Starting from $s_1 =_C s_2$, if $\langle s_1, \mathsf{C}_1\rangle \to \langle s_1', \widetilde{\mathsf{C}_1}\vec{\mathsf{D}_1}\rangle$, then $\langle s_2, \mathsf{C}_1'\rangle \to \langle s_2', \widetilde{\mathsf{C}_1'}\vec{\mathsf{D}_1'}\rangle$ and $\widetilde{\mathsf{C}_1}\vec{\mathsf{D}_1} \approxeq_C \widetilde{\mathsf{C}_1'}\vec{\mathsf{D}_1'}$ with $s_1' =_C s_2'$ (since $\mathsf{C}_1 \approxeq_C \mathsf{C}_1'$). Thus, it must be the case that $\widetilde{\mathsf{C}_1} \approxeq_C \widetilde{\mathsf{C}_1'}$ and $\vec{\mathsf{D}_1} \approxeq_C \vec{\mathsf{D}_1'}$.

Let $\mathbb{C}[\mathsf{C}_1, \mathsf{C}_2] = \mathsf{C}_1; \mathsf{C}_2$ and $\mathbb{C}[\mathsf{C}_1', \mathsf{C}_2'] = \mathsf{C}_1'; \mathsf{C}_2'$. If C_1 spawns threads, $\langle s_1, \mathsf{C}_1; \mathsf{C}_2\rangle \to \langle s_1', (\widetilde{\mathsf{C}_1}; \mathsf{C}_2)\vec{\mathsf{D}_1}\rangle$, then C_1' do the same $\langle s_2, \mathsf{C}_1'; \mathsf{C}_2'\rangle \to \langle s_2', (\widetilde{\mathsf{C}_1'}; \mathsf{C}_2)\vec{\mathsf{D}_1'}\rangle$ and $\widetilde{\mathsf{C}_1} \approxeq_C \widetilde{\mathsf{C}_1'}$, $\vec{\mathsf{D}_1} \approxeq_C \vec{\mathsf{D}_1'}$, $s_1' =_C s_2'$. Commands so obtained are included in R and produce low-equivalent states. Note that, if $\langle s_1, \mathsf{C}_1\rangle \to \langle s_1', \varepsilon\rangle$, then $\langle s_2, \mathsf{C}_1'\rangle \to \langle s_2', \varepsilon\rangle$ and $s_1' =_C s_2'$ so it holds that if $\langle s_1, \mathsf{C}_1; \mathsf{C}_2\rangle \to \langle s_1', C_2\rangle$ then $\langle s_2, C_1'; C_2'\rangle \to \langle s_2', C_2'\rangle$ and $s_1' =_C s_2'$ ($\mathsf{C}_2 \approxeq_C \mathsf{C}_2'$ by hypothesis). These latter two configurations are included in the bisimulation, concluding the proof of the case.

while b_L do $[\bullet_1]$

In this case contexts are $\mathbb{C}[\mathsf{C}_1]$ = while b_L do C_1 and $\mathbb{C}[\mathsf{C}_1']$ = while b_L do C_1'. Here we choose R to be
$\{(\ (\widetilde{\mathsf{C}_1}; \text{while } b_L \text{ do } \mathsf{C}_1)\vec{\mathsf{D}_1}, (\widetilde{\mathsf{C}_1'}; \text{while } b_L \text{ do } \mathsf{C}_1')\vec{\mathsf{D}_1'}\ ) \mid$
$\widetilde{\mathsf{C}_1} \approxeq_C \widetilde{\mathsf{C}_1'},\ \vec{\mathsf{D}_1} \approxeq_C \vec{\mathsf{D}_1'}\} \cup \approxeq_C$.

Starting with $s_1 =_C s_2$, b_L will be evaluated to the same boolean value on both states by Corollary 3.1. Computation either terminates in one step or moves to the commands C_1; while b_L do C_1 and C_1'; while b_L do C_1' with low-equivalent (unchanged) states. In the first case proof is concluded, for the latter one we need to show that if the obtained commands move, then they reduce to programs still contained in R. Indeed, $\mathsf{C}_1 \approxeq_C \mathsf{C}_1'$ and this let us say that, if $\langle s_1, \mathsf{C}_1\rangle \to \langle s_1', \widetilde{\mathsf{C}_1}\vec{\mathsf{D}_1}\rangle$, then $\langle s_2, \mathsf{C}_1'\rangle \to \langle s_2', \widetilde{\mathsf{C}_1'}\vec{\mathsf{D}_1'}\rangle$ and $s_1' =_C s_2'$. Thus,
$\langle s_1, \mathsf{C}_1; \text{while } b_L \text{ do } \mathsf{C}_1\rangle \to \langle s_1', (\widetilde{\mathsf{C}_1}; \text{while } b_L \text{ do } \mathsf{C}_1)\vec{\mathsf{D}_1}\rangle$ implies
$\langle s_2, \mathsf{C}_1'; \text{while } b_L \text{ do } \mathsf{C}_1'\rangle \to \langle s_2', (\widetilde{\mathsf{C}_1}; \text{while } b_L \text{ do } \mathsf{C}_1\vec{\mathsf{D}_1'}\rangle$
These commands are in R and the states produced are low-equivalent.

$\langle\!|\ [\bullet_1][\bullet_2]\ |\!\rangle$

Let R be $\{(\vec{\mathsf{C}_1}\vec{\mathsf{C}_2}, \vec{\mathsf{C}_1'}\vec{\mathsf{C}_2'}) \mid \mathsf{C}_i \approxeq_C \mathsf{C}_i', i : 1, 2\} \cup \approxeq_C$. From $\vec{\mathsf{C}_1} \approxeq_C \vec{\mathsf{C}_1'}$ we know that $\vec{\mathsf{C}_1}, \vec{\mathsf{C}_1'}$ will be of the form $\langle\!|\ \mathsf{C}_1^1 \dots \mathsf{C}_1^n\ |\!\rangle$ and $\langle\!|\ \mathsf{C}_1'^1 \dots \mathsf{C}_1'^m\ |\!\rangle$. The same is true for $\vec{\mathsf{C}_2} \approxeq_C \vec{\mathsf{C}_2'}$: $\langle\!|\ \mathsf{C}_2^1 \dots \mathsf{C}_2^m\ |\!\rangle$, $\langle\!|\ \mathsf{C}_2'^1 \dots \mathsf{C}_2'^m\ |\!\rangle$. It holds that $\forall s_1 =_C s_2,\ i : 1, \dots, n\ j : 1, \dots, m$:

$$\text{if } \langle s_1, \mathsf{C}_1^i\rangle \to \langle s_1', \vec{\mathsf{C}}\rangle \text{ then } \langle s_2, \mathsf{C}_1'^i\rangle \to \langle s_2', \vec{\mathsf{C}'}\rangle$$
$$\text{if } \langle s_2, \mathsf{C}^j\rangle \to \langle s_1', \vec{\mathsf{C}}\rangle \text{ then } \langle s_2, \mathsf{C}_2'^j\rangle \to \langle s_2', \vec{\mathsf{C}'}\rangle$$

and $\vec{\mathsf{C}} \approxeq_C \vec{\mathsf{C}'}$, $s_1' =_C s_2'$. This proves the case.

□

Lemma. 3.3 ***(Expression equivalence)***
If $e : \mathsf{L}$ then e is low according to Definition 3.11.

Proof. We prove the following stronger fact:
Let $s_1 = (\mathsf{M}_1, \sigma_2)$, $s_2 = (\mathsf{M}_2, \sigma_2)$ with $s_1 =_C s_2$ and $v_1 = e\downarrow^{\mathsf{M}_1}$, $v_2 = e\downarrow^{\mathsf{M2}}$

$$e : \mathsf{L} \Rightarrow \exists\rho : \ \mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho,\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$$

By induction on the structure of e.

x

Since $s_1 =_C s_2$, $\exists\rho :\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$. Considering $\mathsf{p}_\ell(\mathsf{M}_1(x))$ and $\mathsf{p}_\ell(\mathsf{M}_2(x))$ we are taking a subset of the original patterns, indeed $x : \mathsf{L}$ so $\Gamma(x) = \mathsf{L}$. Thus, it holds that $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho$ using the same confounder substitution adopted to match the patterns on the whole states.

$\mathsf{pair}(e_1, e_2)$

Let $\mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$.

By (exp-l) $\mathsf{pair}(e_1, e_2) : \mathsf{L}$, $e_1 : \mathsf{L}$ and $e_2 : \mathsf{L}$. Let $v_i' = e_1 \downarrow^{\mathsf{M}_i}$, $v_i'' = e_2 \downarrow^{\mathsf{M}_i}$ with $i : 1, 2$. By induction on e_1 and e_2 we can say that

$$\exists \rho_1 : \quad \mathsf{p}_\ell(v_1') = \mathsf{p}_\ell(v_2')\rho_1, \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho_1$$
$$\exists \rho_2 : \quad \mathsf{p}_\ell(v_1'') = \mathsf{p}_\ell(v_2'')\rho_2, \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho_2$$

Thus, it holds that if $\rho(c) = c'$ then also $\rho_1(c) = c'$ and $\rho_2(c) = c'$ for all confounders substituted by ρ, i.e., ρ_1 and ρ_2 perform the same substitutions of ρ on $\mathsf{sp}(s_2)$. That also means that new substitutions added to both ρ_i acts on fresh (different) confouders. Let $\rho' = \rho[\rho_1, \rho_2]$ be the confounder substitution which acts as ρ plus substitutions performed by ρ_1 and ρ_2. It follows that: $\mathsf{p}_\ell(v_1) = (\mathsf{p}_\ell(v_1'), \mathsf{p}_\ell(v_1'')) = (\mathsf{p}_\ell(v_2'), \mathsf{p}_\ell(v_2''))\rho' = \mathsf{p}_\ell(v_2)\rho'$ and $\mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho'$.

$\mathsf{fst}(e_1)$

Let $\mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$. Standing that $\mathsf{fst}(e_1) : \mathsf{L}$, then $e_1 : \mathsf{L}$ by (exp-l). Let $v_i' = e_1 \downarrow^{mem_i}$ $i : 1, 2$. By induction on e_1:

$$\exists \rho_1 : \mathsf{p}_\ell(v_1') = \mathsf{p}_\ell(v_2')\rho_1, \ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho_1$$

Thus v_1' and v_2' has the same structure. If $v_1' = (v_1'', v_1''')$ then $v_2' = (v_2'', v_2''')$ and $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_1'') = \mathsf{p}_\ell(v_2'')\rho_1 = \mathsf{p}_\ell(v_2)\rho_1$. This prove the sub-case. Otherwise, if v_1' is not a couple, $v_1' \neq (v_1'', v_1''')$, then $\mathsf{fst}(e_1) \downarrow^{\mathsf{M}_i} \perp$ on both M_1 and M_2. $\mathsf{p}_\ell(\perp) = \perp$, thus it holds that $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho, \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$, where ρ is the same map we assumed at the beginning of the case.

snd

This case is all way similar to the fst one.

newkey

Since newkey : K by (newkey), this case does not satisfy Lemma hypothesis.

$\mathsf{enc}(e_1, e_2)$

Let $\mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$. We can analyze two distinct cases:

- $e_1 : \mathsf{H}$: this is the case where we are encrypting secret data. The type system requires that e_2 is a variable x and $\Gamma(x) = \mathsf{K}$ by rule (enc-sec). Thus, $\mathsf{p}_\ell(v_1) = \square_{c_1}$ and $\mathsf{p}_\ell(v_2) = \square_{c_2}$, indeed by Definition 3.7 $\mathsf{M}_i(x) \in KEY$, $i : 1, 2$ (item 1) and $\mathsf{sp}(s_1) \not\vdash \mathsf{M}_1(x)$, $\mathsf{sp}(s_2) \not\vdash \mathsf{M}_2(x)$ (item 2). Since c_1 and c_2 are fresh confounders, they are not mapped in ρ so we can extend it as follows: $\rho' = \rho[c_2 \mapsto c_1]$. It holds that $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho'$, $\mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho'$.

- $e_1 : \mathsf{L}$: by rule (exp-l) $e_1 : \mathsf{L}$, $e_2 : \mathsf{L}$. Let $v_i' = e_1 \downarrow^{\mathsf{M}_i},\ v_i'' = e_2 \downarrow^{\mathsf{M}_i}, i : 1, 2$. By induction on e_1 and e_2 we can say:

$$\begin{array}{ll} \exists \rho_1 : & \mathsf{p}_\ell(v_1') = \mathsf{p}_\ell(v_2')\rho_1,\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho_1 \\ \exists \rho_2 : & \mathsf{p}_\ell(v_1'') = \mathsf{p}_\ell(v_2'')\rho_2,\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho_2 \end{array}$$

  If v_i'' is not an atomic value n, then enc evaluates to $\perp$ on both states. Thus, $\mathsf{p}_\ell(\perp) =\perp$ and it holds that $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho, \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$. If v_i'' is an atomic value, then we have $\mathsf{p}_\ell(v_1'') = n = \mathsf{p}_\ell(v_2'')$. We are creating two new cyphertexts so, let c_1 and c_2 be two new fresh confounders not mapped in ρ_1. Let us extend such a substitution as follows: $\rho' = \rho_1[c_2 \mapsto c_1]$. It holds that $\mathsf{p}_\ell(v_1) = \{\mathsf{p}_\ell(v_1'), c_1\}_n = \{\mathsf{p}_\ell(v_2'), c_2\}_n\rho' = \mathsf{p}_\ell(v_2)\rho',\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho'$.

$\mathsf{dec}(e_1, e_2)$

By rule (exp-l) $e_1 : \mathsf{L}$ and $e_2 : \mathsf{L}$. By hypothesis $s_1 =_C s_2$ and so $\mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$. Let $v_i' = e_1 \downarrow^{\mathsf{M}_i}$, $v_i'' = e_2 \downarrow^{\mathsf{M}_i}$, $i : 1, 2$. By induction on e_1 and e_2 it holds that.

$$\begin{array}{ll} \exists \rho_1 : & \mathsf{p}_\ell(v_1') = \mathsf{p}_\ell(v_2')\rho_1,\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho_1 \\ \exists \rho_2 : & \mathsf{p}_\ell(v_1'') = \mathsf{p}_\ell(v_2'')\rho_2,\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho_2 \end{array}$$

If v_1'' is not an atomic value n, then also v_2'' will not be an atomic values and so $v_1 =\perp= v_2$.
Since $\mathsf{p}_\ell(\perp) =\perp$, it follows $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho,\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$. Otherwise, if $v_1'' = n$, two cases are possible:

- $v_1' = \{v_1''', c_1\}_n$: in this case v_2' has the same structure, i.e., $v_2' = \{v_1''', c_1'\}_n$. We know, by induction on e_1, that $\mathsf{p}_\ell(v_1') = \{v_1''', c_1\}_n = \{v_2''', c_1'\}_n\rho_1 = \mathsf{p}_\ell(v_2')\rho_1$. Evaluation of e_1, in this case, yields the two values $v_1 = v_1'''$ and $v_2 = v_2'''$. As just noted ρ_1 is a confounder substitution which let us say $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho_1,\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho_1$.
- otherwise: in this case dec will be evaluated to $\perp$ in both states. So, $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho,\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$.

$e_1 = e_2$

Let $\mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$. Since evaluation of this operators leads to a boolean value or $\perp$ we always use ρ as a confounder substitution but we need to show that $\mathsf{p}_\ell(v_1) = b = \mathsf{p}_\ell(v_2)$. Since $e : \mathsf{L}$, by (exp-l), $e_1 : \mathsf{L}$ and $e_2 : \mathsf{L}$. Let $v_i' = e_1 \downarrow^{\mathsf{M}_i},\ v_i'' = e_2 \downarrow^{\mathsf{M}_i}, i : 1, 2$. By induction on e_1 and e_2 it holds that.

$$\begin{array}{ll} \exists \rho_1 : & \mathsf{p}_\ell(v_1') = \mathsf{p}_\ell(v_2')\rho_1,\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho_1 \\ \exists \rho_2 : & \mathsf{p}_\ell(v_1'') = \mathsf{p}_\ell(v_2'')\rho_2,\ \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho_2 \end{array}$$

Cases $v'_1 = v''_1$ and $v'_1 \neq v''_1$ are handled similarly. We present here just the first case, the latter is symmetric.

Let $v'_1 = v''_1$ we then, want to prove that this also imply $v'_2 = v''_2$. Proof follows by induction on the structure of the value v'_1.

$v'_1 = n$
: Since $\mathsf{p}_\ell(v'_1) = n = \mathsf{p}_\ell(v'_2)\rho_1$ and $\mathsf{p}_\ell(v''_1) = n = \mathsf{p}_\ell(v''_2)\rho_2$, it is easy to show that $\mathsf{p}_\ell(v'_2) = n = \mathsf{p}_\ell(v''_2)$ indeed no confounder substitution can be performed on such value.

$v'_1 = b$
: This case is very close to the above one.

$v'_1 = (v'''_1, v^{iv}_1)$
: The case follows by induction hypothesis. Let $v''_1 = (v^1_2, v^2_2)$ since $v'_1 = v''_1$ it holds that $v'''_1 = v^1_2$ and $v^{iv}_1 = v^2_2$. Suppose that $v'_2 = (v^{1*}_1, v^{2*}_1)$ and $v''_2 = (v^{1*}_2, v^{2*}_2)$. Thus, by induction $v^{1*}_1 = v^{1*}_2$ and $v^{2*}_1 = v^{2*}_2$ that prove the case.

$v'_1 = \{v'''_1, c_1\}_n$
: In order to have two equal cyphertexts both encryptions must be already contained in the state patterns: creating new encryption will lead to a new (fresh) value as discussed. Let $v'_2 = \{v'''_2, c_2\}_n$ and $v''_2 = \{v'''_2, c_3\}$. Note that we used the same value for the message contained in both encryptions since it holds by induction. Let us take the pattern built on a state composed only by v'_i and v''_i. It must hold that $\mathsf{sp}(v'_1, v''_1) = \mathsf{sp}(v'_2, v''_2)\rho$ since $\mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$ and these are subset of s_1 and s_2 respectively. Thus, let x and y be two variables used just to build our example patterns

$$s' = \mathsf{sp}(v'_1, v''_1) =$$

$$\{(x, \{\mathsf{p}_\ell(val'''_1), c_1\}_n), (y, \{\mathsf{p}_\ell(v'''_1), c_1\}_n)\}$$

and

$$s'' = \mathsf{sp}(v'_2, v''_2) =$$

$$\{(x, \{\mathsf{p}_\ell(val'''_2), c_2\}_n), (y, \{\mathsf{p}_\ell(v'''_2), c_3\}_n)\}$$

but $s' = s''\rho$ and so it must holds that $c_2 = c_3$, proving the case.

$v'_1 = \{v'''_1, c_1\}_k$
: As stated for the above case, the two compared cyphertexts must already be included in the state patterns $\mathsf{sp}(s_1)$ and $\mathsf{sp}(s_2)$. Suppose $v'_2 = \{v'''_2, c_2\}_{k'}$ and $v''_2 = \{val'''_2, c_3\}_{k''}$. Proof follows by taking into account state patterns generate from (v'_1, v''_1) and (v'_2, v''_2) as for the above case, letting us to prove that $c_2 = c_3$. By well-formed definition on state (Definition 3.6) this assure that the two values are the same.

e_1 op e_2 (op $\neq=$)
Let $\mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$. e types L, so e_1 : L and e_2 : L by (exp-l). If op is applied to cyphertexts, it will be evaluated to $\bot$. So, if $v_1 =\bot$, then $v_2 =\bot$ and $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(val_2)\rho, \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$. Otherwise, if all the sub-expressions evaluate to cleartexts then by induction on e_1, e_2 we can say:

$$v_1^1 = n_1 = v_2^1, v_1^2 = n_2 = v_2^2$$

where $v_j^i = e_i \downarrow^{\mathsf{M}_j}, i : 1, 2 j : 1, 2$. It holds $v_1 = n = v_2$ (or $v_1 = b = v_2$) and thus, $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho, \mathsf{sp}(s_1) = \mathsf{sp}(s_2)\rho$.

□

Theorem. 3.2
If $\vec{\mathsf{C}} \hookrightarrow \vec{\mathsf{C}'} : \vec{Sl}$ *then* $\vec{\mathsf{C}'} \approxeq_C \vec{Sl}$.

Proof. By induction on the structure of $\vec{\mathsf{C}'}$.

Cases skip : skip, $l := e : l := e$ (e : L), $h := e$: skip, $k := e$: skip and message sending and receiving send(cid, e) : send(cid, e) (e : L), receive(cid, x) : receive($cid, \hat{x}$), are easy to prove since the program $\vec{\mathsf{C}}$ is clearly low-bisimilar to its low slice.

Cases $\mathsf{C}_1; \mathsf{C}_2 : Sl_1; Sl_2$, while b do C_1 : while b do Sl (b : L), $\langle\!|\, \mathsf{C}_1 \dots \mathsf{C}_n \,|\!\rangle$: $\langle\!|\, Sl_1 \dots Sl_n \,|\!\rangle$, fork($\mathsf{C}_1 \vec{\mathsf{C}_2}$) : fork($Sl_1 \vec{Sl_2}$) and the branch based on a low-expression test if b then C_1 else C_2 : if b then Sl_1 else Sl_2 (b : L) provide secure contexts $[\bullet_1]; [\bullet_2]$, while b do $[\bullet_1]$ (b : L), $\langle [\bullet_1] \dots [\bullet_n] \rangle$, fork($[\bullet_1][\vec{\bullet_2}]$) and if b then $[\bullet_1]$ else $[\bullet_2]$. Thus, since by induction hypothesis sub-commands have a low-bisimilar slice, Theorem 3.1 let us prove these cases.

One last case is left: the branch with a secret guard. if b then $\mathsf{C}'_1; Sl_2$ else $Sl_1; \mathsf{C}'_2$: skip; $Sl_1; Sl_2$. To apply secure congruence theorem we need to show that the transformation made by the type system lead two low-bisimilar branches, i.e., $\mathsf{C}'_1; Sl_2 \approxeq_C Sl_1; \mathsf{C}'_2$. This follows providing the secure context $\mathbb{C} = [\bullet_1]; [\bullet_2]$. By induction hypothesys we know that $\mathsf{C}'_1 \approxeq_C Sl_1$ and $\mathsf{C}'_2 \approxeq_C Sl_2$ which let us conclude that $\mathbb{C}[\mathsf{C}'_1, Sl_2] \approxeq_C \mathbb{C}[Sl_1, \mathsf{C}'_2]$. This is enough to show that the following relation is a strong cryptographic low bisimulation concluding the proof:
$\{($if b then $\mathsf{C}'_1; Sl_2$ else $Sl_1; \mathsf{C}'_2$, skip; $Sl_1; Sl_2)\} \cup \approxeq_C$ □

B

Proving integrity by equality - Formal Proofs

This appendix proves that the type system presented in Chpater 4 enforces both secret-sensitive and integrity noninterference.

Expression evaluation Simple security is a standard lemma used to prove the soundness of security type systems dealing with information flow as discussed in Chapter 2 (see Lemma 2.1). It can be seen that rule (hash-b) breaks such a proposition, indeed it takes a big secret as an input but its result will be typed public. In this setting, instead, it can be shown that if an expression types at security level ℓ then its value can be assigned to an observable variable of two ℓ-equivalent memories without breaking their equivalence.

Given two digest substitutions ρ and ρ', ρ is included in ρ', noted $\rho \subseteq \rho'$ if all the mappings performed by ρ is also done in ρ'. A digest substitution ρ is said to be *minimal* with respect to M_1 and M_2 if $\mathsf{M}_{1|\ell} = \mathsf{M}_{2|\ell}\rho$ and $\forall(\mathsf{h}^{\mathsf{b}}(v_2), \mathsf{h}^{\mathsf{b}}(v_1)) \in \rho, \exists x.\ \mathsf{M}_1(x) = \mathsf{h}^{\mathsf{b}}(v_1) \ \wedge \mathsf{M}_2(x) = \mathsf{h}^{\mathsf{b}}(v_2)$.

Proposition B.1. *Suppose* $\mathsf{M}_1 =_\ell^h \mathsf{M}_2$ *and let* $\mathsf{M}_{1|\ell} = \mathsf{M}_{2|\ell}\rho$ *where* ρ *is a minimal digest substitution for* M_1 *and* M_2*. If* $\mathsf{M}_1(x) = v_1 \in Val_b$, $\mathsf{M}_2(x) = v_2 \in Val_b$ *and* $\mathsf{L}(\Delta(x)) = \mathsf{H}_{\mathsf{b}}\ell_I$*, then* $\mathsf{h}(v_2) \in dom(\rho)$ *implies* $(\mathsf{h}(v_2), \mathsf{h}(v_1)) \in \rho$.

Proof. The two memories are comparable, let $\mathsf{r}_\ell(\mathsf{M}_1) = \mathsf{r}_\ell(\mathsf{M}_2)\mu$, it must be that $(v_2, v_1) \in \mu$ since $\mathsf{M}_i(x) = v_i \in Val_b$, $i : 1, 2$ and $\mathsf{L}(\Delta(x)) = \mathsf{H}_{\mathsf{b}}\ell_I$. The digest substitution ρ is minimal for $\mathsf{M}_1, \mathsf{M}_2$ and $\mathsf{h}(v_2) \in dom(\rho)$ thus $(\mathsf{h}(v_2), \mathsf{h}(v^*)) \in \rho$ and there must be a variable y such that $\mathsf{L}(\Delta(y)) \sqsubseteq \ell$, $\mathsf{M}_1(y) = \mathsf{h}(v^*)$ and $\mathsf{M}_2(y) = \mathsf{h}(v_2)$. Suppose $v^* \neq v_1$, then $(v_2, v^*) \in \mu$ which is impossible since μ is a bijection and $(v_2, v_1) \in \mu$. It follows $v^* = v_1$ and so $(\mathsf{h}(v_2), \mathsf{h}(v_1)) \in \rho$. □

It is straightforward to prove that a typed expression respects value types as shown by the next statement.

Proposition B.2. *Let* M *be a well-formed memory, if* $\Delta \vdash e : \tau$ *and* $e \downarrow^{\mathsf{M}} v$ *then it holds:*

1. $\tau = \mathsf{PH}_{\mathsf{s}}\ell_I$ *implies* $\vdash v : vt$ *with* $vt \in \{\mathsf{S}, \mathsf{B}\}$
2. $\tau = \mathsf{PH}_{\mathsf{b}}\ell_I$ *implies* $\vdash v : \mathsf{B}$
3. $\tau = \mathsf{PL}\ell_I$ *implies* $\vdash v : \mathsf{S}$
4. $\tau = \mathsf{D}\ell$ *implies* $\vdash v : vt$ *with* $vt \in \{\mathsf{S}^\#, \mathsf{B}^\#\}$

Proof. By induction on the structure of e.

x

This case follows directly by considering that M is well-formed.

$\mathsf{hash}(x)$

It holds $\Delta \vdash \mathsf{hash}(x) : \tau' \leq \tau$ and one of (hash-b) or (hash-s) has been used. In both cases notice that $\tau' = \mathsf{D}\ell'$ and $\Delta \vdash x : \mathsf{P}\ell$ thus by induction on x it holds that $\vdash \mathsf{M}(x) : vt$ with $vt \in \{\mathsf{S}, \mathsf{B}\}$. By hash semantics then it follows $\mathsf{hash}(x) \downarrow^{\mathsf{M}} v \in Val_s^d \cup Val_b^d$ thus $\vdash v : vt \in \{\mathsf{S}^\#, \mathsf{B}^\#\}$.

e_1 op e_2 with $\mathsf{op} \neq =$

The type system says that $\Delta \vdash e_1$ op $e_2 : \mathsf{P}\ell'$ and $\mathsf{P}\ell' \leq \tau = \mathsf{P}\ell$. By (op) $\Delta \vdash e_i : \mathsf{P}\ell''$ and $\ell' = \ell''^{\sqcup} \sqsubseteq \ell$, thus ℓ' could either be $\mathsf{L}\ell_I$ or $\mathsf{H_s}\ell_I$. Two different cases have to be considered: $\ell'' = \mathsf{L}\ell_I$ and $\ell'' \not\sqsubseteq \mathsf{LL}$.

Let $\ell'' = \mathsf{L}\ell_I$, then $\ell' = \mathsf{L}\ell_I$. Suppose $e_i \downarrow^{\mathsf{M}} v_i$, by induction $\vdash v_1 : \mathsf{S}$ and $\vdash v_2 : \mathsf{S}$ from which the thesis follows.

Otherwise $\ell'' \not\sqsubseteq \mathsf{LL}$ and $\ell' = \mathsf{H_s}\ell_I$. By induction, if $\ell'' = \mathsf{H_b}\ell_I$ then $\vdash v_i : \mathsf{B}$, while $\ell'' = \mathsf{H_s}\ell_I$ implies $\vdash v_i : vt$ with $vt \in \{\mathsf{S}, \mathsf{B}\}$. The evaluation of the expression then will result either in a big or small values thus giving the thesis.

$e_1 = e_2$

It holds $\Delta \vdash e_1 = e_2 : \mathsf{P}\ell'$ by (eq) and $\mathsf{P}\ell' \leq \tau = \mathsf{P}\ell$ with $\ell' \sqsubseteq \ell$. From the typing rule follows that $\Delta \vdash e_i : \tau'$, $\mathsf{L}(\tau') = \ell''$ and $\ell' = \ell''^{\sqcup}$ thus ℓ' could either be $\mathsf{L}\ell_I$ or $\mathsf{H_s}\ell_I$. The equal operator gives back a boolean which is a small value then it holds $\vdash e_1 = e_2 : \mathsf{S}$ which prove the case for both possible values of ℓ'.

□

A simple consequence of the above proposition is that if a given expression types $\mathsf{P}\ell$ then its evaluation on a well-formed memory M gets a plain value, i.e., a value which is not a digest.

Corollary B.1. *If* $\Delta \vdash e : \mathsf{P}\ell$ *and* $e \downarrow^{\mathsf{M}} v$ *then* $\vdash v : vt$ *with* $vt \in \{\mathsf{S}, \mathsf{B}\}$

Proof. The fact follows directly from Proposition B.2 conditions 1,2 and 3. □

Given two ℓ-equivalent memories and an expression which types as a digest with a security level less or equal to ℓ, its evaluation on these memories have to result in two values belonging to the same domain.

Proposition B.3. *Let* M_1, M_2 *be two well-formed memory such that* $\mathsf{M}_1 =_\ell^h \mathsf{M}_2$ *and* $e \downarrow^{\mathsf{M}_i} v_i$. *If* $\Delta \vdash e : \tau$ *and* $\mathsf{T}(\tau) = \mathsf{D}$, $\mathsf{L}(\tau) \sqsubseteq \ell$ *it holds* $v_1 \in Val_b^d \Leftrightarrow v_2 \in Val_b^d$.

Proof. Note that the expression could have been typed $\Delta \vdash e : \tau' \leq \tau$ only by (var), (hash-b) or (hash-h). It follows $\mathsf{L}(\tau') \sqsubseteq \mathsf{L}(\tau)$ thus $\mathsf{L}(\tau') \sqsubseteq \ell$. The proof follows by cases on the typing rule used to type e.

If rule (var) has been applied then let e be x, it follows $\Delta \vdash x : \mathsf{D}\ell''$ and since $\ell'' \sqsubseteq \ell$ then $\mathsf{M}_1(x) = \mathsf{M}_2(x)\rho$ thus the two variables store values from the same domain.

If rule (hash-b) hash been used then e is $\mathsf{hash}(x)$ and $\Delta \vdash x : \mathsf{PH_bH}$ thus by memory well-formedness and Proposition B.2 $x \downarrow^{\mathsf{M}_i} v^i \in Val_b$ so by hash semantics $\mathsf{hash}(x) \downarrow^{\mathsf{M}_i} v_i \in Val_b^d$.

Finally if rule (hash-s) derive the type judgment then $\Delta \vdash x : \mathsf{P}\ell''$ and it must be that $\mathsf{M}_1(x) = \mathsf{M}_2(x)$ indeed $\ell'' \sqsubseteq \ell$ and the values stored in the variables are not subject to substitution by ρ (by Corollary B.1) thus it will be $\mathsf{M}_i(x) \in Val_b$ or $\mathsf{M}_i(x) \in Val_s$ which will get the wanted result. □

Take two ℓ-equivalent memories $\mathsf{M}_1 =_\ell^h \mathsf{M}_2$, they will also be comparable. Suppose $\mathsf{r}_\ell(\mathsf{M}_1) = \mathsf{r}_\ell(\mathsf{M}_2)\mu$. Let $\Delta \vdash e : \tau$ with $\mathsf{L}(\tau) \sqsubseteq \ell$ and $e \downarrow^{\mathsf{M}_i} v_i$. Since there is no expression which can produce new random values it holds that the same random substitution can be used to make $\mathsf{M}_1[x \mapsto v_1]$ and $\mathsf{M}_2[x \mapsto v_2]$ ℓ-comparable.

Proposition B.4. *Let M_1, M_2 be two well-formed memories such that $\mathsf{M}_1 =_\ell^h \mathsf{M}_2$. Suppose $\mathsf{r}_\ell(\mathsf{M}_1) = \mathsf{r}_\ell(\mathsf{M}_2)\mu$. If $\Delta \vdash e : \tau$ with $\mathsf{L}(\tau) = \ell' \sqsubseteq \ell$, $e \downarrow^{\mathsf{M}_i} v_i$ and $\mathsf{L}(\Delta(x)) = \ell'$ then $\mathsf{r}_\ell(\mathsf{M}_1[x \mapsto v_1]) = \mathsf{r}_\ell(\mathsf{M}_2[x \mapsto v_2])\mu$.*

Proof. By cases on the structure of the expression e.

y

$\Delta \vdash y : \tau$ so $\Delta(y) = \tau' \leq \tau$. It follows $\mathsf{L}(\Delta(y)) = \ell'' \sqsubseteq \ell$ and two cases are considered:

- $\mathsf{T}(\tau) = \mathsf{P}$: By Corollary B.1 $v_i \in Val_s \cup Val_b$ so from $\mathsf{M}_1 =_\ell^h \mathsf{M}_2$ it follows $\mathsf{M}_1(y) = \mathsf{M}_2(y) = v$, indeed the variable is observable at level ℓ and the values stored in it are not in the domain of a digest substitution.

  If $\ell'' = \mathsf{H_b}\ell_I$ then subtyping cannot be used (so $\ell' = \mathsf{H_b}\ell$) and by memory well-formedness $v \in Val_b$ and it also holds $(v, v) \in \mu$ proving the case.

  Otherwise, $\ell'' \neq \mathsf{H_b}\ell_I$ and $\mathsf{r}_\ell(v, \ell') = 0$ from which the thesis follows.
- $\mathsf{T}(\tau) = \mathsf{D}$: By Proposition B.2 $v_i \in Val_s^d \cup Val_b^d$. If $v_1 \in Val_b^d$ then $v_2 \in Val_b^d$ by Proposition B.3 and suppose $v_i = \mathsf{h}(v_i')$, it follows $\mathsf{r}_\ell(v_i, \ell') = v_i' \in Val_b$ and since $\ell'' \sqsubseteq \ell'$ then $\ell'' \sqsubseteq \ell$ so $(v_2', v_1') \in \mu$ proving the case. If instead $v_1 \in Val_s^d$ then also $v_2 \in Val_s^d$ (always by Proposition B.3) thus $\mathsf{r}_\ell(v_i, \ell') = 0$ proving the case.

$\mathsf{hash}(y)$

In this case $\mathsf{T}(\tau) = \mathsf{D}$ so by Proposition B.2 $v_i \in Val_s^d \cup Val_b^d$. The case follows like the one above (sub-case $\mathsf{T}(\tau) = \mathsf{D}$).

e_1 op e_2

Let $\Delta \vdash e : \tau' \leq \tau$ be derived either by (op) or (eq). If $\Delta \vdash e_i : \tau''$ with $\mathsf{L}(\tau'') = \ell''$ then $\mathsf{L}(\tau') = \ell''^{\sqcup}$, i.e., $\mathsf{L}(\tau') = \mathsf{L}\ell_I$ or $\mathsf{L}(\tau') = \mathsf{H_s}\ell_I$ and since $\mathsf{T}(\tau') = \mathsf{P}$ by subtyping definition τ could be either $\mathsf{PL}\ell_I'$ or $\mathsf{PH_s}\ell_I'$. It follows that $v_i \in Val_s \cup Val_b$ by Corollary B.1 thus $\mathsf{r}_\ell(v_i, \ell') = 0$ proving the case.

□

The main proposition on typed expressions follows. If an expression types at security level ℓ then its value can be assigned to an observable variable of two ℓ-equivalent memories without breaking their equivalence.

Lemma B.1. (**Expression ℓ-equivalence**)
Let M_1, M_2 be two well-formed memories such that $\mathsf{M}_1 =^h_\ell \mathsf{M}_2$, $\Delta \vdash e : \tau$ with $\mathsf{L}(\tau) = \ell' \sqsubseteq \ell$ and $e \downarrow^{\mathsf{M}_i} v_i$. If $\mathsf{L}(\Delta(x)) = \ell'$ then $\mathsf{M}_1[x \mapsto v_1] =^h_\ell \mathsf{M}_2[x \mapsto v_2]$.

Proof. Let $\mathsf{M}_{1|\ell} = \mathsf{M}_{2|\ell}\rho$ with ρ being a minimal digest substitution for M_1 and M_2. It must be proved that:

1. $\mathsf{M}_1[x \mapsto v_1] \bowtie \mathsf{M}_2[x \mapsto v_2]$

2. there exists a digest substitution ρ' such that $\mathsf{M}_1[x \mapsto v_1]_{|\ell} = \mathsf{M}_2[x \mapsto v_2]_{|\ell}\rho'$ and $\rho \subseteq \rho'$.

The first part is proved by Proposition B.4 while the second statement is proved by induction on the structure of the expression.

y

From $\Delta \vdash y : \tau$ it follows $\Delta(y) = \tau' \leq \tau$, i.e., $\mathsf{L}(\tau') \sqsubseteq \mathsf{L}(\tau)$, indeed the typing can be derived by (var) and (sub). The variable is directly observable at level ℓ so $\mathsf{M}_1(y) = \mathsf{M}_2(y)\rho$ which proves the case by choosing $\rho' = \rho$.

$\mathsf{hash}(y)$

$\Delta \vdash \mathsf{hash}(y) : \tau' \leq \tau$ could by derived either by (hash-b) or (hash-s). Suppose that $\Delta \vdash y : \tau''$, two distinct cases are considered:

1. $\tau'' = \mathsf{PH_bH}$: In this case rule (hash-b) will be used to derive the type judgement.

   $\Delta \vdash \mathsf{hash}(y) : \mathsf{DLH} \leq \tau = \mathsf{D}\ell'$. Note that $\Delta \vdash y : \mathsf{PH_bH}$ implies $\Delta(y) = \mathsf{PH_bH}$ indeed the subtyping rule cannot be applied in this case. By hash semantics $v_i = \mathsf{h}(v_i')$, indeed memories are well-formed so $\mathsf{M}_i(y) = v_i' \in \mathit{Val}_b$. If $v_2 \in \mathit{dom}(\rho)$ then by Proposition B.1 $(v_2, v_1) \in \rho$ thus giving the thesis by $\rho' = \rho$. If, instead $v_2 \notin \rho$, it follows $\nexists z.\ \mathsf{M}_2(z) = \mathsf{h}(v_2')$. Since $\mathsf{M}_1 \bowtie_\ell \mathsf{M}_2$, let $\mathsf{r}_\ell(\mathsf{M}_1) = \mathsf{r}_\ell(\mathsf{M}_2)\mu$, it will be $(v_2', v_1') \in \mu$ indeed $\mathsf{L}(\Delta(y)) = \mathsf{H_bH}$. Suppose that there exists a variable w such that $\mathsf{L}(\Delta(w)) \sqsubseteq \ell$, $\mathsf{M}_2(w) = \mathsf{h}(v^*)$ with $v^* \neq v_2'$ and $\mathsf{M}_1(w) = \mathsf{h}(v_1')$, then it should be that $(v^*, v_1') \in \mu$ but it cannot be the case since μ is a bijection thus there is no variable in M_1 that stores the digest $\mathsf{h}(v_1')$. The thesis now follows by adding $(\mathsf{h}(v_2'), \mathsf{h}(v_1'))$ to ρ, i.e., $\rho'(v) = \rho(v)\ \forall v \in \mathit{dom}(\rho)$ and $\rho'(\mathsf{h}(v_2')) = \mathsf{h}(v_1')$, indeed note that $\rho \subseteq \rho'$.

2. $\tau'' = \mathsf{P}\ell,\ \ell \neq \mathsf{H_bH}$: In this case rule (hash-s) will be used to type the expression.

   $\Delta \vdash \mathsf{hash}(y) : \mathsf{D}\ell'' \leq \tau = \mathsf{D}\ell'$, and obviously $\ell'' \sqsubseteq \ell$. The type system state that $\Delta \vdash y : \mathsf{P}\ell''$, i.e., variable y is directly observable at level ℓ and it stores a plain value by Corollary B.1 so let $\mathsf{M}_1(y) = \mathsf{M}_2(y) = v$. If $v \in Val_s$ then the value is not on the domain of a digest substitution and the thesis follows by $\rho' = \rho$. Otherwise, if $v \in Val_b$ by Proposition B.1 if $\mathsf{h}(v) \in dom(\rho)$ then $(\mathsf{h}(v), \mathsf{h}(v)) \in \rho$ and the case is proved choosing ρ' to be ρ; if, instead $v \notin dom(\rho)$ then there not exists a variable z such that $\mathsf{M}_2(z) = \mathsf{h}(v)$. As shown above from $\mathsf{M}_1 \bowtie_\ell \mathsf{M}_2$ it will be possible to derive the fact that there will also not be any observable variable w such that $\mathsf{M}_1(w) = \mathsf{h}(v)$. The thesis follows by $\rho'(v) = \rho(v)\ \forall v \in dom(\rho)$ and $\rho'(\mathsf{h}(v)) = \mathsf{h}(v)$ which also gives $\rho \subseteq \rho'$.

e_1 op e_2

Two cases have to be considered:

1. $\mathsf{op} \neq =$: It holds $\Delta \vdash e_1\ \mathsf{op}\ e_2 : \mathsf{P}\ell''$ with $\mathsf{P}\ell'' \leq \tau = \mathsf{P}\ell'$ and also $\Delta \vdash e_1 : \mathsf{P}\ell'''$ and $\Delta \vdash e_2 : \mathsf{P}\ell'''$ and $\ell'' = \ell'''^{\sqcup}$. So, $\ell'' \sqsubseteq \ell$ and $\ell''' \sqsubseteq \ell'$ from which follows $\ell''' \sqsubseteq \ell'$. Let $e_1 \downarrow^{\mathsf{M}_i} v_i^1$ and $e_2 \downarrow^{\mathsf{M}_i} v_i^2$. By induction on e_1 it holds $\mathsf{M}_1[x \mapsto v_1^1]_{|\ell} = \mathsf{M}_2[x \mapsto v_2^1]_{|\ell}\rho'$, the values are plain as stated by Corollary B.1 thus they are not in the domain of a digest substitution so it holds $v_1^1 = v_2^1$, the same can be proved by induction on e_2 obtaining $v_1^2 = v_2^2$. These two equalities directly gives $e_1\ \mathsf{op}\ e_2 \downarrow^{\mathsf{M}_i} v,\ i : 1, 2$ from which the thesis follows with $\rho' = \rho$.

2. op is $=$: $\Delta \vdash e_1\ \mathsf{op}\ e_2 : \mathsf{P}\ell''$ by (eq) with $\mathsf{P}\ell'' \leq \tau = \mathsf{P}\ell'$. It is $\Delta \vdash e_1 : \tau''$, $\Delta \vdash e_2 : \tau''$ and $\mathsf{L}(\tau'') = \ell'''$ with $\ell'' = \ell'''^{\sqcup}$. Let $e_1 \downarrow^{\mathsf{M}_i} v_i^1$ and $e_2 \downarrow^{\mathsf{M}_i} v_i^2$ with $i : 1, 2$.

   If $\mathsf{T}(\tau'') = \mathsf{P}$ then by Corollary B.1 e_1 and e_2 will both evaluate to plain values. By induction on e_1 it follows $\mathsf{M}_1[x_1 \mapsto v_1^1]_{|\ell} = \mathsf{M}_2[x_1 \mapsto v_2^1]_{|\ell}\rho'$ but more precisely it is $v_1^1 = v_2^1$ since the values are not in the domain of a digest substitution. The same can be said about e_2 yielding $v_1^2 = v_2^2$. Thus the thesis is proved.

   If $\mathsf{T}(\tau'') = \mathsf{D}$, (op) rule cannot be applied to type the expressions e_1 and e_2 so they must both be an invocation of the hash function or a variable. Note that $v_1^1 \in Val_b^d \Leftrightarrow v_2^1 \in Val_b^d$ by Proposition B.3 and the same holds for v_i^2: $v_1^2 \in Val_b^d \Leftrightarrow v_2^2 \in Val_b^d$.

   Suppose $v_1^1 \in Val_b^d$ and $v_1^2 \in Val_s^d$, then obviously $v_1^1 \neq v_1^2$ and it will also be that $v_2^1 \in Val_b^d$ and $v_2^2 \in Val_s^d$ thus $v_2^1 \neq v_1^2$ which prove the case, indeed $e_1 = e_2 \downarrow^{\mathsf{M}_i} \mathsf{false}$. The case in which $v_1^1 \in Val_s^d$ and $v_1^2 \in Val_b^d$ is all the way similar.

Let $v_1^1 \in Val_b^d$ and $v_1^2 \in Val_b^d$, more precisely suppose that $v_i^1 = \mathsf{h}(v_i^+)$ and $v_i^2 = \mathsf{h}(v_i^*)$ and note that $v_i^+ \in Val_b$ and $v_i^* \in Val_b$. Since both e_j can be either $\mathsf{hash}(x_j)$ or x_j different cases are analyzed:

- $e_1 = x_1$, $e_2 = x_2$: In this case $\mathsf{M}_1(x_1) = \mathsf{M}_2(x_1)\rho$ and $\mathsf{M}_1(x_2) = \mathsf{M}_2(x_2)\rho$ it follows $(\mathsf{h}(v_2^+), \mathsf{h}(v_1^+)) \in \rho$ and $(\mathsf{h}(v_2^*), \mathsf{h}(v_1^*)) \in \rho$. If $\mathsf{h}(v_1^+) = \mathsf{h}(v_1^*)$ then since ρ is a bijection it holds $\mathsf{h}(v_2^+) = \mathsf{h}(v_2^*)$. In the same way from $\mathsf{h}(v_1^+) \neq \mathsf{h}(v_1^*)$ and the fact that ρ is a bijection it follows $\mathsf{h}(v_2^+) \neq \mathsf{h}(v_2^*)$.
- $e_1 = x_1$, $e_2 = \mathsf{hash}(x_2)$: It must be that $\mathsf{M}_1(x_1) = \mathsf{M}_2(x_1)\rho$, i.e., $(\mathsf{h}(v_2^+), \mathsf{h}(v_1^+)) \in \rho$. If $(\mathsf{h}(v_2^*), \mathsf{h}(v_1^*)) \in \rho$, then the case follows as in the previous case by the fact that ρ is a bijection and the two substitution are contained in it. Otherwise, if $(\mathsf{h}(v_2^*), \mathsf{h}(v_1^*)) \notin \rho$, by induction on e_2 $\mathsf{M}_1[x \mapsto \mathsf{h}(v_1^*)]_{|\ell} = \mathsf{M}_2[x \mapsto \mathsf{h}(v_2^*)]_{|\ell}\rho'$ and $\rho \subseteq \rho'$, thus there exists a digest substitution ρ' such that $(\mathsf{h}(v_2^+), \mathsf{h}(v_1^+)) \in \rho'$ (since $\rho \subseteq \rho'$) and $(\mathsf{h}(v_2^*), \mathsf{h}(v_1^*)) \in \rho'$ which give the case as shown above.
- $e_1 = \mathsf{hash}(x_1)$, $e_2 = x_2$: This case is symmetric to the above one.
- $e_1 = \mathsf{hash}(x_1)$, $e_2 = \mathsf{hash}(x_2)$: Three different cases must be considered.
  - Both expressions have been typed by (hash-b): From $\mathsf{M}_1 =_\ell^h \mathsf{M}_2$ it follows $\mathsf{M}_1 \bowtie_\ell \mathsf{M}_2$. It must be that $\Delta(x_i) = \mathsf{PH_bH}$ so $(v_2^+, v_1^+) \in \mu$ and $(v_2^*, v_1^*) \in \mu$ which implies that $v_1^+ = v_1^*$ if and only if $v_2^+ = v_2^*$ proving the case.
  - Only one of the expressions type by (hash-b): The only case in which this could happen is when $\Delta(x_1) = \mathsf{PH_bH}$ and $\Delta(x_2) = \mathsf{P}\ell'$ with $\ell' \neq \mathsf{H_bH}$ and $\ell' \sqsubseteq \ell$. If $\ell' = \mathsf{L}\ell_I$ then it will hold that x_2 can store only a small value by Proposition B.2 and so $\mathsf{hash}(x_1) = \mathsf{hash}(x_2) \downarrow^{\mathsf{M}_i} \mathsf{false}$ proving the case; Otherwise $\ell' = \mathsf{H_bL}$ but then also x_1 is observable and $\mathsf{M}_1(x_1) = \mathsf{M}_1(x_2)$ if and only $\mathsf{M}_2(x_1) = \mathsf{M}_2(x_2)$ (indeed $\mathsf{M}_1(x_1) = \mathsf{M}_2(x_1)$ and $\mathsf{M}_1(x_2) = \mathsf{M}_2(x_2)$) proving the case.
  - Both expression are typed using (hash-s): The two variables (x_1 and x_2) are observable and store plain values by Proposition B.2 so the case follows by observing that $\mathsf{M}_1(x_1) = \mathsf{M}_2(x_1)$ and $\mathsf{M}_1(x_2) = \mathsf{M}_2(x_2)$.

If, instead, $v_1^1 \in Val_s^d$ and $v_1^2 \in Val_s^d$, then by induction on e_1 it holds $\mathsf{M}_1[x \mapsto v_1^1]_{|\ell} = \mathsf{M}_2[x \mapsto v_2^1]_{|\ell}\rho$ indeed the values are not subject to a digest substitution and thus $v_1^1 = v_2^1$; the same reasoning can be applied to e_2 to get $v_1^2 = v_2^2$. These facts easily prove the case. In all the above cases it will be that $\mathsf{M}_1[x \mapsto v_1]_{|\ell} = \mathsf{M}_2[x \mapsto v_2]_{|\ell}\rho$ indeed the equal operator gets back a boolean, i.e., a plain value thus no new substitution will be

needed to map the result.

□

Subject reduction A subtyping relation for command types is introduced here. This has not been done before since it is not directly used by the type system, anyway it is useful to make some of the proofs cleaner: $(w_1, t_1, f_1) \leq (w_2, t_2, f_2)$ if $w_2 \sqsubseteq w_1$, $t_1 \sqsubseteq t_2$ and $f_1 \sqsubseteq f_2$ [29].

Note that the type system allows an untyped assignment in rule (int-test). The following proposition characterize the only command type which could lead to such scenario.

Proposition B.5. *If* $\Delta \vdash \mathsf{c} : (w, t, f)$ *and* $\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{\alpha} \langle \mathsf{M}', x := y; \mathsf{c}' \rangle$ *with* $\Delta(x) = \mathsf{P}\ell_C\mathsf{H}$*,* $\Delta \vdash y : \mathsf{P}\ell_C\mathsf{L}$*,* $\ell_C \sqsubseteq_C \mathsf{H_b}$ *and* $\Delta \vdash \mathsf{c}' : (w', t', f')$ *it holds* $(w, t, f) = (\ell_C\mathsf{H}, t' \sqcup \ell_C\mathsf{L}, \uparrow)$*.*

Proof. Note that the assignment $x := y$ is not typed and since c type-checks it must be that rule (int-hash) has been used. The thesis then follows by applying the typing rule to the hypothesis. □

The following Lemma proves a kind of subject reduction, indeed the untyped assignment discussed above must be considered as a special case.

Lemma B.2. (**Subject reduction**)
If $\Delta \vdash \mathsf{c} : (w, t, f)$ *and* $\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}' \rangle$ *then either:*

1. $\Delta \vdash \mathsf{c}' : (w', t', f') \leq (w, t, f)$ *or*
2. $\mathsf{c}' = x := y; \mathsf{c}''$ *and* $\Delta(x) = \mathsf{P}\ell_C\mathsf{H}$*,* $\Delta \vdash y : \mathsf{P}\ell_C\mathsf{L}$ *and* $\Delta \vdash \mathsf{c}' : (w', t', f') \leq (w, t, f)$ *or*
3. $\mathsf{c}' = \varepsilon$ *or* $\mathsf{c}' = \mathsf{FAIL}$*.*

Proof. By induction on the structure of c.

skip
: $\Delta \vdash \mathsf{skip} : (\mathsf{H_sL}, \mathsf{LH}, \downarrow)$ and $\langle \mathsf{M}, \mathsf{skip} \rangle \xrightarrow{\cdot} \langle \mathsf{M}, \varepsilon \rangle$.

$x := e$
: This case is similar to the above one, indeed the command reduce to ε in one step.

if e then c_1 else c_2
: Three different cases are analyzed depending on the typing rule used to derive the judgement.

- (if): It holds $\Delta \vdash e : \tau$, $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$ and $\mathsf{L}(\tau) \sqsubseteq w_i$ from which follows $\Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 : (w_1 \sqcap w_2, t_1 \sqcup t_2, f_1 \sqcup f_2)$.
  If $e \downarrow^{\mathsf{M}} \mathsf{true}$ then $\langle \mathsf{M}, \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 \rangle \dot{\rightarrow} \langle \mathsf{M}, \mathsf{c}_1 \rangle$ and the case is proved by the typing of c_1, indeed $w_1 \sqcap w_2 \sqsubseteq w_1$, $t_1 \sqsubseteq t_1 \sqcup t_2$ and $f_1 \sqsubseteq f_1 \sqcup f_2$. The case in which $e \downarrow^{\mathsf{M}} \mathsf{false}$ is all the way similar.
- (int-test): In this case e is $x = y$ and c_2 is FAIL. The command is typed $(\ell_C \mathsf{H}, \ell_C \mathsf{L}, \uparrow)$. It holds $\Delta \vdash x : \tau$, $\Delta \vdash y : \tau'$ with $\mathsf{T}(\tau) = \mathsf{T}(\tau')$ and $\mathsf{L}(\tau) = \ell_C \mathsf{L}$, $\mathsf{L}(\tau') = \ell_C \mathsf{H}$ and $\ell_C \sqsubseteq_C \mathsf{H_b}$. If $x = y \downarrow^{\mathsf{M}} \mathsf{false}$ then $\langle \mathsf{M}, \mathsf{if}\ x = y\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{FAIL} \rangle \dot{\rightarrow} \langle \mathsf{M}, \mathsf{FAIL} \rangle$ concluding the case. Instead, if $x = y \downarrow^{\mathsf{M}} \mathsf{true}$ then $\langle \mathsf{M}, \mathsf{if}\ x = y\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{FAIL} \rangle \dot{\rightarrow} \langle \mathsf{M}, \mathsf{c}_1 \rangle$ from which the thesis follows by $\Delta \vdash \mathsf{c}_1 : (\ell_C \mathsf{H}, t, f)$ indeed $\ell_C \mathsf{H} \sqsubseteq \ell_C \mathsf{H}$, $t \sqsubseteq \ell_C \mathsf{L} \sqcup t$ and $\uparrow \sqsubseteq f \sqcup \uparrow$.
- (int-hash): In this case e is $\mathsf{hash}(x) = y$ and c_1 is $z := x; \mathsf{c}_1'$ and c_2 is FAIL.
  If $\mathsf{hash}(x) = y \downarrow^{\mathsf{M}} \mathsf{false}$ then $\langle \mathsf{M}, \mathsf{if}\ \mathsf{hash}(x) = y\ \mathsf{then}\ z := x; \mathsf{c}_1'\ \mathsf{else}\ \mathsf{FAIL} \rangle \dot{\rightarrow} \langle \mathsf{M}, \mathsf{FAIL} \rangle$ proving the case.
  Otherwise, $\langle \mathsf{M}, \mathsf{if}\ \mathsf{hash}(x) = y\ \mathsf{then}\ z := x; \mathsf{c}_1'\ \mathsf{else}\ \mathsf{FAIL} \rangle \dot{\rightarrow} \langle \mathsf{M}, z := x; \mathsf{c}_1' \rangle$ and by the type system $\Delta(z) = \mathsf{P}\ell_C \mathsf{H}$, $\Delta \vdash x : \mathsf{P}\ell_C \mathsf{L}$ and $\Delta \vdash \mathsf{c}_1' : (\ell_C \mathsf{H}, t, f) \leq (\ell_C \mathsf{H}, t \sqcup \ell_C \mathsf{L}, \uparrow)$ which prove the case.

$\mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'$

The command is typed $(w, t \sqcup \mathsf{L}(\tau) \uparrow,)$ by rule (while) provided that $\Delta \vdash e : \tau$, $\Delta \vdash \mathsf{c}' : (w, t, f)$ and if $t = \mathsf{H_s}\ell_I$ then $t \sqsubseteq w$. If $e \downarrow^{\mathsf{M}} \mathsf{false}$ then $\langle \mathsf{M}, \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}' \rangle \dot{\rightarrow} \langle \mathsf{M}, \varepsilon \rangle$ which proves the case. Otherwise, $\langle \mathsf{M}, \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}' \rangle \dot{\rightarrow} \langle \mathsf{M}, \mathsf{c}'; \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}' \rangle$. If $t = \mathsf{H_s}\ell_I$ and $f = \uparrow$ then by (seq-2) it follows $\Delta \vdash \mathsf{c}'; \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}' : (w, t \sqcup \mathsf{L}(\tau), \uparrow)$ thus $w \sqsubseteq w$, $t \sqsubseteq t \sqcup \mathsf{L}(\tau)$ and $f \sqsubseteq \uparrow$ which prove the case, otherwise if $t \sqsubseteq \mathsf{H_b L}$ or $f = \downarrow$ by (seq-1) it is $\Delta \vdash \mathsf{c}'; \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}' : (w, t \sqcup \mathsf{L}(\tau), \uparrow)$ proving the case.

$\mathsf{c}_1; \mathsf{c}_2$

Let $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$. If $\langle \mathsf{M}, \mathsf{c}_1 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}_1' \rangle$ and $\mathsf{c}_1' \neq \varepsilon$, $\mathsf{c}_1' \neq \mathsf{FAIL}$ then $\langle \mathsf{M}, \mathsf{c}_1; \mathsf{c}_2 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}_1'; \mathsf{c}_2 \rangle$. By induction on c_1 it either holds (1) $\Delta \vdash \mathsf{c}_1' : (w_1', t_1', f_1') \leq (w_1, t_1, f_1)$ or (2) $\mathsf{c}_1' = x := y; \mathsf{c}_1''$ and $\Delta(x) = \mathsf{P}\ell_C \mathsf{H}$, $\Delta \vdash y : \mathsf{P}\ell_C \mathsf{L}$and $\Delta \vdash \mathsf{c}_1'' : (w_1', t_1', f_1') \leq (w_1, t_1, f_1)$, these two cases are considered:

1. Suppose that the type judgement of $\mathsf{c}_1; \mathsf{c}_2$ has been derived by (seq-1), then $t_1 \sqsubseteq \mathsf{H_b L}$ or $f_1 = \downarrow$. Since $\Delta \vdash \mathsf{c}_1' : (w_1', t_1', f_1') \leq (w_1, t_1, f_1)$ then $w_1 \sqsubseteq w_1'$, $t_1' \sqsubseteq t_1$ and $f_1' \sqsubseteq f_1$. If $t_1 \sqsubseteq \mathsf{H_b L}$ then $t_1' \sqsubseteq \mathsf{H_b L}$ thus by applying (seq-1) again the thesis follows: Indeed $\mathsf{c}_1'; \mathsf{c}_2$ will type $(w_1' \sqcap w_2, t_1' \sqcup t_2, f_1' \sqcup f_2) \leq (w_1 \sqcap w_2, t_1 \sqcup t_2, f_1 \sqcup f_2)$; if $f_1 = \downarrow$ then $f_1' = \downarrow$ indeed it is the only way in which $f_1' \sqsubseteq f_1$ is satisfied (and indeed if a program terminates all its subcommands do so) thus again the proof follows by (seq-1): $\mathsf{c}_1'; \mathsf{c}_2$ will type as above.

If $\mathsf{c}_1; \mathsf{c}_2$ has been typed by (seq-2) then $t_1 = \mathsf{H_s}\ell_I$,$f_1 = \uparrow$ and $t_1 \sqsubseteq w_2$ so $t'_1 \sqsubseteq t_1$ and $f'_1 \sqsubseteq f_1$. If $t'_1 \sqsubseteq \mathsf{H_bL}$ or $f'_1 = \downarrow$ then by (seq-1) $\mathsf{c}'_1; \mathsf{c}_2$ types $(w'_1 \sqcap w_2, t'_1 \sqcup t_2, f'_1 \sqcup f_2) \leq (w_1 \sqcap w_2, t_1 \sqcup t_2, \uparrow)$. Otherwise, $t'_1 = \mathsf{H_s}\ell_I$ and $f'_1 = \uparrow$ and by (seq-2) $\mathsf{c}'_1; \mathsf{c}_2$ types
$(w'_1 \sqcap w_2, t'_1 \sqcup t_2, \uparrow) \leq (w_1 \sqcap w_2, t_1 \sqcup t_2, \uparrow)$.

2. It holds $w_1 \sqsubseteq w'_1$, $t'_1 \sqsubseteq t_1$ and $f'_1 \sqsubseteq f_1$. By Proposition B.5 $(w_1, t_1, f_1) = (\ell_C\mathsf{H_b}, t'_1 \sqcup \ell_C\mathsf{L}, \uparrow)$ and $\ell_C \sqsubseteq_C \mathsf{H_b}$ so if $t'_1 \sqcup \ell_C\mathsf{L} \sqsubseteq \mathsf{H_bL}$ then $t'_1 \sqsubseteq \mathsf{H_bL}$ and by rule (seq-1) $\mathsf{c}''_1; \mathsf{c}_2$ is typed $(w'_1 \sqcap w_2, t'_1 \sqcup t_2, \uparrow)$ proving the case.
   If $t'_1 \sqcup \ell_C\mathsf{L} = \mathsf{H_sL}$ then $t'_1 = \mathsf{H_s}\ell_I$ and $\mathsf{c}_1; \mathsf{c}_2$ must have been typed by (seq-2) from which follows $t'_1 \sqcup \ell_C\mathsf{L} \sqsubseteq w_2$. It straightforward to derive that $t'_1 \sqsubseteq w_2$ and so $\mathsf{c}''_1; \mathsf{c}_2$ types $(w'_1 \sqcap w_2, t'_1 \sqcup t_2, \uparrow)$ by (seq-2) proving the case.

If $\langle \mathsf{M}, \mathsf{c}_1 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \varepsilon \rangle$ then $\langle \mathsf{M}, \mathsf{c}_1; \mathsf{c}_2 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}_2 \rangle$ and both (seq-1) and (seq-2) typing rules state that $\Delta \vdash \mathsf{c}_2 : (w_2, t_2, f_2)$ from which the case follows: Indeed by (seq-1) it would be $\Delta \vdash \mathsf{c}_2 : (w_2, t_2, f_2) \leq (w_1 \sqcap w_2, t_1 \sqcup t_2, f_1 \sqcup f_2)$ and by (seq-2) $\Delta \vdash \mathsf{c}_2 : (w_2, t_2, f_2) \leq (w_1 \sqcap w_2, t_1 \sqcup t_2, \uparrow)$.

If $\langle \mathsf{M}, \mathsf{c}_1 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{FAIL} \rangle$ then $\langle \mathsf{M}, \mathsf{c}_1; \mathsf{c}_2 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{FAIL}; \mathsf{c}_2 \rangle$ but since FAIL has no semantics it is equivalent to state that $\langle \mathsf{M}, \mathsf{c}_1; \mathsf{c}_2 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{FAIL} \rangle$ proving the case.

□

It is useful to note that untyped program (which is not an end of the computation or a FAIL) originated from a typed one could only come from an integrity test by hash.

Proposition B.6. $\Delta \vdash \mathsf{c} : (w, t, f)$, $\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{\alpha} \langle \mathsf{M}, \mathsf{c}' \rangle$, $\mathsf{c}' \neq \mathsf{FAIL}$, $\mathsf{c}' \neq \varepsilon$ *and* c' *does not type if and only if* c *is* $\mathsf{if}\ \mathsf{hash}(x) = y\ \mathsf{then}\ z := x; \mathsf{c}''\ \mathsf{else}\ \mathsf{FAIL}; \mathsf{c}'''$.

Proof. Note that there is only rule in the type system which allow an untyped assignment: (int-hash). □

The following proposition proves that typed commands preserve memory well-formedness.

Proposition B.7. *If* $\Delta \vdash \mathsf{c} : (w, t, f)$, M *is well-formed and* $\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}' \rangle$ *then* M' *is well-formed and it either holds:*

1. $\Delta \vdash \mathsf{c}' : (w', t', f')$ *or*

2. $\mathsf{c}' = x := y; \mathsf{c}''$ *and* $\Delta(x) = \mathsf{P}\ell_C\mathsf{H}$, $\Delta \vdash y : \mathsf{P}\ell_C\mathsf{L}$*and* $\Delta \vdash \mathsf{c}'' : (w', t', f')$ *and it holds* $\langle \mathsf{M}', x := y \rangle \xrightarrow{x} \langle \mathsf{M}'', \varepsilon \rangle$ *with* M'' *a well-formed memory or*

3. $\mathsf{c}' = \varepsilon$ *or* $\mathsf{c}' = \mathsf{FAIL}$.

Proof. By induction on the structure of the command c.

skip

This case is trivial, indeed $\langle \mathsf{M}, \mathsf{skip} \rangle \dot{\rightarrow} \langle \mathsf{M}, \varepsilon \rangle$.

$x := e$

Rule (assign) states that $\Delta(x) = \tau$ and $\Delta \vdash e : \tau$. By Proposition B.2 $e \downarrow^{\mathsf{M}} v$ and if $\tau = \mathsf{PH_s}\ell_I$ then $\vdash v : vt \in \{\mathsf{S}, \mathsf{B}\}$, if $\tau = \mathsf{PH_b}\ell_I$ then $\vdash v : \mathsf{B}$, if $\tau = \mathsf{PL}\ell_I$ then $\vdash v : \mathsf{S}$ and if $\tau = \mathsf{D}\ell$ then $\vdash v : vt \in \{\mathsf{S}^\#, \mathsf{B}^\#\}$.
$\langle \mathsf{M}, x := e \rangle \xrightarrow{x} \langle \mathsf{M}[x \mapsto v], \varepsilon \rangle$ which prove the case.

if e then c_1 else c_2

By cases on the rule used to type the command.

Suppose rule (if) has been used. It holds $\Delta \vdash e : \tau$ and $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$. It then follows $\langle \mathsf{M}, \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 \rangle \dot{\rightarrow} \langle \mathsf{M}, \mathsf{c}_i \rangle$ and the result is proved.

If rule (int-test) has been applied then e is $x = y$ and c_2 is FAIL and $\Delta \vdash \mathsf{c}_1 : (\ell'_C\mathsf{H}, t, f)$. If $\langle \mathsf{M}, \mathsf{if}\ x = y\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{FAIL} \rangle \dot{\rightarrow} \langle \mathsf{M}, \mathsf{c}_1 \rangle$ the result follows, otherwise $\langle \mathsf{M}, \mathsf{if}\ x = y\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{FAIL} \rangle \dot{\rightarrow} \langle \mathsf{M}, \mathsf{FAIL} \rangle$ and the case is proved.

If rule (int-hash) is used then e is $\mathsf{hash}(x) = y$, c_1 is $z := x; \mathsf{c}'_1$ and c_2 is FAIL. It also holds $\Delta \vdash x : \mathsf{P}\ell_C\mathsf{L}$, $\Delta \vdash y : \mathsf{D}\ell_C\mathsf{H}$, $\Delta \vdash z : \mathsf{P}\ell_C\mathsf{H}$ and $\Delta \vdash \mathsf{c}'_1 : (\ell'_C\mathsf{H}, t, f)$. Suppose that $\langle \mathsf{M}, \mathsf{if}\ \mathsf{hash}(x) = y\ \mathsf{then}\ z := x; \mathsf{c}'_1\ \mathsf{else}\ \mathsf{FAIL} \rangle \dot{\rightarrow} \langle \mathsf{M}, z := x; \mathsf{c}'_1 \rangle$. Let $\mathsf{M}(x) = v$, it is now proved that $\mathsf{M}[z \mapsto v]$ is well-formed. Three cases can be distinguished: If $\Delta \vdash x : \mathsf{PLL}$ then $v \in Vals_s$ by Proposition B.2 and thus since $\Delta(z) = \mathsf{PLH}$ its assignment to v do not break memory well-formedness; If $\Delta \vdash x : \mathsf{PH_b L}$ then by Proposition B.2 $v \in Val_b$ thus the assignment to z which has type $\mathsf{PH_b H}$ do not break well-formedness; Finally if $\Delta \vdash x : \mathsf{PH_s L}$ $v \in Val_s \cup Val_b$ (again by Proposition B.2) but since $\Delta(z) = \mathsf{PH_s H}$ that is exactly what is expected by memory well-formedness. Hence $\mathsf{M}[z \mapsto v]$ is well-formed. It holds $\langle \mathsf{M}, z := x \rangle \xrightarrow{z} \langle \mathsf{M}[z \mapsto v], \varepsilon \rangle$ which proves the case.
Otherwise, $\langle \mathsf{M}, \mathsf{if}\ \mathsf{hash}(x) = y\ \mathsf{then}\ z := x; \mathsf{c}'_1\ \mathsf{else}\ \mathsf{FAIL} \rangle \dot{\rightarrow} \langle \mathsf{M}, \mathsf{FAIL} \rangle$ and the thesis follows.

while e do c'

Two cases are considered: $\langle \mathsf{M}, \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}' \rangle \dot{\rightarrow} \langle \mathsf{M}, \varepsilon \rangle$ which get the thesis.
Otherwise $\langle \mathsf{M}, \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}' \rangle \dot{\rightarrow} \langle \mathsf{M}, \mathsf{c}'; \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}' \rangle$ and by (while) $\Delta \vdash \mathsf{c}' : (w, t', f')$, the proof follows by cases on t:

- $t \leq \mathsf{H_b L}$: $t' \sqsubseteq \mathsf{H_b L}$ and rule (seq-1) types $\mathsf{c}'; \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'$.
- $t = \mathsf{H_s}\ell_I$: If $t' = \mathsf{H_s}\ell'_I$ then $t' \sqsubseteq w$ by (while) and if $f' =\uparrow \mathsf{c}'; \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'$ types by (seq-2) otherwise by (seq-1). If instead $t' \sqsubseteq \mathsf{H_b L}$, $\mathsf{c}'; \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'$ types by rule (seq-1).

$\mathsf{c}_1; \mathsf{c}_2$

$\Delta \vdash \mathsf{c}_1; \mathsf{c}_2 : (w, t, f)$ could derive either from (seq-1) or (seq-2), in both cases $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$. If $\langle \mathsf{M}, \mathsf{c}_1 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}_1' \rangle$ then by subject reduction (Lemma B.2) it holds either (1) $\Delta \vdash \mathsf{c}' : (w', t', f') \leq (w, t, f)$ or (2) $\mathsf{c}' = x := y; \mathsf{c}''$ and $\Delta(x) = \mathsf{P}\ell_C\mathsf{H}$, $\Delta \vdash y : \mathsf{P}\ell_C\mathsf{L}$ and $\Delta \vdash \mathsf{c}' : (w', t', f') \leq (w, t, f)$ or (3) $\mathsf{c}' = \varepsilon$ or $\mathsf{c}' = \mathsf{FAIL}$. The proof follows by these three cases:

1. By induction on c_1, $\langle \mathsf{M}, \mathsf{c}_1 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}_1' \rangle$ and M_1' is well-formed. It holds $\langle \mathsf{M}, \mathsf{c}_1; \mathsf{c}_2 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}_1'; \mathsf{c}_2 \rangle$ which types by subject reduction (Lemma B.2).
2. By induction on c_1 $\langle \mathsf{M}', x := y \rangle \xrightarrow{x} \langle \mathsf{M}'', \varepsilon \rangle$ and M'' is well-formed. It follows $\langle \mathsf{M}, \mathsf{c}_1; \mathsf{c}_2 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', x := y; \mathsf{c}_1'; \mathsf{c}_2' \rangle$ proving the case, indeed $\mathsf{c}_1'; \mathsf{c}_2$ types by subject reduction (Lemma B.2).
3. If $\mathsf{c}_1' = \varepsilon$ then $\langle \mathsf{M}, \mathsf{c}_1; \mathsf{c}_2 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}_2 \rangle$ and by induction on c_1 it follows M' is well-formed and by subject reduction c_2 types. If $\mathsf{c}_1' = \mathsf{FAIL}$ then $\langle \mathsf{M}, \mathsf{c}_1; \mathsf{c}_2 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{FAIL}; \mathsf{c}_2 \rangle$ but since FAIL has no semantics it could be stated that $\langle \mathsf{M}, \mathsf{c}_1; \mathsf{c}_2 \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{FAIL} \rangle$ proving the case since M' will be well-formed by induction on c_1.

□

Information flow properties Some standard lemmas [62] for information flow security type systems are rephrased in the following. They confirm the intuitions given above in Section 4.5 when defining the role of each command type component [29].

If a command types with a writing effect w then it assigns only to variables whose security level is at or above w.

Lemma B.3. (**Confinement**)
If $\Delta \vdash \mathsf{c} : (w, t, f)$ then for every variable x assigned to in c it holds $w \sqsubseteq \mathsf{L}(\Delta(x))$.

Proof. By induction on the structure of the command c.

skip

No assignment is done, thus the thesis trivially holds.

$x := e$

By rule (assign) $\Delta(x) = \tau$, $\Delta \vdash e : \tau$ and $\Delta \vdash x := e : (\mathsf{L}(\tau), \mathsf{LH}, \downarrow)$ which proves the case.

if e then c_1 else c_2

Three different could be applied, the proof of the case proceed by cases on the rule used:

- (if): It is $\Delta \vdash e : \tau$, $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$, $\mathsf{L}(\tau) \sqsubseteq w_i$ and $\Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 : (w_1 \sqcap w_2, t_1 \sqcup t_2 \sqcup \mathsf{L}(tau), f_1 \sqcup f_2)$. By induction on c_1 for every variable x assigned to in it $w_1 \sqsubseteq \mathsf{L}(\Delta(x))$ and by induction on c_2 for every variable x assigned to in it $w_2 \sqsubseteq \mathsf{L}(\Delta(x))$. The case is then proved, indeed $w_1 \sqcap w_2 \sqsubseteq w_1$ and also $w_1 \sqcap w_2 \sqsubseteq w_2$.
- (int-test): In this case e is $x = y$ and c_2 must be FAIL. The command types $(\ell_C\mathsf{H}, t \sqcup \ell_C\mathsf{L}, \uparrow)$. The else-branch do not perform any assignment thus the thesis vacuously holds for it, for the then-branch the case follows by induction on c_1 indeed $\Delta \vdash \mathsf{c}_1 : (\ell_C\mathsf{H}, t, f)$ thus for any variable x assigned to in c_1, $\ell_C\mathsf{H} \sqsubseteq \mathsf{L}(\Delta(x))$ which prove the case.
- (int-hash): It must be that e is $\mathsf{hash}(x) = y$, c_1 is $z := x; \mathsf{c}_1'$ and c_2 is FAIL. The command is typed as $(\ell_C\mathsf{H}, t \sqcup \ell_C\mathsf{L}, \uparrow)$. It also holds that $\Delta(z) = \mathsf{P}\ell_C\mathsf{H}$ and by induction on c_1, since $\Delta \vdash \mathsf{c}_1' : (\ell_C\mathsf{H}, t, f)$, for any variable x assigned to in c_1' it holds $\ell_C\mathsf{H} \sqsubseteq \mathsf{L}(\Delta(x))$ proving the case.

$\mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'$: By rule (while) $\Delta \vdash e : \tau$, $\Delta \vdash \mathsf{c}' : (w, t, f)$ and $\Delta \vdash \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}' : (w, t \sqcup \mathsf{L}(\tau), \uparrow)$ thus the case follows by induction on c'.

$\mathsf{c}_1; \mathsf{c}_2$: The command could be typed by (seq-1) or (seq-2), however notice that both rules states that $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$ and $\Delta \vdash \mathsf{c}_1; \mathsf{c}_2 : (w_1 \sqcap w_2, t_1 \sqcup t_2, f')$. By induction on c_1 for every variable x assigned to in c_1 it holds $w_1 \sqsubseteq \mathsf{L}(\Delta(x))$ and by induction on c_2 for every variable assigned by it follows $w_2 \sqsubseteq \mathsf{L}(\Delta(x))$. Since $w_1 \sqcap w_2 \sqsubseteq w_i, i : 1, 2$ the case is proved.

□

Observing the termination of c gives information on variables at most at level t, so all the guard of the program is based on variables whose security level is below t.

Lemma B.4. (Guard safety)
If $\Delta \vdash \mathsf{c} : (w, t, f)$ then for every while loop or conditional guard e in c it holds $\Delta \vdash e : \tau$ with $\mathsf{L}(\tau) \sqsubseteq t$ or for every variable x in the guard it holds $\Delta \vdash x : \tau$ with $\mathsf{L}(\tau) \sqsubseteq t$.

Proof. By induction on the structure of the command c.

Commands skip and $x := e$ do not contain any guard so the thesis vacuously holds for them, the interesting cases are analyzed below.

$\mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2$
Three different rules could be applied to type the command:

- (if): It is $\Delta \vdash e : \tau$, $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$, and $\Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 : (w_1 \sqcap w_2, t_1 \sqcup t_2 \sqcup \mathsf{L}(\tau), f_1 \sqcup f_2)$. By induction, for every while loop or conditional guard e in c_1 it holds $\Delta \vdash e : \tau'$ with $\mathsf{L}(\tau') \sqsubseteq t_1 \sqsubseteq t_1 \sqcup t_2 \sqcup \mathsf{L}(\tau)$

or for every variable x in e $\Delta \vdash x : \tau'$ with $\mathsf{L}(\tau') \sqsubseteq t_1 \sqsubseteq t_1 \sqcup t_2 \sqcup \mathsf{L}(\tau)$ and also for every while loop or conditional guard e in c_2 it is $\Delta \vdash e : \tau'$ with $\mathsf{L}(\tau') \sqsubseteq t_2 \sqsubseteq t_1 \sqcup t_2 \sqcup \mathsf{L}(\tau)$ or for every variable x in e $\Delta \vdash x : \tau'$ with $\mathsf{L}(\tau') \sqsubseteq t_2 \sqsubseteq t_1 \sqcup t_2 \sqcup \mathsf{L}(\tau)$. To conclude the case it just suffice to note that $\mathsf{L}(\tau) \sqsubseteq t_1 \sqcup t_2 \sqcup \mathsf{L}(\tau)$.

- (int-test): In this case e is $x = y$. The command types $(\ell_C\mathsf{H}, t \sqcup \ell_C\mathsf{L}, \uparrow)$. By induction on c_1 for every while loop or conditional guard e contained in c_1 it holds $\Delta \vdash e : \tau'$ with $\mathsf{L}(\tau') \sqsubseteq t \sqsubseteq t \sqcup \ell_C\mathsf{L}$ or for every variable x in e $\Delta \vdash x : \tau'$ with $\mathsf{L}(\tau') \sqsubseteq t \sqsubseteq t \sqcup \ell_C\mathsf{L}$, indeed $\Delta \vdash \mathsf{c}_1 : (\ell_C\mathsf{H}, t, f)$. The test expression could either types $\tau = \mathsf{PLL}$ by (eq) from which $\mathsf{L}(\tau) = \mathsf{LL} \sqsubseteq t \sqcup \mathsf{LL}$ or if $\ell_C = \mathsf{H_b}$ the expression do not type but it holds $\Delta \vdash y : \mathsf{PH_bH}$ and $\mathsf{H_bH} \sqsubseteq t \sqcup \mathsf{H_bL}$ and also $\Delta \vdash x : \mathsf{PH_bL}$ and $\mathsf{H_bL} \sqsubseteq t \sqcup \mathsf{H_bL}$ proving the case.
- (int-hash): It must be that e is $\mathsf{hash}(x) = y$, c_1 is $z := x; \mathsf{c}_1'$ and c_2 is FAIL. The command is typed as $(\ell_C\mathsf{H}, t \sqcup \ell_C\mathsf{L}, \uparrow)$ and $\Delta \vdash \mathsf{c}_1' : (\ell_C\mathsf{H}, t, f)$. By induction, for every while loop or conditional guard e in c_1' it holds $\Delta \vdash e : \tau'$ with $\mathsf{L}(\tau') \sqsubseteq t \sqsubseteq t \sqcup \ell_C\mathsf{L}$ or for every variable x in the guards it holds $\Delta \vdash x : \tau'$ with $\mathsf{L}(\tau') \sqsubseteq t \sqsubseteq t \sqcup \ell_C\mathsf{L}$. The test expression is not typed by (eq) anyway $\Delta \vdash x : \mathsf{PH_bL}$ and $\mathsf{H_bL} \sqsubseteq t \sqcup \mathsf{H_bL}$ and also $\Delta \vdash y : \mathsf{DH_bH}$ and $\mathsf{H_bH} \sqsubseteq t \sqcup \mathsf{H_bH}$ proving the case.

$\mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'$: By rule (while) $\Delta \vdash e : \tau$, $\Delta \vdash \mathsf{c}' : (w, t, f)$ and $\Delta \vdash \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}' : (w, t \sqcup \mathsf{L}(\tau), \uparrow)$. By induction on c' for every while loop or conditional guard e in c' it holds $\Delta \vdash e : \tau'$ $\mathsf{L}(\tau') \sqsubseteq t \sqsubseteq t \sqcup \mathsf{L}(\tau)$ or for every variable x in e $\Delta \vdash x : \tau'$ with $\mathsf{L}(\tau') \sqsubseteq t \sqsubseteq t \sqcup \mathsf{L}(\tau)$ and also $\mathsf{L}(\tau) \sqsubseteq t \sqcup \mathsf{L}(\tau)$ proving the case.

$\mathsf{c}_1; \mathsf{c}_2$: The command could be typed by (seq-1) or (seq-2), however notice that both rules states that $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$ and $\Delta \vdash \mathsf{c}_1; \mathsf{c}_2 : (w_1 \sqcap w_2, t_1 \sqcup t_2, f')$. By induction for every while loop or conditional guard e in c_1 it holds $\Delta \vdash e : \tau$ with $\mathsf{L}(\tau) \sqsubseteq t_1 \sqsubseteq t_1 \sqcup t_2$ or for every x in e it is $\Delta \vdash x : \tau$ with $\mathsf{L}(\tau) \sqsubseteq t_1 \sqsubseteq t_1 \sqcup t_2$ and for every while loop or conditional guard e in c_2 it is $\Delta \vdash e : \tau$ with $\mathsf{L}(\tau) \sqsubseteq t_2 \sqsubseteq t_1 \sqcup t_2$ or for every variable x in e it holds $\Delta \vdash x : \tau$ with $\mathsf{L}(\tau) \sqsubseteq t_2 \sqsubseteq t_1 \sqcup t_2$.

□

Finally, the termination flag $\downarrow$ is given to program which always terminate.

Lemma B.5. (**Termination**)
If $\Delta \vdash \mathsf{c} : (w, t, \downarrow)$ *then* c *terminates on all memories.*

Proof. It is proved that if $\Delta \vdash \mathsf{c} : (w, t, \downarrow)$ then $\langle \mathsf{M}, \mathsf{c} \rangle \to^* \langle \mathsf{M}', \varepsilon \rangle$. By induction on the structure of the command c.

skip
: $\Delta \vdash \mathsf{skip} : (\mathsf{H_sL}, \mathsf{LH}, \downarrow)$ and $\langle \mathsf{M}, \mathsf{skip}\rangle \xrightarrow{\cdot} \langle \mathsf{M}, \varepsilon\rangle$ proving the case.

$x := e$
: $\Delta \vdash x := e : (w, t, \downarrow,$,a)nd if $e \downarrow^{\mathsf{M}} v$ then $\langle \mathsf{M}, x := e\rangle \xrightarrow{x} \langle \mathsf{M}[x \mapsto v], \varepsilon\rangle$ proving the case.

if e then c_1 else c_2
: The command could be typed by three different rules, note that if rule (int-test) or rule (int-hash) have been used then the command is potentially nonterminating so the hypothesis does not hold. It must be that the command is typed by (if), $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$ and
$\Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2 : (w_1 \sqcap w_2, t_1 \sqcup t_2, f_1 \sqcup f_2)$. Since $f_1 \sqcup f_2 =\downarrow$ it must be that $f_i :\downarrow$, $i : 1, 2$ which give the thesis by induction on c_1 or c_2.

while e do c'
: By (while) $\Delta \vdash \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}' : (w, t, \uparrow)$ so the hypothesis does not hold.

$\mathsf{c}_1; \mathsf{c}_2$
: Note that to meet the hypothesis only rule (seq-1) could be used to type $\mathsf{c}_1; \mathsf{c}_2$, it follows $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$ and $f_1 \sqcup f_2 =\downarrow$ form which the thesis follows by induction on c_1 and c_2.

□

SSNI by typing It is now proved that the type system assures that a typed program obeys to secret-sensitive NI. The proofs below follow the scheme of the proofs by Demange and Sands [29].

A command c is said to be a $\mathsf{H_s}$ command if $\Delta \vdash \mathsf{c} : (\mathsf{H_s}\ell_I, t, f)$, alternatively it is a $\mathsf{H_b}$ command if $\Delta \vdash \mathsf{c} : (w, t, f)$ with $\mathsf{H_bH} \sqsubseteq w$. The following propositions show that the fact of being an $\mathsf{H_s}$ or $\mathsf{H_b}$ program is preserved by typing.

Proposition B.8. *If* c *is* $\mathsf{H_s}$ *and* $\langle \mathsf{M}, \mathsf{c}\rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}'\rangle$ *then* c' *is a* $\mathsf{H_s}$ *command.*

Proof. By subject reduction (Lemma B.2) noting that clause (2) will never be triggered since rule (int-hash) type the command as $(\ell_C\mathsf{H}, t', \uparrow)$ with $\ell_C \sqsubseteq_C \mathsf{H_b}$. □

Proposition B.9. *If* c *is* $\mathsf{H_b}$ *and* $\langle \mathsf{M}, \mathsf{c}\rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}'\rangle$ *then either*

1. $\Delta \vdash \mathsf{c}' : (w', t', f')$ *and* c' *is a* $\mathsf{H_b}$ *command, or*
2. $\mathsf{c}' = x := y; \mathsf{c}''$ *with* $\Delta(x) = \mathsf{PH_bH}$*,* $\Delta \vdash y : \mathsf{PH_bL}$ *and* c'' *is* $\mathsf{H_b}$*.*

Proof. By subject reduction (Lemma B.2), in case 2 notice that $\Delta \vdash \mathsf{c} : (\ell_C\mathsf{H}, t, f)$ by Proposition B.5 and $\ell_C = \mathsf{H_b}$ since the program is $\mathsf{H_b}$, by subject reduction $\Delta \vdash \mathsf{c}'' : (w', t', f') \leq (w, t, f)$ proving that c'' is $\mathsf{H_b}$. □

It is straightforward to show that $\mathsf{H_s}$ commands are $\mathsf{H_b L}$-TSB and $\mathsf{H_b}$ one are LL-TIB.

Proposition B.10.
$\{(\langle \mathsf{M}_1, \mathsf{c}_1\rangle, \langle \mathsf{M}_2, \mathsf{c}_2\rangle) |\ \mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ *and* $\mathsf{c}_1, \mathsf{c}_2$ *are* $\mathsf{H_s}\}$ *is an* $\mathsf{H_bL}$-*TSB.*

Proof. The relation is symmetric by definition. Commands are typed as $\Delta \vdash \mathsf{c}_i : (\mathsf{H_s}\ell^i_I, t_i, f_i)$.

If $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \xrightarrow{\alpha} \langle \mathsf{M}'_1, \mathsf{c}'_1\rangle$ then by Confinement (Lemma B.3) the command assigns only to variables whose level is greater or equal to $\mathsf{H_s}\ell^1_I$ thus it holds $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \dot{\rightarrow}_{\mathsf{H_bL}} \langle \mathsf{M}'_1, \mathsf{c}'_1\rangle$ and $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}'_1$. The transition can be matched by $\langle \mathsf{M}_2, \mathsf{c}_2\rangle \overset{0}{\dot{\rightarrow}}_{\mathsf{H_bL}} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$, indeed since $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ then $\mathsf{M}'_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$. Command c'_1 is still an $\mathsf{H_s}$ command by Proposition B.8. The two configurations are thus contained in the relation proving the claim. □

Proposition B.11.
Let $\mathcal{B}$ be a relation on command configurations defined as $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \mathcal{B} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$ *if and only if* $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}_2$ *and it either holds:*

- c_i *is* $\mathsf{H_b}$ *or*
- $\mathsf{c}_i = x := y; \mathsf{c}'$ *with* $\Delta(x) = \mathsf{PH_bH}$, $\Delta \vdash y : \mathsf{PH_bL}$ *and* c' *a* $\mathsf{H_b}$ *command.*

It holds that $\mathcal{B}$ is a LL-*TIB.*

Proof. The relation is symmetric by definition and the proof follows by induction on the definition of $\mathcal{B}$.

$\mathsf{c}_1, \mathsf{c}_2$ are $\mathsf{H_b}$
: If $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \xrightarrow{\alpha} \langle \mathsf{M}'_1, \mathsf{c}'_1\rangle$ then by Confinement (Lemma B.3) the command assigns only to variables whose level is greater or equal to $\mathsf{H_bH}$ thus it holds $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \dot{\rightarrow}_{\mathsf{LL}} \langle \mathsf{M}'_1, \mathsf{c}'_1\rangle$ and $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}'_1$. The transition can be matched by $\langle \mathsf{M}_2, \mathsf{c}_2\rangle \overset{0}{\dot{\rightarrow}}_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$, indeed since $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}_2$ then $\mathsf{M}'_1 =^h_{\mathsf{LL}} \mathsf{M}_2$. The two configurations are still in $\mathcal{B}$ indeed c_2 is $\mathsf{H_b}$ and c_1 satisfies one of the two conditions by Proposition B.9.

c_1 is $\mathsf{H_b}$ and c_2 satisfies the second condition
: If $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}'_1\rangle$ then by Confinement (Lemma B.3) the command assigns only to variables whose level is greater or equal to $\mathsf{H_bH}$ thus it holds $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \dot{\rightarrow}_{\mathsf{LL}} \langle \mathsf{M}'_1, \mathsf{c}'_1\rangle$ and $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}'_1$. The transition can be matched by $\langle \mathsf{M}_2, \mathsf{c}_2\rangle \overset{0}{\dot{\rightarrow}}_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$, indeed since $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}_2$ then $\mathsf{M}'_1 =^h_{\mathsf{LL}} \mathsf{M}_2$. The two configurations are still in $\mathcal{B}$ indeed c_2 still satisfies the second condition and c_1 satisfies one of the two requirements by Proposition B.9.

c_1 satisfies the second condition and c_2 is $\mathsf{H_b}$

$\langle \mathsf{M}_1, x := y; \mathsf{c}' \rangle \xrightarrow{x} \langle \mathsf{M}_1[x \mapsto \mathsf{M}(y)], \mathsf{c}' \rangle$. Since $\Delta(x) = \mathsf{PH_bH}$ then $\langle \mathsf{M}_1, x := y; \mathsf{c}' \rangle \dot{\rightarrow}_{\mathsf{LL}} \langle \mathsf{M}_1[x \mapsto \mathsf{M}(y)], \mathsf{c}' \rangle$ and $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}'_1$. The transition can be matched by $\langle \mathsf{M}_2, \mathsf{c}_2 \rangle \dot{\rightarrow}^0_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}_2 \rangle$, indeed since $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}_2$ then $\mathsf{M}'_1 =^h_{\mathsf{LL}} \mathsf{M}_2$. The two configurations are still in $\mathcal{B}$ indeed c' and c_2 are $\mathsf{H_b}$.

c_1 and c_2 satisfy the second condition

$\langle \mathsf{M}_1, x := y; \mathsf{c}' \rangle \xrightarrow{x} \langle \mathsf{M}_1[x \mapsto \mathsf{M}(y)], \mathsf{c}' \rangle$. Since $\Delta(x) = \mathsf{PH_bH}$ then $\langle \mathsf{M}_1, x := y; \mathsf{c}' \rangle \dot{\rightarrow}_{\mathsf{LL}} \langle \mathsf{M}_1[x \mapsto \mathsf{M}(y)], \mathsf{c}' \rangle$ and $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}'_1$. The transition can be matched by $\langle \mathsf{M}_2, \mathsf{c}_2 \rangle \dot{\rightarrow}^0_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}_2 \rangle$, indeed since $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}_2$ then $\mathsf{M}'_1 =^h_{\mathsf{LL}} \mathsf{M}_2$. The two configurations are still in $\mathcal{B}$ indeed c' is $\mathsf{H_b}$ and c_2 satisfies the second condition.

□

Memories equivalence at level LL and $\mathsf{H_bL}$ is preserved according to the termination effect of a command.

Proposition B.12.
If $\Delta \vdash \mathsf{c} : (w, \mathsf{L}\ell_I, f)$ *and* $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}_2$ *then* $\langle \mathsf{M}_1, \mathsf{c} \rangle \xrightarrow{\alpha} \langle \mathsf{M}'_1, \mathsf{c}' \rangle$ *implies* $\langle \mathsf{M}_2, \mathsf{c} \rangle \xrightarrow{\alpha} \langle \mathsf{M}'_2, \mathsf{c}' \rangle$ *with* $\mathsf{M}'_1 =^h_{\mathsf{LL}} \mathsf{M}'_2$.

Proof. By induction on the structure of the command c.

skip

This case is obvious.

$x := e$

It holds $\Delta \vdash e : \tau$ and $\mathsf{L}(\Delta(x)) = \tau$ and let $e \downarrow^{\mathsf{M}_i} v_i$. It follows $\langle \mathsf{M}_i, x := e \rangle \xrightarrow{x} \langle \mathsf{M}_i[x \mapsto v_i], \varepsilon \rangle$. If $\tau \sqsubseteq \mathsf{LL}$ then by Expression ℓ-equivalence (Lemma B.1) it follows $\mathsf{M}_1[x \mapsto v_1] =^h_{\mathsf{LL}} \mathsf{M}_2[x \mapsto v_2]$ which proves the case. If instead $\tau \not\sqsubseteq \mathsf{LL}$ then the variable to which the value is assigned is not observable thus it easily holds that $\mathsf{M}_1[x \mapsto v_1] =^h_{\mathsf{LL}} \mathsf{M}_2[x \mapsto v_2]$.

if e then c_1 else c_2

Three different cases are considered depending on the rule used to type the command.

- (if): It holds $\Delta \vdash e : \tau$, $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$ and $\mathsf{L}(\tau) \sqsubseteq w_i$. Notice that since the termination effect of the whole command is $\mathsf{L}\ell_I$ it must be that $\mathsf{L}(\tau) \sqsubseteq \mathsf{L}\ell_I$ and also $t_i \sqsubseteq \mathsf{L}\ell_I$. Since the expression types with a security level equal or below LL then the expression will be evaluated to the same boolean value thus the same branch will be followed in both the executions: $\langle \mathsf{M}_i, \text{if } e \text{ then } \mathsf{c}_1 \text{ else } \mathsf{c}_2 \rangle \dot{\rightarrow} \langle \mathsf{M}_i, \mathsf{c}_j \rangle$ which proves the case.

- (int-test): In this case e is $x = y$. Note that $\Delta \vdash x : \tau$ and $\Delta \vdash y : \tau'$ and it must be that $\mathsf{L}(\tau) = \mathsf{LL}$ and $\mathsf{L}(\tau') = \mathsf{LH}$ thus $\mathsf{M}_1(x) = \mathsf{M}_2(x)$ and $\mathsf{M}_1(y) = \mathsf{M}_2(y)$. The two executions will execute the same branch and get to two configurations with the same memory by a silent transition.
- (int-hash): The case follows similarly to the above one.

$\mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'$

It is $\Delta \vdash e : \tau$ and $\Delta \vdash \mathsf{c}' : (w, t, f)$. Since the termination effect is $\mathsf{L}\ell_I$ it must be that $\mathsf{L}(\tau) \sqsubseteq \mathsf{L}\ell_I$ and $t \sqsubseteq \mathsf{L}\ell_I$. The expression is thus evaluated to the same value in both memories: If it gets evaluated to false then $\langle \mathsf{M}_i, \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'\rangle \dot{\rightarrow} \langle \mathsf{M}_i, \varepsilon\rangle$, otherwise $\langle \mathsf{M}_i, \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'\rangle \xrightarrow{\alpha} \langle \mathsf{M}_i, \mathsf{c}'; \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'\rangle$.

$\mathsf{c}_1; \mathsf{c}_2$

Since the termination effect is $\mathsf{L}\ell_I$ only rule (seq-1) can be used and also $\Delta \vdash \mathsf{c}_1 : (w_1, t_1, f_1)$ and $\Delta \vdash \mathsf{c}_2 : (w_2, t_2, f_2)$ with $t_1 \sqsubseteq \mathsf{L}\ell_I$, $t_2 \sqsubseteq \mathsf{L}\ell_I$. If $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \xrightarrow{\alpha} \langle \mathsf{M}_1', \mathsf{c}_1'\rangle$ then by induction on c_1 it holds $\langle \mathsf{M}_2, \mathsf{c}_1\rangle \xrightarrow{\alpha} \langle \mathsf{M}_2', \mathsf{c}_1'\rangle$ with $\mathsf{M}_1' =^h_{\mathsf{LL}} \mathsf{M}_2'$. By command semantics $\langle \mathsf{M}_1, \mathsf{c}_1; \mathsf{c}_2\rangle \xrightarrow{\alpha} \langle \mathsf{M}_1', \mathsf{c}_1'; \mathsf{c}_2\rangle$ and $\langle \mathsf{M}_2, \mathsf{c}_1; \mathsf{c}_2\rangle \xrightarrow{\alpha} \langle \mathsf{M}_2', \mathsf{c}_1'; \mathsf{c}_2\rangle$ proving the case.

If $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \xrightarrow{\alpha} \langle \mathsf{M}_1', \varepsilon\rangle$ then by induction on c_1 it is $\langle \mathsf{M}_2, \mathsf{c}_1\rangle \xrightarrow{\alpha} \langle \mathsf{M}_2', \varepsilon\rangle$ with $\mathsf{M}_1' =^h_{\mathsf{LL}} \mathsf{M}_2'$.
The case follows by command semantics: $\langle \mathsf{M}_i, \mathsf{c}_1; \mathsf{c}_2\rangle \xrightarrow{\alpha} \langle \mathsf{M}_i', \mathsf{c}_2\rangle$.

□

Proposition B.13.
If $\Delta \vdash \mathsf{c} : (w, \mathsf{H_b}\ell_I, f)$ *and* $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ *then* $\langle \mathsf{M}_1, \mathsf{c}\rangle \xrightarrow{\alpha} \langle \mathsf{M}_1', \mathsf{c}'\rangle$ *implies* $\langle \mathsf{M}_2, \mathsf{c}\rangle \xrightarrow{\alpha} \langle \mathsf{M}_2', \mathsf{c}'\rangle$ *with* $\mathsf{M}_1' =^h_{\mathsf{H_bL}} \mathsf{M}_2'$.

Proof. The proof follows by induction on the structure of the command and is all the way similar to the one above for Proposition B.12. □

Given two $\mathsf{H_bL}$ memories an execution of a hash integrity test on them will follow the same branch.

Proposition B.14.
If $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ *and* $\Delta \vdash \mathsf{if}\ \mathsf{hash}(x) = y\ \mathsf{then}\ z := x; \mathsf{c}''\ \mathsf{else}\ \mathsf{FAIL} : (w, t, f)$ *then* $\langle \mathsf{M}_1, \mathsf{c}\rangle \dot{\rightarrow} \langle \mathsf{M}_1, \mathsf{c}'\rangle$ *implies* c' *does not type,* $\langle \mathsf{M}_2, \mathsf{c}\rangle \dot{\rightarrow} \langle \mathsf{M}_2, \mathsf{c}'\rangle$ *and* $\mathsf{c}' = z := x; \mathsf{c}''$ *with* $\Delta(z) = \mathsf{P}\ell_C\mathsf{H}$, $\Delta \vdash x : \mathsf{D}\ell_C\mathsf{L}$, $\ell_C \sqsubseteq_C \mathsf{H_b}$ *and* $\Delta \vdash \mathsf{c}'' : (w', t', f')$ *or* $\mathsf{c}' = \mathsf{FAIL}$.

Proof. The command can only be typed by (int-hash) and $\mathsf{hash}(x) = y$ will evaluate to the same boolean value on the two memories since $\Delta(x) = \mathsf{P}\ell_C\mathsf{L}$, $\Delta \vdash y : \mathsf{D}\ell_C\mathsf{H}$ with $\ell_C \sqsubseteq_C \mathsf{H_b}$. □

The following lemma proves that typed commands are related by a $\mathsf{H_bL}$-TSB.

Lemma B.6. *If* $\Delta \vdash \mathsf{c} : (w, t, f)$ *and* $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ *then* $\langle \mathsf{M}_1, \mathsf{c}\rangle \simeq_{\mathsf{H_bL}} \langle \mathsf{M}_2, \mathsf{c}\rangle$.

Proof. Take $\mathcal{S}$ a relation on command configurations defined as $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$ if and only if $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ and either:

(a) $\Delta \vdash \mathsf{c}_1 : (w_1, t_1, f_1)$, $\Delta \vdash \mathsf{c}_2 : (w_2, t_2, f_2)$ and either:

1. c_1 and c_2 are $\mathsf{H_s}$
2. $\mathsf{c}_1 = \mathsf{c}_2$
3. $\mathsf{c}_1 = \mathsf{c}_1'; \mathsf{c}''$, $\mathsf{c}_2 = \mathsf{c}_2'; \mathsf{c}''$ with $\langle \mathsf{M}_1, \mathsf{c}_1'\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}_2'\rangle$

or (b) $\mathsf{c}_1 = \mathsf{c}_2 = \mathsf{FAIL}$ or $\mathsf{c}_1 = \mathsf{c}_2 = \varepsilon$
or (c) $\mathsf{c}_1 = \mathsf{c}_2 = x := y; \mathsf{c}'$ with $\Delta(x) = \mathsf{P}\ell_C\mathsf{H}$, $\Delta \vdash y : \mathsf{P}\ell_C\mathsf{L}$ with $\ell_C \sqsubseteq_C \mathsf{H_b}$ and $\Delta \vdash \mathsf{c}' : (w', t', f')$.

In the following $\mathcal{S}$ is shown to be an $\mathsf{H_bL}$-TSB. Since $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ by definition, it remains to prove that if $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \xrightarrow{\alpha}_{\mathsf{H_bL}} \langle \mathsf{M}_1', \mathsf{c}_1'\rangle$ then $\langle \mathsf{M}_2, \mathsf{c}_2\rangle \xrightarrow{\alpha}{}^*_{\mathsf{H_bL}} \langle \mathsf{M}_2', \mathsf{c}_2'\rangle$ and $\langle \mathsf{M}_1', \mathsf{c}_1'\rangle \mathcal{S} \langle \mathsf{M}_2', \mathsf{c}_2'\rangle$. By induction on the definition of $\mathcal{S}$.
Case (a):

1. By Proposition B.10
2. In this case $\mathsf{c}_1 = \mathsf{c}_2$ thus the proof proceed by induction on the structure of the command c_1.

   skip
   This case is obvious.

   $x := e$
   The type system states that $\Delta \vdash e : \tau$ and $\Delta(x) = \tau$. Let $e \downarrow^{\mathsf{M}_i} v_i$. Two cases must be considered.

   If $\mathsf{L}(\Delta(x)) \sqsubseteq \mathsf{H_bL}$ then by Lemma B.1 $\mathsf{M}_1[x \mapsto v_1] =^h_{\mathsf{H_bL}} \mathsf{M}_2[x \mapsto v_2]$. Note that $\langle \mathsf{M}_i, x := e\rangle \xrightarrow{x}_{\mathsf{H_bL}} \langle \mathsf{M}_i[x \mapsto v_i], \varepsilon\rangle$, this prove the case since the two memories are equivalent and the configurations are related by clause (b).

   Otherwise, the assignment will not be observable, thus it easily holds that $\mathsf{M}_1[x \mapsto v_1] =^h_{\mathsf{H_bL}} \mathsf{M}_2[x \mapsto v_2]$ and the case follows as for the previous one.

   $\mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}_1'\ \mathsf{else}\ \mathsf{c}_2'$
   Three different cases are considered depending on the rule used to type the command.

- (if): It holds $\Delta \vdash e : \tau$ and $\Delta \vdash \mathsf{c}'_i : (w'_i, t'_i, f'_i)$. If $\mathsf{L}(\tau) \sqsubseteq \mathsf{H_bL}$ then the expression will evaluate to the same value in both M_1 and M_2 thus the same branch will be followed: If $e \downarrow^{\mathsf{M}_i} \mathsf{false}$ then $\langle \mathsf{M}_i, \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}'_1\ \mathsf{else}\ \mathsf{c}'_2 \rangle \dot{\rightarrow}_{\mathsf{H_bL}} \langle \mathsf{M}_i, \mathsf{c}'_2 \rangle$, clearly the resulting memories are equivalent since they are unchanged and the configurations are related by clause 2.

  If, instead $\mathsf{L}(\tau) \not\sqsubseteq \mathsf{H_bL}$ then the type system assure that $\mathsf{L}(\tau) \sqsubseteq w_i$ thus $w_i = \mathsf{H_s}\ell^i_I, i : 1, 2$, i.e., the two branches are $\mathsf{H_s}$ commands. The action is silent and do not change memories. Moreover requirement 1 is satisfied proving the case.
- (int-test): In this case e is $x = y$ and c'_2 is FAIL. It holds $\Delta \vdash x : \tau$, $\Delta \vdash y : \tau'$ $\mathsf{T}(\tau) = \mathsf{T}(\tau')$, $\mathsf{L}(\tau) = \ell_C\mathsf{L}$, $\mathsf{L}(\tau') = \ell_C\mathsf{H}$ and $\ell_C \sqsubseteq_C \mathsf{H_b}$. The variables are observable, thus if $\mathsf{T}(\tau) = \mathsf{P}$ then it follows $\mathsf{M}_1(x) = \mathsf{M}_2(x)$ and $\mathsf{M}_1(y) = \mathsf{M}_2(y)$ hence the test expression will evaluate to the same boolean on both memories. If, instead, $\mathsf{T}(\tau) = \mathsf{D}$ then by Proposition B.3 $\mathsf{M}_1(x) \in \mathit{Val}_{db} \Leftrightarrow \mathsf{M}_2(x) \in \mathit{Val}_{db}$: If $\mathsf{M}_1(x) \in \mathit{Val}_{ds}$ then $\mathsf{M}_1(x) = \mathsf{M}_2(x)$ otherwise $\mathsf{p}(\mathsf{M}_1(x)) = \mathsf{p}(\mathsf{M}_2(x))\rho$, and the same holds for y. Thus $x = y$ will evaluate to the same value on both memories: Indeed if the values are digest of small values nothing special is needed otherwise the fact that ρ is a bijection prove the statement. Suppose that $x = y \downarrow^{\mathsf{M}_i} \mathsf{false}$, then $\langle \mathsf{M}_i, \mathsf{if}\ x = y\ \mathsf{then}\ \mathsf{c}'_1\ \mathsf{else}\ \mathsf{FAIL} \rangle \dot{\rightarrow}_{\mathsf{H_bL}} \langle \mathsf{M}_i, \mathsf{FAIL} \rangle$ which proves the case by satisfying requirement (b). If the expression gets evaluated to true $\langle \mathsf{M}_i, \mathsf{if}\ x = y\ \mathsf{then}\ \mathsf{c}'_1\ \mathsf{else}\ \mathsf{FAIL} \rangle \dot{\rightarrow}_{\mathsf{H_bL}} \langle \mathsf{M}_i, \mathsf{c}'_1 \rangle$ concluding the case, indeed $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ and the configurations are related by clause 2.
- (int-hash): Here e is $\mathsf{hash}(x) = y$, c'_1 is $z := x; \mathsf{c}''_1$ and c'_2 is FAIL. It holds $\Delta \vdash x : \mathsf{P}\ell_C\mathsf{L}$, $\Delta \vdash y : \mathsf{D}\ell_C\mathsf{H}$, $\Delta(z) = \mathsf{P}\ell_C\mathsf{H}$ and $\ell_C \sqsubseteq \mathsf{H_b}$. It follows $\mathsf{M}_1(x) = \mathsf{M}_2(x)$ and $\mathsf{M}_1(y) \in \mathit{Val}_{db} \Leftrightarrow \mathsf{M}_2(y) \in \mathit{Val}_{db}$. Suppose $\mathsf{M}_1(y) \in \mathit{Val}_{ds}$ (indeed memories are well formed thus it must be a digest) it must be that $\mathsf{M}_1(y) = \mathsf{M}_2(y)$ thus $\mathsf{hash}(x) \downarrow^{\mathsf{M}_1} = \mathsf{M}_1(y)$ if and only if $\mathsf{hash}(x) \downarrow^{\mathsf{M}_2} = \mathsf{M}_2(y)$. Otherwise if $\mathsf{M}_1(y) \in \mathit{Val}_{db}$ it holds $\mathsf{p}(\mathsf{M}_1(y)) = \mathsf{p}(\mathsf{M}_2(y))\rho$ but $\mathsf{M}_1(x) = \mathsf{M}_2(x)$ thus since the hash operator is deterministic the test gets evaluated to the same value in both memories. If the guard evaluates to true then
  $\langle \mathsf{M}_i, \mathsf{if}\ \mathsf{hash}(x) = y\ \mathsf{then}\ z := x; \mathsf{c}''_1\ \mathsf{else}\ \mathsf{FAIL} \rangle \dot{\rightarrow} \langle \mathsf{M}_i, z := x; \mathsf{c}''_1 \rangle$ proving the case by $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ and clause (c), indeed $\Delta(x) = \mathsf{P}\ell_C\mathsf{H}$ $\Delta \vdash y : \mathsf{P}\ell_C\mathsf{L}$ with $\ell_C \sqsubseteq_C \mathsf{H_b}$ and $\Delta \vdash \mathsf{c}''_1 : (\ell_C\mathsf{H}, t, f)$. If the guard evaluates to false then the two configurations are related by clause (b).

$\mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'$

The case is similar to the conditional one sub-case (if).

$\mathsf{c}_1; \mathsf{c}_2$

Let $\Delta \vdash \mathsf{c}_i : (w_i, t_i, f_i)$. Two cases are considered.

If $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \xrightarrow{\alpha}_{\mathsf{H_bL}} \langle \mathsf{M}_1', \mathsf{c}_1'\rangle$ and $\mathsf{c}_1' \neq \varepsilon$ then $\langle \mathsf{M}_1, \mathsf{c}_1; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{H_bL}} \langle \mathsf{M}_1', \mathsf{c}_1'; \mathsf{c}_2\rangle$. By induction on c_1 $\langle \mathsf{M}_2, \mathsf{c}_1\rangle \xrightarrow{\alpha}{}^*_{\mathsf{H_bL}} \langle \mathsf{M}_2', \mathsf{c}_2'\rangle$, $\mathsf{M}_1' =^h_{\mathsf{H_bL}} \mathsf{M}_2'$ and $\langle \mathsf{M}_1', \mathsf{c}_1'\rangle \mathcal{S} \langle \mathsf{M}_2', \mathsf{c}_2'\rangle$.
By command semantics it follows $\langle \mathsf{M}_2, \mathsf{c}_1; \mathsf{c}_2\rangle \xrightarrow{\alpha}{}^*_{\mathsf{H_bL}} \langle \mathsf{M}_2', \mathsf{c}_2'; \mathsf{c}_2\rangle$.
If $\Delta \vdash \mathsf{c}_i'; \mathsf{c}_2 : (w_i', t_i', f_i')$ then two cases are possible:

- $\mathsf{c}_1' = \mathsf{c}_2'$: It follows $\mathsf{c}_1'; \mathsf{c}_2 = \mathsf{c}_2'; \mathsf{c}_2$ proving the case by requirement 2.
- $\mathsf{c}_1' \neq \mathsf{c}_2'$: The two configurations are in $\mathcal{S}$ by clause 3.

If c_1' does not type then by subject reduction (Lemma B.2) it could be $\mathsf{c}_1' = x := y; \mathsf{c}_1''$ and $\Delta(x) = \mathsf{P}\ell_C\mathsf{H}$, $\Delta \vdash y : \mathsf{P}\ell_C\mathsf{L}$ and $\Delta \vdash \mathsf{c}_1'' : (w_1'', t_1'', f_1'')$ in which case the same holds for c_2', i.e., $\mathsf{c}_2' = \mathsf{c}_1'$ and also $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \dot{\rightarrow} \langle \mathsf{M}_1, \mathsf{c}_1'\rangle$, $\langle \mathsf{M}_2, \mathsf{c}_1\rangle \dot{\rightarrow} \langle \mathsf{M}_2, \mathsf{c}_1'\rangle$ by Proposition B.14. By command semantics $\langle \mathsf{M}_1, \mathsf{c}_1; \mathsf{c}_2\rangle \dot{\rightarrow}_{\mathsf{H_bL}} \langle \mathsf{M}_1, \mathsf{c}_1'; \mathsf{c}_2\rangle$ and $\langle \mathsf{M}_2, \mathsf{c}_1; \mathsf{c}_2\rangle \dot{\rightarrow}_{\mathsf{H_bL}} \langle \mathsf{M}_2, \mathsf{c}_1'; \mathsf{c}_2\rangle$ and the two configurations are related by clause (c), indeed $\mathsf{c}_1''; \mathsf{c}_2$ is typed by subject reduction. It could also be that $\mathsf{c}_1' = \mathsf{c}_2' = \mathsf{FAIL}$ which prove the case by requirement (b).

Otherwise $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \xrightarrow{\alpha}_{\mathsf{H_bL}} \langle \mathsf{M}_1', \varepsilon\rangle$ and by induction on c_1 $\langle \mathsf{M}_2, \mathsf{c}_1\rangle \xrightarrow{\alpha}{}^*_{\mathsf{H_bL}} \langle \mathsf{M}_2', \varepsilon\rangle$ and $\mathsf{M}_1' =^h_{\mathsf{H_bL}} \mathsf{M}_2'$. It follows $\langle \mathsf{M}_1, \mathsf{c}_1; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{H_bL}} \langle \mathsf{M}_1', \mathsf{c}_2\rangle$ and $\langle \mathsf{M}_2, \mathsf{c}_1; \mathsf{c}_2\rangle \xrightarrow{\alpha} \langle \mathsf{M}_2', \mathsf{c}_2\rangle$ proving the case by clause 2.

3. $\mathsf{c}_1 = \mathsf{c}_1'; \mathsf{c}''$, $\mathsf{c}_2 = \mathsf{c}_2'; \mathsf{c}''$ with $\langle \mathsf{M}_1, \mathsf{c}_1'\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}_2'\rangle$.
   If $\langle \mathsf{M}_1, \mathsf{c}_1'\rangle \xrightarrow{\alpha} \mathsf{H_bL}\langle \mathsf{M}_1', \mathsf{c}_1''\rangle$ by induction hypothesis $\langle \mathsf{M}_2, \mathsf{c}_2'\rangle \xrightarrow{\alpha}{}^*_{\mathsf{H_bL}} \langle \mathsf{M}_2', \mathsf{c}_2''\rangle$ and $\langle \mathsf{M}_1', \mathsf{c}_1''\rangle \mathcal{S} \langle \mathsf{M}_2', \mathsf{c}_2''\rangle$ from which follows $\mathsf{M}_1' =^h_{\mathsf{H_bL}} \mathsf{M}_2'$. It holds $\langle \mathsf{M}_1, \mathsf{c}_1'; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{H_bL}} \langle \mathsf{M}_1', \mathsf{c}_1''; \mathsf{c}_2\rangle$ and $\langle \mathsf{M}_1, \mathsf{c}_1'; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{H_bL}} \langle \mathsf{M}_1', \mathsf{c}_1''; \mathsf{c}_2\rangle$ proving the case by clause 3.

   If instead $\langle \mathsf{M}_1, \mathsf{c}_1'\rangle \xrightarrow{\alpha}_{\mathsf{H_bL}} \langle \mathsf{M}_1', \varepsilon\rangle$ then by induction hypothesis $\langle \mathsf{M}_2, \mathsf{c}_2'\rangle \xrightarrow{\alpha}{}^*_{\mathsf{H_bL}} \langle \mathsf{M}_2', \varepsilon\rangle$ with $\mathsf{M}_1' =^h_{\mathsf{H_bL}} \mathsf{M}_2'$. It follows $\langle \mathsf{M}_1, \mathsf{c}_1'; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{H_bL}} \langle \mathsf{M}_1', \mathsf{c}_2\rangle$ and $\langle \mathsf{M}_2, \mathsf{c}_2'; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{H_bL}} \langle \mathsf{M}_2', \mathsf{c}_2\rangle$ which prove the case by relating the configurations by constraint 2.

Case (b): Note that $\langle \mathsf{M}_1, \mathsf{c}_1\rangle$ and $\langle \mathsf{M}_2, \mathsf{c}_2\rangle$ either cannot move or both silently diverge, getting the result.

Case (c): Let $\mathsf{M}_i(y) = v_i$ it follows $\langle \mathsf{M}_1, x := y; \mathsf{c}'\rangle \xrightarrow{x}_{\mathsf{H_bL}} \langle \mathsf{M}_1[x \mapsto v_1], \mathsf{c}'\rangle$ and $\langle \mathsf{M}_2, x := y; \mathsf{c}'\rangle \xrightarrow{x}_{\mathsf{H_bL}} \langle \mathsf{M}_2[x \mapsto v_2], \mathsf{c}'\rangle$. Note that $\Delta \vdash y : \mathsf{PH_bL}$ so by Corollary B.1 v_1 and v_2 are not in the domain of a digest substitution so from $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ it follows $\mathsf{M}_1(y) = \mathsf{M}_2(y)$ hence $\mathsf{M}_1[x \mapsto v_1] =^h_{\mathsf{H_bL}}$
$\mathsf{M}_2[x \mapsto v_2]$. The case is then proved since c' is typed and so $\langle \mathsf{M}_1', \mathsf{c}'\rangle \mathcal{S} \langle \mathsf{M}_2', \mathsf{c}'\rangle$ by clause 2.

It follows that $\mathcal{S}$ is $\mathsf{H_bL}$-TSB and the lemma is proved by clause 2. □

The following lemma proves, instead, that typed commands are related by a LL-TIB.

Lemma B.7. *If* $\Delta \vdash \mathsf{c} : (w, t, f)$ *and* $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}_2$ *then* $\langle \mathsf{M}_1, \mathsf{c} \rangle \approx_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c} \rangle$.

Proof. Take $\mathcal{S}$ a relation on command configurations defined as
$\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}_2 \rangle$ if and only if $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}_2$ and either:

(a) $\Delta \vdash \mathsf{c}_1 : (w_1, t_1, f_1)$, $\Delta \vdash \mathsf{c}_2 : (w_2, t_2, f_2)$ and either:

1. c_1 and c_2 are $\mathsf{H_b}$

2. $\mathsf{c}_1 = \mathsf{c}_2$

3. $\mathsf{c}_1 = \mathsf{c}'_1; \mathsf{c}''$, $\mathsf{c}_2 = \mathsf{c}'_2; \mathsf{c}''$ with $\langle \mathsf{M}_1, \mathsf{c}'_1 \rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}'_2 \rangle$

or (b) $\mathsf{c}_1 = \mathsf{FAIL}$ or $\mathsf{c}_2 = \mathsf{FAIL}$
or (c) $\mathsf{c}_1 = \mathsf{c}_2 = \varepsilon$
or (d) $\mathsf{c}_1 = \mathsf{c}_2 = x := y; \mathsf{c}'$ with $\Delta(x) = \mathsf{P}\ell_C\mathsf{H}$, $\Delta \vdash y : \mathsf{P}\ell_C\mathsf{L}$ with $\ell_C \sqsubseteq_C \mathsf{H_b}$ and $\Delta \vdash \mathsf{c}' : (w', t', f')$.

In the following $\mathcal{S}$ is shown to be an LL-TIB. Since $\mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}_2$ by definition, it remains to prove that if $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \xrightarrow{\alpha}_{\mathsf{LL}} \langle \mathsf{M}'_1, \mathsf{c}'_1 \rangle$ then either $\langle \mathsf{M}_2, \mathsf{c}_2 \rangle \xrightarrow{\alpha}^*_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}'_2 \rangle$ and $\langle \mathsf{M}'_1, \mathsf{c}'_1 \rangle \mathcal{S} \langle \mathsf{M}'_2, \mathsf{c}'_2 \rangle$ or $\langle \mathsf{M}_2, \mathsf{c}_2 \rangle \Uparrow$.

By induction on the definition of $\mathcal{S}$.

Case (a):

1. By Proposition B.11

2. In this case $\mathsf{c}_1 = \mathsf{c}_2$ thus the proof proceed by induction on the structure of the command c_1.

   $\mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}'_1\ \mathsf{else}\ \mathsf{c}'_2$
   Three different cases are considered depending on the rule used to type the command.

   - (if): It holds $\Delta \vdash e : \tau$ and $\Delta \vdash \mathsf{c}'_i : (w'_i, t'_i, f'_i)$. If $\mathsf{L}(\tau) \sqsubseteq \mathsf{LL}$ then the expression will evaluate to the same value in both M_1 and M_2 thus the same branch will be followed: If $e \downarrow^{\mathsf{M}_i} \mathsf{false}$ then $\langle \mathsf{M}_i, \mathsf{if}\ e\ \mathsf{then}\ \mathsf{c}'_1\ \mathsf{else}\ \mathsf{c}'_2 \rangle \xrightarrow{\cdot}_{\mathsf{LL}} \langle \mathsf{M}_i, \mathsf{c}'_2 \rangle$, clearly the resulting memories are equivalent since they are unchanged and the configurations are related by clause 2.

     If, instead $\mathsf{L}(\tau) \not\sqsubseteq \mathsf{LL}$ then the type system assure that $\mathsf{L}(\tau) \sqsubseteq w_i$ thus $w_i = \mathsf{H_b}\ell^i_I, i : 1, 2$, i.e., the two branches are $\mathsf{H_b}$ commands. The action is silent and do not change memories. Moreover requirement 1 is satisfied proving the case.

- (int-test): In this case e is $x = y$ and c'_2 is FAIL. It holds $\Delta \vdash x : \tau$, $\Delta \vdash y : \tau'$ $\mathsf{T}(\tau) = \mathsf{T}(\tau')$, $\mathsf{L}(\tau) = \ell_C\mathsf{L}$, $\mathsf{L}(\tau') = \ell_C\mathsf{H}$ and $\ell_C \sqsubseteq_C \mathsf{H_b}$.
  Let $\langle \mathsf{M}_1, \mathsf{if}\ x = y\ \mathsf{then}\ \mathsf{c}'_1\ \mathsf{else}\ \mathsf{FAIL}\rangle \dot{\rightarrow}_{\mathsf{LL}} \langle \mathsf{M}_1, \mathsf{c}''_1\rangle$. If either one of the two runs follow the else-branch then the case is proved by clause (b) relating either $\langle \mathsf{M}_1, \mathsf{FAIL}\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$ or $\langle \mathsf{M}_1, \mathsf{c}''_1\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{FAIL}\rangle$. The case in which both runs follow the then-branch is the only one that remains to prove, let $\langle \mathsf{M}_2, \mathsf{if}\ x = y\ \mathsf{then}\ \mathsf{c}'_1\ \mathsf{else}\ \mathsf{FAIL}\rangle \dot{\rightarrow}_{\mathsf{LL}} \langle \mathsf{M}_1, \mathsf{c}'_1\rangle$ and also $\mathsf{c}''_1 = \mathsf{c}'_1$.
  If $\ell_C = \mathsf{H_b}$ then by (int-test) c'_1 is a $\mathsf{H_b}$ command thus the case is proved by relating the two configuration using constraint 1.
  Otherwise, if $\ell_C = \mathsf{L}$ then $\Delta \vdash \mathsf{c}'_1 : (\mathsf{LH}, t, f)$ and the configurations are related by requirements 2.
- (int-hash): Here e is $\mathsf{hash}(x) = y$, c'_1 is $z := x; \mathsf{c}''_1$ and c'_2 is FAIL.
  Let $\langle \mathsf{M}_1, \mathsf{if}\ \mathsf{hash}(x) = y\ \mathsf{then}\ z := x; \mathsf{c}''_1\ \mathsf{else}\ \mathsf{FAIL}\rangle \dot{\rightarrow}_{\mathsf{LL}} \langle \mathsf{M}_1, \mathsf{c}^*_1\rangle$. If either one of the two runs follow the else-branch then the case is proved by clause (b) relating either $\langle \mathsf{M}_1, \mathsf{FAIL}\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$ or $\langle \mathsf{M}_1, \mathsf{c}^*_1\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{FAIL}\rangle$.
  The case in which both runs follow the then-branch is the only one that remains to prove.
  Let $\langle \mathsf{M}_2, \mathsf{if}\ \mathsf{hash}(x) = y\ \mathsf{then}\ z := x\mathsf{c}''_1\ \mathsf{else}\ \mathsf{FAIL}\rangle \dot{\rightarrow}_{\mathsf{LL}} \langle \mathsf{M}_1, z := x; \mathsf{c}'_1\rangle$ and also $\mathsf{c}^*_1 = x := z; \mathsf{c}''_1$. The configurations are related by clause (d) concluding the case.

The other cases are similar to the proof of Lemma B.6.

3. $\mathsf{c}_1 = \mathsf{c}'_1; \mathsf{c}''$, $\mathsf{c}_2 = \mathsf{c}'_2; \mathsf{c}''$ with $\langle \mathsf{M}_1, \mathsf{c}'_1\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}'_2\rangle$.
   If $\langle \mathsf{M}_1, \mathsf{c}'_1\rangle \xrightarrow{\alpha} \mathsf{LL}\langle \mathsf{M}'_1, \mathsf{c}''_1\rangle$ by induction hypothesis $\langle \mathsf{M}_2, \mathsf{c}'_2\rangle \xrightarrow{\alpha}{}^*_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}''_2\rangle$ and $\langle \mathsf{M}'_1, \mathsf{c}''_1\rangle \mathcal{S} \langle \mathsf{M}'_2, \mathsf{c}''_2\rangle$ from which follows $\mathsf{M}'_1 =^h_{\mathsf{LL}} \mathsf{M}'_2$ or $\langle \mathsf{M}_2, \mathsf{c}'_2\rangle \Uparrow$. It holds $\langle \mathsf{M}_1, \mathsf{c}'_1; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{LL}} \langle \mathsf{M}'_1, \mathsf{c}''_1; \mathsf{c}_2\rangle$ and $\langle \mathsf{M}_2, \mathsf{c}'_2; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}''_1; \mathsf{c}_2\rangle$ proving the case by clause 3 or $\langle \mathsf{M}_2, \mathsf{c}'_2; \mathsf{c}_2\rangle \Uparrow$.

   If instead $\langle \mathsf{M}_1, \mathsf{c}'_1\rangle \xrightarrow{\alpha}_{\mathsf{LL}} \langle \mathsf{M}'_1, \varepsilon\rangle$ then by induction hypothesis $\langle \mathsf{M}_2, \mathsf{c}'_2\rangle \xrightarrow{\alpha}{}^*_{\mathsf{LL}} \langle \mathsf{M}'_2, \varepsilon\rangle$ with $\mathsf{M}'_1 =^h_{\mathsf{H_bL}} \mathsf{M}'_2$ or $\langle \mathsf{M}_2, \mathsf{c}'_2\rangle \Uparrow$. It follows $\langle \mathsf{M}_1, \mathsf{c}'_1; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{LL}} \langle \mathsf{M}'_1, \mathsf{c}_2\rangle$ and $\langle \mathsf{M}_2, \mathsf{c}'_2; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}_2\rangle$ which prove the case by relating the configurations by constraint 2 or $\langle \mathsf{M}_2, \mathsf{c}'_2; \mathsf{c}_2\rangle \Uparrow$.

Case (b): Note that FAIL is a silent diverging program thus if $\mathsf{c}_1 = \mathsf{FAIL}$ every moves will be matched by $\langle \mathsf{M}_2, \mathsf{c}_2\rangle \dot{\rightarrow}^0_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$, otherwise if $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \xrightarrow{\alpha}_{\mathsf{LL}} \langle \mathsf{M}'_1, \mathsf{c}'_1\rangle$ it holds $\langle \mathsf{M}_2, \mathsf{FAIL}\rangle \Uparrow$.

Case (c): The configurations do not move proving the case.

Case (d): Let $\mathsf{M}_i(y) = v_i$. If $\ell_C = \mathsf{H_b}$ then it follows $\langle \mathsf{M}_i, x := y; \mathsf{c}'\rangle \dot{\rightarrow}_{\mathsf{LL}} \langle \mathsf{M}_i[x \mapsto v_i], \mathsf{c}'\rangle$ and obviously $\mathsf{M}_1[x \mapsto v_1] =^h_{\mathsf{LL}} \mathsf{M}_2[x \mapsto v_2]$ and the two configurations are related by clause 2, indeed c' is typed. Otherwise, $\ell_C = \mathsf{L}$ and it follows

$\langle \mathsf{M}_1, x := y; \mathsf{c}' \rangle \xrightarrow{x}_{\mathsf{LL}} \langle \mathsf{M}_1[x \mapsto v_1], \mathsf{c}' \rangle$ and $\langle \mathsf{M}_2, x := y; \mathsf{c}' \rangle \xrightarrow{x}_{\mathsf{LL}} \langle \mathsf{M}_2[x \mapsto v_2], \mathsf{c}' \rangle$. Note that $\Delta \vdash y : \mathsf{PLL}$ so by Corollary B.1 v_1 and v_2 are not in the domain of a digest substitution so from $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ it follows $\mathsf{M}_1(y) = \mathsf{M}_2(y)$ hence $\mathsf{M}_1[x \mapsto v_1] =^h_{\mathsf{LL}} \mathsf{M}_2[x \mapsto v_2]$. The case is then proved since c' is typed and so $\langle \mathsf{M}'_1, \mathsf{c}' \rangle \mathcal{S} \langle \mathsf{M}'_2, \mathsf{c}' \rangle$ by clause 2.

It follows that $\mathcal{S}$ is LL-TIB and the lemma is proved by clause 2. □

It then follows that a typed command obeys to secret-sensitive NI.

Theorem. (**SSNI by typing**)
If $\Delta \vdash \mathsf{c} : (w, t, f)$ *then* c *satisfies SSNI.*

Proof. It holds that if $\Delta \vdash \mathsf{c} : (w, t, f)$ then $\forall \mathsf{M}_1 =^h_{\mathsf{LL}} \mathsf{M}_2$

1. $\langle \mathsf{M}_1, \mathsf{c} \rangle \approx_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c} \rangle$ by Lemma B.7 and
2. $\mathsf{M}_1 =^h_{\mathsf{H_bL}} \mathsf{M}_2$ implies $\langle \mathsf{M}_1, \mathsf{c} \rangle \simeq_{\mathsf{H_bL}} \langle \mathsf{M}_2, \mathsf{c} \rangle$ by Lemma B.6.

□

Integrity NI by typing It is now proved that the type system assures that a typed program obeys to integrity NI.

Proposition B.15. *If* $\Delta \vdash \mathsf{c} : (\ell_C \mathsf{L}, t, f)$ *and* $\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}' \rangle$ *then* $\Delta \vdash \mathsf{c}' : (\ell'_C \mathsf{L}, t', f')$

Proof. By subject reduction (Lemma B.2), notice that case 2 cannot be applied since c' is typed $(\ell_C \mathsf{H}, t'', f'')$ by Proposition B.5 and $(\ell_C \mathsf{H}, t'', f'') \not\leq (\ell_C \mathsf{L}, t, f)$. □

Note that $\mathsf{M}_1 =_{\mathsf{H_sH}} \mathsf{M}_2$ implies $\mathsf{M}_1 =^h_{\mathsf{H_sH}} \mathsf{M}_2$ indeed it holds $\mathsf{M}_{1|\mathsf{H_sH}} = \mathsf{M}_{2|\mathsf{H_sH}}\rho$ where ρ is the identity function.

Proposition B.16.
$\{(\langle \mathsf{M}_1, \mathsf{c}_1 \rangle, \langle \mathsf{M}_2, \mathsf{c}_2 \rangle) | \ \mathsf{M}_1 =_{\mathsf{H_sH}} \mathsf{M}_2$ *and* $\Delta \vdash cmd_i : (\ell^i_C \mathsf{L}, t_i, f_i)\}$ *is a* $\mathsf{H_sH}$*-TIB.*

Proof. The relation is symmetric by definition. Commands are typed as $\Delta \vdash \mathsf{c}_i : (\ell^i_C \mathsf{L}, t_i, f_i)$.

If $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \xrightarrow{\alpha} \langle \mathsf{M}'_1, \mathsf{c}'_1 \rangle$ then by Confinement (Lemma B.3) the command assigns only to variables whose level is greater or equal to $\ell^i_C \mathsf{L}$ thus it holds $\langle \mathsf{M}_1, \mathsf{c}_1 \rangle \dot{\rightarrow}_{\mathsf{H_sH}} \langle \mathsf{M}'_1, \mathsf{c}'_1 \rangle$ and $\mathsf{M}_1 =_{\mathsf{H_sH}} \mathsf{M}'_1$. The transition can be matched by $\langle \mathsf{M}_2, \mathsf{c}_2 \rangle \overset{0}{\dot{\rightarrow}}_{\mathsf{H_sH}} \langle \mathsf{M}_2, \mathsf{c}_2 \rangle$, indeed since $\mathsf{M}_1 =_{\mathsf{H_sH}} \mathsf{M}_2$ then $\mathsf{M}'_1 =_{\mathsf{H_sH}} \mathsf{M}_2$. Command c'_1 is still such that $\Delta \vdash \mathsf{c}' : (\ell'_C \mathsf{L}, t', f')$ by Proposition B.15. The two configurations are thus contained in the relation proving the claim. □

Lemma B.8. *If* $\Delta \vdash \mathsf{c} : (w, t, f)$ *and* $\mathsf{M}_1 =_{\mathsf{H_sH}} \mathsf{M}_2$ *then* $\langle \mathsf{M}_1, \mathsf{c} \rangle \approx_{\mathsf{H_sH}} \langle \mathsf{M}_2, \mathsf{c} \rangle$.

Proof. Let $\mathcal{S}$ be a relation on command configurations defined as $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$ if and only if $\mathsf{M}_1 =_{\mathsf{H_sH}} \mathsf{M}_2$ and either:

(a) $\Delta \vdash \mathsf{c}_1 : (w_1, t_1, f_1)$, $\Delta \vdash \mathsf{c}_2 : (w_2, t_2, f_2)$ and either:

1. $\Delta \vdash \mathsf{c}_i : (\ell_C^i \mathsf{L}, t_i, f_i)$
2. $\mathsf{c}_1 = \mathsf{c}_2$
3. $\mathsf{c}_1 = \mathsf{c}_1'; \mathsf{c}''$, $\mathsf{c}_2 = \mathsf{c}_2'; \mathsf{c}''$ with $\langle \mathsf{M}_1, \mathsf{c}_1'\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}_2'\rangle$

or (b) $\mathsf{c}_1 = \mathsf{FAIL}$ or $\mathsf{c}_2 = \mathsf{FAIL}$
or (c) $\mathsf{c}_1 = \mathsf{c}_2 = \varepsilon$
or (d) $\mathsf{c}_1 = \mathsf{c}_2 = x := y; \mathsf{c}'$ with $\Delta(x) = \mathsf{P}\ell_C\mathsf{H}$, $\Delta \vdash y : \mathsf{P}\ell_C\mathsf{L}$ with $\ell_C \sqsubseteq_C \mathsf{H_b}$ and $\Delta \vdash \mathsf{c}' : (w', t', f')$ and $\mathsf{M}_1(y) = \mathsf{M}_2(y)$.

In the following $\mathcal{S}$ is shown to be an $\mathsf{H_sH}$-TIB. Since $\mathsf{M}_1 =^h_{\mathsf{H_sH}} \mathsf{M}_2$ by definition, it remains to prove that if $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \xrightarrow{\alpha}_{\mathsf{H_sH}} \langle \mathsf{M}_1', \mathsf{c}_1'\rangle$ then either $\langle \mathsf{M}_2, \mathsf{c}_2\rangle \xrightarrow{\alpha}{}^*_{\mathsf{H_sH}} \langle \mathsf{M}_2', \mathsf{c}_2'\rangle$ and $\langle \mathsf{M}_1', \mathsf{c}_1'\rangle \mathcal{S} \langle \mathsf{M}_2', \mathsf{c}_2'\rangle$ or $\langle \mathsf{M}_2, \mathsf{c}_2\rangle \Uparrow$.

By induction on the definition of $\mathcal{S}$.

Case (a):

1. By Proposition B.16
2. In this case $\mathsf{c}_1 = \mathsf{c}_2$ so the proof follows by induction on the structure of the command c_1

   skip
   This case is obvious.

   $x := e$
   It holds $\Delta(x) = \tau$ and $\Delta \vdash \exp : \tau$. Let $e \downarrow^{\mathsf{M}_i} v_i$, it follows $\langle \mathsf{M}_i, x := e\rangle \xrightarrow{x} \langle \mathsf{M}_i[x \mapsto v_i], \varepsilon\rangle$. If $\mathsf{L}(\tau) = \ell_C\mathsf{L}$ then $\langle \mathsf{M}_i, x := e\rangle \dot{\rightarrow}_{\mathsf{H_sH}} \langle \mathsf{M}_i[x \mapsto v_i], \varepsilon\rangle$ and $\mathsf{M}_1[x \mapsto v_2] =_{\mathsf{H_sH}} \mathsf{M}_2[x \mapsto v_2]$ proving the case by requirement (c). Otherwise the case follows similarly by Expression ℓ-equivalence (Lemma B.1).

   if e then c_1' else c_2'
   Three cases are considered depending on the rule used to type the command.

   - (if): It holds $\Delta \vdash e : \tau$, $\Delta \vdash \mathsf{c} : (w_i, t_i, f_i)$ and $\mathsf{L}(\tau) \sqsubseteq w_i$.
     If $\mathsf{L}(\tau) = \ell_C\mathsf{L}$ then the $w_i = \ell_C^i\mathsf{L}$, it follows $\langle \mathsf{M}_1, \text{if } e \text{ then } \mathsf{c}_1' \text{ else } \mathsf{c}_2'\rangle \dot{\rightarrow}_{\mathsf{H_sH}} \langle \mathsf{M}_1, \mathsf{c}_j'\rangle$ and $\langle \mathsf{M}_2, \text{if } e \text{ then } \mathsf{c}_1' \text{ else } \mathsf{c}_2'\rangle \dot{\rightarrow}_{\mathsf{H_sH}} \langle \mathsf{M}_2, \mathsf{c}_z'\rangle$ so the configurations are related by clause 1.
     Otherwise, $\mathsf{L}(\tau) = \ell_C\mathsf{H}$ and so the expression evaluates to the same boolean value on both memories by Expression ℓ-equivalence thus $\langle \mathsf{M}_i, \text{if } e \text{ then } \mathsf{c}_1' \text{ else } \mathsf{c}_2'\rangle \dot{\rightarrow}_{\mathsf{H_sH}} \langle \mathsf{M}_1, \mathsf{c}_j'\rangle$ and the configurations are related by constraint 2.

- (int-test): In this case e is $x = y$, c'_2 is FAIL, $\Delta(x) = \tau$, $\Delta \vdash y : \tau'$ with $\mathsf{L}(\tau) = \ell_C\mathsf{L}$, $\mathsf{L}(\tau') = \ell_C\mathsf{H}$, $\mathsf{T}(\tau) = \mathsf{T}(\tau')$.
  Let $\langle \mathsf{M}_1, \mathsf{if}\ x = y\ \mathsf{then}\ \mathsf{c}'_1\ \mathsf{else\ FAIL}\rangle \dot{\rightarrow}_{\mathsf{H_sH}} \langle \mathsf{M}_1, \mathsf{c}''_1\rangle$. If either one of the two runs follow the else-branch then the case is proved by clause (b) relating either $\langle \mathsf{M}_1, \mathsf{FAIL}\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$ or $\langle \mathsf{M}_1, \mathsf{c}''_1\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{FAIL}\rangle$. The case in which both runs follow the then-branch is the only one that remains to prove, let $\langle \mathsf{M}_2, \mathsf{if}\ x = y\ \mathsf{then}\ \mathsf{c}'_1\ \mathsf{else\ FAIL}\rangle \dot{\rightarrow}_{\mathsf{H_sH}} \langle \mathsf{M}_1, \mathsf{c}'_1\rangle$ and also $\mathsf{c}''_1 = \mathsf{c}'_1$ thus the two configurations are related by clause 2.
- (int-hash): Here e is $\mathsf{hash}(x) = y$, c'_1 is $z := x; \mathsf{c}''_1$ and c'_2 is FAIL.
  Let $\langle \mathsf{M}_1, \mathsf{if\ hash}(x) = y\ \mathsf{then}\ z := x; \mathsf{c}''_1\ \mathsf{else\ FAIL}\rangle \dot{\rightarrow}_{\mathsf{H_sH}} \langle \mathsf{M}_1, \mathsf{c}^*_1\rangle$. If either one of the two runs follow the else-branch then the case is proved by clause (b) relating either $\langle \mathsf{M}_1, \mathsf{FAIL}\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$ or $\langle \mathsf{M}_1, \mathsf{c}^*_1\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{FAIL}\rangle$.
  The case in which both runs follow the then-branch is the only one that remains to prove.
  Let $\langle \mathsf{M}_2, \mathsf{if\ hash}(x) = y\ \mathsf{then}\ z := x; \mathsf{c}''_1\ \mathsf{else\ FAIL}\rangle \dot{\rightarrow}_{\mathsf{H_sH}} \langle \mathsf{M}_1, z := x; \mathsf{c}'_1\rangle$ and also $\mathsf{c}^*_1 = x := z; \mathsf{c}''_1$.
  It holds $\mathsf{hash}(x) = y \downarrow^{\mathsf{M}_i} \mathsf{true}$, $\mathsf{M}_1(y) = \mathsf{M}_2(y) = v$ so if $v = \mathsf{h}(v')$ then it follows $\mathsf{M}_1(x) = \mathsf{M}_2(x) = v'$, otherwise $v = \mathsf{h^b}(v'')$ and $\mathsf{M}_1(x) = \mathsf{M}_2(x) = v''$. The configurations are related by clause (d) concluding the case.

$\mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}'$

This case is similar to the conditional one sub-case (if).

$\mathsf{c}'_1; \mathsf{c}'_2$

Let $\Delta \vdash \mathsf{c}'_i : (w'_i, t'_i, f'_i)$. Two cases are considered. If $\langle \mathsf{M}_1, \mathsf{c}'_1\rangle \xrightarrow{\alpha}_{\mathsf{H_sH}} \langle \mathsf{M}'_1, \mathsf{c}''_1\rangle$ and $\mathsf{c}''_1 \neq \varepsilon$ then $\langle \mathsf{M}_1, \mathsf{c}'_1; \mathsf{c}'_2\rangle \xrightarrow{\alpha}_{\mathsf{H_sH}} \langle \mathsf{M}'_1, \mathsf{c}''_1; \mathsf{c}_2\rangle$. By induction on c'_1 $\langle \mathsf{M}_2, \mathsf{c}'_1\rangle \xrightarrow{\alpha}{}^{*}_{\mathsf{H_sH}} \langle \mathsf{M}'_2, \mathsf{c}''_2\rangle$, $\mathsf{M}'_1 =^h_{\mathsf{H_sH}} \mathsf{M}'_2$ and $\langle \mathsf{M}'_1, \mathsf{c}''_1\rangle \mathcal{S} \langle \mathsf{M}'_2, \mathsf{c}''_2\rangle$ or $\langle \mathsf{M}_2, \mathsf{c}'_1\rangle \Uparrow$. In the latter case the proof is done.

Otherwise by command semantics it follows $\langle \mathsf{M}_2, \mathsf{c}'_1; \mathsf{c}'_2\rangle \xrightarrow{\alpha}{}^{*}_{\mathsf{H_sH}} \langle \mathsf{M}'_2, \mathsf{c}''_2; \mathsf{c}_2\rangle$. If $\Delta \vdash \mathsf{c}''_i; \mathsf{c}_2 : (w''_i, t''_i, f''_i)$ then two cases are possible:

- $\mathsf{c}'_1 = \mathsf{c}'_2$: It follows $\mathsf{c}'_1; \mathsf{c}_2 = \mathsf{c}'_2; \mathsf{c}_2$ proving the case by requirement 2.
- $\mathsf{c}'_1 \neq \mathsf{c}'_2$: The two configurations are in $\mathcal{S}$ by clause 3.

If c''_1 does not type then by induction it could be $\mathsf{c}''_1 = x := y; \mathsf{c}^*_1$ and $\Delta(x) = \mathsf{P}\ell_C\mathsf{H}$, $\Delta \vdash y : \mathsf{P}\ell_C\mathsf{L}$, $\mathsf{M}_1(y) = \mathsf{M}_2(y)$ and $\Delta \vdash \mathsf{c}^*_1 : (w'''_1, t'''_1, f'''_1)$ in which case the same holds for c''_2, i.e., $\mathsf{c}''_2 = \mathsf{c}''_1$. By command semantics $\langle \mathsf{M}_1, \mathsf{c}'_1; \mathsf{c}'_2\rangle \dot{\rightarrow}_{\mathsf{H_sH}} \langle \mathsf{M}_1, \mathsf{c}''_1; \mathsf{c}_2\rangle$ and $\langle \mathsf{M}_2, \mathsf{c}'_1; \mathsf{c}'_2\rangle \dot{\rightarrow}_{\mathsf{H_sH}} \langle \mathsf{M}_2, \mathsf{c}''_1; \mathsf{c}_2\rangle$ and the two configurations are related by clause (c), indeed $\mathsf{c}''_1; \mathsf{c}_2$ is typed by subject reduction. It could also be that $\mathsf{c}'_1 = \mathsf{c}'_2 = \mathsf{FAIL}$ which prove the case by requirement (b).

Otherwise $\langle \mathsf{M}_1, \mathsf{c}'_1\rangle \xrightarrow{\alpha}_{\mathsf{H_sH}} \langle \mathsf{M}'_1, \varepsilon\rangle$ and by induction on c'_1 $\langle \mathsf{M}_2, \mathsf{c}'_1\rangle \xrightarrow{\alpha}{}^{*}_{\mathsf{H_sH}}$

$\langle \mathsf{M}'_2, \varepsilon\rangle$ and $\mathsf{M}'_1 =^h_{\mathsf{H_sH}} \mathsf{M}'_2$. It follows $\langle \mathsf{M}_1, \mathsf{c}'_1; \mathsf{c}'_2\rangle \xrightarrow{\alpha}_{\mathsf{H_sH}} \langle \mathsf{M}'_1, \mathsf{c}'_2\rangle$ and $\langle \mathsf{M}_2, \mathsf{c}'_1; \mathsf{c}'_2\rangle \xrightarrow{\alpha}_{\mathsf{H_sH}} \langle \mathsf{M}'_2, \mathsf{c}'_2\rangle$ proving the case by clause 2.

3. $\mathsf{c}_1 = \mathsf{c}'_1; \mathsf{c}''$, $\mathsf{c}_2 = \mathsf{c}'_2; \mathsf{c}''$ with $\langle \mathsf{M}_1, \mathsf{c}'_1\rangle \mathcal{S} \langle \mathsf{M}_2, \mathsf{c}'_2\rangle$.
   If $\langle \mathsf{M}_1, \mathsf{c}'_1\rangle \xrightarrow{\alpha} \mathsf{H_sH} \langle \mathsf{M}'_1, \mathsf{c}''_1\rangle$ by induction hypothesis $\langle \mathsf{M}_2, \mathsf{c}'_2\rangle \xrightarrow{\alpha}{}^*_{\mathsf{H_sH}} \langle \mathsf{M}'_2, \mathsf{c}''_2\rangle$ and $\langle \mathsf{M}'_1, \mathsf{c}''_1\rangle \mathcal{S} \langle \mathsf{M}'_2, \mathsf{c}''_2\rangle$ from which follows $\mathsf{M}'_1 =_{\mathsf{H_sH}} \mathsf{M}'_2$ or $\langle \mathsf{M}_2, \mathsf{c}'_2\rangle \Uparrow$.
   It holds $\langle \mathsf{M}_1, \mathsf{c}'_1; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{H_sH}} \langle \mathsf{M}'_1, \mathsf{c}''_1; \mathsf{c}_2\rangle$ and $\langle \mathsf{M}_1, \mathsf{c}'_2; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{H_sH}} \langle \mathsf{M}'_1, \mathsf{c}''_1; \mathsf{c}_2\rangle$ proving the case by clause 3 or $\langle \mathsf{M}_2, \mathsf{c}'_2; \mathsf{c}_2\rangle \Uparrow$.

   If instead $\langle \mathsf{M}_1, \mathsf{c}'_1\rangle \xrightarrow{\alpha}_{\mathsf{H_sH}} \langle \mathsf{M}'_1, \varepsilon\rangle$ then by induction hypothesis $\langle \mathsf{M}_2, \mathsf{c}'_2\rangle \xrightarrow{\alpha}{}^*_{\mathsf{H_sH}} \langle \mathsf{M}'_2, \varepsilon\rangle$ with $\mathsf{M}'_1 =_{\mathsf{H_sH}} \mathsf{M}'_2$ or $\langle \mathsf{M}_2, \mathsf{c}'_2\rangle \Uparrow$. It follows $\langle \mathsf{M}_1, \mathsf{c}'_1; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{H_sH}} \langle \mathsf{M}'_1, \mathsf{c}_2\rangle$ and $\langle \mathsf{M}_2, \mathsf{c}'_2; \mathsf{c}_2\rangle \xrightarrow{\alpha}_{\mathsf{H_sH}} \langle \mathsf{M}'_2, \mathsf{c}_2\rangle$ which prove the case by relating the configurations by constraint 2 or $\langle \mathsf{M}_2, \mathsf{c}'_2; \mathsf{c}_2\rangle \Uparrow$.

Case (b): Note that FAIL is a silent diverging program thus if $\mathsf{c}_1 = \mathsf{FAIL}$ every moves will be matched by $\langle \mathsf{M}_2, \mathsf{c}_2\rangle \xrightarrow{\cdot}{}^0_{\mathsf{H_sH}} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$, otherwise if $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \xrightarrow{\alpha}_{\mathsf{LL}} \langle \mathsf{M}'_1, \mathsf{c}'_1\rangle$ it holds $\langle \mathsf{M}_2, \mathsf{FAIL}\rangle \Uparrow$.

Case (c): The configurations do not move proving the case.

Case (d): Let $\mathsf{M}_i(y) = v$, it follows $\langle \mathsf{M}_1, x := y; \mathsf{c}'\rangle \xrightarrow{x}_{\mathsf{H_sH}} \langle \mathsf{M}_1[x \mapsto v], \mathsf{c}'\rangle$ and $\langle \mathsf{M}_2, x := y; \mathsf{c}'\rangle \xrightarrow{x}_{\mathsf{H_sH}} \langle \mathsf{M}_2[x \mapsto v], \mathsf{c}'\rangle$ so from $\mathsf{M}_1 =_{\mathsf{H_sH}} \mathsf{M}_2$ it easily follows $\mathsf{M}_1[x \mapsto v] =_{\mathsf{H_sH}} \mathsf{M}_2[x \mapsto v]$. The case is then proved since c' is typed and so $\langle \mathsf{M}'_1, \mathsf{c}'\rangle \mathcal{S} \langle \mathsf{M}'_2, \mathsf{c}'\rangle$ by clause 2.

It follows that $\mathcal{S}$ is a $\mathsf{H_sH}$-TIB and the lemma is proved by clause 2. □

Theorem. (**Integrity NI by typing**)
If $\Delta \vdash \mathsf{c} : (w, t, f)$ *then* c *satisfies integrity NI.*

Proof. It must be proved that if $\Delta \vdash \mathsf{c} : (w, t, f)$ then $\forall \mathsf{M}_1 =_{\mathsf{H_sH}} \mathsf{M}_2\ \langle \mathsf{M}_1, \mathsf{c}\rangle \approx_{\mathsf{H_sH}} \langle \mathsf{M}_2, \mathsf{c}\rangle$. The claim follows directly by Lemma B.8. □

C

Type checking PIN Verification APIs - Formal Proofs

C.1 Closed key types

In some typing rules we need to recursively collect the level of the types of an encryption or MAC key, by descending into pairs and into high level encryptions. This is achieved by the following function $\mathsf{LT}_\Delta(\delta)$

$$\begin{array}{rcl} \mathsf{LT}_\Delta(\delta) & = & \{\delta\} \\ \mathsf{LT}_\Delta((\tau_1,\tau_2)) & = & \mathsf{LT}_\Delta(\tau_1) \cup \mathsf{LT}_\Delta(\tau_2) \\ \mathsf{LT}_\Delta(\mathsf{enc}_\delta\ \kappa) & = & \begin{cases} \mathsf{LT}_\Delta(\tau) & \text{if } \mathsf{K}(\kappa) = \mathsf{cK}^{\mu}_{H\mathsf{C}}(\tau)\ \kappa \\ \{LL\} & \text{otherwise} \end{cases} \end{array}$$

Notice that, if there is a cycle of encryption labels the computation of $\mathsf{LT}_\Delta(\delta)$ does not terminate. From now on we will always assume key types are acyclic. We also need to check when a type does not contains randomized encryptions.

$$\begin{array}{rcl} \mathsf{Det}(\delta) & = & \mathsf{true} \\ \mathsf{Det}((\tau_1,\tau_2)) & = & \mathsf{Det}(\tau_1) \wedge \mathsf{Det}(\tau_2) \\ \mathsf{Det}(\mathsf{enc}_\delta\ \kappa) & = & \mathsf{Det}(\tau) \wedge (\mathsf{K}(\kappa) = \mathsf{cK}_\delta(\tau)\ \kappa) \end{array}$$

Based on this function, we define some useful sets and predicates:

$$\begin{array}{rcl} \mathsf{IRs}(\tau) & = & \{\mathsf{D}\ |\delta_C[\mathsf{D}] \in \mathsf{LT}_\Delta(\tau)\} \\ \mathsf{Dep}(\tau) & = & \{\mathsf{D} \mid \mathsf{D} \in \tilde{\mathsf{D}}, \delta_C[\mathsf{D}' : \tilde{\mathsf{D}}] \in \mathsf{LT}_\Delta(\tau)\} \\ \mathsf{DD}(\tau) & \textit{iff} & \delta_C\delta_I \in \mathsf{LT}_\Delta(\tau) \text{ implies } \mathsf{C} \not\sqsubseteq \delta_I \\ \mathsf{Closed}(\tau) & \textit{iff} & \mathsf{Dep}(\tau) \subseteq \mathsf{IRs}(\tau) \\ \mathsf{CloseDD}(\tau) & \textit{iff} & \mathsf{Closed}(\tau) \wedge \mathsf{DD}(\tau) \\ \mathsf{CloseDD}^{\mathsf{det}}(\tau) & \textit{iff} & \mathsf{Closed}(\tau) \wedge \mathsf{DD}(\tau) \wedge \mathsf{Det}(\tau) \end{array}$$

respectively, the set $\mathsf{IRs}(\tau)$ of the integrity representatives in τ; the domains $\mathsf{Dep}(\tau)$ on which types of τ depend; the fact, written $\mathsf{DD}(\tau)$, that all the integrity levels are below C, i.e., are all of the form $[\mathsf{D} : \tilde{\mathsf{D}}]$; a predicate $\mathsf{Closed}(\tau)$ checking the closure of dependent domains of τ with respect to integrity representatives. For example, $([\mathsf{D}], [\mathsf{D}' : \mathsf{D}])$ is closed while $([\mathsf{D}], [\mathsf{D}' : \mathsf{D}''])$ is not; $\mathsf{CloseDD}(\tau)$ and $\mathsf{CloseDD}^{\mathsf{det}}(\tau)$, additionally requiring $\mathsf{DD}(\tau)$ and $\mathsf{Det}(\tau)$.

C.2 Memory well-formedness

Well-formedness for values is given in Table C.1. Θ is an injective mapping from key values to their types. An integrity representative assumes the constant value determined by function $g(\mathsf{D})$ where D is the domain of the representative (v-ir). A value which depends on a set of domains $\{\mathsf{D_1}, \ldots \mathsf{D_m}\} = \tilde{\mathsf{D}}'$ and belongs to the dependent domain $\mathsf{D} : \tilde{\mathsf{D}}'$ is mapped by a function $f_{\mathsf{D}:\mathsf{D_1},\ldots\mathsf{D_m}}$ taking the values of each of the integrity representative of the domains contained in $\tilde{\mathsf{D}}'$ and returning the value of the resulting domain (v-dd). The ordering of the D_i is important here to keep the parameters in the correct sequence. While g can change depending on the

integrity context, we assume f is fixed a-priory for a memory and never changes while checking well-formedness. This amounts to fix the dependency among dependent domains without however fixing a-priori their values, which is instead determined once g is also fixed. We will always assume that f and g never returns keys or confounders. Finally, while g is partial, we always assume f is defined for each dependent domain $\mathsf{D} : \tilde{\mathsf{D}}'$.

MACs and values encrypted with trusted keys that should always contain 'high-integrity' values of type τ (v-enc) and (v-mac), and a confounder in case of randomized encryption (v-enc-R). In fact, those values can only be generated by trusted, well-typed, programs. These rules also instantiate all the integrity representatives corresponding to domains appearing in τ via a function g; this is to avoid that the same representative assumes different values in the same high integrity context.

All the other rules work as expected. Notice that ciphertexts obtained through high level keys can be given confidentiality level L independently of the confidentiality level of the plaintext, reflecting the possibility of generating low-confidential ciphertexts containing high-confidentiality plaintexts. As well-formedness only aims at checking the integrity of values, we do not constraint the value of δ_C as done, instead, when typing expressions.

It can be easily proved that any value in which only low-level keys appear is well-formed and can be given any possible low-integrity type, meaning that we do no restrict attacker ability to manipulate low-integrity memories apart from subvalues generated via high level keys. (Recall that types at level $\delta_C L$ are all implicitly considered the same.)

Intuitively, a memory M is well-formed if all its variables stores well-formed values. For dependent domains we fix a-priori all the representatives via a specific g. This is to account for run-time situations in which a MAC check has been performed. Let g_τ denote ϵ when $H \sqsubseteq \mathcal{L}_I(\tau)$ and g otherwise.

Definition C.1 (Memory well-formedness)**.** *Let $\Theta : k \mapsto \mathsf{K}_\delta(\tau)\ \kappa$ be a mapping from keys to key-types such that $k \in \mathcal{K}_{\mathcal{L}(\Theta(k))}$, and injective for high level keys, i.e., $k, k' \in \mathcal{K}_{H\mathsf{C}}$ implies $\Theta(k) \neq \Theta(k')$. A memory M is well-formed with respect to Δ and g ($\neq \epsilon$), written $\Delta \vdash^f_g \mathsf{M}$, if the followings hold*

(i) $\Delta(x) = \tau$ *and* $\mathsf{M}(x) = v$ *imply* $\Theta \vdash^f_{g_\tau} v : \tau$

(ii) If values $\{\!|v, r|\!\}_k$, $\{\!|v', r'|\!\}_k$ *occur in* M*, then* $\Theta(k) = \mathsf{cK}^R_{H\mathsf{C}}(\tau)\ \kappa$ *and* $v \neq v'$ *imply* $r \neq r'$*.*

Proofs

This appendix contains all the proofs of the results. In some cases we also add a few technical details that have been skipped in Chapter 5 to easy the presentation.

Table C.1 Values well-formedness

Note: In rules (v-enc), (v-enc-r), (v-mac), $g = g'$ when $g \neq \epsilon$ or $\mathcal{L}_I(\delta) = L$

$$\text{(v-name)}\ \frac{n \notin \mathcal{K} \quad \delta_I \neq [\mathsf{D} : \tilde{\mathsf{D}}']}{\Theta \vdash_g^f n : \delta_C \delta_I} \qquad \text{(v-key)}\ \frac{\Theta(k) = \tau}{\Theta \vdash_g^f k : \tau} \qquad \text{(v-sub)}\ \frac{\Theta \vdash_g^f v : \tau' \quad \tau' \leq \tau}{\Theta \vdash_g^f v : \tau}$$

$$\text{(v-pair)}\ \frac{\Theta \vdash_g^f v_1 : \tau_1 \quad \Theta \vdash_g^f v_2 : \tau_2}{\Theta \vdash_g^f (v_1, v_2) : (\tau_1, \tau_2)}$$

$$\text{(v-ir)}\ \frac{g(\mathsf{D})\downarrow \quad n = g(\mathsf{D})}{\Theta \vdash_g^f n : \delta_C[\mathsf{D}]} \qquad \text{(v-dd)}\ \frac{g(\mathsf{D}_i)\downarrow \quad n = f_{\mathsf{D}:\mathsf{D}_1,\ldots\mathsf{D}_m}(g(\mathsf{D}_1), \ldots, g(\mathsf{D}_m))}{\Theta \vdash_g^f n : \delta_C[\mathsf{D} : \{\mathsf{D}_1, \ldots, \mathsf{D}_m\}]}$$

$$\text{(v-mac)}\ \frac{\Theta(k) = \mathsf{mK}_\delta(\tau) \quad \Theta \vdash_{g'}^f v : \tau}{\Theta \vdash_g^f \langle v \rangle_k : \delta_C L}$$

$$\text{(v-enc)}\ \frac{\Theta(k) = \mathsf{cK}_\delta(\tau)\ \kappa \quad \Theta \vdash_{g'}^f v : \tau}{\Theta \vdash_g^f \{\!|v|\!\}_k : \mathsf{enc}_{\delta_C \mathcal{L}_I(\delta) \sqcup \mathcal{L}_I(\tau)}\ \kappa} \qquad \text{(v-enc-r)}\ \frac{\Theta(k) = \mathsf{cK}_{H\mathsf{C}}^R(\tau)\ \kappa \quad \Theta \vdash_{g'}^f v : \tau}{\Theta \vdash_g^f \{\!|v, r|\!\}_k : \mathsf{enc}_{\delta_C \mathsf{C} \sqcup \mathcal{L}_I(\tau)}\ \kappa}$$

Preliminaries

We first formalize various assumption that have been given (a few of them omitted for the sake of simplicity).

Assumption C.1 (Types)**.** *We always assume that:*

1. *In* (τ_1, τ_2) *we have* $\mathcal{L}_C(\tau_1) = \mathcal{L}_C(\tau_2)$*. If* $\mathsf{H} \sqsubseteq \mathcal{L}_I((\tau_1, \tau_2))$ *then also* $\mathcal{L}_I(\tau_1) = \mathcal{L}_I(\tau_2)$*;*
2. *In* $\mathsf{cK}_\delta^\mu(\tau)\ \kappa$ *and* $\mathsf{cK}_{\delta'}^{\mu'}(\tau')\ \kappa'$*,* $\kappa = \kappa'$ *implies* $\mathsf{cK}_\delta^\mu(\tau)\ \kappa = \mathsf{cK}_{\delta'}^{\mu'}(\tau')\ \kappa'$*.*
3. *In* $\mathsf{K}_\delta(\tau)\ \kappa$ *and* (τ_1, τ_2) *types* $\tau, \tau_1, \tau_2 \neq \mathsf{K}_{\mathsf{HC}}(\tau)\ \kappa$
4. *For types referring to the same key label as* $\mathsf{enc}_\delta\ \kappa$ *and* $\mathsf{cK}_{\delta'}^\mu(\tau)\ \kappa$*, we always assume* $\mathcal{L}_I(\delta) = \mathcal{L}_I(\delta') \sqcup \mathcal{L}_I(\tau)$

Intuitively, (1) states that, in pairs, the two types are required to have the same confidentiality level and, for integrity levels at or above H, even the same integrity level; (2) formalizes that κ is a label that uniquely identifies one key type; (3) states that we disallow the encryption (and the MAC) of high level keys; (4) requires that the integrity level of encryption is coherent with the one of the key (notice that this is not limiting, as if the integrity level is not the right one it is impossible to type encryptions and decryptions).

Table C.2 Command Semantics

$$[skip]\ \langle \mathsf{M}, \mathsf{skip}\rangle \Rightarrow \mathsf{M} \qquad [assign]\ \frac{e \downarrow^{\mathsf{M}} v}{\langle \mathsf{M}, x := e\rangle \Rightarrow \mathsf{M}[x \mapsto v]} \qquad [seq]\ \frac{\langle \mathsf{M}, \mathsf{c}_1\rangle \Rightarrow \mathsf{M}' \quad \langle \mathsf{M}', \mathsf{c}_2\rangle \Rightarrow \mathsf{M}''}{\langle \mathsf{M}, \mathsf{c}_1; \mathsf{c}_2\rangle \Rightarrow \mathsf{M}''}$$

$$[ift]\ \frac{b \downarrow^{\mathsf{M}} \mathsf{true} \quad \langle \mathsf{M}, \mathsf{c}_1\rangle \Rightarrow \mathsf{M}'}{\langle \mathsf{M}, \mathsf{if}\ b\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\rangle \Rightarrow \mathsf{M}'} \qquad [iff]\ \frac{b \downarrow^{\mathsf{M}} \mathsf{false}, \bot \quad \langle \mathsf{M}, \mathsf{c}_2\rangle \Rightarrow \mathsf{M}'}{\langle \mathsf{M}, \mathsf{if}\ b\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\rangle \Rightarrow \mathsf{M}'}$$

$$[whilet]\ \frac{b \downarrow^{\mathsf{M}} \mathsf{true} \quad \langle \mathsf{M}, \mathsf{c}\rangle \Rightarrow \mathsf{M}' \quad \langle \mathsf{M}', \mathsf{while}\ b\ \mathsf{do}\ \mathsf{c}\rangle \Rightarrow \mathsf{M}''}{\langle \mathsf{M}, \mathsf{while}\ b\ \mathsf{do}\ \mathsf{c}\rangle \Rightarrow \mathsf{M}''} \qquad [whilef]\ \frac{b \downarrow^{\mathsf{M}} \mathsf{false}, \bot}{\langle \mathsf{M}, \mathsf{while}\ b\ \mathsf{do}\ \mathsf{c}\rangle \Rightarrow \mathsf{M}}$$

Table C.3 Expression Semantics

NOTE: We let $e \downarrow^{\mathsf{M}} v$, $e_1 \downarrow^{\mathsf{M}} v_1$, $e_2 \downarrow^{\mathsf{M}} v_2$ and $x \downarrow^{\mathsf{M}} k$.

$$\begin{array}{rll}
\mathsf{new}() \downarrow^{\mathsf{M}} & r & r \leftarrow C \\
\mathsf{enc}_x(e) \downarrow^{\mathsf{M}} & \{\!|v|\!\}_k & \\
\mathsf{dec}_x(e') \downarrow^{\mathsf{M}} & v & \text{if } e' \downarrow^{\mathsf{M}} \{\!|v|\!\}_k \\
\mathsf{mac}_x(e) \downarrow^{\mathsf{M}} & \langle v\rangle_k & \\
\mathsf{pair}(e_1, e_2) \downarrow^{\mathsf{M}} & (v_1, v_2) & \\
\mathsf{fst}(e') \downarrow^{\mathsf{M}} & v_1 & \text{if } e' \downarrow^{\mathsf{M}} (v_1, v_2) \\
\mathsf{snd}(e') \downarrow^{\mathsf{M}} & v_2 & \text{if } e' \downarrow^{\mathsf{M}} (v_1, v_2)
\end{array}$$

All remaining cases, e.g., decryption with an incorrect key, evaluate to the special value $\bot$ making the expressions total.

Assumption C.2 (Expressions)**.** *For generic expressions it always holds*

1. $e_1\ \mathsf{op}\ e_2 \downarrow^{\mathsf{M}} n$ *with* $n \notin \mathcal{K} \cup C$
2. $e_1\ \mathsf{op}\ e_2 \not\downarrow^{\mathsf{M}} \bot$ *then either* op *is '='* *(i.e., the equality check) or* $e_i \downarrow^{\mathsf{M}} n_i$ *with* $n_i \notin \mathcal{K} \cup C$
3. *when the observation level* ℓ *is* HL *(top) the equality check is defined as* $e_1 = e_2 \downarrow^{\mathsf{M}} \mathsf{true}$ *iff* $e_i \downarrow^{\mathsf{M}} v_i$ *and* $\mathsf{p}_{\mathsf{HL}}(v_1) = \mathsf{p}_{\mathsf{HL}}(v_2)$ *(instead of* $v_1 = v_2$*);* $e_1 = e_2 \downarrow^{\mathsf{M}}$ false *otherwise.*

Intuitively, (1) generic expressions never generate keys or confounders; (2) they fail when applied to names which are not atomic and different from keys/confounders, unless the operation is the equality check; (3) equality check $e_1 = e_2$ is evaluated as expected (i.e., syntactic equality of the resulting values) apart when the observation level is HL; in this particular case we require it is consistent with equality of patterns at the actual observation level HL; this assumption is needed to prove noninterference when $\ell = \mathsf{HL}$. We might even remove this assumption and restrict noninterference result (theorem C.2) to LL and HH which are anyway the

levels of interest since they respectively represent the attacker and the trusted users. Robustness, in fact, is based on these two levels and does not depend on assumption C.2(3). Technically, under this assumption, expression evaluation, and consequently program execution, depend on ℓ; to keep the notation as much simple as possible, we however prefer to omit to decorate with ℓ program semantics. When not explicitly specified, we will always intend that programs are executed below HL, i.e., with the expected semantics for equality check ($e_1 = e_2 \downarrow^{\mathsf{M}} \mathsf{true}$ iff $e_i \downarrow^{\mathsf{M}} v_i$ and $v_1 = v_2$, and $e_1 = e_2 \downarrow^{\mathsf{M}} \mathsf{false}$ otherwise.)

Assumption C.3 (Confounders). *Confounders r occur in memories only as $\{\!|v, r|\!\}_k$ with $\Theta(k) = \mathsf{cK}^{R}_{\mathsf{HC}}(\tau)\ \kappa$.*

This assumption comes from the fact that (typed) programs only use confounders in the expected way, i.e., inside randomized ciphertexts. It is trivial to show that the assumption is preserved when running (typed) programs: it is enough to observe that, by assumption C.2(1) above, the only expression which returns confounders is new and it is only (implicitly) typed by rule (enc-r), while randomized decryption always 'throws away' the confounder.

Memory well-formedness

We now give some results on value well-formedness. To do so, we first need to formalize what is a subvalue. In particular, we formalize a notion of subvalues at level δ, i.e., subvalues observable by only knowing keys at or below δ. To simplify a bit the notation, we will often write v to denote the set $\{v\}$ only containing value v. Recall that we write $\mathcal{K}_{\sqsubseteq\delta}$ to denote the set of keys at or below δ, i.e., $\cup_{\delta'\sqsubseteq\delta}\mathcal{K}_{\delta'}$.

Definition C.2. *The set of δ-subvalues of a value v, written $\mathsf{Sub}_\delta(v)$ is recursively defined as:*

$$\begin{array}{rcl} \mathsf{Sub}_\delta(n) & = & n \\ \mathsf{Sub}_\delta((v_1, v_2)) & = & (v_1, v_2) \cup \mathsf{Sub}_\delta(v_1) \cup \mathsf{Sub}_\delta(v_2) \\ \mathsf{Sub}_\delta(\langle v\rangle_k) & = & \langle v\rangle_k \\ \mathsf{Sub}_\delta(\{\!|v|\!\}_k) & = & \left\{ \begin{array}{ll} \{\!|v|\!\}_k & \text{if } k \notin \mathcal{K}_{\sqsubseteq\delta} \\ \{\!|v|\!\}_k \cup \mathsf{Sub}_\delta(v) & \text{otherwise} \end{array} \right. \end{array}$$

We write $\mathsf{Sub}(v)$ to denote $\mathsf{Sub}_{\mathsf{HL}}(v)$, i.e., the set of all the subvalues of v.

Notice that keys used to create ciphertexts and MACs are not considered as subvalues. We actually assume it is never possible to deduce a key from a ciphertext or MAC. Those keys, when needed, can be extracted from a value by just looking at all the ciphertexts/MACs appearing as a subvalue. Formally

$$\mathcal{K}(v) = \{k \mid \{\!|v'|\!\}_k \in \mathsf{Sub}(v) \vee \langle v'\rangle_k \in \mathsf{Sub}(v)\}$$

Next lemma states that any value in which only low-level keys appear is well-formed and can be given any possible low-integrity type, meaning that we do no restrict attacker ability to manipulate low-integrity memories apart from subvalues generated via high level keys.

Lemma C.1. *If* $\mathcal{K}(v) \subseteq \mathcal{K}_{\mathsf{LL}}$ *and* $\mathsf{Sub}(v) \cap \mathcal{K}_{\mathsf{HC}} = \emptyset$ *then* $\Theta \vdash^f v : \mathsf{LL}$.

Proof. It is sufficient to consider Θ *such that* $\Theta(k) = \mathsf{LL}$ *for all* $k \in \mathcal{K}(v)$*. We now proceed by induction on the structure of* v*.*

Base case:

n

By (v-name) we have that all names n*, except keys, are such that* $\Theta \vdash^f n : \mathsf{LL}$*. Since* $\mathsf{Sub}(v) \cap \mathcal{K}_{\mathsf{HC}} = \emptyset$ *we have that* $n \notin \mathcal{K}_{\mathsf{HC}}$*. When* $n \in \mathcal{K}_{\mathsf{LL}}$*, by* $\Theta(k) = \mathsf{LL}$ *for all* $k \in \mathcal{K}(v)$ *and (v-key) we obtain* $\Theta \vdash^f k : \mathsf{LL}$*.*

Inductive case:

(v_1, v_2)

Since $\mathsf{Sub}(v_1), \mathsf{Sub}(v_2) \subseteq \mathsf{Sub}(v)$ *we also have* $\mathcal{K}(v_i) \subseteq \mathcal{K}(v) \subseteq \mathcal{K}_{\mathsf{LL}}$ *and* $\mathsf{Sub}(v_i) \cap \mathcal{K}_{\mathsf{HC}} \subseteq \mathsf{Sub}(v) \cap \mathcal{K}_{\mathsf{HC}} = \emptyset$*. Thus by inductive hypothesis we know that* $\Theta \vdash^f v_i : \mathsf{LL}$*. By applying (v-pair) we directly obtain the thesis since* $(\mathsf{LL}, \mathsf{LL}) \equiv \mathsf{LL}$*.*

$\{\!|v'|\!\}_k$

As above, by induction we know that $\Theta \vdash^f v' : \mathsf{LL}$ *and by* $\mathcal{K}(v) \subseteq \mathcal{K}_{\mathsf{LL}}$ *we know that* $\Theta(k) = \mathsf{LL}$*. Recall we identify* LL *with every low-integrity/confidentiality type as, in particular,* $\mathsf{cK}_{\mathsf{LL}}(\mathsf{LL})\ \kappa \equiv \mathsf{LL}$*. By (v-enc) we thus get* $\Theta \vdash^f \{\!|v'|\!\}_k : \mathsf{enc}_{\mathsf{LL}}\ \kappa \equiv \mathsf{LL}$*.*

$\langle v' \rangle_k$

We proceed exactly as in the previous case. □

□

We now prove that in any well-formed value, which is not itself a high level key, high level keys k never appear as subvalues, even if they can appear as keys as, e.g., in $\{\!|v|\!\}_k$.

Lemma C.2. *Let* $\Theta \vdash^f_g v : \tau$*. Then* $v \notin \mathcal{K}_{\mathsf{HC}}$ *implies* $\mathsf{Sub}(v) \cap \mathcal{K}_{\mathsf{HC}} = \emptyset$ *.*

Proof. By induction on the structure of v*:*

Base case:

n, $\langle v \rangle_k$

Trivial since $\mathsf{Sub}(v) = \{v\}$*. By* $v \notin \mathcal{K}_{\mathsf{HC}}$ *we directly have that* $\mathsf{Sub}(v) \cap \mathcal{K}_{\mathsf{HC}} = \emptyset$*.*

Inductive cases:

(v_1, v_2)

Let $\Theta \vdash_g^f (v_1, v_2) : \tau$*. Since pairs are not in the subtyping relation, the judgement comes from (v-pair) implying that* $\tau = (\tau_1, \tau_2)$*,* $\Theta \vdash_g^f v_1 : \tau_1$ *and* $\Theta \vdash_g^f v_2 : \tau_2$*. By assumption C.1(3), we have that* $\tau_i \neq \mathsf{cK}^{\mu}_{\mathsf{HC}}(\tau)\ \kappa$*. Notice that if* $v_i \in \mathcal{K}_{\mathsf{HC}}$ *it would be necessarily be typed as* $\mathsf{cK}^{\mu}_{\mathsf{HC}}(\tau)\ \kappa$ *since the only possible rule for high keys is (v-key) and high level key-type is not in the subtype relation. So* $v_i \notin \mathcal{K}_{\mathsf{HC}}$ *and, by induction,* $\mathsf{Sub}(v_i) \cap \mathcal{K}_{\mathsf{HC}} = \emptyset$*. Thus,*

$$\begin{aligned}&\mathsf{Sub}((v_1, v_2)) \cap \mathcal{K}_{\mathsf{HC}} = \\ &\quad = (\mathsf{Sub}(v_1) \cup \mathsf{Sub}(v_2) \cup (v_1, v_2)) \cap \mathcal{K}_{\mathsf{HC}} = \emptyset\end{aligned}$$

$\{\!|v'|\!\}_k$

Let $\Theta \vdash_g^f \{\!|v|\!\}_k : \tau$*. The judgement might derive from (v-sub) and one among (v-enc) and (v-enc-r), meaning* $\Theta(k) = \mathsf{cK}^{\mu}_{\delta}(\tau')\ \kappa$ *and* $\Theta \vdash_{g'}^f v' : \tau'$*. By assumption C.1(3), we have that* $\tau' \neq \mathsf{cK}^{\mu}_{\mathsf{HC}}(\tau)\ \kappa$*. As for the above case, this means that* $v' \notin \mathcal{K}_{HC}$*. By induction we thus get* $\mathsf{Sub}(v_i) \cap \mathcal{K}_{\mathsf{HC}} = \emptyset$*. Thus,*

$$\begin{aligned}&\mathsf{Sub}(\{\!|v'|\!\}_k) \cap \mathcal{K}_{\mathsf{HC}} \subseteq \\ &\quad \subseteq (\mathsf{Sub}(v') \cup \{\!|v'|\!\}_k) \cap \mathcal{K}_{\mathsf{HC}} = \emptyset\end{aligned}$$

□

□

Intuitively, judgments $\Theta \vdash^f v : \mathsf{LL}$, $\Theta \vdash^f v : \tau$ and $\Theta \vdash_g^f v : \tau$ have increasing discriminating power, as they are used to check the well-formedness of values at increasing integrity levels. We formally show that $\Theta \vdash_g^f v : \tau$ always implies $\Theta \vdash^f v : \mathsf{LL}$. Moreover, when $\mathcal{L}_I(\tau) = \mathsf{H}$, it also implies $\Theta \vdash^f v : \tau$. This does not hold for high level key values whose integrity cannot be reduced by subtyping. For example, a high level key k can be typed as $\Theta \vdash_g^f k : \tau$ but it can never be typed $\Theta \vdash^f k : \delta_C\mathsf{L}$.

Lemma C.3. *Let* $v \notin \mathcal{K}_{\mathsf{HC}}$*. Then*

(i) $\Theta \vdash_g^f v : \tau$ *implies* $\Theta \vdash^f v : \mathsf{LL}$*;*

(ii) $\Theta \vdash_g^f v : \tau$ *and* $\mathcal{L}_I(\tau) = \mathsf{H}$ *imply* $\Theta \vdash^f v : \tau$*.*

Proof. By induction on the structure of v.

Base case:

n

We have that all names n, except keys, are such that $\Theta \vdash^f n : \mathsf{LL}$, thanks to (v-name). Moreover, the condition $k \in \mathcal{K}_{\mathcal{L}(\Theta(k))}$ on Θ ensures that low keys $k \in \mathcal{K}_{\mathsf{LL}}$ are such that $\Theta(k) = \tau' \equiv \mathsf{LL}$ and by (v-key) $\Theta \vdash^f k : \mathsf{LL}$. By hypothesis $n \notin \mathcal{K}_{\mathsf{HC}}$.

Inductive cases:

(v_1, v_2)
: Let $\Theta \vdash^f_g (v_1, v_2) : \tau$. Since pairs are not in the subtyping relation, the judgement comes from (v-pair) implying that $\tau = (\tau_1, \tau_2)$, $\Theta \vdash^f_g v_i : \tau_i$. By lemma C.2 we know that $v_i \notin \mathcal{K}_{\mathsf{HC}}$. Thus, inductive hypothesis applies and we obtain $\Theta \vdash^f v_i : \mathsf{LL}$ and, by (v-pair), $\Theta \vdash^f (v_1, v_2) : (\mathsf{LL}, \mathsf{LL}) \equiv \mathsf{LL}$.

$\{\!|v'|\!\}_k$
: Let $\Theta \vdash^f_g \{\!|v'|\!\}_k : \tau$. The judgment might derive from (v-sub) and one among (v-enc) and (v-enc-r), meaning $\Theta(k) = \mathsf{cK}^{\mu}_{\delta}(\tau')\ \kappa$ and $\Theta \vdash^f_{g'} v' : \tau'$. By lemma C.2 we know that $v' \notin \mathcal{K}_{\mathsf{HC}}$. Thus, inductive hypothesis applies and we obtain $\Theta \vdash^f v' : \mathsf{LL}$. By reapplying the very same rule (v-enc) or (v-enc-r) with empty g and $\delta_C = \mathsf{L}$ we obtain $\Theta \vdash^f \{\!|v'|\!\}_k : \mathsf{enc}_{\mathsf{LL}}\ \kappa \equiv \mathsf{LL}$.

$\langle v' \rangle_k$
: This case is identical to the previous one.

(ii) Again, by induction on the structure of v:

Base case:

n
: By rule (v-name), we have that all names n, except keys, are such that $\Theta \vdash^f n : \tau'$, with $\mathcal{L}_I(\tau') = \mathsf{H}$. When $\mathcal{L}_I(\tau') = \mathsf{H}$ we also obtain $\Theta \vdash^f n : \tau$, by taking $\tau' = \tau$. The condition $k \in \mathcal{K}_{\mathcal{L}(\Theta(k))}$ on Θ ensures that low keys $k \in \mathcal{K}_{\mathsf{LL}}$ are such that $\Theta(k) = \tau' \equiv \mathsf{LL}$ and by (v-key) $\Theta \vdash^f k : \mathsf{LL}$. Thus, low keys can never be typed at level $\mathcal{L}_I(\tau) \sqsubseteq_I \mathsf{H}$, meaning the $n \notin \mathcal{K}_{\mathsf{LL}}$. Moreover, by hypothesis $n \notin \mathcal{K}_{\mathsf{HC}}$. Thus n cannot be a key.

Inductive cases:

(v_1, v_2)
: Let $\Theta \vdash^f_g (v_1, v_2) : \tau$ with $\mathcal{L}_I(\tau) = \mathsf{H}$. Since pairs are not in the subtyping relation, the judgement comes from (v-pair) implying that $\tau = (\tau_1, \tau_2)$, $\Theta \vdash^f_g v_i : \tau_i$. By assumption C.1(1), we also have $\mathcal{L}_I(\tau_1), \mathcal{L}_I(\tau_2) = \mathsf{H}$. Thus, induction gives $\Theta \vdash^f v_i : \tau_i$, and from (v-pair) with empty g we directly obtain $\Theta \vdash^f (v_1, v_2) : (\tau_1, \tau_2)$.

$\{\!|v_1|\!\}_k$
: Let $\Theta \vdash^f_g \{\!|v_1|\!\}_k : \tau$ with $\mathcal{L}_I(\tau) = \mathsf{H}$. The judgment might derive from (v-sub) and one among (v-enc) and (v-enc-r), meaning $\Theta(k) = \mathsf{cK}^{\mu}_{\delta}(\tau_1)\ \kappa$ and $\Theta \vdash^f_{g'} v_1 : \tau_1$. By reapplying the very same rules with $g = \epsilon$ we thus obtain $\Theta \vdash^f \{\!|v_1|\!\}_k : \tau$.

$\langle v \rangle_k$
MACs are never typed at high integrity. □

□

Corollary C.1. *Let* $\mathsf{H} \sqsubseteq_I \mathcal{L}_I(\tau)$. *Then,* $\Theta \vdash^f_g v : \tau$ *if and only if* $\Theta \vdash^f v : \tau$.

Proof. Notice that $\Theta \vdash^f v : \tau$ *denotes* $\Theta \vdash^f_g v : \tau$ *with an empty* g. *So we only need to prove that* $\Theta \vdash^f_g v : \tau$, *with* $\mathsf{H} \sqsubseteq_I \mathcal{L}_I(\tau)$ *and non-empty* g, *implies* $\Theta \vdash^f v : \tau$. *By* $\mathsf{H} \sqsubseteq_I \mathcal{L}_I(\tau)$ *we have two easy cases: when* $\mathcal{L}_I(\tau) = \mathsf{L}$, *by lemma C.3(i) we obtain that* $\Theta \vdash^f v : \mathsf{LL} \equiv \tau$. *When* $\mathcal{L}_I(\tau) = \mathsf{H}$, *by lemma C.3(ii) we directly obtain* $\Theta \vdash^f v : \tau$. □ □

Next lemma states that value subtyping works as expected even in the presence of g_τ. Recall that g_τ denotes ϵ when $\mathsf{H} \sqsubseteq \mathcal{L}_I(\tau)$ and g otherwise.

Lemma C.4. $\Theta \vdash^f_{g_{\tau'}} v : \tau'$ *and* $\tau' \leq \tau$ *then* $\Theta \vdash^f_{g_\tau} v : \tau$.

Proof. By (v-sub) we have that $\Theta \vdash^f_{g_{\tau'}} v : \tau'$ *and* $\tau' \leq \tau$ *imply* $\Theta \vdash^f_{g_{\tau'}} v : \tau$. *Now we have three cases. If* $\mathsf{H} \not\sqsubseteq \mathcal{L}_I(\tau)$ *then also* $\mathsf{H} \not\sqsubseteq \mathcal{L}_I(\tau')$ *and* $g_\tau = g_{\tau'} = g$ *which concludes the proof. If* $\mathsf{H} \sqsubseteq \mathcal{L}_I(\tau)$ *and* $\mathsf{H} \sqsubseteq \mathcal{L}_I(\tau')$ *then* $g_\tau = g_{\tau'} = \epsilon$ *and even this case is concluded. The only interesting case is when* $\mathsf{H} \sqsubseteq \mathcal{L}_I(\tau)$ *and* $\mathsf{H} \not\sqsubseteq \mathcal{L}_I(\tau')$ *which implies* $g_\tau = \epsilon$ *and* $g_{\tau'} = g$. *Intuitively, this happens when subtyping 'crosses' the* H *integrity boundary. By corollary C.1(ii) we obtain that* $\Theta \vdash^f_g v : \tau$ *implies* $\Theta \vdash^f v : \tau$ *giving the thesis.* □ □

Key-safety of well-formed memories We now show that memory well-formedness implies key-safety: when observing a memory at δ we can only learn keys of level δ or below. We first need to formalize the set of values deducible from a set of values V by only using deducible keys, written $\mathsf{knows}(V)$. It is defined as the least set such that

(1) $V \subseteq \mathsf{knows}(V)$;

(2) if $(v_1, v_2) \in \mathsf{knows}(V)$ then $v_1, v_2 \in \mathsf{knows}(V)$;

(3) if $k, \{\!|v|\!\}_k \in \mathsf{knows}(V)$ then $v \in \mathsf{knows}(V)$.

What is deducible at δ from a memory M, written $\mathsf{knows}_\delta(\mathsf{M})$, can be now expressed as:

$$\mathsf{knows}_\delta(\mathsf{M}) = \mathsf{knows}(img(\mathsf{M}|_\delta))$$

where $img(\mathsf{M}|_\delta) = \{\mathsf{M}|_\delta(x) \;.\; x \in dom(\mathsf{M}|_\delta)\}$. We can now define memory key-safety as follows:

Definition C.3 (Key-safety of memories)**.** *A memory* M *is* key-safe *iff* $\mathsf{knows}_\delta(\mathsf{M}) \cap \mathcal{K} \subseteq \mathcal{K}_{\sqsubseteq\delta}$.

In order to prove that well-formed memories are key-safe we need a few lemmas. First, it is trivial to show that $\mathsf{knows}_\delta(\mathsf{M})$ only contains subvalues of $img(\mathsf{M}|_\delta)$, written $\mathsf{Sub}(\mathsf{M}|_\delta) = \cup_{v \in img(\mathsf{M}|_\delta)} \mathsf{Sub}(v)$.

Lemma C.5. $\mathsf{knows}_\delta(\mathsf{M}) \subseteq \mathsf{Sub}(\mathsf{M}|_\delta)$.

Proof. Let $V = img(\mathsf{M}|_\delta)$. We proceed by induction on $|\mathsf{knows}_\delta(\mathsf{M})|$.

Base case: $|\mathsf{knows}_\delta(\mathsf{M})| = |V|$. We have $\mathsf{knows}_\delta(\mathsf{M}) = V \subseteq \cup_{v \in V} \mathsf{Sub}(v) = \mathsf{Sub}(\mathsf{M}|_\delta)$.

Inductive case: $|\mathsf{knows}_\delta(\mathsf{M})| > |V|$. We pick $v \in \mathsf{knows}_\delta(\mathsf{M})$. If $v \in V$ we are done as $V \subseteq \cup_{v \in V} \mathsf{Sub}(v) = \mathsf{Sub}(\mathsf{M}|_\delta)$. If, instead, $v \notin V$ we remove it from $\mathsf{knows}_\delta(\mathsf{M})$, obtaining $V' = \mathsf{knows}_\delta(\mathsf{M}) \setminus \{v\}$. Doing so we necessarily break condition (2) or (3), otherwise we would have $\mathsf{knows}_\delta(\mathsf{M})$ is not the least set satisfying those constraints (since V' is included in it). We now have different subcases depending on which condition we break:

(2) We have either $(v, v') \in V'$ or $(v', v) \in V'$. Without loss of generality, assume $(v, v') \in V'$. By inductive hypothesis we have that $(v, v') \in \mathsf{Sub}(v'')$ for at least one $v'' \in V$ which implies $v \in \mathsf{Sub}(v'') \subseteq \mathsf{Sub}(\mathsf{M}|_\delta)$ giving the thesis.

(3) We have that $k, \{\!|v|\!\}_k$ belong to V'. By inductive hypothesis we have that $\{\!|v|\!\}_k \in \mathsf{Sub}(v'')$ for at least one $v'' \in V$ which implies $v \in \mathsf{Sub}(v'') \subseteq \mathsf{Sub}(\mathsf{M}|_\delta)$ giving the thesis. □

□

Next lemma states that values typed at a certain integrity level contains subvalues which are typed at the same integrity level or below (i.e., higher since integrity levels are countervariant).

Lemma C.6. *Let $\Theta \vdash^f_g v : \tau$ and let $v' \in \mathsf{Sub}(v)$. Then, $\Theta \vdash^f_{g'} v' : \tau'$ and $\mathcal{L}_I(\tau') \sqsubseteq_I \mathcal{L}_I(\tau)$.*

Proof. Observe that if $\Theta \vdash^f_g v : \tau$ then $\Theta \vdash^f_{g'} v' : \tau'$ occurs in the proof of the former judgement since all subvalues are recursively judged.

Thus the only fact we need to prove is that $\mathcal{L}_I(\tau') \sqsubseteq_I \mathcal{L}_I(\tau)$. It is sufficient to proceed by induction on the length of the derivation from $\Theta \vdash^f_{g'} v' : \tau'$ to $\Theta \vdash^f_g v : \tau$. For length 0 we have that $v = v'$ which trivially gives the thesis. For length $i > 0$, by an inspection of the rules, we can see that, when v is not atomic, judgement $\Theta \vdash^f_g v : \tau$ always depends on $\Theta \vdash^f_{g''} v'' : \tau$ such that v'' is a subvalue of v and, in all cases, we have $\mathcal{L}_I(\tau'') \sqsubseteq_I \mathcal{L}_I(\tau)$. Since we also have that $v' \in \mathsf{Sub}_\delta(v'')$ (in case of pairs we will choose the appropriate subvalue), by inductive hypothesis we obtain $\mathcal{L}_I(\tau') \sqsubseteq_I \mathcal{L}_I(\tau'')$ from which the thesis. □ □

We can now prove that low keys never occurs as subvalues of values which have integrity level at or below H.

Lemma C.7. *Let* $\Theta \vdash_g^f v : \tau$ *and* $\mathcal{L}_I(\tau) \sqsubseteq_I \mathsf{H}$. *Then* $\mathsf{Sub}(v) \cap \mathcal{K}_{\mathsf{LL}} = \emptyset$.

Proof. By lemma C.6 we know that $v' \in \mathsf{Sub}(v)$ is such that $\Theta \vdash_{g'}^f v' : \tau'$ with $\mathcal{L}_I(\tau') \sqsubseteq_I \mathcal{L}_I(\tau) \sqsubseteq_I \mathsf{H}$. Since low keys can only be typed via (v-key) and (v-sub) as $\delta_C\mathsf{L}$, we conclude that $v' \notin \mathcal{K}_{\mathsf{LL}}$ from which the thesis. □

Proposition C.1 (Key-safety)**.** *Let* $\Delta \vdash_g^f \mathsf{M}$, *then* M *is key-safe.*

Proof. We want to prove that $\mathsf{knows}_\delta(\mathsf{M}) \cap \mathcal{K} \subseteq \mathcal{K}_{\sqsubseteq\delta}$. *First recall that* $\mathsf{knows}_\delta(\mathsf{M}) = \mathsf{knows}(img(\mathsf{M}|_\delta))$ *and, by definition,* $\mathsf{M}|_\delta$ *is the submemory of* M *only containing variables whose level is less then or equal* δ. *Let* $v_1, \ldots, v_n$ *be all the values in* $img(\mathsf{M}|_\delta)$. *From* $\Delta \vdash_g^f \mathsf{M}$ *we know that* $\Theta \vdash_{g_{\tau_i}}^f v_i : \tau_i$ *with* $\mathcal{L}(\tau_i) \sqsubseteq \delta$, *for all* $i = 1 \ldots n$.

If $\mathsf{LL} \not\sqsubseteq \delta$ *we necessarily have that* $\mathcal{L}_I(\delta) = \mathsf{H}$. *By lemma C.7 we know that* $\mathsf{Sub}(v_i) \cap \mathcal{K}_{\mathsf{LL}} = \emptyset$. *By lemma C.5 we obtain that* $\mathsf{knows}_\delta(\mathsf{M}) \cap \mathcal{K}_{\mathsf{LL}} \subseteq \cup_{i=1}^n \mathsf{Sub}(v_i) \cap \mathcal{K}_{\mathsf{LL}} = \emptyset$.

If $\mathsf{H}C \not\sqsubseteq \delta$ *we know that* $v_i \notin \mathcal{K}_{\mathsf{HC}}$ *since high keys can only be typed at level* HC. *By lemma C.2 we obtain that* $\mathsf{Sub}(v_i) \cap \mathcal{K}_{\mathsf{HC}} = \emptyset$. *Again by lemma C.5 we obtain that* $\mathsf{knows}_\delta(\mathsf{M}) \cap \mathcal{K}_{\mathsf{HC}} \subseteq \cup_{i=1}^n \mathsf{Sub}(v_i) \cap \mathcal{K}_{\mathsf{HC}} = \emptyset$.

We have thus proved that $\mathsf{knows}_\delta(\mathsf{M}) \cap \mathcal{K}_{\not\sqsubseteq\delta} = \emptyset$, *where* $\mathcal{K}_{\not\sqsubseteq\delta} = \mathcal{K} \setminus \mathcal{K}_{\sqsubseteq\delta}$. *Now we can write*

$$\begin{aligned}
&\mathsf{knows}_\delta(\mathsf{M}) \cap \mathcal{K} = \\
&\quad = \mathsf{knows}_\delta(\mathsf{M}) \cap (\mathcal{K}_{\sqsubseteq\delta} \cup \mathcal{K}_{\not\sqsubseteq\delta}) \\
&\quad = (\mathsf{knows}_\delta(\mathsf{M}) \cap \mathcal{K}_{\sqsubseteq\delta}) \cup (\mathsf{knows}_\delta(\mathsf{M}) \cap \mathcal{K}_{\not\sqsubseteq\delta}) \\
&\quad \subseteq \mathcal{K}_{\sqsubseteq\delta} \cup \emptyset \\
&\quad = \mathcal{K}_{\sqsubseteq\delta}
\end{aligned}$$

giving the thesis. □ □

Evaluation of well-typed expressions

We now prove that well-typed expressions of type τ evaluated on well-formed memories, always give a well-formed value of type τ. Moreover, randomized ciphertexts are guaranteed to be either identical copies of already existing ones or completely new ones, i.e., with a fresh confounder. Recall, we write $\mathsf{Sub}(\mathsf{M})$ to denote the set of subvalues of the whole memory M, i.e., $\cup_{v \in img(\mathsf{M})} \mathsf{Sub}(v)$.

Proposition C.2. *Let* $\Delta \vdash_g^f \mathsf{M}$, $\Delta \vdash e : \tau$ *and* $e \downarrow^{\mathsf{M}} v$. *Then*

(*i*) $\Theta \vdash_{g_\tau}^f v : \tau$

(*ii*) *if* $\{\!|v', r|\!\}_k \in \mathsf{Sub}(v)$, *with* $k \in \mathcal{K}_{\mathsf{HC}}^R$, *then either* $\{\!|v', r|\!\}_k \in \mathsf{Sub}(\mathsf{M})$ *or* r *has been extracted fresh, noted* $r \leftarrow C$, *during the evaluation of* e.

Proof. By induction on the structure of e.

x

By $\Delta \vdash x : \tau$ we have that $\Delta(x) = \tau'$ with $\tau' \leq \tau$. Since $x \downarrow^{\mathsf{M}} \mathsf{M}(x)$ by (i) of definition C.1 we directly obtain that $\Theta \vdash^f_{g_{\tau'}} v : \tau'$ and by lemma C.4 we obtain thesis (i). Thesis (ii) is easily obtained by observing that $\mathsf{Sub}(v) \subseteq \mathsf{Sub}(\mathsf{M})$.

e_1 op e_2

By $\Delta \vdash e_1 \ \mathsf{op}\ e_2 : \delta$ we have $\mathcal{L}_I(\delta) \neq [\mathsf{D} : \tilde{\mathsf{D}}]$. Here we do not even need induction: by assumption C.2(1) we have that $e_1 \ \mathsf{op}\ e_2 \downarrow^{\mathsf{M}} n$ with $n \notin \mathcal{K}$. Thesis (i) is a direct consequence of (v-name), thesis (ii) holds since $\mathsf{Sub}(v) = \{n\}$.

pair(e_1, e_2)

By $\Delta \vdash \mathsf{pair}(e_1, e_2) : (\tau_1, \tau_2)$ we have $\Delta \vdash e_i : \tau_i'$ with $\tau_i' \leq \tau_i$. Let $e_i \downarrow^{\mathsf{M}} v_i$ and notice that $\mathsf{pair}(e_1, e_2) \downarrow^{\mathsf{M}} (v_1, v_2)$. By induction and lemma C.4 we directly obtain that $\Theta \vdash^f_{g_{\tau_i}} v_i : \tau_i$. We have two cases: if $\mathsf{H} \sqsubseteq_I \mathcal{L}_I((\tau_1, \tau_2))$, by assumption C.1, we know that $\mathsf{H} \sqsubseteq_I \mathcal{L}_I(\tau_1) = \mathcal{L}_I(\tau_2)$ thus $g_{\tau_1} = g_{\tau_2} = g_\tau = \epsilon$ and we obtain the thesis by (v-pair). If $\mathsf{H} \not\sqsubseteq_I \mathcal{L}_I((\tau_1, \tau_2))$ we also have $\mathsf{H} \not\sqsubseteq_I \mathcal{L}_I(\tau_1), \mathcal{L}_I(\tau_2)$, since, by definition, $\mathcal{L}((\tau_1, \tau_2)) = \mathcal{L}(\tau_1) \sqcup \mathcal{L}(\tau_2)$. Thus $g_{\tau_1} = g_{\tau_2} = g_\tau = g$ and, again, by (v-pair) we obtain thesis (i). Now observe that $\mathsf{Sub}(v_1, v_2) = (v_1, v_2) \cup \mathsf{Sub}(v_1) \cup \mathsf{Sub}(v_2)$, thus every value of the form $\{\!|v', r|\!\}_k \in \mathsf{Sub}(v_1, v_2)$ occurs either in $\mathsf{Sub}(v_1)$ or $\mathsf{Sub}(v_2)$. Since, by induction, (ii) holds on v_1 and v_2, it also holds for (v_1, v_2).

fst(e_1), snd(e_1)

We show the proof for $\mathsf{fst}(e_1)$ as the one for $\mathsf{snd}(e_1)$ is identical. By $\Delta \vdash \mathsf{fst}(e_1) : \tau$ we have $\Delta \vdash e_1 : \tau'' = (\tau', \tau_2')$ with $\tau' \leq \tau$. Let $e_1 \downarrow^{\mathsf{M}} v_1$. By induction we know that $\Theta \vdash^f_{g_{\tau''}} v_1 : \tau''$. Now we have three cases: if $v_1 = (v, v_2)$ and $\mathsf{H} \sqsubseteq_I \mathcal{L}_I(\tau'')$, as for the above case, we have that $\mathsf{H} \sqsubseteq_I \mathcal{L}_I(\tau')$, thus $g_{\tau''} = g_{\tau'} = \epsilon$, implying $\Theta \vdash^f_{g_{\tau'}} v : \tau'$ and, by lemma C.4 we obtain $\Theta \vdash^f_{g_\tau} v : \tau$. The case $v_1 = (v, v_2)$ and $\mathsf{H} \not\sqsubseteq_I \mathcal{L}_I(\tau'')$ implies also $\mathsf{H} \not\sqsubseteq_I \mathcal{L}_I(\tau')$, giving $g_{\tau''} = g_{\tau'} = g$, and we proceed as above. The last case is when v is not a pair. Here we have that $\mathsf{fst}(e)$ returns $\bot$. The only way a non-pair can be typed as a pair is when $\tau'' \equiv \mathsf{LL} \equiv (\mathsf{LL}, \mathsf{LL})$. By (v-name) we obtain $\Theta \vdash^f \bot : \tau \equiv \mathsf{LL}$. Thesis (ii) is trivial in the case the result is $\bot$. If, instead, the result is v it is sufficient to observe that $\mathsf{Sub}(v) \subseteq \mathsf{Sub}(v_1)$ and that, by induction, (ii) holds for v_1.

enc$^\mu_x(e_1)$

In this case $\Delta \vdash e : \mathsf{enc}_\delta\ \kappa$. Let $e_1 \downarrow^{\mathsf{M}} v_1$ and $x \downarrow^{\mathsf{M}} v_2$. The proof follows by cases on the key type.

If $\Delta(x) = \mathsf{cK}_{\mathsf{LL}}(\tau)\ \kappa$ then the expression has been typed by (enc). It follows that $\delta = \delta_C\mathsf{L}$ and $\Delta \vdash e_1 : \tau'$ with $\tau' \leq \tau$. We have two cases: if $v_2 \notin \mathcal{K}$ we have $e \downarrow^{\mathsf{M}} \bot$ and by (v-name) we directly obtain $\Theta \vdash^f \bot : \delta_C\mathsf{L} \equiv \mathsf{enc}_\delta\ \kappa$. If, instead,

$v_2 = k \in \mathcal{K}$, we have $e \downarrow^{\mathsf{M}} \{|v_1|\}_k$. By induction hypothesis, $\Theta \vdash^f_{g_{\tau'}} v_1 : \tau'$ and $\Theta(v_2) = \mathsf{cK}_{\mathsf{LL}}(\tau)\ \kappa \equiv \mathsf{cK}_{\mathsf{LL}}(\mathsf{LL})\ \kappa$; by lemma C.3($i$) we obtain $\Theta \vdash^f v_1 : \mathsf{LL}$ and by (v-enc) we directly obtain that $\Theta \vdash^f \{|v_1|\}_k : \delta_C\mathsf{L} \equiv \mathsf{enc}_\delta\ \kappa$.

If $\Delta(x) = \mathsf{cK}_{\mathsf{HC}}(\tau)\ \kappa$ then the expression can be typed either by (enc) or (enc-d). In both cases $\Delta \vdash e_1 : \tau'$ with $\tau' \leq \tau$ and $\mathcal{L}_I(\delta) = C \sqcup \mathcal{L}_I(\tau)$. So by induction and lemma C.4, $\Theta \vdash^f_{g_\tau} v_1 : \tau$ and $\Theta(k) = \mathsf{cK}_{\mathsf{HC}}(\tau)\ \kappa$. Notice that, since $\mathcal{L}_I(\delta) = C \sqcup \mathcal{L}_I(\tau)$, we have $g_{\mathsf{enc}_\delta\ \kappa} = g_\tau$. We can now apply (v-enc) to get $\Theta \vdash^f_{g_\tau} \{|v_1|\}_k : \mathsf{enc}_\delta\ \kappa$. Notice that (v-enc) does not constraint the confidentiality level of δ, which is useful to fulfill both (enc) and (enc-d) requirements.

The case $\Delta(x) = \mathsf{cK}^R_{\mathsf{HC}}(\tau)\ \kappa$ is exactly as the above one.

Thesis (ii) is trivial in the case the result is $\perp$. If, instead, the result is $\{|v_1|\}_k$, with $\Theta(k) \neq \mathsf{cK}^R_{\mathsf{HC}}(\tau)\ \kappa$, we observe that $\mathsf{Sub}(\{|v_1|\}_k) = \{|v_1|\}_k \cup \mathsf{Sub}(v_1)$ and, since by induction we know that (ii) holds on $\mathsf{Sub}(v_1)$, we obtain the thesis. The only interesting case is when $\Theta(k) = \mathsf{cK}^R_{\mathsf{HC}}(\tau)\ \kappa$ which gives $\{|v_1, r|\}_k$ and consequently $\mathsf{Sub}(\{|v_1, r|\}_k) = \{|v_1, r|\}_k \cup \mathsf{Sub}(v_1) \cup r$. Since $r \leftarrow C$ during the evaluation of this encryption and by induction we know that (ii) holds on v_1, we obtain it also holds for $\{|v_1, r|\}_k$.

$\mathsf{dec}^\mu_x(e_1)$

We have $\Delta \vdash \mathsf{dec}_x(e_1) : \tau$ and $\Delta \vdash e_1 : \mathsf{enc}_\delta\ \kappa$. Let $e_1 \downarrow^{\mathsf{M}} v_1$ and $x \downarrow^{\mathsf{M}} v_2$. By induction hypothesis, $\Theta \vdash^f_{g_{\tau''}} v_1 : \tau''$ with $\tau'' = \mathsf{enc}_\delta\ \kappa$. The proof follows by cases on the key type.

If $\Delta(x) = \mathsf{cK}_{\mathsf{LL}}(\tau')\ \kappa$ then the expression has been typed with $\tau = \mathsf{LL} \sqcup \delta$ by (dec). It follows that $\tau = \delta_C\mathsf{L}$. We have two cases: if $v_1 \neq \{|v|\}_{v_2}$ or $v_2 \notin \mathcal{K}$ we have $e \downarrow^{\mathsf{M}} \perp$ and by (v-name) we directly obtain $\Theta \vdash^f \perp : \delta_C\mathsf{L} \equiv \tau$. If, instead, $v_1 = \{|v|\}_k$ and $v_2 = k \in \mathcal{K}$, we have $e \downarrow^{\mathsf{M}} v$. By induction hypothesis, $\Theta \vdash^f v_1 : \mathsf{enc}_\delta\ \kappa \equiv \delta_C\mathsf{L}$. This implies, by (v-enc), that $\Theta \vdash^f v : \tau'$ and, by lemma C.3(i) we obtain $\Theta \vdash^f v : \mathsf{LL} \leq \delta_C\mathsf{L}$.

If $\Delta(x) = \mathsf{cK}_{\mathsf{HC}}(\tau')\ \kappa$, since high keys can only be typed via (v-key), we know that $v_2 = k$ and $\Theta(k) = \mathsf{cK}_{\mathsf{HC}}(\tau')\ \kappa$. The expression can be typed either by (dec) or (dec-μ).

If the expression has been typed with $\tau = \mathsf{H}C \sqcup \delta$ by (dec) we have two subcases: if $v_1 \neq \{|v|\}_k$ we have $e \downarrow^{\mathsf{M}} \perp$. The only way v_1 can have type $\mathsf{enc}_\delta\ \kappa$ without being a ciphertext is when $\mathcal{L}_I(\delta) = \mathsf{L}$ thus $\mathcal{L}_I(\mathsf{H}C \sqcup \delta) = \mathsf{L}$. By (v-name) we directly obtain $\Theta \vdash^f \perp : \tau$. If, instead, $v_1 = \{|v|\}_k$, we have $e \downarrow^{\mathsf{M}} v$. By induction hypothesis, $\Theta \vdash^f_{g_{\tau''}} v_1 : \tau''$ with $\tau'' = \mathsf{enc}_\delta\ \kappa$ and by (v-enc) we know that $\Theta \vdash^f_{g'} v : \tau'$, with $C \sqcup \mathcal{L}_I(\tau') = \mathcal{L}_I(\tau'')$. This implies $\mathsf{H} \sqsubseteq_I \mathcal{L}_I(\tau'')$ iff

$\mathsf{H} \sqsubseteq_I \mathcal{L}_I(\tau')$ which means $g_{\tau''} = g_{\tau'}$. Notice also that g' can be different from $g_{\tau''}$ only when $g_{\tau''} = \epsilon$ but this happens when $\mathsf{H} \sqsubseteq_I \mathcal{L}_I(\tau')$ and by corollary C.1 we obtain that $\Theta \vdash^f_{g'} v : \tau'$ implies $\Theta \vdash^f v : \tau'$. Thus $\Theta \vdash^f_{g_{\tau''}} v : \tau'$ and since $g_{\tau''} = g_{\tau'}$, $\Theta \vdash^f_{g_{\tau'}} v : \tau'$. Now it can be easily shown that $\tau = \mathsf{H}\delta_I$ with $\delta_I = C \sqcup \mathcal{L}_I(\tau')$. In fact, $\tau = \mathsf{HC} \sqcup \delta = \mathsf{HC} \sqcup \mathcal{L}_I(\delta) = \mathsf{HC} \sqcup \mathcal{L}_I(\tau'') = \mathsf{HC} \sqcup \mathcal{L}_I(\tau')$. Thus $\tau' \leq \tau$ and by lemma C.4 we obtain $\Theta \vdash^f_{g_\tau} v : \tau$.

If the expression has been typed with $\tau = \tau'$ by (dec-μ) two subcases must be considered: if $v_1 \neq \{\!|v|\!\}_k$ we have $e \downarrow^{\mathsf{M}} \bot$. The only way v_1 can have type $\mathsf{enc}_\delta\ \kappa$ without being a ciphertext is when $\mathcal{L}_I(\delta) = \mathcal{L}_I(\mathsf{H}C \sqcup \mathcal{L}_I(\tau')) = \mathsf{L}$, i.e., $\mathcal{L}_I(\tau') = \mathsf{L}$. By (v-name) we directly obtain $\Theta \vdash^f \bot : \delta_C\mathsf{L} \equiv \tau'$. Otherwise $v_1 = \{\!|v|\!\}_k$ and we have $e \downarrow^{\mathsf{M}} v$. If $\mathcal{L}_I(\tau') = \mathsf{L}$ the proof follows directly by (v-name). If, instead, $\mathcal{L}_I(\tau') \sqsubseteq_I \mathsf{H}$ it follows $\mathcal{L}_I(\delta) \sqsubseteq_I \mathsf{H}$ and v_1 must have been typed via (v-enc). The fact $\Theta \vdash^f_{g_{\tau'}} v : \tau'$ is obtained exactly as done for the previous case.

The case $\Delta(x) = \mathsf{cK}^R_{\mathsf{HC}}(\tau')\ \kappa$ is exactly as the above one except for $v_1 = \{\!|v, r|\!\}_k$ which is typed via (v-enc-r) in place of (v-enc).

Thesis (ii) is trivial in the case the result is $\bot$. If, instead, the result is v it is sufficient to observe that $\mathsf{Sub}(v) \subseteq \mathsf{Sub}(\{\!|v|\!\}_k)$ and that, by induction, (ii) holds for $\{\!|v|\!\}_k$.

$\mathsf{mac}_k(e_1)$

Proof is analogous to the one for encryption: The expression has been typed by rule (mac) with $\Delta(x) = \mathsf{mK}_{\delta_C\delta_I}(\tau)$. It follows that $\Delta \vdash e_1 : \tau'$ with $\tau' \leq \tau$. We have two cases: if $v_2 \notin \mathcal{K}$ we have $\delta_I = \mathsf{L}$ and $e \downarrow^{\mathsf{M}} \bot$ and by (l-name) we directly obtain $\Theta \vdash^f \bot : \mathsf{LL}$. If, instead, $v_2 = k \in \mathcal{K}$, we have $e \downarrow^{\mathsf{M}} \langle v_1 \rangle_k$. If the key is of level LL, by induction hypothesis, $\Theta \vdash^f_{g_{\tau'}} v_1 : \tau'$ and $\Theta(v_2) = \mathsf{mK}_{\mathsf{LL}}(\tau) \equiv \mathsf{mK}_{\mathsf{LL}}(\mathsf{LL})$ and by lemma C.3(i) we obtain $\Theta \vdash^f v_1 : \mathsf{LL}$ which by (v-mac) gives $\Theta \vdash^f \langle v_1 \rangle_k : \mathsf{LL}$. If instead the key is of level $\mathsf{H}C$, by induction and lemma C.4, $\Theta \vdash^f_{g_\tau} v_1 : \tau$ and $\Theta(k) = \mathsf{mK}_{\mathsf{H}C}(\tau)$. Thus, by (v-mac) $\Theta \vdash^f_{g_\tau} \langle v_1 \rangle_k : \delta_C\mathsf{L}$ and by lemma C.3(i) $\Theta \vdash^f \langle v_1 \rangle_k : \mathsf{LL} \sqsubseteq \delta_C\mathsf{L}$.

Thesis (ii) is trivial in the case the result is $\bot$. If, instead, the result is $\langle v_1 \rangle_k$ we observe that $\mathsf{Sub}(\langle v_1 \rangle_k) = \langle v_1 \rangle_k \cup \mathsf{Sub}(v_1)$ and, since by induction we know that (ii) holds on $\mathsf{Sub}(v_1)$, we obtain the thesis. □

□

It is useful to prove that well-formedness of values of type $\delta_C\delta_I$ does not constrain in any way the confidentiality level.

Lemma C.8. *If* $\Theta \vdash^f_g v : \mathsf{H}\delta_I$ *then* $\Theta \vdash^f_g v : \mathsf{L}\delta_I$.

Proof. By induction on the structure of v.

Base case:

n
: We know that $\Theta \vdash_g^f n : \delta'_C\delta'_I$ with $\delta'_C\delta'_I \leq \mathsf{H}\delta_I$. Notice that $n \notin \mathcal{K}_{\mathsf{H}C}$ as high keys can never be typed as $\mathsf{H}\delta_I$. Low keys are only typed LL and HL which directly gives the thesis. For $n \notin \mathcal{K}$ we observe that rules (v-name), (v-ir) and (v-dd) do not restrict the confidentiality level. By reapplying the same rule as above with $\delta'_C = \mathsf{L}$ we thus obtain $\Theta \vdash_g^f n : \mathsf{L}\delta'_I$. From $\mathsf{L}\delta'_I \leq \mathsf{L}\delta_I$ we obtain the thesis.

Inductive cases:

(v_1, v_2)
: The only case a pair can be typed $\mathsf{H}\delta_I$ is when $\delta_I = \mathsf{L}$, since $\mathsf{HL} \equiv (\mathsf{HL}, \mathsf{HL})$. Thus, by assumption C.1(1) and (v-pair), we know that $\Theta \vdash_g^f v_i : \mathsf{HL}$. By induction we get $\Theta \vdash_g^f v_i : \mathsf{LL}$ and by (v-pair) directly $\Theta \vdash_g^f v : (\mathsf{LL}, \mathsf{LL}) \equiv \mathsf{LL}$.

$\{\!|v|\!\}_k$
: We know that $\Theta \vdash_g^f \{\!|v|\!\}_k : \mathsf{enc}_{\delta'_C\delta'_I}\ \kappa$ with $\mathsf{enc}_{\delta'_C\delta'_I}\ \kappa \leq \delta'_C\mathsf{L} \leq \mathsf{HL}$. The first judgment is either (v-enc) or (v-enc-r): we reapply it with $\delta'_C = \mathsf{L}$ getting $\Theta \vdash_g^f \{\!|v|\!\}_k : \mathsf{enc}_{\mathsf{L}\delta'_I}\ \kappa \leq \mathsf{LL}$.

$\langle v \rangle_k$
: This case proved exactly as above. □

□

Theorem C.1. *If* $\Delta, pc \vdash \mathsf{c}$*,* $\Delta \vdash_g^f \mathsf{M}$ *and* $\langle \mathsf{M}, \mathsf{c} \rangle \Rightarrow \mathsf{M}'$ *then* $\Delta \vdash_g^f \mathsf{M}'$

Proof. By induction on the derivation length of $\langle \mathsf{M}, \mathsf{c} \rangle \Rightarrow \mathsf{M}'$.

Base case, length 1:

[*skip*], [*whilef*]
: Since $\langle \mathsf{M}, \mathsf{skip} \rangle \Rightarrow \mathsf{M}$ and $\Delta \vdash_g^f \mathsf{M}$ we have nothing to prove. The same holds for while loops when the guard is false, since memory is untouched.

[*assign*]
: We have $\langle \mathsf{M}, x := e \rangle \Rightarrow \mathsf{M}[x \mapsto v] = \mathsf{M}'$ given that $e \downarrow^{\mathsf{M}} v$. We now prove condition (i) of definition C.1. Since the only change from M to M' is the value of x, which is set to v, we just need to check that $\Delta(x) = \tau$ implies $\Theta \vdash_{g_\tau}^f v : \tau$. For (assign) this is directly achieved via proposition C.2(i), since the rule requests $\Delta \vdash e : \tau$. For (declassify) we know that $\Delta(x) = \delta_C\mathsf{H}$ and $\Delta \vdash e : \delta'_C\mathsf{H}$ and, by proposition C.2(i), $\Theta \vdash^f v : \delta'_C\mathsf{H}$. By lemma C.8 we have that $\Theta \vdash^f v : \mathsf{LH} \leq \delta_C\mathsf{H}$.

Notice that $\mathsf{Sub}(\mathsf{M}') \subseteq \mathsf{Sub}(\mathsf{M}) \cup \mathsf{Sub}(v)$. Thus if we prove condition (ii) of definition C.1 on $\mathsf{Sub}(\mathsf{M}) \cup \mathsf{Sub}(v)$ we obtain that it holds even for M'. We consider three different cases: Let $\Theta(k) = \mathsf{cK}^R_{\mathsf{HC}}(\tau)\ \kappa$ and $\{\!|v, r|\!\}_k$, $\{\!|v', r'|\!\}_k \in \mathsf{Sub}(\mathsf{M})$. Since $\Delta \vdash^f_g \mathsf{M}$ we directly know that $r \neq r'$. If, instead, $\{\!|v, r|\!\}_k, \{\!|v', r'|\!\}_k \in \mathsf{Sub}(v)$, by proposition C.2$(ii)$ we know that either $\{\!|v', r'|\!\}_k \in \mathsf{Sub}(\mathsf{M})$, which leads to the previous case, or r has been extracted fresh, noted $r \leftarrow C$, during the evaluation of e, which directly gives $r \neq r'$. Finally, if $\{\!|v, r|\!\}_k$, $\{\!|v', r'|\!\}_k \in \mathsf{Sub}(v)$ we have that either both values also appear in M, which one more time leads to the first case, or one of the confounder has been extracted fresh during the evaluation of e which directly gives $r \neq r'$.

Inductive case, length n. We consider the last rule applied:

[***seq***]
We have $\langle \mathsf{M}, \mathsf{c}_1; \mathsf{c}_2 \rangle \Rightarrow \mathsf{M}'$ since $\langle \mathsf{M}, \mathsf{c}_1 \rangle \Rightarrow \mathsf{M}''$ and also $\langle \mathsf{M}'', \mathsf{c}_2 \rangle \Rightarrow \mathsf{M}'$. By rule (seq) we have that $\Delta, pc \vdash \mathsf{c}_i$. By induction on first command we get $\Delta \vdash^f_g \mathsf{M}''$, then by induction on the second one $\Delta \vdash^f_g \mathsf{M}'$.

[***ift***], [***iff***]
If the command is typed with rule (if) we proceed by induction on the executed branch and we directly get $\Delta \vdash^f_g \mathsf{M}'$. The interesting case is when the command is typed via (if-MAC). If the mac check is false the command do not terminate and we have nothing to prove. We analyze the case when $\mathsf{mac}_x(z, e) = e'$ is true, i.e., we have:

$$\langle \mathsf{M}, \mathsf{if}\ \mathsf{mac}_x(z, e) = e'\ \mathsf{then}\ (y := e; \mathsf{c}_1)\ \mathsf{else}\ \mathsf{c}_2; \bot_{\mathsf{MAC}} \rangle \Rightarrow \mathsf{M}'$$

because of $\langle \mathsf{M}, y := e \rangle \Rightarrow \mathsf{M}''$ and $\langle \mathsf{M}'', \mathsf{c}_1 \rangle \Rightarrow \mathsf{M}'$. By rule (if-MAC) we know that $\Delta(x) = \mathsf{mK}_{\mathsf{HC}}(\mathsf{L}[\mathsf{D}], \tau)$. By $\Delta \vdash^f_g \mathsf{M}$ we know that $\mathsf{M}(x)$ has type $\mathsf{mK}_{\mathsf{HC}}(\mathsf{L}[\mathsf{D}], \tau)$, i.e., $\mathsf{M}(x) = k$ and $\Theta(k) = \mathsf{mK}_{\mathsf{HC}}(\mathsf{L}[\mathsf{D}], \tau)$ thus $\mathsf{mac}_x(z, e) \downarrow^{\mathsf{M}} \langle v, v' \rangle_k$ and $e' \downarrow^{\mathsf{M}} \langle v, v' \rangle_k$. We know that $\Delta \vdash e' : \mathsf{LL}$, thus by proposition C.2 we know that $\Theta \vdash^f_{g_\tau} \langle v, v' \rangle_k : \mathsf{LL}$. By (v-mac) and (v-pair) we are guaranteed that $\Theta \vdash^f_{g'} v : \mathsf{L}[\mathsf{D}]$ and $\Theta \vdash^f_{g'} v' : \tau$. By rule (if-MAC) we also know that $\Delta \vdash z : \mathsf{L}[\mathsf{D}]$ and from $\Delta \vdash^f_g \mathsf{M}$ we have $\Theta \vdash^f_g v : \mathsf{L}[\mathsf{D}]$, with the expected g since the type integrity is lower than H. We obtain that $g(\mathsf{D}) = g'(\mathsf{D})$. From $\mathsf{IRs}(\mathsf{L}[\mathsf{D}], \tau) = \{\mathsf{D}\}$ and $\mathsf{Closed}(\mathsf{L}[\mathsf{D}], \tau)$ we easily obtain that $\mathcal{L}(\tau) = \delta_C[\bullet : \mathsf{D}]$ and also that $\Theta \vdash^f_{g'} v' : \tau$ implies $\Theta \vdash^f_g v' : \tau$, since by lemma C.6 nothing above $[\bullet : \mathsf{D}]$ will ever appear as subvalue of v' and g and g' are the same on domain D. This is what we need to prove to check the well-formedness of the memory obtained after the assignment (item (ii) of well-formedness is dealt with as in the case [*assign*]). Thus $\Delta \vdash^f_g \mathsf{M}''$. By induction on the derivation for c_1 we have that also $\Delta \vdash^f_g \mathsf{M}'$.

[***whilet***]
This case is analogous to [*seq*]. Execution $\langle \mathsf{M}, \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c} \rangle \Rightarrow \mathsf{M}'$ derives

from $\langle \mathsf{M}, \mathsf{c}\rangle \Rightarrow \mathsf{M}''$ and $\langle \mathsf{M}'', \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}\rangle \Rightarrow \mathsf{M}'$. Rule (while) ensures that $\Delta, pc' \vdash \mathsf{c}$. Thus, by induction, we get $\Delta \vdash^f_g \mathsf{M}''$ and consequently $\Delta \vdash^f_g \mathsf{M}'$. Notice that we can apply induction on $\langle \mathsf{M}'', \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}\rangle \Rightarrow \mathsf{M}'$ as, even if the command is the same as the one we are analysing, the derivation length is $n-1$. □

□

Indistinguishability of well-formed memories We now extend the equivalence notion on memories, so to respect dependent domains. Notice that observation point is placed in the four-point lattice (i.e., we write ℓ and not δ), i.e., variables below integrity H are always observed as a whole, at the appropriate confidentiality level.

We say that a substitution ρ *f-respects* Θ if $\rho(\Box_{\{\!|v_1|\!\}_k}) = \Box_{\{\!|v_2|\!\}_k})$ and $\Theta(k) = \mathsf{K}_{\delta'}(\tau)\ \kappa$ imply $\exists g'$ such that $\Theta \vdash^{f_i}_{g'} v_i : \tau$. Moreover, $g' = \epsilon$ iff $\mathsf{H} \sqsubseteq_I \mathcal{L}_I(\tau)$.

Definition C.4 (WF-Indistinguishability)**.** M_1 *and* M_2 *are WF-indistinguishable at level* ℓ*, written* $\mathsf{M}_1 \approx^\Delta_\ell \mathsf{M}_2$*, if*

1. $\Delta \vdash^{f_i}_g \mathsf{M}_i$*;*
2. $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\ \rho$ *(meaning that* $\mathsf{M}_1 \approx_\ell \mathsf{M}_2$*);*
3. ρ *f-respects* Θ*.*

Intuitively, two memories are WF-indistinguishable if (1) they are well-formed; (2) they are indistinguishable; (3) encrypted hidden values mapped by ρ can be typed with the same g' (at the expected type τ).

We now need a few lemmas that characterize $\mathsf{CloseDD}(\tau)$ types, defined in section C.1.

Lemma C.9. $\mathsf{DD}(\tau)$*, with* $\tau \neq \mathsf{K}_{\mathsf{HC}}(\tau')\ \kappa$*, implies* $\mathcal{L}_I(\tau) \sqsubset_I \mathsf{H}$*.*

Proof. Notice that stating $\mathcal{L}_I(\tau) \sqsubset_I \mathsf{H}$ is equivalent to $\mathsf{H} \not\sqsubseteq_I \mathcal{L}_I(\tau)$ since every integrity level is comparable with H. We proceed by induction on the structure of τ:

Base case:

$\delta_C\delta_I$
: Trivial since $\mathsf{DD}(\delta_\mathsf{C}\delta_\mathsf{I})$ directly requires that $\mathsf{C} \not\sqsubseteq_I \delta_I$ and $\mathsf{C} \sqsubseteq_I \mathsf{H} \sqsubseteq \mathsf{L}$. Thus $\delta_I \neq \mathsf{L}, \mathsf{H}$, i.e., $\mathcal{L}_I(\tau) \sqsubset_I \mathsf{H}$.

Inductive case:

$\mathsf{K}_{\delta'}(\tau')\ \kappa$
: We have assumed $\tau \neq \mathsf{K}_{\mathsf{HC}}(\tau)\ \kappa$ thus $\delta' = \mathsf{LL}$, which is anyway forbidden as $\mathsf{C} \sqsubseteq_I \mathsf{L}$. Thus this case never occurs.

$\mathsf{enc}_{\delta'}\ \kappa$

Notice that if κ refers to a low key we have LL in $\mathsf{LT}_{\Delta}(\mathsf{enc}_{\delta'}\ \kappa)$, making $\mathsf{DD}(\tau)$ false. Thus we only consider the case in which κ refers to a high key $\mathsf{cK}^{\mu}_{\mathsf{HC}}(\tau')\ \kappa$, By assumption C.1(4) we have $\mathcal{L}_I(\delta') = \mathsf{C} \sqcup \mathcal{L}_I(\tau')$. We also have $\mathsf{DD}(\mathsf{enc}_{\delta'}\ \kappa)$ iff $\mathsf{DD}(\tau')$ and τ' cannot be a high key by assumption C.1(3) . Thus, by induction we get $\mathcal{L}_I(\tau') \sqsubset_I \mathsf{H}$ which implies $\mathcal{L}_I(\delta') = \mathcal{L}_I(\mathsf{enc}_{\delta'}\ \kappa) \sqsubset_I \mathsf{H}$.

(τ_1, τ_2)

We have $\mathsf{DD}(\tau)$ iff $\mathsf{DD}(\tau_1)$ and $\mathsf{DD}(\tau_2)$ and by assumption C.1(3) τ_i cannot be high level key types. Thus by induction $\mathcal{L}_I(\tau_i) \sqsubset_I \mathsf{H}$ and so $\mathcal{L}_I(\tau) \sqsubset_I \mathsf{H}$. □

□

We let $\mathsf{gcore}_\tau = (\mathsf{IRs}(\tau) \cup \mathsf{Dep}(\tau)) \times \mathit{Val}$. When we intersect a function g with gcore_τ we get its restriction on the set of domains $\mathsf{IRs}(\tau) \cup \mathsf{Dep}(\tau)$. Intuitively, these are the domains which are needed to type a value as τ, since they are all the integrity representatives plus all the domains which type τ depends on. Next lemma formalizes this fact. Moreover it proves that any other g' which includes the above mentioned restriction of g, can be used in place of g to type the value.

Lemma C.10. *Let* $\Theta \vdash^f_g v : \tau$, $g \neq \epsilon$, $\mathsf{DD}(\tau)$. *Then*

(*i*) $\mathit{dom}(g) \supseteq \mathsf{IRs}(\tau) \cup \mathsf{Dep}(\tau)$

(*ii*) $g' \supseteq g \cap \mathsf{gcore}_\tau$ *implies* $\Theta \vdash^f_{g'} v : \tau$.

Proof. By induction on the structure of v:

Base case:

n

First notice that types $\delta_C[\mathsf{D}]$ and $\delta_C[\mathsf{D} : \{\mathsf{D_1}, \ldots, \mathsf{D_m}\}]$ have no subtypes. The only rules where the value of g matters are (v-ir) and (v-dd) which respectively gives judgements $\Theta \vdash^f_g n : \delta_C[\mathsf{D}]$ and $\Theta \vdash^f_g n : \delta_C[\mathsf{D} : \{\mathsf{D_1}, \ldots, \mathsf{D_m}\}]$. In the former case we have $\mathsf{D} \in \mathsf{IRs}(\delta_C[\mathsf{D}]) = \{\mathsf{D}\}$. We know that $g(\mathsf{D})\downarrow$ and $n = g(\mathsf{D})$. Since $\mathsf{Dep}(\delta_C[\mathsf{D}]) = \emptyset$ we directly have (i). In order to prove (ii), by lemma hypothesis we get $g'(\mathsf{D})\downarrow$, $g'(\mathsf{D}) = g(\mathsf{D})$ and thus $n = g'(\mathsf{D})$ which implies $\Theta \vdash^f_{g'} n : \delta_C[\mathsf{D}]$. In the latter case we have that $\mathsf{D}_1, \ldots, \mathsf{D}_m \in \mathsf{Dep}(\delta_C[\mathsf{D} : \{\mathsf{D_1}, \ldots, \mathsf{D_m}\}]) = \{\mathsf{D}_1, \ldots, \mathsf{D}_m\}$. We additionally know that $g(\mathsf{D}_i)\downarrow$ and $n = f_{\mathsf{D:D_1,\ldots D_m}}(g(\mathsf{D}_1), \ldots, g(\mathsf{D}_m))$. By lemma hypothesis we thus get $g'(\mathsf{D}_i)\downarrow$, $g'(\mathsf{D}_i) = g(\mathsf{D}_i)$ thus $n = f_{\mathsf{D:D_1,\ldots D_m}}(g'(\mathsf{D}_1), \ldots, g'(\mathsf{D}_m))$ which implies $\Theta \vdash^f_{g'} n : \delta_C[\mathsf{D} : \{\mathsf{D_1}, \ldots, \mathsf{D_m}\}]$, since f is the same function.

Inductive case:

(v_1, v_2)

We have $\Theta \vdash_g^f v : \tau = (\tau_1, \tau_2)$ because of $\Theta \vdash_g^f v_i : \tau_i$ (pairs are not in the subtype relation). We have that $\mathsf{DD}(\tau)$ iff $\mathsf{DD}(\tau_1)$ and $\mathsf{DD}(\tau_2)$; moreover, $\mathsf{IRs}(\tau) \cup \mathsf{Dep}(\tau) = \cup_i \mathsf{IRs}(\tau_i) \cup \mathsf{Dep}(\tau_i)$. Thus, $g' \supseteq g \cap (\mathsf{IRs}(\tau_i) \cup \mathsf{Dep}(\tau_i)) \times Val$ meaning that, by induction, we get $dom(g) \supseteq \mathsf{IRs}(\tau_i) \cup \mathsf{Dep}(\tau_i)$ thus $dom(g) \supseteq \mathsf{IRs}(\tau) \cup \mathsf{Dep}(\tau)$, and $\Theta \vdash_{g'}^f v_i : \tau_i$ which, from (v-pair), trivially gives thesis (ii).

$\{\!|v'|\!\}_k$

We have $\Theta(k) = \mathsf{K}_\delta(\tau')\ \kappa$ and $\Theta \vdash_g^f \{\!|v'|\!\}_k : \tau$ because of one of the (v-enc) rules. Moreover notice that type $\mathsf{enc}_\delta\ \kappa$ derived from those rules is always such that $C \sqsubseteq \mathcal{L}_I(\delta)$. Thus by $\mathsf{DD}(\tau)$ we know that $\tau = \mathsf{enc}_\delta\ \kappa$; in fact subtyping would give $\mathsf{enc}_\delta\ \kappa \leq \delta'$ with $C \sqsubseteq \mathcal{L}_I(\delta') = \mathsf{L}$ forbidden by $\mathsf{DD}(\tau)$. We consider (v-enc): since by hypothesis $g \neq \epsilon$ we have that $\Theta \vdash_g^f v' : \tau'$. Now we have $\mathsf{DD}(\tau)$ iff $\mathsf{DD}(\tau')$. Notice, in fact, that $\mathsf{LT}_\Delta(\mathsf{enc}_\delta\ \kappa) = \mathsf{LT}_\Delta(\tau')$ where τ' is the type transported by the unique key relative to label κ. For the same reason, $\mathsf{IRs}(\tau) \cup \mathsf{Dep}(\tau) = \mathsf{IRs}(\tau') \cup \mathsf{Dep}(\tau')$. By induction, we thus get $dom(g) \supseteq \mathsf{IRs}(\tau') \cup \mathsf{Dep}(\tau') = \mathsf{IRs}(\tau) \cup \mathsf{Dep}(\tau)$ and $\Theta \vdash_{g'}^f v' : \tau'$ and, again by (v-enc), we get $\Theta \vdash_{g'}^f \{\!|v'|\!\}_k : \tau$. Rule (v-enc-r) is analyzed in the very same way apart from the fact $v' = (v'', r)$.

$\langle v' \rangle_k$

This case is not possible as MACs can never be typed at a level $\delta_C \delta_I$ such that $\mathsf{C} \not\sqsubseteq_I \delta_I$, as required by $\mathsf{DD}(\tau)$. □

□

We now prove that when v is typed τ under g and g', then g and g' are the same when restricted to the set of integrity representatives of τ. We let $\mathsf{gcoreIR}_\tau = \mathsf{IRs}(\tau) \times Val$.

Lemma C.11. *Let* $\Theta \vdash_g^f v : \tau$, $\Theta \vdash_{g'}^f v : \tau$, $g, g' \neq \epsilon$, $\mathsf{DD}(\tau)$, *with* $\tau \neq \mathsf{K}_{\mathsf{HC}}(\tau')\ \kappa$. *Then* $g \cap \mathsf{gcoreIR}_\tau = g' \cap \mathsf{gcoreIR}_\tau$.

Proof. We proceed by induction on the structure of τ:

Base case:

$\delta_C \delta_I$

Since $\mathsf{DD}(\delta_\mathsf{C}\delta_\mathsf{I})$ requires that $\mathsf{C} \not\sqsubseteq_I \delta_I$ the only possible types are $\delta_C[\mathsf{D}]$ and $\delta_C[\mathsf{D} : \{\mathsf{D}_1, \ldots, \mathsf{D}_\mathsf{m}\}]$. The only one which has a non empty $\mathsf{IRs}(\tau)$ is the former. In fact, $\mathsf{IRs}(\delta_C[\mathsf{D}]) = \{\mathsf{D}\}$, and it can only type atomic names n. Thus $v = n$ with $n = g(\mathsf{D}) = g'(\mathsf{D})$, from which the thesis.

Inductive case:

$\mathsf{K}_{\delta'}(\tau')\ \kappa$

We have assumed $\tau \neq \mathsf{K}_{\mathsf{HC}}(\tau)\ \kappa$ thus $\delta' = \mathsf{LL}$, which is anyway forbidden as $\mathsf{C} \sqsubseteq_I \mathsf{L}$. Thus this case never occurs.

$\mathsf{enc}_{\delta'}\ \kappa$

Notice that if κ refers to a low key we have LL in $\mathsf{LT}_\Delta(\mathsf{enc}_{\delta'}\ \kappa)$, making $\mathsf{DD}(\tau)$ false. Thus we only consider the case in which κ refers to a high key $\mathsf{cK}^{\mu}_{\mathsf{HC}}(\tau')\ \kappa$. We also have $\mathsf{IRs}(\mathsf{enc}_{\delta'}\ \kappa) = \mathsf{IRs}(\tau')$ and $\mathsf{DD}(\tau)$ iff $\mathsf{DD}(\tau')$. Since, by (v-enc), $v = \{\!|v'|\!\}_k$ and $\Theta \vdash^f_g v' : \tau'$, $\Theta \vdash^f_{g'} v' : \tau'$, by induction we get $g \cap \mathsf{gcoreIR}_\tau = g' \cap \mathsf{gcoreIR}_\tau$, which gives the thesis. The case for (v-enc-r) is done analogously.

(τ_1, τ_2)

We have $\mathsf{IRs}((\tau_1, \tau_2)) = \cup_i \mathsf{IRs}(\tau_i)$. We have that $\mathsf{DD}(\tau)$ iff $\mathsf{DD}(\tau_1)$ and $\mathsf{DD}(\tau_2)$. By rule (v-pair) we have that $\Theta \vdash^f_g v_i : \tau'$, $\Theta \vdash^f_{g'} v_i : \tau'$ and, by induction, we get $g \cap \mathsf{gcoreIR}_{\tau_i} = g' \cap \mathsf{gcoreIR}_{\tau_i}$. Since $g \cap \mathsf{gcoreIR}_\tau = \cup_i g \cap \mathsf{gcoreIR}_{\tau_i}$ and $g' \cap \mathsf{gcoreIR}_\tau = \cup_i g' \cap \mathsf{gcoreIR}_{\tau_i}$ we obtain that $g \cap \mathsf{gcoreIR}_\tau = g' \cap \mathsf{gcoreIR}_\tau$. □

□

If we type v_1 and v_2 as τ under the same g and f, and the $\mathsf{DD}(\tau)$, $\mathsf{Det}(\tau)$, then we can conclude that the two values are the same:

Lemma C.12. *Let* $\Theta \vdash^f_g v_1 : \tau$, $\Theta \vdash^f_g v_2 : \tau$ *and* $g \neq \epsilon$, $\mathsf{DD}(\tau)$, $\mathsf{Det}(\tau)$, *with* $\tau \neq \mathsf{K}_{\mathsf{HC}}(\tau')\ \kappa$. *Then* $v_1 = v_2$.

Proof. We proceed by induction on the structure of τ:

Base case:

$\delta_C\delta_I$

Since $\mathsf{DD}(\delta_\mathsf{C}\delta_\mathsf{I})$ requires that $\mathsf{C} \not\sqsubseteq_I \delta_I$ the only possible types are $\delta_C[\mathsf{D}]$ and $\delta_C[\mathsf{D} : \{\mathsf{D}_1, \ldots, \mathsf{D}_\mathsf{m}\}]$. In the former case $v_1, v_2 = g(\mathsf{D})$, in the latter case $v_1, v_2 = f_{\mathsf{D}:\mathsf{D}_1,\ldots\mathsf{D}_\mathsf{m}}(g(\mathsf{D}_1), \ldots, g(\mathsf{D}_m))$.

Inductive case:

$\mathsf{K}_{\delta'}(\tau')\ \kappa$

We have assumed $\tau \neq \mathsf{K}_{\mathsf{HC}}(\tau)\ \kappa$ thus $\delta' = \mathsf{LL}$, which is anyway forbidden as $\mathsf{C} \sqsubseteq_I \mathsf{L}$. Thus this case never occurs.

$\mathsf{enc}_{\delta'}\ \kappa$

Notice that if κ refers to a low key we have LL in $\mathsf{LT}_\Delta(\mathsf{enc}_{\delta'}\ \kappa)$, making $\mathsf{DD}(\tau)$ false. Thus we only consider the case in which κ refers to a high deterministic key $\mathsf{cK}_{\mathsf{HC}}(\tau')\ \kappa$, given that randomized keys are excluded by condition $\mathsf{Det}(\tau)$. Now we have $\mathsf{DD}(\tau)$ iff $\mathsf{DD}(\tau')$ and $\mathsf{Det}(\tau)$ iff $\mathsf{Det}(\tau')$. Since, by (v-enc), $v_i = \{\!|v'_i|\!\}_k$ and $\Theta \vdash^f_g v'_1 : \tau'$, $\Theta \vdash^f_g v'_2 : \tau'$, by induction we get $v'_1 = v'_2$, which gives the thesis.

(τ_1, τ_2)

Let $v_i = (v_i^1, v_i^2)$. We have that $\mathsf{DD}(\tau), \mathsf{Det}(\tau)$ iff $\mathsf{DD}(\tau_1), \mathsf{Det}(\tau_1)$ and $\mathsf{DD}(\tau_2), \mathsf{Det}(\tau_1)$; By rule (v-pair) we have that $\Theta \vdash_g^f v_i^j : \tau$ and, by induction, we get $v_1^j = v_2^j$ from which $v_1 = (v_1^1, v_1^2) = (v_2^1, v_2^2) = v_2$. □

□

When we remove the condition on deterministic cryptography, we can prove $\mathsf{p}_{\mathsf{HH}}(v_1) = \mathsf{p}_{\mathsf{HH}}(v_2)$ instead of $v_1 = v_2$:

Lemma C.13. *Let $\Theta \vdash_g^f v_1 : \tau$, $\Theta \vdash_g^f v_2 : \tau$ and $g \neq \epsilon$, $\mathsf{DD}(\tau)$, with $\tau \neq \mathsf{K}_{\mathsf{HC}}(\tau')\ \kappa$.. Then $\mathsf{p}_{\mathsf{HH}}(v_1) = \mathsf{p}_{\mathsf{HH}}(v_2)$.*

Proof. Proof of this lemma is exactly as the one above apart from randomized encryptions. Notice, in fact, that we have removed the condition $\mathsf{Det}(\tau)$. The equality only holds above HH as, intuitively, at that level we can enter randomized encryptions and disregards confounders. Formally, we are in the case $\mathsf{enc}_{\delta'}\ \kappa$ with $\mathsf{cK}_{\mathsf{HC}}^R(\tau')\ \kappa$. By (v-enc), $v_i = \{\!|v_i', r_i|\!\}_k$ and $\Theta \vdash_g^f v_1' : \tau'$, $\Theta \vdash_g^f v_2' : \tau'$, and by induction we get $\mathsf{p}_{\mathsf{HH}}(v_1') = \mathsf{p}_{\mathsf{HH}}(v_2')$. Now it is sufficient to notice that $\mathsf{p}_{\mathsf{HH}}(\{\!|v_i', r_i|\!\}_k) = \{\!|\mathsf{p}_{\mathsf{HH}}(v_i'), \perp|\!\}_k$, which gives the thesis. □ □

Proposition C.3. *Let $\Theta \vdash_g^{f_i} v_i : \tau$, $\Theta \vdash_{g'}^{f_i} v_i' : \tau$, $g, g' \neq \epsilon$ and $\mathsf{CloseDD}^{\mathsf{det}}(\tau)$, with $\tau \neq \mathsf{K}_{\mathsf{HC}}(\tau')\ \kappa$. Then $v_1 = v_1'$ iff $v_2 = v_2'$.*

Proof. We prove the $\Rightarrow$ implication. The other direction is completely analogous. By lemma C.10(ii) we get that

$$\Theta \vdash_{g \cap \mathsf{gcore}_\tau}^{f_i} v_i : \tau \qquad \Theta \vdash_{g' \cap \mathsf{gcore}_\tau}^{f_i} v_i' : \tau$$

By $\mathsf{CloseDD}(\tau)$ we know that $\mathsf{Dep}(\tau) \subseteq \mathsf{IRs}(\tau)$ which implies $\mathsf{gcore}_\tau = \mathsf{gcoreIR}_\tau$. Thus

$$\Theta \vdash_{g \cap \mathsf{gcoreIR}_\tau}^{f_i} v_i : \tau \qquad \Theta \vdash_{g' \cap \mathsf{gcoreIR}_\tau}^{f_i} v_i' : \tau$$

By lemma C.11 we get that $v_1 = v_1'$ implies $g \cap \mathsf{gcoreIR}_\tau = g' \cap \mathsf{gcoreIR}_\tau = \tilde{g}$. Thus we get

$$\Theta \vdash_{\tilde{g}}^{f_2} v_2 : \tau \qquad \Theta \vdash_{\tilde{g}}^{f_2} v_2' : \tau$$

From lemma C.12 we finally get $v_2 = v_2'$. □ □

No well typed expression will ever return a confounder.

Lemma C.14. *Let $\Delta \vdash_g^f \mathsf{M}$. Then $\Delta \vdash e : \tau$ and $e \downarrow^{\mathsf{M}} v$ imply v satisfies assumption C.3.*

Proof. Easy by assumption C.3 and by induction on the structure of expressions. The only interesting cases are (i) randomized encryptions, which generate one confounder r in the value $\{\!|v, r|\!\}_k$ thus respecting the assumption, and (ii) randomized decryptions, which always disregard the confounder. Assumption C.2(1) ensures that no other expression will ever generate confounders. □ □

It is now useful to prove that expressions evaluated at level $\delta_C\mathsf{H}$ never return ciphertext, MACs or pairs.

Lemma C.15. *Let $\Delta \vdash^f_g \mathsf{M}$. Then $\Delta \vdash e : \delta_C\mathsf{H}$ and $e \downarrow^{\mathsf{M}} v$ imply $v = n$.*

Proof. Ciphertexts, MACs and pairs can never be typed as $\delta_C\mathsf{H}$. Notice, in particular, that MACs always have low-integrity, ciphertexts can be promoted only to low-integrity and pairs are not in the subtype relation. The fact that $n \notin C$ derives from assumptions C.2(1) and C.3 □ □

Notice that when we have $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\rho$ it might be the case that some hidden values in M_2 are not in the domain of ρ. This means that these values $\Box_v$ are untouched by ρ but we might of course extend ρ so that $\rho(\Box_v) = \Box_v$. We note $\bar{\rho}$ this 'closure' of ρ and we obviously have $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\bar{\rho}$. Notice that if item (3) of definition C.4 holds for ρ it trivially holds also for $\bar{\rho}$. In fact, by lemma C.6, since v is subvalue of a values stored in M_2, which is well-formed, we have that $\Theta \vdash^f_{g'} v : \tau'$. By corollary C.1 we obtain that $\Theta \vdash^f v : \tau'$ when $\mathsf{H} \sqsubseteq_I \mathcal{L}_I(\tau)$, as required by item (3) of definition C.4. In the next result, when $\ell = \mathsf{HL}$ we adopt the nonstandard semantics of equality check formalized in assumption C.2(3).

Proposition C.1 (Expression equivalence)**.** *Let $\mathsf{M}_1 \approx^\Delta_\ell \mathsf{M}_2$ and let $\Delta \vdash e : \tau$ with $\mathcal{L}(\tau) \sqsubseteq \ell$. Then, $e \downarrow^{\mathsf{M}_i} v_i$ and $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\rho$ and ρ f-respects Θ, imply $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho'$ such that $\bar{\rho} \subseteq \rho'$ and ρ' f-respects Θ.*

Proof. By induction on the structure of the expression e:

x

From $\Delta \vdash x : \tau$ we know, by (var) and (sub), that $\Delta(x) = \tau'$ with $\mathcal{L}(\tau') \sqsubseteq \mathcal{L}(\tau) \sqsubseteq \ell$. As a consequence the variable is observable, i.e., $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\rho$ implies $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\bar{\rho}$, which gives the thesis (with $\bar{\rho} = \rho'$).

$\mathsf{pair}(e_1, e_2)$

We have $\Delta \vdash \mathsf{pair}(e_1, e_2) : (\tau_1, \tau_2)$. By (pair) and since pairs are not in the subtype relation, we know that $\Delta \vdash e_1 : \tau_1$ and $\Delta \vdash e_2 : \tau_2$. It always holds $\mathcal{L}(\tau_i) \sqsubseteq \mathcal{L}(\tau_1, \tau_2) \sqsubseteq \ell$. Let $e_i \downarrow^{\mathsf{M}_j} v^i_j$. By induction on e_1 we can say $\mathsf{p}_\ell(v^1_1) = \mathsf{p}_\ell(v^1_2)\rho'_1$ and $\mathsf{p}_\ell(v^1_1) = \mathsf{p}_\ell(v^1_2)\bar{\rho}'_1$ with $\bar{\rho} \subseteq \rho'_1 \subseteq \bar{\rho}'_1$ and $\rho'_1, \bar{\rho}'_1$ f-respect Θ. Since $\bar{\rho}$ maps all the hidden values in M_2 we also have $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\bar{\rho}'_1$. By induction on e_2 we now obtain $\mathsf{p}_\ell(v^2_1) = \mathsf{p}_\ell(v^2_2)\rho'_2$ with $\bar{\rho}'_1 \subseteq \rho'_2$ and ρ'_2 f-respects Θ. Notice that, as above, we also have $\mathsf{p}_\ell(v^1_1) = \mathsf{p}_\ell(v^1_2)\rho'_2$, thus $\mathsf{p}_\ell((v^1_1, v^1_2)) = \mathsf{p}_\ell((v^2_1, v^2_2))\rho'_2$ and since $\bar{\rho} \subseteq \rho'_1 \subseteq \bar{\rho}'_1 \subseteq \rho'_2$, we obtain the thesis with $\rho' = \rho'_2$.

$\mathsf{fst}(e')$

We have $\Delta \vdash \mathsf{fst}(e') : \tau$ with $\mathcal{L}(\tau) \sqsubseteq \ell$. By (pair) and (sub) we know that $\Delta \vdash e' : (\tau_1, \tau_2)$ and $\tau_1 \leq \tau$ thus $\mathcal{L}(\tau_1) \sqsubseteq \mathcal{L}(\tau) \sqsubseteq \ell$. By assumption C.1 we

know that either $\mathcal{L}(\tau_1) = \mathcal{L}(\tau_2)$ (when $\mathsf{H} \sqsubseteq \mathcal{L}_I(\tau_1)$) or $\mathsf{H} \not\sqsubseteq \mathcal{L}_I(\tau_1), \mathcal{L}_I(\tau_2)$. In both cases we obtain that $\mathcal{L}((\tau_1, \tau_2)) \sqsubseteq \ell$. Let $e' \downarrow^{\mathsf{M}_j} v_j$. By induction we get $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho'_1$ such that $\bar{\rho} \subseteq \rho'_1$ and ρ'_1 f-respects Θ. From this we know that either $v_i = (v_i^1, v_i^2)$ or none of them is a pair, respectively giving $e \downarrow^{\mathsf{M}_j} v_j^1$ or $e \downarrow^{\mathsf{M}_j} \bot$. In the former case it is sufficient to observe that $\mathsf{p}_\ell(v_1^1) = \mathsf{p}_\ell(v_2^1)\rho'_1$ and we thus obtain the thesis with $\rho' = \rho'_1$. The latter case trivially give the thesis with $\rho' = \bar{\rho}$ since no new mapping is required to match $\bot$ with $\bot$.

$\mathsf{snd}(e')$

Exactly as above.

$\mathsf{enc}_x^\mu(e')$

The expression considered by this case is $e = \mathsf{enc}_x^\mu(e')$. From the type system we know that $\Delta(x) = \mathsf{cK}_{\delta'}^\mu(\tau')\ \kappa$, $\Delta \vdash e' : \tau'$ and $\Delta \vdash e : \mathsf{enc}_\delta\ \kappa$ with $\mathsf{enc}_\delta\ \kappa \leq \tau$ meaning that $\delta \sqsubseteq \mathcal{L}(\tau) \sqsubseteq \ell$. Let $e' \downarrow^{\mathsf{M}_j} v_j$ and $x \downarrow^{\mathsf{M}_j} v_j^k$. We proceed by cases depending on the typing rule used to get $\Delta \vdash e : \mathsf{enc}_\delta\ \kappa$:

(enc) we have that $\delta = \delta' \sqcup \mathcal{L}(\tau')$, meaning that $\delta', \mathcal{L}(\tau') \sqsubseteq \ell$, i.e., both the key and the subexpression are observable at ℓ. By induction we get $\mathsf{p}_\ell(v_1^k) = \mathsf{p}_\ell(v_2^k)\rho'_k$, meaning that either $v_i^k = k \in \mathcal{K}$ or none of them is a key, respectively giving $e \downarrow^{\mathsf{M}_j} \{\!|v_j|\!\}_k$ or $e \downarrow^{\mathsf{M}_j} \bot$. The latter case trivially give the thesis with $\rho' = \bar{\rho}$ since no new mapping is required to match $\bot$ with $\bot$. In the former case we just observe that, since the key is below ℓ, we have $\mathsf{p}_\ell(\{\!|v_j|\!\}_k) = \{\!|\mathsf{p}_\ell(v_j)|\!\}_k$. By induction (since $\mathcal{L}(\tau') \sqsubseteq \ell$) we get $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho'_1$ such that $\bar{\rho} \subseteq \rho'_1$ and ρ'_1 f-respect Θ, from which the thesis with $\rho' = \rho'_1$.

(enc-r) we have $\Delta(x) = \mathsf{cK}_{\mathsf{HC}}^R(\tau')\ \kappa$ and $e = \mathsf{enc}_x^R(e')$. By $\Delta \vdash_g^{f_i} \mathsf{M}_i$ we are guaranteed that $\mathsf{M}_i(x) = k \in \mathcal{K}_{\mathsf{HC}}$ where k is the unique key such that $\Theta(k) = \mathsf{cK}_{\mathsf{HC}}^R(\tau')\ \kappa$ (recall Θ is injective on high keys). Thus $e \downarrow^{\mathsf{M}_j} \{\!|v_j, r_j|\!\}_k$, with r_j extracted fresh. We have two cases: if $\mathsf{HC} \sqsubseteq \ell$ the key is observable and we have $\mathsf{p}_\ell(\{\!|v_j, r_j|\!\}_k) = \{\!|\mathsf{p}_\ell(v_j), \bot\ |\!\}_k$. By rule (enc-r) we know that $\delta = \mathsf{LC} \sqcup \mathcal{L}_I(\tau')$, meaning that $\mathcal{L}_I(\tau') \sqsubseteq_I \mathcal{L}_I(\delta) \sqsubseteq_I \mathcal{L}_I(\ell)$ and, since in this case we have $\mathsf{HC} \sqsubseteq \ell$, we obtain $\mathcal{L}(\tau') \sqsubseteq \ell$. We can thus apply induction and we get $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho'_1$ such that $\bar{\rho} \subseteq \rho'_1$ and ρ'_1 f-respect Θ, from which the thesis with $\rho' = \rho'_1$. If $\mathsf{HC} \not\sqsubseteq \ell$ the key is not observable and we have $\mathsf{p}_\ell(\{\!|v_j, r_j|\!\}_k) = \Box_{\{\!|v_j, r_j|\!\}_k}$. Thesis is trivially obtained with $\rho' = \bar{\rho} \cup [\Box_{\{\!|v_2, r_2|\!\}_k} \mapsto \Box_{\{\!|v_1, r_1|\!\}_k}]$. Even in this case the fact confounders are fresh trivially guarantees that the extension is possible and by proposition C.2 we get $\Theta \vdash_{g_\tau}^{f_i} \{\!|v_i, r_i|\!\}_k : \tau$, i.e., ρ' f-respects Θ.

(enc-d) We have $\Delta(x) = \mathsf{cK}_{\mathsf{HC}}(\tau')\ \kappa$ and $e = \mathsf{enc}_x(e')$. As above, by $\Delta \vdash_g^{f_i} \mathsf{M}_i$ we are guaranteed that $\mathsf{M}_i(x) = k \in \mathcal{K}_{\mathsf{HC}}$ where k is the unique key such that $\Theta(k) = \mathsf{cK}_{\mathsf{HC}}(\tau')\ \kappa$. Thus $e \downarrow^{\mathsf{M}_j} \{\!|v_j|\!\}_k$. We have two cases: if $\mathsf{HC} \sqsubseteq \ell$ the key is observable and we have $\mathsf{p}_\ell(\{\!|v_j|\!\}_k) = \{\!|\mathsf{p}_\ell(v_j)|\!\}_k$. As for the above case, by rule (enc-d) we know that $\delta = \mathsf{LC} \sqcup \mathcal{L}_I(\tau')$ from which we derive $\mathcal{L}(\tau') \sqsubseteq \ell$.

Thus, by induction we get $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho'_1$ such that $\bar{\rho} \subseteq \rho'_1$ and ρ'_1 f-respect Θ, from which the thesis with $\rho' = \rho'_1$. if $\mathsf{HC} \not\sqsubseteq \ell$ the key is not observable and we have $\mathsf{p}_\ell(\{\!|v_j|\!\}_k) = \Box_{\{\!|v_j|\!\}_k}$. If $\Box_{\{\!|v_2|\!\}_k} \notin dom(\bar{\rho})$ and $\Box_{\{\!|v_1|\!\}_k} \notin img(\bar{\rho})$ we trivially obtain the thesis with $\rho' = \bar{\rho} \cup [\Box_{\{\!|v_2|\!\}_k} \mapsto \Box_{\{\!|v_1|\!\}_k}]$ (notice that by proposition C.2 we get $\Theta \vdash^{f_i}_{g_\tau} \{\!|v_i|\!\}_k : \tau$, i.e., ρ' f-respects Θ). If, instead, $\Box_{\{\!|v_2|\!\}_k} \in dom(\bar{\rho})$ or $\Box_{\{\!|v_1|\!\}_k} \in img(\bar{\rho})$ we show that $\bar{\rho}(\Box_{\{\!|v_2|\!\}_k}) = \Box_{\{\!|v_1|\!\}_k}$, i.e., the needed mapping is already in $\bar{\rho}$ and we thus have the thesis by simply taking $\rho' = \bar{\rho}$. Assume, then, that $\Box_{\{\!|v_2|\!\}_k} \in dom(\bar{\rho})$. By the constraints on hidden value substitutions we know that $\bar{\rho}(\Box_{\{\!|v_2|\!\}_k}) = \Box_{\{\!|v'|\!\}_k}$, since the mapping is required to respect keys. By the fact that $\bar{\rho}$ *f-respects* Θ we know that $\exists g'$ such that $\Theta \vdash^{f_i}_{g'} v_2 : \tau'$ and $\Theta \vdash^{f_i}_{g'} v' : \tau'$. By lemma C.9 and since $\mathsf{CloseDD}(\tau')$ and, by assumption C.1(3), $\tau' \neq \mathsf{K}_{\mathsf{HC}}(\tau)\ \kappa$ we get that $\mathcal{L}_I(\tau') \sqsubset_I \mathsf{H}$, thus $g' \neq \epsilon$. Since $\Delta \vdash e' : \tau'$ and $\Delta \vdash^{f_i}_{g} \mathsf{M}_i$, by proposition C.2 we get $\Theta \vdash^{f_i}_{g} v_i : \tau'$. By $\mathsf{CloseDD}^{\mathsf{det}}(\tau')$ and proposition C.3 we directly obtain that $v_1 = v'$, showing that the new mapping is the same already in $\bar{\rho}$. (The other direction is proved similarly.)

$\mathsf{mac}_x(e')$

The expression considered by this case is $e = \mathsf{mac}_x(e')$. From the type system we know that $\Delta(x) = \mathsf{cK}^{\mu}_{\delta'_C\delta'_I}(\tau')\ \kappa$, $\Delta \vdash e' : \tau'$ and $\Delta \vdash e : \mathsf{LL} \sqcup \mathcal{L}(\tau')$ with $\mathsf{LL} \sqcup \mathcal{L}(\tau') \leq \tau$ meaning that $\mathcal{L}(\tau') \sqsubseteq \mathcal{L}(\tau) \sqsubseteq \ell$, i.e., the subexpression is observable at ℓ. Let $e' \downarrow^{\mathsf{M}_j} v_j$ and $x \downarrow^{\mathsf{M}_j} v^k_j$. We proceed by cases depending on level $\delta'_C\delta'_I$ of the key.

If $\delta'_C\delta'_I = \mathsf{LL}$, we have that also the key is observable at ℓ. By induction we get $\mathsf{p}_\ell(v^k_1) = \mathsf{p}_\ell(v^k_2)\rho'_k$, meaning that either $v^k_i = k \in \mathcal{K}$ or none of them is a key, respectively giving $e \downarrow^{\mathsf{M}_j} \langle v_j\rangle_k$ or $e \downarrow^{\mathsf{M}_j} \bot$. The latter case trivially give the thesis with $\rho' = \bar{\rho}$ since no new mapping is required to match $\bot$ with $\bot$. In the former case we just observe that $\mathsf{p}_\ell(\langle v_j\rangle_k) = \langle \mathsf{p}_\ell(v_j)\rangle_k$. By induction we get $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho'_1$ such that $\bar{\rho} \subseteq \rho'_1$ and ρ'_1 f-respect Θ, from which the thesis with $\rho' = \rho'_1$.

If $\delta'_C\delta'_I = \mathsf{HC}$, the key might be not observable at ℓ. However, since the key is high and the memories are well-formed, by $\Delta \vdash^{f_i}_{g} \mathsf{M}_i$ we are guaranteed that $\mathsf{M}_i(x) = k \in \mathcal{K}_{\mathsf{HC}}$ where k is the unique key such that $\Theta(k) = \mathsf{cK}^{R}_{\mathsf{HC}}(\tau')\ \kappa$ (recall Θ is injective on high keys). Thus $e \downarrow^{\mathsf{M}_j} \langle v_j\rangle_k$. Now observe that $\mathsf{p}_\ell(\langle v_j\rangle_k) = \langle \mathsf{p}_\ell(v_j)\rangle_k$ (this is independent of the level of the key). By induction, as above, we get $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho'_1$ such that $\bar{\rho} \subseteq \rho'_1$ and ρ'_1 f-respect Θ, from which the thesis with $\rho' = \rho'_1$.

$\mathsf{dec}_x(e')$

The expression is $e = \mathsf{dec}_x(e')$ and $\Delta \vdash e : \tau$ with $\mathcal{L}(\tau) \sqsubseteq \ell$. From the type system we know that $\Delta(x) = \mathsf{cK}^{\mu}_{\delta'}(\tau')\ \kappa$, $\Delta \vdash e' : \mathsf{enc}_{\delta''}\ \kappa$. Let $e' \downarrow^{\mathsf{M}_j} v'_j$ and $x \downarrow^{\mathsf{M}_j} v^k_j$. In case of rule (dec), we know that $\delta' \sqcup \delta'' \leq \tau$ meaning

that $\delta', \delta'' \sqsubseteq \ell$, i.e., both the key and the subexpression are observable at ℓ. For (dec-μ) we know that $\tau' \leq \tau$. By the fact $\mathcal{L}_C(\tau) = \mathsf{H}$ we know that $\ell = \mathsf{H}\ell_I$ thus the key is observable, and by the typing rule we know that $\mathcal{L}_I(\delta'') = \mathcal{L}_I(\delta') \sqcup \mathcal{L}_I(\tau') = \mathsf{C} \sqcup \mathcal{L}_I(\tau')$ which implies $\mathcal{L}_I(\delta'') \sqsubseteq_I \ell_I$. Thus $\delta'' \sqsubseteq \ell$ meaning that also the subexpression is observable. By induction we get $\mathsf{p}_\ell(v_1^k) = \mathsf{p}_\ell(v_2^k)\rho'_k$ and $\mathsf{p}_\ell(v'_1) = \mathsf{p}_\ell(v'_2)\rho'_1$ meaning that either $v_i^k = k \in \mathcal{K}$ or none of them is a key, and either $v'_i = \{\!|v_j|\!\}_k$ or none of them is a ciphertext based on key k. When $v_i^k \notin \mathcal{K}$ or $v'_i \neq \{\!|v_j|\!\}_k$ we obtain $e \downarrow^{\mathsf{M}_j} \bot$. We easily get the thesis with $\rho' = \bar{\rho}$ since no new mapping is required to match $\bot$ with $\bot$. Otherwise, we get $e \downarrow^{\mathsf{M}_j} v_j$. Observe that, since the key is below ℓ, we have $\mathsf{p}_\ell(\{\!|v_j|\!\}_k) = \{\!|\mathsf{p}_\ell(v_j)|\!\}_k$. By the above induction we had that $\bar{\rho} \subseteq \rho'_1$ and ρ'_1 f-respect Θ, from which the thesis with $\rho' = \rho'_1$.

e_1 op e_2
: $\Delta \vdash e_1$ op $e_2 : \delta$. By (op) we know that $\Delta \vdash e_1 : \delta'$ and $\Delta \vdash e_2 : \delta'$ with $\delta' \sqsubseteq \delta \sqsubseteq \ell$, thus both subsxpressions are observable. Let $e_i \downarrow^{\mathsf{M}_j} v_j^i$. By induction on e_1 we can say $\mathsf{p}_\ell(v_1^1) = \mathsf{p}_\ell(v_2^1)\rho'_1$ and $\mathsf{p}_\ell(v_1^1) = \mathsf{p}_\ell(v_2^1)\bar{\rho}'_1$ with $\bar{\rho} \subseteq \rho'_1 \subseteq \bar{\rho}'_1$ and $\rho'_1, \bar{\rho}'_1$ f-respect Θ. Since $\bar{\rho}$ maps all the hidden values in M_2 we also have $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\bar{\rho}'_1$. By induction on e_2 we now obtain $\mathsf{p}_\ell(v_1^2) = \mathsf{p}_\ell(v_2^2)\rho'_2$ with $\bar{\rho}'_1 \subseteq \rho'_2$ and ρ'_2 f-respects Θ. Notice that, as above, we also have $\mathsf{p}_\ell(v_1^1) = \mathsf{p}_\ell(v_2^1)\rho'_2$. We consider two cases: if e_1 op e_2 is the equality test $e_1 = e_2$ by lemma C.14 and C.15 we know that for $\delta \sqsubseteq \mathsf{HH}$ we have $v_j^i = n_j^i \notin C$. Thus, easily, $v_j^1 = v_j^2$ iff $\mathsf{p}_\ell(v_j^1) = \mathsf{p}_\ell(v_j^2)$. By lemma C.14 we have that the same holds for $\delta = \mathsf{LL}$, since no confounders will be abstracted in the patterns (since high keys are not known) and hidden values are decorated by the whole encrypted message. When $\delta = \mathsf{HL}$ we exploit assumption C.2(3). We obtain that $v_1^1 = v_1^2$ iff $v_2^1 = v_2^2$. Thus we get the thesis with $\rho' = \rho'_2$. For all the other generic expressions, by assumption C.2(2), we know that e_1 op $e_2 \not\downarrow^{\mathsf{M}} \bot$ implies $e_i \downarrow^{\mathsf{M}} n_i$ with $n_i \notin \mathcal{K} \cup C$. By $\mathsf{p}_\ell(v_1^i) = \mathsf{p}_\ell(v_2^i)\rho'_2$ we obtain that $e_i \downarrow^{\mathsf{M}_1} n_i$ iff $e_i \downarrow^{\mathsf{M}_2} n_i$, $(n_i \notin \mathcal{K} \cup C)$ meaning that the expression e_1 op e_2 either evaluate to $\bot$ or to the very same n in both memories. We thus get the thesis with $\rho' = \rho'_2$. □

□

Corollary C.2. *Let* $\mathsf{M}_1 \approx_\ell^\Delta \mathsf{M}_2$ *and let* $\Delta \vdash e : \tau$ *and* $e \downarrow^{\mathsf{M}_i} v_i$. *If* $\mathcal{L}(\tau) \sqsubseteq \ell$ *or* $\mathcal{L}(\Delta(x)) \not\sqsubseteq \ell$ *then* $\mathsf{M}_1[x \mapsto v_i] \approx_\ell^\Delta \mathsf{M}_2[x \mapsto v_i]$.

Proof. If $\mathcal{L}(\tau) \sqsubseteq \ell$, by $\mathsf{M}_1 \approx_\ell^\Delta \mathsf{M}_2$ we know that there exists ρ such that $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\rho$ and ρ f-respects Θ. Now, by proposition C.1 we directly have $\mathsf{p}_\ell(v_1) = \mathsf{p}_\ell(v_2)\rho'$ such that $\bar{\rho} \subseteq \rho'$ and ρ' f-respects Θ. Of course we also have $\mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\rho'$ and, consequently, $\mathsf{p}_\ell(\mathsf{M}_1)[x \mapsto v_1] = \mathsf{p}_\ell(\mathsf{M}_2)[x \mapsto v_2]\rho'$. If, instead, $\mathcal{L}(\Delta(x)) \not\sqsubseteq \ell$ then the variable x is above the observation level and we trivially have $\mathsf{M}_i|_\ell = \mathsf{M}_i[x \mapsto v_i]|_\ell$ which gives $\mathsf{p}_\ell(\mathsf{M}_1)[x \mapsto v_1] = \mathsf{p}_\ell(\mathsf{M}_1) = \mathsf{p}_\ell(\mathsf{M}_2)\rho =$

$\mathsf{p}_\ell(\mathsf{M}_2)[x \mapsto v_2]\rho$. We still need to prove that $\Delta \vdash_g^{f_i} \mathsf{M}_i[x \mapsto v_i]$. This is directly achieved by proposition C.2: $\Delta \vdash_g^{f_i} \mathsf{M}_i$ and $\Delta \vdash e : \tau$ imply $\Theta \vdash_{g_\tau}^{f} v_i : \tau$ which implies $\Delta \vdash_g^{f_i} \mathsf{M}_i[x \mapsto v_i]$. □ □

Lemma C.16. (**Confinement**) *If $\Delta, pc \vdash \mathsf{c}$ then for every variable x assigned to in c and such that $\Delta(x) = \tau$ it holds that $pc \sqsubseteq \mathcal{L}(\tau) \sqcup \mathsf{LH}$.*

Proof. We proceed by induction on the strucuture of c.

skip

The command does not assign to any variable thus the lemma trivially holds.

$x := e$

This command can be typed by two different rules: (assign) and (declassify). The former requires $pc \sqsubseteq \mathcal{L}(\tau) \sqcup \mathsf{LH}$ while the latter $pc \sqsubseteq \mathcal{L}(\tau)$, which directly give the thesis.

$\mathsf{c}_1; \mathsf{c}_2$

Since $\Delta, pc \vdash \mathsf{c}_i$, the thesis follows directly by induction on c_1 and c_2.

if b then c_1 else c_2

Two different rules may be used to type this command.

(if)

The typing rule assures that the expression b types τ' and $\Delta, \mathcal{L}(\tau') \sqcup pc \vdash \mathsf{c}_i$. By induction on c_1 and c_2 it holds that for every variable x assigned to in c_1, c_2 such that $\Delta(x) = \tau$ $\mathcal{L}(\tau') \sqcup pc \sqsubseteq \mathcal{L}(\tau) \sqcup \mathsf{LH}$, thus $pc \sqsubseteq \mathcal{L}(\tau) \sqcup \mathsf{LH}$.

(if-MAC)

For variable y we directly have $pc \sqsubseteq \mathcal{L}(\tau) \sqcup \mathsf{LH}$. Since $\Delta, pc \vdash \mathsf{c}_1$ and $\Delta, pc \vdash \mathsf{c}_2$, by induction on the two commands we get the thesis.

while e do c

This case is analogous to the former one of the if command. □

□

We now prove noninterference on all the level but LH. In fact, on LH, the property does not hold since, intuitively, randomized but high-equivalent messages (i.e., messages encrypted with high keys differing only for the values of confounders) might be distinguished by LH users, who know no keys, via traffic analysis. Once more, this is like the attacker fouling himself, by 'incompetently' sending messages on which he can perform traffic analysis. We disregard this uninteresting behaviour by just requiring $\ell \sqsupset \mathsf{LH}$. As for expression equivalence, in the next result, when $\ell = \mathsf{HL}$ we adopt the nonstandard semantics of equality check formalized in assumption C.2(3).

Theorem C.2 (Noninterference). *Let* c *be a program which does not contain any declassification statement. If* $\Delta, pc \vdash \mathsf{c}$ *then* c *satisfies noninterference, i.e.,* $\forall\ \ell \sqsupseteq \mathsf{LH}, \mathsf{M}_1, \mathsf{M}_2$. $\mathsf{M}_1 \approx_\ell^\Delta \mathsf{M}_2$ *implies* $\langle \mathsf{M}_1, \mathsf{c}\rangle \approx_\ell^\Delta \langle \mathsf{M}_2, \mathsf{c}\rangle$.

Proof. Recall that $\langle \mathsf{M}_1, \mathsf{c}\rangle \approx_\ell^\Delta \langle \mathsf{M}_2, \mathsf{c}\rangle$ denotes weak-indistinguishability, i.e., whenever $\langle \mathsf{M}_i, \mathsf{c}\rangle \Rightarrow \mathsf{M}'_i$ we have $\mathsf{M}'_1 \approx_\ell^\Delta \mathsf{M}'_2$. We proceed by induction on derivation length of $\langle \mathsf{M}_i, \mathsf{c}\rangle \Rightarrow \mathsf{M}'_i$.

Base case, length 1:

[***skip***]

Since $\langle \mathsf{M}_i, \mathsf{skip}\rangle \Rightarrow \mathsf{M}_i$ we have nothing to prove.

[***assign***]

We have $\langle \mathsf{M}_i, x := e\rangle \Rightarrow \mathsf{M}_i[x \mapsto v_i] = \mathsf{M}'_i$ given that $e \downarrow^{\mathsf{M}_i} v_i$. If rule (assign) have been used then $\Delta(x) = \tau$, $\Delta \vdash e : \tau$. Notice that either $\mathcal{L}(\tau) \sqsubseteq \ell$ or $\ell \sqsubset \mathcal{L}(\tau) = \mathcal{L}(\Delta(x))$. Corollary C.2 directly gives the thesis. By hypothesis, we do not have declassify commands, thus rule (declassify) cannot have been used to type the assignment.

Inductive case, length n. We consider the last rule applied:

[***seq***]

We have $\langle \mathsf{M}_i, \mathsf{c}_1; \mathsf{c}_2\rangle \Rightarrow \mathsf{M}'_i$ since $\langle \mathsf{M}_i, \mathsf{c}_1\rangle \Rightarrow \mathsf{M}''_i$ and also $\langle \mathsf{M}''_i, \mathsf{c}_2\rangle \Rightarrow \mathsf{M}'_i$. By rule (seq) we have that $\Delta, pc \vdash \mathsf{c}_i$. By induction on c_1 we have $\mathsf{M}''_1 \approx_\ell^\Delta \mathsf{M}''_2$ and by induction on c_2 we obtain $\mathsf{M}'_1 \approx_\ell^\Delta \mathsf{M}'_2$.

[***ift***], [***iff***]

Let $\langle \mathsf{M}_i, \mathsf{if}\ b\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\rangle \Rightarrow \mathsf{M}'_i$. The typing can be made by rules (if) and (if-MAC).

For rule (if) it holds that $\Delta \vdash b : \tau$, $\Delta, \mathcal{L}(\tau) \sqcup pc \vdash \mathsf{c}_1$ and $\Delta, \mathcal{L}(\tau) \sqcup pc \vdash \mathsf{c}_2$. If $\mathcal{L}(\tau) \sqsubseteq \ell$ then by proposition C.1 the boolean expression b necessarily evaluates to two identical boolean values (or $\bot$) so the same branch will be followed. Suppose $e \downarrow^{\mathsf{M}_i} \mathsf{true}$, we have that $\langle \mathsf{M}_i, \mathsf{c}_1\rangle \Rightarrow \mathsf{M}'_i$, thus by induction on c_1, $\mathsf{M}'_1 \approx_\ell^\Delta \mathsf{M}'_2$ (analogously for $\mathsf{false}/\bot$ and c_2). If instead $\ell \sqsubset \mathcal{L}(\tau)$ by lemma C.16 for every variable x assigned by c_1 and c_2 such that $\Delta(x) = \tau'$ it holds that $\mathcal{L}(\tau) \sqcup pc \sqsubseteq \mathcal{L}(\tau') \sqcup \mathsf{LH}$, thus $\ell \sqsubset \mathcal{L}(\tau') \sqcup \mathsf{LH}$ which implies $\ell \sqsubset \mathcal{L}(\tau')$ (since ℓ is at least LH). Intuitively, all the assignments performed by the branches are above the level of observation. By corollary C.2 we directly obtain $\mathsf{M}'_1 \approx_\ell^\Delta \mathsf{M}'_2$.

In case (if-MAC) has been applied then the command has the following form:

$$\mathsf{if}\ \mathsf{mac}_x(z, e) = e'\ \mathsf{then}\ (y := e; \mathsf{c}_1)\ \mathsf{else}\ \mathsf{c}_2; \bot_{\mathsf{MAC}}$$

and it must be that $\Delta(x) = \mathsf{mK_{HC}}(\mathsf{L[D]}, \tau)$, $\Delta \vdash z : \mathsf{L[D]}$, $\Delta \vdash e : \mathsf{LL}$, $\Delta(y) = \tau$, $\Delta \vdash e' : \mathsf{LL}$. Moreover $\Delta, pc \vdash \mathsf{c}_1$ and $\Delta, pc \vdash \mathsf{c}_2$. Note that if one of the execution takes the else branch it will not terminate, giving $\langle \mathsf{M}_1, \mathsf{c}\rangle \approx_\ell^\Delta \langle \mathsf{M}_2, \mathsf{c}\rangle$. We thus consider the case in which both executions take the if branch. Let $e \downarrow^{\mathsf{M}_i} v_i$, we consider different cases depending on level ℓ:

Let $\mathsf{LL} \sqsubseteq \ell$, by corollary C.2 we get that $\mathsf{M}_1[y \mapsto v_i] \approx_\ell^\Delta \mathsf{M}_2[y \mapsto v_i]$.

If $\mathsf{LL} \not\sqsubseteq \ell$ By hypothesis $\ell \sqsupset \mathsf{LH}$ thus the only possibility is $\ell = \mathsf{HH}$: as in the proof of theorem C.1 (case [*ift*]) we get that $\mathsf{mac}_x(z, e) \downarrow^{\mathsf{M}_i} \langle v_i, v_i'\rangle_k$ and $e' \downarrow^{\mathsf{M}_i} \langle v_i, v_i'\rangle_k$ and $\Theta \vdash_g^{f_i} v_i' : \tau$. Since memories are well-formed and, by lemma C.9 we have that $\mathcal{L}_I(\tau) \sqsubset_I \mathsf{H}$, we know that $\Theta \vdash_g^{f_i} \mathsf{M}_i(y) : \tau$. Now by lemma C.13 we have that $\mathsf{p_{HH}}(v_i') = \mathsf{p_{HH}}(\mathsf{M}_i(y))$, meaning that the assignment does not change in any way the equality of the two memories, i.e., $\mathsf{M}_1[y \mapsto v_i] \approx_{\mathsf{HH}}^\Delta \mathsf{M}_2[y \mapsto v_i]$.

We have thus proved that, for all $\ell \sqsupset \mathsf{LH}$, $\mathsf{M}_1[y \mapsto v_i] \approx_\ell^\Delta \mathsf{M}_2[y \mapsto v_i]$. Now, by induction on c_1 we get the thesis.

[*whilet*], [*whilef*]

For rule (while) it holds that $\Delta \vdash b : \tau$, $\Delta, \mathcal{L}(\tau) \sqcup pc \vdash \mathsf{c}$. If $\mathcal{L}(\tau) \sqsubseteq \ell$ then by proposition C.1 the boolean expression b evaluates to two identical boolean values (or $\perp$) so the same branch will be followed. Suppose $e \downarrow^{\mathsf{M}_i} \mathsf{true}$, we have that $\langle \mathsf{M}_i, \mathsf{c}\rangle \Rightarrow \mathsf{M}_i''$ and $\langle \mathsf{M}_i'', \mathsf{while}\ e\ \mathsf{do}\ \mathsf{c}\rangle \Rightarrow \mathsf{M}_i'$, thus by induction (on the length of the derivation), $\mathsf{M}_1'' \approx_\ell^\Delta \mathsf{M}_2''$ and then $\mathsf{M}_1' \approx_\ell^\Delta \mathsf{M}_2'$. If instead $\ell \sqsubset \mathcal{L}(\tau)$ by lemma C.16 for every variable x assigned by c such that $\Delta(x) = \tau'$ it holds that $\mathcal{L}(\tau) \sqcup pc \sqsubseteq \mathcal{L}(\tau') \sqcup \mathsf{LH}$, thus $\ell \sqsubset \mathcal{L}(\tau') \sqcup \mathsf{LH}$ which implies $\ell \sqsubset \mathcal{L}(\tau')$ (since ℓ is at least LH). Intuitively, all the assignments performed by the loops are above the level of observation. By corollary C.2 we directly obtain $\mathsf{M}_1' \approx_\ell^\Delta \mathsf{M}_2'$. When the guard is false the result trivially holds, since memories remain untouched. □

□

When we re-introduce declassification statements, noninterference still holds for $\ell = \mathsf{HH}$ as, as expected, integrity is not broken by declasssifying information. This is proved in the following theorem:

Theorem C.3. *If* $\Delta, pc \vdash \mathsf{c}$ *then* $\forall\ \mathsf{M}_1, \mathsf{M}_2$ *such that* $\mathsf{M}_1 \approx_{\mathsf{HH}}^\Delta \mathsf{M}_2$ *it holds* $\langle \mathsf{M}_1, \mathsf{c}\rangle \approx_{\mathsf{HH}}^\Delta \langle \mathsf{M}_2, \mathsf{c}\rangle$

Proof. The proof of this theorem is a simple extension of the one of theorem C.2, instantiated with $\ell = \mathsf{HH}$, to programs which contain declassification. The only case that we need to extend is the assignment, since it is where declassification may occur: We have $\langle \mathsf{M}_i, x := \mathsf{declassify}(e')\rangle \Rightarrow \mathsf{M}_i[x \mapsto v_i] = \mathsf{M}_i'$ given that $e \downarrow^{\mathsf{M}_i} v_i$.

From the type system it follows $\Delta \vdash e' : \delta'_C\mathsf{H} \sqsubseteq \mathsf{HH}$. By corollary C.2 we obtain the thesis. □

□

We can now state our final results on robustness. We will consider programs that assign declassified data to special variables assigned only once. This can be easily achieved syntactically , e.g., by using one different variable for each declassification statement (which we label for clarity), i.e., $x_1 := \mathsf{declassify_1}(e_1), \ldots, x_m := \mathsf{declassify_m}(e_m)$, and avoiding to place declassifications inside while loops. These special variables are nowhere else assigned. We call this class of programs *CD-programs.*

Lemma C.17 (CD-programs)**.** *In a CD-program* c*, if* $\langle \mathsf{M}_1, x := \mathsf{declassify}(e')\rangle \Rightarrow \mathsf{M}_1[x \mapsto v_1]$ *and* $\langle \mathsf{M}_2, y := e\rangle \Rightarrow \mathsf{M}_2[y \mapsto v_2]$ *occur in the derivation of* $\langle \mathsf{M}, \mathsf{c}\rangle \Rightarrow \mathsf{M}'$ *then* $x \neq y$*.*

Proof. Easy by induction on the structure of commands, by observing that the only command re-executing the same syntactic portion of a program is the while-loop. □ □

Let $\Delta, pc \vdash \mathsf{c}$. We now prove that any value v declassified during the computation of c on a well-formed memory is an atomic name.

Lemma C.18. *Let* $\Delta, pc \vdash \mathsf{c}$ *and* $\Delta \vdash^f_g \mathsf{M}$*. Then any* $\langle \mathsf{M}_1, x := \mathsf{declassify}(e')\rangle \Rightarrow \mathsf{M}_1[x \mapsto v_1]$ *occurring in the derivation of* $\langle \mathsf{M}, \mathsf{c}\rangle \Rightarrow \mathsf{M}'$ *is such that* $v_1 = n$ *and* $n \notin C$*.*

Proof. It is sufficient to notice that the declassified value is expected to be of type $\delta_C\mathsf{H}$ and apply lemma C.15. □ □

Information leakage in clearly declassifying programs can be always observed by inspecting the equality of declassifying variables as proved in the following:

Lemma C.19. *Let* c *be a CD-program on variables* $x_1, \ldots, x_k$*. If* $\Delta, pc \vdash \mathsf{c}$*,* $\mathsf{M}_1 \approx^\Delta_{\mathsf{LL}} \mathsf{M}_2$*,* $\langle \mathsf{M}_i, \mathsf{c}\rangle \Rightarrow \mathsf{M}'_i$ *and* $\mathsf{M}'_1 \not\approx^\Delta_{\mathsf{LL}} \mathsf{M}'_2$ *then* $\exists i$ *such that* $\mathsf{M}'_1(x_i) = n \neq n' = \mathsf{M}'_2(x_i)$*.*

Proof. Lemma C.18 already proves that declassified values are all atomic names. We now proceed by induction on the number of declassifying variables in c.

Base case (no declassification): We have $\mathsf{M}'_1 \approx^\Delta_{\mathsf{LL}} \mathsf{M}'_2$ by theorem C.2, thus we have nothing to prove.

Inductive case: Assume we have $x_i, \ldots, x_m$ declassifying variables. Let $\mathsf{M}'_1 \not\approx^\Delta_{\mathsf{LL}} \mathsf{M}'_2$. If $\mathsf{M}'_1(x_m) \neq \mathsf{M}'_2(x_m)$ we are done. Otherwise it will be $\mathsf{M}'_1(x_m) = \mathsf{M}'_2(x_m) = n$. We replace $x_m := \mathsf{declassify_m}(e_m)$ with $x_m := n$ obtaining c'' and re-execute it, i.e., $\langle \mathsf{M}_i, \mathsf{c}''\rangle \Rightarrow \mathsf{M}''_i$. By lemma C.17 we know that the above one was the only assignment done to x_m during the derivation, thus $\mathsf{M}'_i = \mathsf{M}''_i$, thus $\mathsf{M}''_1 \not\approx^\Delta_{\mathsf{LL}} \mathsf{M}''_2$. By induction we now have that $\exists i \in [1, m-1]$ such that $\mathsf{M}'(x_i) = \mathsf{M}''_1(x_i) = n \neq n' = \mathsf{M}''_2(x_i) = \mathsf{M}'_2(x_i)$. □ □

Lemma C.20. *Let* $\mathsf{c}_1;\mathsf{c}_2$ *be a CD-program such that* $\Delta, pc \vdash \mathsf{c}_1;\mathsf{c}_2$. *Then* $\langle \mathsf{M}_1, \mathsf{c}_1;\mathsf{c}_2\rangle \simeq^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}_1;\mathsf{c}_2\rangle$ *implies* $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \simeq^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}_1\rangle$.

Proof. $\langle \mathsf{M}_1, \mathsf{c}_1;\mathsf{c}_2\rangle \simeq^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}_1;\mathsf{c}_2\rangle$ means that $\langle \mathsf{M}_i, \mathsf{c}_1;\mathsf{c}_2\rangle \Rightarrow \mathsf{M}'_i$ and $\mathsf{M}'_1 \approx^{\Delta}_{\mathsf{LL}} \mathsf{M}'_2$. Notice, in fact, that $\simeq^{\Delta}_{\mathsf{LL}}$ is strong and requires both executions to terminate. We know that $\langle \mathsf{M}_i, \mathsf{c}_1\rangle \Rightarrow \mathsf{M}''_i$ and $\langle \mathsf{M}''_i, \mathsf{c}_2\rangle \Rightarrow \mathsf{M}'_i$. Since $\mathsf{c}_1;\mathsf{c}_2$ is a CD-program, so are c_1 and c_2, as the condition is purely syntactic. Assume, by contradiction, that $\mathsf{M}''_1 \not\approx^{\Delta}_{\mathsf{LL}} \mathsf{M}''_2$. By lemma C.19 there exist a declassifying variable x such that $\mathsf{M}''_1(x) = n \neq n' = \mathsf{M}''_2(x)$. By lemma C.17 we know that x will never be assigned again in $\langle \mathsf{M}''_i, \mathsf{c}_2\rangle \Rightarrow \mathsf{M}'_i$, thus giving $\mathsf{M}'_1(x) = n \neq n' = \mathsf{M}'_2(x)$ and the contradiction $\mathsf{M}'_1 \not\approx^{\Delta}_{\mathsf{LL}} \mathsf{M}'_2$. □ □

Next result finally proves robustness of well-typed CD-programs. Notice that it only adopts the standard semantics of equality check and is thus not based on assumption C.2(3).

Theorem C.4 (Robustness)**.** *If a CD-program* c *is such that* $\Delta, pc \vdash \mathsf{c}$ *then* c *satisfies robustness, i.e.,* $\forall \mathsf{M}_1, \mathsf{M}_2, \mathsf{M}'_1, \mathsf{M}'_2$ *such that* $\mathsf{M}_1 \approx^{\Delta}_{\mathsf{LL}} \mathsf{M}_2$, $\mathsf{M}'_1 \approx^{\Delta}_{\mathsf{LL}} \mathsf{M}'_2$ *and* $\mathsf{M}_i \approx^{\Delta}_{\mathsf{HH}} \mathsf{M}'_i$ *it holds*

$$\langle \mathsf{M}_1, \mathsf{c}\rangle \simeq^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}\rangle \textit{ implies } \langle \mathsf{M}'_1, \mathsf{c}\rangle \approx^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}\rangle$$

Proof. The proof follows by induction on the structure of the command c.

skip

By $\langle \mathsf{M}'_i, \mathsf{skip}\rangle \Rightarrow \mathsf{M}'_i$ we directly get $\langle \mathsf{M}'_1, \mathsf{c}\rangle \approx^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}\rangle$.

$x := e$

If the expression e is not a declassification then by Theorem C.2, c satisfies noninterference, thus $\langle \mathsf{M}'_1, x := e\rangle \approx^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}'_2, x := e\rangle$.

Suppose $e = \mathsf{declassify}(e')$. The type system states that $\Delta(x) = \delta_C\mathsf{H}$, $\Delta \vdash e' : \delta'_C\mathsf{H}$ and $pc \sqsubseteq \delta_C\mathsf{H}$. Let $e' \downarrow^{\mathsf{M}_i} v_i$ and $e' \downarrow^{\mathsf{M}'_i} v'_i$. By corollary C.2, since $\mathsf{M}_i \approx^{\Delta}_{\mathsf{HH}} \mathsf{M}'_i$ and $\delta'_C\mathsf{H} \sqsubseteq \mathsf{HH}$, we get $\mathsf{M}_i[x \mapsto v_i] \approx^{\Delta}_{\mathsf{HH}} \mathsf{M}'_i[x \mapsto v'_i]$. By lemma C.18 we know that all these values are atomic and not confounders, thus we get $v_i = v'_i = n_i$. Now, $\langle \mathsf{M}_1, x := e\rangle \simeq^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}_2, x := e\rangle$ implies $\mathsf{M}_1[x \mapsto v_1] \approx^{\Delta}_{\mathsf{LL}} \mathsf{M}_2[x \mapsto v_2]$, meaning $v_1 = v_2$. We get $v'_1 = v_1 = v_2 = v'_2$. Thus $\mathsf{M}'_1[x \mapsto v'_1] \approx^{\Delta}_{\mathsf{LL}} \mathsf{M}'_2[x \mapsto v'_2]$, giving the thesis $\langle \mathsf{M}'_1, x := e\rangle \approx^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}'_2, x := e\rangle$.

if b then c_1 else c_2

If rule (if) has been used to type the command, then $\Delta \vdash b : \tau$ and $\Delta, \mathcal{L}(\tau) \sqcup pc \vdash \mathsf{c}_i$.

If $\mathcal{L}_I(\tau) = \mathsf{L}$ then $\mathcal{L}(\tau) \sqcup pc \not\sqsubseteq \delta_C\mathsf{H}$ thus, by lemma C.16, no declassification may occur in both c_1 and c_2. By Theorem C.2, the whole command is noninterferent giving thesis $\langle \mathsf{M}'_1, \mathsf{c}\rangle \approx^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}\rangle$.

If $\mathcal{L}(\tau) = \mathsf{HH}$ then $\Delta, \mathsf{HH} \sqcup pc \vdash \mathsf{c}_i$ and by Confinemnt lemma (Lemma C.16) it holds that for every variable x assigned to by c_1 and c_2 such that $\Delta(x) = \tau'$ then $\mathsf{HH} \sqcup pc \sqsubseteq \mathcal{L}(\tau') \sqcup \mathsf{LH}$, meaning $\mathcal{L}(\tau') \not\sqsubseteq \mathsf{LL}$. Thus, by corollary C.2, $\langle \mathsf{M}'_1, \mathsf{c}\rangle \approx^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}\rangle$.

Suppose $\mathcal{L}(\tau) = \mathsf{LH}$ then the same branch will be followed by each of the four executions. Let, for example $b \downarrow^{\mathsf{M}_i}$ false and $b \downarrow^{\mathsf{M}'_i}$ false, then it must be that $\langle \mathsf{M}_i, \text{if } e \text{ then } \mathsf{c}_1 \text{ else } \mathsf{c}_2\rangle \Rightarrow N_i$ because of $\langle \mathsf{M}_i, \mathsf{c}_2\rangle \Rightarrow N_i$ (by command semantics rule [*iff*]) thus $\langle \mathsf{M}_1, \mathsf{c}_2\rangle \simeq^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$ which by induction on c_2 gets $\langle \mathsf{M}'_1, \mathsf{c}_2\rangle \approx^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}_2\rangle$. This implies $\langle \mathsf{M}'_1, \text{if } b \text{ then } \mathsf{c}_1 \text{ else } \mathsf{c}_2\rangle \approx^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}'_2, \text{if } b \text{ then } \mathsf{c}_1 \text{ else } \mathsf{c}_2\rangle$. (Analogously for the case $b \downarrow^{\mathsf{M}_i}$ true.)

If rule (if-MAC) has been used to type the command it is

$$\text{if } \mathsf{mac}_x(z, e) = e' \text{ then } (y := e; \mathsf{c}_1) \text{ else } \mathsf{c}_2; \bot_{\mathsf{MAC}}$$

and it must be that $\Delta(x) = \mathsf{mK_{HC}}(\mathsf{L[D]}, \tau)$, $\Delta \vdash z : \mathsf{L[D]}$, $\Delta \vdash e : \mathsf{LL}$, $\Delta(y) = \tau$, $\Delta \vdash e' : \mathsf{LL}$ and also $\Delta, pc \vdash \mathsf{c}_1$,$\Delta, pc \vdash \mathsf{c}_2$. Suppose $e \downarrow^{\mathsf{M}_i} v_i$, $z \downarrow^{\mathsf{M}_i} v^*_i$ and $e \downarrow^{\mathsf{M}'_i} v'_i$, $z \downarrow^{\mathsf{M}'_i} v'^*_i$. Since from the hypothesis the program terminates in both M_1 and M_2 it must be that the then-branch has been followed in both configurations giving $\langle \mathsf{M}_1, y := e; \mathsf{c}_1\rangle \simeq^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}_2, y := e; \mathsf{c}_1\rangle$.

Notice that if one of the executions $\langle \mathsf{M}'_1, \mathsf{c}\rangle$ and $\langle \mathsf{M}'_2, \mathsf{c}\rangle$ takes the else branch it will not terminate, directly giving $\langle \mathsf{M}'_1, \mathsf{c}\rangle \approx^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}\rangle$. We thus consider the case in which both executions take the if branch. By corollary C.2 we get that $\mathsf{M}_1[y \mapsto v_1] \approx^{\Delta}_{\mathsf{LL}} \mathsf{M}_2[y \mapsto v_2]$, $\mathsf{M}'_1[y \mapsto v'_1] \approx^{\Delta}_{\mathsf{LL}} \mathsf{M}'_2[y \mapsto v'_2]$. As in the proof of lemma C.1 (case [*ift*]) and Theorem C.2 we get that $\mathsf{mac}_x(z, e) \downarrow^{\mathsf{M}_i} \langle v^*_i, v_i\rangle_k$, $\mathsf{mac}_x(z, e) \downarrow^{\mathsf{M}'_i} \langle v'^*_i, v'_i\rangle_k$ and $e' \downarrow^{\mathsf{M}_i} \langle v^*_i, v_i\rangle_k$, $e' \downarrow^{\mathsf{M}'_i} \langle v'^*_i, v'_i\rangle_k$ and $\Theta \vdash^{f_i}_g v_i : \tau$,$\Theta \vdash^{f_i}_g v'_i : \tau$. Since memories are well-formed and, by lemma C.9 we have that $\mathcal{L}_I(\tau) \sqsubset_I \mathsf{H}$, we know that $\Theta \vdash^{f_i}_g \mathsf{M}_i(y) : \tau$,$\Theta \vdash^{f_i}_g \mathsf{M}'_i(y) : \tau$. Now by lemma C.13 we have that $\mathsf{p_{HH}}(v_i) = \mathsf{p_{HH}}(\mathsf{M}_i(y))$ and $\mathsf{p_{HH}}(v'_i) = \mathsf{p_{HH}}(\mathsf{M}'_i(y))$, meaning that the assignment does not change in any way the equality of the two memories, i.e., $\mathsf{M}_i[y \mapsto v_i] \approx^{\Delta}_{\mathsf{HH}} \mathsf{M}'_i[y \mapsto v_i]$. Hence it is possible to apply induction on c_1 to conclude the case.

while e do c_1

No declassification is allowed inside a while loop, thus the case follows by Theorem C.2.

$\mathsf{c}_1; \mathsf{c}_2$

Rule (seq) states $\Delta, pc \vdash \mathsf{c}_1$ and $\Delta, pc \vdash \mathsf{c}_2$. By Lemma C.20 since $\langle \mathsf{M}_1, \mathsf{c}_1; \mathsf{c}_2\rangle \simeq^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}_1; \mathsf{c}_2\rangle$ then it must be that $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \simeq^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}_1\rangle$ thus by induction we get $\langle \mathsf{M}'_1, \mathsf{c}_1\rangle \approx^{\Delta}_{\mathsf{LL}} \langle \mathsf{M}'_2, \mathsf{c}_1\rangle$. Let $\langle \mathsf{M}_i, \mathsf{c}_1\rangle \Rightarrow N^1_i$, $\langle \mathsf{M}'_i, \mathsf{c}_1\rangle \Rightarrow N^2_i$ it holds $N^1_1 \approx^{\Delta}_{\mathsf{LL}} N^1_2$, $N^2_1 \approx^{\Delta}_{\mathsf{LL}} N^2_2$ and by theorem C.3 $N^1_i \approx^{\Delta}_{\mathsf{HH}} N^2_i$. The thesis follows by induction on c_2. □

□

Table C.4 PIN-block formats

We report some of the standard PIN block formats. All of the formats below are 16 hexadecimal digits, i.e., 64 bits long, and support PINs from 4 to 12 digits in length.

ISO-0, ANSI X9.8, VISA-1, ECI-1

0	L	P	P	P	P	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	F	F

0 identifies the format, L is the length of the PIN, P are the PIN digits while P/F is either e PIN digit or the pad value F, depending on the PIN length. Then the rightmost 12 digits of the PAN are written as follows:

0	0	0	0	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN

and we now just xor the two numbers:

0	L	P	P	P xor PAN	P xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	F xor PAN	F xor PAN

ISO-1

1	L	P	P	P	P	P/R	P/R	P/R	P/R	P/R	P/R	P/R	P/R	R	R

1 identifies the format, L is the length of the PIN, P are the PIN digits while P/R is either e PIN digit or the pad random value R, depending on the PIN length.

Table C.5 Revised model of the PIN verification API with types.

```
PIN_V(PAN, EPB, len, offset, vdata, dectab) {

    // deriving user PIN with IBM 3624 PIN calculation method:
    x1 := dec_pdk(vdata);            // *decrypt* vdata with pdk - x1 : H[HEX : PAN]
    x2 := left(len, x1);             // takes len leftmost digits - x2 : H[• : PAN]
    x3 := decimalize(dectab, x2);    // decimalizes - x3 : H[• : PAN]
    x4 := sum_mod10(x3, offset);     // sums the offset - x4 : H[• : PAN]

    // recovering the trial PIN from ISO-1 block
    x6 := dec^R_k(EPB);              // decrypts the EPB with k - x6 : H[PIN : PAN]

    // checks trial PIN versus actual user PIN
    x7 := declassify(x4 = x6)        // declassify the result - x7 : LH
    if  (x7)  then  ret := "PIN is correct";  // ret : LL
               else  ret := "PIN is wrong";
}
```

Table C.6 The new PIN_T_M API with MAC-based integrity, with types.

```
PIN_T_M(PAN, EPB_I, EPB_O, MAC_I, MAC_O) {

    // checking the MAC of PIN decryption
    if  (mac_ak(EPB_I, PAN) = MAC_I)  then
    {
        // recovering the trial PIN from ISO-1 block
        x1 := dec^R_k(EPB_I);          // decrypts the EPB with k - x1 : H[PIN : PAN]

        EPB'_O := enc^R_k'(x1, PAN);   // PIN encryption ISO-0, PAN in padding
                                       //  - EPB_O : enc_L[•:PAN] κ_k'
        MAC_O := mac_ak'(EPB_O, PAN); // generates the next MAC - MAC_O : LL
        EPB_O := EPB'_O;               // rises to LL - EPB_O : LL
        ret := "success";              // ret : LL
    }
    else
        ret := "Integrity Check Failed";
}
```

D

Type checking PKCS#11 - Formal Proofs

D.1 Typing values

The following rules type ciphertext, diversified keys and decrypted messages.

$$\frac{\Gamma \vdash_{\mathsf{M,H}} v : \tau' \quad \tau' \leq \tau}{\Gamma \vdash_{\mathsf{M,H}} v : \tau} \qquad \frac{\Gamma \vdash_{\mathsf{M,H}} v : \mathsf{LL} \quad \Gamma \vdash_{\mathsf{M,H}} v' : \tau \neq \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]}{\Gamma \vdash_{\mathsf{M,H}} enc(v, v') : \mathsf{LL}}$$

$$\frac{\Gamma \vdash_{\mathsf{M,H}} v : \tau \quad \Gamma \vdash_{\mathsf{M,H}} v' : \mathsf{W}^{\mathsf{HH}}[\mathsf{HL}]}{\Gamma \vdash_{\mathsf{M,H}} enc(v, v') : \mathsf{LL}} \qquad \frac{\Gamma \vdash_{\mathsf{M,H}} v : \mathsf{HH} \quad \Gamma \vdash_{\mathsf{M,H}} v' : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]}{\Gamma \vdash_{\mathsf{M,H}} enc(v, v') : \mathsf{LL}}$$

$$\frac{\Gamma \vdash_{\mathsf{M,H}} v : \ell}{\Gamma \vdash_{\mathsf{M,H}} enc(\mathtt{D}, v) : \mathsf{D}^{\ell}} \qquad \frac{\Gamma \vdash_{\mathsf{M,H}} v : \ell \quad \ell \neq \mathsf{LL}}{\Gamma \vdash_{\mathsf{M,H}} enc(\mathtt{W}, v) : \mathsf{W}^{\ell}[\mathsf{HL}]} \qquad \frac{\Gamma \vdash_{\mathsf{M,H}} v : \mathsf{LL}}{\Gamma \vdash_{\mathsf{M,H}} enc(\mathtt{W}, v) : \mathsf{LL}}$$

$$\frac{\Gamma \vdash_{\mathsf{M,H}} v : \mathsf{HH}}{\Gamma \vdash_{\mathsf{M,H}} enc(\mathtt{W}^2, v) : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]} \qquad \frac{\Gamma \vdash_{\mathsf{M,H}} v : \mathsf{LL} \quad \Gamma \vdash_{\mathsf{M,H}} v' : \mathsf{D}^{\mathsf{HL}}}{\Gamma \vdash_{\mathsf{M,H}} dec(v, v') : \mathsf{LL}}$$

$$\frac{\Gamma \vdash_{\mathsf{M,H}} v : \mathsf{LL} \quad \Gamma \vdash_{\mathsf{M,H}} v' : \mathsf{W}^{\mathsf{HH}}[\mathsf{HL}]}{\Gamma \vdash_{\mathsf{M,H}} dec(v, v') : \mathsf{HL}} \qquad \frac{\Gamma \vdash_{\mathsf{M,H}} v : \mathsf{LL} \quad \Gamma \vdash_{\mathsf{M,H}} v' : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]}{\Gamma \vdash_{\mathsf{M,H}} dec(v, v') : \mathsf{HH}}$$

D.2 Formal proofs

The sub-type relation permits no type to be a sub-type of a handle.

Proposition D.1. *If $\tau' \leq \tau$ then $\tau' \neq T \Rightarrow \tau \neq T'$*

The following lemma shows that expression evaluation respects value typing.

Lemma D.1. *Let $\Gamma \vdash_P e : \tau$ and $e \downarrow^{\mathsf{M,H}} v$. If $\Gamma \vdash_P \mathsf{M}, \mathsf{H}$ then it holds*

- $\tau \neq T$ *implies* $\Gamma \vdash_{\mathsf{M,H}} v : \tau' \leq \tau$
- $\tau = T$ *implies* $\Gamma \vdash_{\mathsf{M,H}} v : \mathsf{LL}$

Proof. By induction on the derivation of the type judgement $\Gamma \vdash_P e : \tau$, on cases on the last rule applied:

(var)
This case follows directly from $\Gamma \vdash_P \mathsf{M}, \mathsf{H}$.

(sub)
It holds $\Gamma \vdash_P e : \tau''$ and $\tau'' \leq \tau$. Two cases are considered:

- $\tau \neq T$: it could be that $\tau'' = T$ in which case by induction $\Gamma \vdash_{\mathsf{M,H}} v : \mathsf{LL}$ and it also holds $\mathsf{LL} \leq \tau$ since $T \leq \mathsf{LL}$ and $T \leq \tau$; otherwise if $\tau'' \neq T$, by induction on the judgement of $\Gamma \vdash_P e : \tau''$, $\Gamma \vdash_{\mathsf{M,H}} v : \tau'''$ with $\tau''' \leq \tau''$ from which the case follows since the sub-typing relation is transitive.
- $\tau = T$: by Proposition D.1 it must also be that $\tau' = T'$ but since handle types are not in any sub-type relation it holds $\tau' = T$ which gives the result by induction.

(get)
: Note that e must be of the form $\mathsf{getObj}(x)$ and it holds $\Gamma \vdash_P x : T$ and $\vdash T : \tau \neq T$. If $\mathsf{getObj}(x) \downarrow^{\mathsf{M,H}}$ `CKR_HANDLE_INVALID` then the case is trivial since the hypothesis do not hold (`CKR_HANDLE_INVALID` is an error *err* and not a value v). Otherwise $\mathsf{H}(n) = (v, T)$ and $\mathsf{getObj}(x) \downarrow^{\mathsf{M,H}} v$. By $\Gamma \vdash_P \mathsf{M}, \mathsf{H}$ it follows $\Gamma \vdash_{\mathsf{M,H}} v : \tau'' \leq \tau$ proving the case.

(get-un)
: It must be that e is $\mathsf{getObj}(x)$ and $\Gamma \vdash_P x : \mathsf{LL}$. If $\mathsf{getObj}(x) \downarrow^{\mathsf{M,H}}$ `CKR_HANDLE_INVALID` then the case is trivial since the hypothesis do not hold (`CKR_HANDLE_INVALID` is an error *err* and not a value v). Otherwise, $\tau = \mathsf{HL}$ and the result follows noting that for any τ', $\tau' \leq \mathsf{HL}$.

(chk)
: e is $\mathsf{checkTemplate}(x, T)$, it follows $\Gamma \vdash_P x : \mathsf{LL}$ and $\mathsf{topQ}(T, P) = \tau \neq T$. If $\mathsf{checkTemplate}(x, T) \downarrow^{\mathsf{M,H}}$ `CKR_TEMPLATE_INCONSISTENT` then the case is trivially proved since the hypothesis do not hold. Otherwise, it must be that $\mathsf{M}(x) = g$, $\mathsf{H}(g) = (v', T')$, $T \subseteq T'$ and $v = v'$. Let $\vdash T' : \tau''$, from $\Gamma \vdash_P \mathsf{M}, \mathsf{H}$ it follows $\Gamma \vdash_{\mathsf{M,H}} v : \tau''' \leq \tau''$ and also $T' \in P_{GEN} \cup P_{IMP}$, then $\tau'' \leq \tau$ proving the case.

(dk)
: The expression e is $\mathsf{diversifyKey}(\mathtt{D}, x)$. Let $\mathsf{M}(x) = v'$ then $v = enc(\mathtt{D}, v')$. The type system requires that $\Gamma \vdash_P x : \ell$ and states that $\tau = \mathsf{D}^{\ell}$. By induction on the judgement for x it holds $\Gamma \vdash_{\mathsf{M,H}} v' : \tau'' \leq \ell$, so $\Gamma \vdash_{\mathsf{M,H}} v' : \ell$ and the case is proved.

(dw-h)
: The expression e is $\mathsf{diversifyKey}(\mathtt{W}, x)$ and $\Gamma \vdash_P x : \ell \neq \mathsf{LL}$. Let $\mathsf{M}(x) = v'$, then $v = enc(\mathtt{W}, v')$. By induction on $\Gamma \vdash_P x : \ell$ it follows $\Gamma \vdash_{\mathsf{M,H}} v' : \tau'' \leq \ell$ so $\Gamma \vdash_{\mathsf{M,H}} v' : \ell$ proving the case (indeed $\Gamma \vdash_{\mathsf{M,H}} enc(\mathtt{W}, v') : \mathsf{W}^{\ell}[\mathsf{HL}]$).

(dw-l)
: It holds $\Gamma \vdash_P x : \mathsf{LL}$ and if $\mathsf{M}(x) = v'$, $v = enc(\mathtt{W}, v')$. By induction on the type judgement for x if follows $\Gamma \vdash_{\mathsf{M,H}} v' : \tau'' \leq \mathsf{LL}$, so $\Gamma \vdash_{\mathsf{M,H}} v' : \mathsf{LL}$ proving the case by deriving $\Gamma \vdash_{\mathsf{M,H}} enc(\mathtt{W}, v') : \mathsf{LL}$.

(dww)
: e is $\mathsf{diversifyKey}(\mathtt{W}^2, x)$ and it holds $\Gamma \vdash_P x : \mathsf{HH}$. Let $\mathsf{M}(x) = v'$ then $v = enc(\mathtt{W}^2, v')$. By induction on the judgment for x it follows $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v' : \tau'' \leq \mathsf{HH}$ which proves the case: indeed it also holds by sub-typing that $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v' : \mathsf{HH}$ and then $\Gamma \vdash_{\mathsf{M},\mathsf{H}} enc(\mathtt{W}^2, v') : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]$.

(enc)
: In this case $e = \mathsf{enc}(e', x)$. Let $e' \downarrow^{\mathsf{M},\mathsf{H}} v'$ and $\mathsf{M}(x) = v''$, then $v = enc(v', v'')$. The type system requires $\Gamma \vdash_P x : \mathsf{HL}$, $\Gamma(x) \neq \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]$ and $\Gamma \vdash_P e' : \mathsf{LL}$ then $\tau = \mathsf{LL}$. By induction on the judgment for x and e' it holds that $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v'' : \tau'' \leq \mathsf{HL}$ and $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v' : \tau''' \leq \mathsf{LL}$. It directly follows that $\Gamma \vdash_{\mathsf{M},\mathsf{H}} enc(v', v'') : \mathsf{LL}$ from the value-typing rule for an encrypted message whose payload is a public byte-stream.

(wrap)
: Let e be $\mathsf{enc}(e', x)$, $e' \downarrow^{\mathsf{M},\mathsf{H}} v'$ and $\mathsf{M}(x) = v''$, it holds $v = enc(v', v'')$. Rule (wrap) states that $\Gamma \vdash_P x : \mathsf{W}^{\mathsf{HH}}[\mathsf{HL}]$ and $\Gamma \vdash_P e' : \mathsf{HL}$. By induction $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v' : \tau'' \leq \mathsf{HL}$, $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v'' : \mathsf{W}^{\mathsf{HH}}[\mathsf{HL}]$ from which $\Gamma \vdash_{\mathsf{M},\mathsf{H}} enc(v', v'') : \mathsf{LL}$.

(dec)
: Expression e is of the form $\mathsf{dec}(e', x)$. It holds $\Gamma \vdash_P x : \mathsf{D}^{\mathsf{HL}}$ and $\Gamma \vdash_P e : \mathsf{LL}$. Let $e' \downarrow^{\mathsf{M},\mathsf{H}} v'$ and $\mathsf{M}(x) = v''$ then two cases are considered:

- $v' = enc(v''', v'')$: in this case $v = v'''$. By induction on x and e' it follows $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v'' : \tau'' \leq \mathsf{D}^{\mathsf{HL}}$ and $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v' : \mathsf{LL}$ (since the value is an encryption and every encrypted message types LL). The only rule which can be used to derive such a typing on a encrypted message, given the typing of v'', is the one that states that $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v''' : \mathsf{LL}$ concluding the case.
- $v' \neq enc(v''', v'')$: in this case $v = dec(v', v'')$. By induction on x and e' it follows $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v'' : \tau'' \leq \mathsf{D}^{\mathsf{HL}}$ and $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v' : \tau''' \leq \mathsf{LL}$, then $\Gamma \vdash_{\mathsf{M},\mathsf{H}} dec(v', v'') : \mathsf{LL}$ as needed.

(unwrap)
: Let e be $\mathsf{dec}(e', x)$ with $e' \downarrow^{\mathsf{M},\mathsf{H}} v'$ and $\mathsf{M}(x) = v''$. It holds $\Gamma \vdash_P x : \mathsf{HL}$ and $\Gamma \vdash_P e : \mathsf{LL}$. Two cases are considered:

- $v' = enc(v''', v'')$: in this case $v = v'''$. By induction it follows $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v' : \mathsf{LL}$ (indeed it is an encrypted message) and $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v'' : \tau''' \leq \mathsf{HL}$. Since the encrypted message (v') types, it holds $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v''' : \tau^*$ and it will always be that $\tau^* \leq \mathsf{HL}$ proving the case.
- $v' \neq enc(v''', v'')$: it follows $v = dec(v', v'')$. By induction it holds $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v' : \tau'' \leq \mathsf{LL}$ and $\Gamma \vdash_{\mathsf{M},\mathsf{H}} v'' : \tau''' \leq \mathsf{HL}$. It is enough to state that $\Gamma \vdash_{\mathsf{M},\mathsf{H}} dec(v', v'') : \tau^{iv}$ for some τ^{iv} and it will always be $\tau^{iv} \leq \mathsf{HL}$ proving the case.

(hh-w)

Let e be $\mathsf{enc}(e', x)$, $e' \downarrow^{\mathsf{M,H}} v'$ and $\mathsf{M}(x) = v''$, it holds $\Gamma \vdash_P x : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]$ and $\Gamma \vdash_P e' : \mathsf{HH}$. The expression evaluates to $v = enc(v', v'')$ and by induction $\Gamma \vdash_{\mathsf{M,H}} v' : \tau'' \leq \mathsf{HH}$, $\Gamma \vdash_{\mathsf{M,H}} v'' : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]$ from which $\Gamma \vdash_{\mathsf{M,H}} enc(v', v'') : \mathsf{LL}$.

(hh-u)

The expression e is $\mathsf{dec}(e', x)$ and $\Gamma \vdash_P x : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]$, $\Gamma \vdash_P e' : \mathsf{LL}$. Let $e' \downarrow^{\mathsf{M,H}} v'$ and $\mathsf{M}(x) = v''$, two cases are considered:

- $v' = enc(v''', v'')$: in this case $v = v'''$. By induction, $\Gamma \vdash_{\mathsf{M,H}} v' : \mathsf{LL}$ and $\Gamma \vdash_{\mathsf{M,H}} v'' : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]$ then it must be that $\Gamma \vdash_{\mathsf{M,H}} v''' : \mathsf{HH}$ by the only rule typing a value with such a key.
- $v' \neq enc(v''', v'')$: then $v = dec(v', v'')$. By induction, $\Gamma \vdash_{\mathsf{M,H}} v' : \tau'' \leq \mathsf{LL}$ and $\Gamma \vdash_{\mathsf{M,H}} v'' : \mathsf{W}^{\mathsf{HH}}[\mathsf{HH}]$ which directly give the thesis: indeed it follows $\Gamma \vdash_{\mathsf{M,H}} dec(v', v'') : \mathsf{HH}$.

□

Note that no expression will ever produce new handles, this functionality is delegated to the dedicated assignment commands using genKey and importKey. It follows that if an expression types as a handle then it can only be a variable.

Lemma D.2. *If* $\Gamma \vdash_P e : T$ *then* e *is a variable.*

Proof. The proof follows by observing that every expression e different from a variable is always typed as $\tau \neq T$ and such type cannot be sub-typed to a handle type. □

It is essential to prove that every expression typed LL does not reveal any secret, in fact this confirms that every value "leaked" to the public gives no information about sensitive keys.

Lemma D.3. *Let* $\Gamma \vdash_P \mathsf{M}, \mathsf{H}$, $\Gamma \vdash_P e : \mathsf{LL}$, $e \downarrow^{\mathsf{M,H}} v$ *and* $v' \in G$ *such that* $\mathsf{priv}_{\Gamma,\mathsf{M}}(v')$ *and* $\mathsf{botH}(v', \mathsf{H}) = \tau \neq \mathsf{LL}$. *It holds* $v \neq v'$ *and* $v \neq enc(tag, v')$.

Proof. By induction on the structure of the value v:

n

This case trivially holds since $n \in N$ and $N \cap G = \emptyset$, and also $n \neq enc(v_1, v_2)$ for any v_1, v_2.

g

By Lemma D.1, $\Gamma \vdash_{\mathsf{M,H}} g : \tau \leq \mathsf{LL}$ so it must be that $\neg(\mathsf{priv}_{\Gamma,\mathsf{M}}(g))$ or $\nexists$ $\mathsf{botH}(v, \mathsf{H})$ or $\mathsf{botH}(v, \mathsf{H}) = LL$, proving that $v \neq v'$. Notice also that $g \neq enc(v_1, v_2)$ for any v_1, v_2.

$enc(v_1, v_2)$
: It trivially holds that $v \neq v'$ since an encrypted message is never equal to an atomic value in the G set. Moreover note that for any value v'' it holds $v'' \neq tag$ proving the fact that $v \neq enc(tag, v')$.

$enc(tag, v_1)$
: It holds $\Gamma \vdash_{\mathsf{M,H}} enc(tag, v_1) : \tau \leq \mathsf{LL}$ by Lemma D.1, it then follows $\Gamma \vdash_{\mathsf{M,H}} v_1 : \mathsf{LL}$ which proves the case.

$dec(v_1, v_2)$
: This case is obvious since $dec(v_1, v_2) \neq v''$ for any atomic v'' and also $dec(v_1, v_2) \neq enc(tag, v_3)$ for any v_3.

□

The main result is proved by the following theorem stating that well-typed programs preserve memory and handle-map well-formedness and that if a well-typed program c reduce to c' then c' is well-typed.

Theorem. *Let* $\Gamma \vdash_P \mathsf{M}, \mathsf{H}$ *and* $\Gamma \vdash_P \mathsf{c}$*. If* $\langle \mathsf{M}, \mathsf{H}, \mathsf{c} \rangle \to \langle \mathsf{M}', \mathsf{H}', \mathsf{c}' \rangle$ *then*

- *if* $\mathsf{c}' \neq \varepsilon$ *then* $\Gamma \vdash_P \mathsf{c}'$ *and*
- $\Gamma \vdash_P \mathsf{M}', \mathsf{H}'$.

Proof. By induction on the structure of the program c.

$\mathsf{FAIL}[err]$
: This command represents an error state and does not move, the case trivially holds.

$x := e$
: If $e \downarrow^{\mathsf{M,H}} v$, $\langle \mathsf{M}, \mathsf{H}, x := e \rangle \to \langle \mathsf{M}[x \mapsto v], \mathsf{H}, \varepsilon \rangle$. It is only necessary to prove that $\Gamma \vdash_P \mathsf{M}[x \mapsto v], \mathsf{H}$. It holds $\Gamma(x) = \tau$ and $\Gamma \vdash_P e : \tau$. Two cases are considered. If $\tau \neq T$ then by Lemma D.1 it follows $\Gamma \vdash_{\mathsf{M,H}} v : \tau' \leq \tau$ and the case is proved. Otherwise the same Lemma states that $\Gamma \vdash_{\mathsf{M,H}} v : \mathsf{LL}$ and by Lemma D.2 e must be a variable y, the case follows by well-formedness requirements on y.

  If, instead $e \downarrow^{\mathsf{M,H}} err$ then $\langle \mathsf{M}, \mathsf{H}, x := e \rangle \to \langle \mathsf{M}, \mathsf{H}, \mathsf{FAIL}[err] \rangle$ concluding the proof, indeed $\Gamma \vdash_P \mathsf{FAIL}[err]$ and the memory and handle-map are unchanged thus well-formed by hypothesis.

$x := \mathsf{genKey}(T)$
: It holds $\langle \mathsf{M}, \mathsf{H}, x := \mathsf{genKey}(T) \rangle \to \langle \mathsf{M}[x \mapsto g'], \mathsf{H}[g' \mapsto (g, \oplus(T))], \varepsilon \rangle$. It must be proved that $\Gamma \vdash_P \mathsf{M}[x \mapsto g'], \mathsf{H}[g' \mapsto (g, \oplus(T))]$. The type system requires that $T \in P_{GEN}$, and $\Gamma(x) = \oplus(T)$. It immediately holds that if $S \notin T$

then $\oplus(T) \in P_{GEN}$ (indeed $\oplus(T) = T$) and otherwise $\oplus(T) \setminus \{A\} \in P_{GEN}$ indeed $\oplus(T) = T \cup \{A\}$. Suppose $\vdash \oplus(T) : \tau$, it remains to be proved that $\Gamma \vdash_{\mathsf{M}',\mathsf{H}'} g' : \mathsf{LL}$ and $\Gamma \vdash_{\mathsf{M}',\mathsf{H}'} v : \tau' \leq \tau$. The new handle g' is extracted fresh from G and assigned to x and $\Gamma(x) = \oplus(T) \leq \mathsf{LL}$ so $\neg\mathsf{priv}_{\Gamma,\mathsf{M}'}(g')$ from which $\Gamma \vdash_{\mathsf{M}',\mathsf{H}'} g' : \mathsf{LL}$. In the same way g is a fresh value and the only template in H' which is associated to it is $\oplus(T)$ then $\Gamma \vdash_{\mathsf{M}',\mathsf{H}'} g : \tau$. Notice also that $\mathsf{priv}_{\Gamma,\mathsf{M}'}$ since $\forall x \in dom(\mathsf{M})$ it holds $\mathsf{M}(x) \neq g$ (well-formedness requirement 1 is thus satisfied) and the case is proved.

$x := \mathsf{importKey}(y, T)$

The command reduces to ε yielding memory $\mathsf{M}[x \mapsto g]$ and handle-map $\mathsf{H}[g \mapsto (v, T)]$ (starting from M and H) where $g \leftarrow G$ and $\mathsf{M}(y) = v$. Two cases are considered:

- $\Gamma \vdash_P y : \mathsf{HL}$: the type system requires that $\Gamma(x) = T$, $T \in P_{IMP}$ and $\vdash T : \mathsf{HL}$. Note that since $T \in P_{IMP}$, $A \notin T$ and also it must be that $S \in T$ since $\vdash T : \mathsf{HL}$. It holds $\Gamma \vdash_{\mathsf{M}',\mathsf{H}'} g : \mathsf{LL}$ since $\Gamma(x) = T \leq \mathsf{LL}$ (thus $\neg\mathsf{priv}_{\Gamma,\mathsf{M}'}(g)$). By Lemma D.1 $\Gamma \ \vdash_{\mathsf{M},\mathsf{H}} \ v : \ \tau \leq \mathsf{HL}$ proving the case.
- $\Gamma \vdash_P y : \mathsf{HH}$: it must be $\Gamma(x) = T$, $T \setminus \{A\} \in P_{GEN}$ and $\vdash T : \mathsf{HH}$. It holds $\Gamma \vdash_{\mathsf{M}',\mathsf{H}'} g : \mathsf{LL}$ since $\Gamma(x) = T \leq \mathsf{LL}$ (thus $\neg\mathsf{priv}_{\Gamma,\mathsf{M}'}(g)$). By Lemma D.1 $\Gamma \ \vdash_{\mathsf{M},\mathsf{H}} \ v : \ \tau \leq \mathsf{HH}$ proving the case since the requirement on the handle-map is verified and also point 1 of well-formedness, indeed $A \in T$ (from $\vdash T : \mathsf{HH}$) and M is well-formed so $\mathsf{priv}_{\Gamma,\mathsf{M}}(v)$ (in fact public variable can only store values typing $\tau \leq \mathsf{LL}$ and $\mathsf{HH} \not\leq \mathsf{LL}$).

$\mathsf{c}_1; \mathsf{c}_2$

Rule (seq) requires that $\Gamma \vdash_P \mathsf{c}_1$ and $\Gamma \vdash_P \mathsf{c}_2$. Let $\langle \mathsf{M}, \mathsf{H}, \mathsf{c}_1 \rangle \rightarrow \langle \mathsf{M}', \mathsf{H}', \mathsf{c}_1' \rangle$, by induction on c_1 $\Gamma \vdash_P \mathsf{M}', \mathsf{H}'$. If $\mathsf{c}_1' = \varepsilon$ then $\langle \mathsf{M}, \mathsf{H}, \mathsf{c}_1; \mathsf{c}_2 \rangle \rightarrow \langle \mathsf{M}', \mathsf{H}', \mathsf{c}_2 \rangle$ proving the case, otherwise $\langle \mathsf{M}, \mathsf{H}, \mathsf{c}_1; \mathsf{c}_2 \rangle \rightarrow \langle \mathsf{M}', \mathsf{H}', \mathsf{c}_1'; \mathsf{c}_2 \rangle$ and by induction on c_1 it holds $\Gamma \vdash_P \mathsf{c}_1'$ from which the proof follows applying rule (seq) to $\mathsf{c}_1'; \mathsf{c}_2$.

□

It must also be proved that sensitive values are never leaked.

Theorem. *Let* $\Gamma \vdash_P \mathsf{M}, \mathsf{H}$ *and* $\Gamma \vdash_P \mathsf{c}$*. If* $\langle \mathsf{M}, \mathsf{H}, \mathsf{c} \rangle \rightarrow \langle \mathsf{M}', \mathsf{H}', \mathsf{c}' \rangle$ *and* $v \in G$ *then*

1. $\mathsf{priv}_{\Gamma,\mathsf{M}}(v)$ *and* $\mathsf{botH}(v, \mathsf{H}) = \tau \neq \mathsf{LL}$ *implies* $\mathsf{priv}_{\Gamma,\mathsf{M}'}(v)$ *and* $\mathsf{botH}(v, \mathsf{H}') = \tau \neq \mathsf{LL}$*, and*
2. $\mathsf{priv}_{\Gamma,\mathsf{M}}(enc(tag, v))$ *and* $\mathsf{botH}(enc(tag, v), \mathsf{H}) = \tau \neq \mathsf{LL}$ *implies* $\mathsf{priv}_{\Gamma,\mathsf{M}'}(enc(tag, v))$ *and* $\mathsf{botH}(enc(tag, v), \mathsf{H}') = \tau \neq \mathsf{LL}$.

Proof. By induction on the structure of the command c.

$\mathsf{FAIL}[err]$

This case is obvious since $\mathsf{M}' = \mathsf{M}$ and $\mathsf{H}' = \mathsf{H}$.

$x := e$

The type system states that $\Gamma(x) = \tau$ and $\Gamma \vdash_P e : \tau$. If $e \downarrow^{\mathsf{M},\mathsf{H}} v_1$ then $\mathsf{M}' = \mathsf{M}[x \mapsto v_1]$ and $\mathsf{H}' = \mathsf{H}$. Two different cases are considered:

- $\tau \leq \mathsf{LL}$: in this case by Lemma D.3 $v' \neq v$ and also $v' \neq enc(tag, v)$ thus proving the case.
- $\tau \not\leq \mathsf{LL}$: then it holds $\mathsf{priv}_{\Gamma,\mathsf{M}}(v) \Rightarrow \mathsf{priv}_{\Gamma,\mathsf{M}'}(v)$ (and the same for $enc(tag, v)$) proving the case.

$x := \mathsf{genKey}(T)$

Note that this command does not assign any value already contained in M to a variable and also store in the handle map a fresh value so the case follows immediately.

$x := \mathsf{importKey}(y, T)$

The value assigned to x is fresh and is leaked since it is used as a handle. Let $\mathsf{M}(y) = v'$, if v' is not a sensitive value $(\neg(\mathsf{priv}_{\Gamma,\mathsf{M}}(v') \wedge \mathsf{botH}(v', \mathsf{H}) = \tau' \neq \mathsf{LL})$ then nothing has to be proved, otherwise let $\vdash T : \tau$, it follows $\mathsf{botH}(v', \mathsf{H}') = \mathsf{botH}(v', \mathsf{H}) \sqcap \tau$, two different cases are considered: if the command types by rule (imp-l) then $\tau = \mathsf{HL}$ and $\mathsf{botH}(v', \mathsf{H}) \sqcap \mathsf{HL} = \mathsf{botH}(v', \mathsf{H})$ proving the case; otherwise rule (imp-h) must have been used and $\tau = \mathsf{HH}$, from $\mathsf{botH}(v', \mathsf{H}) = \tau' \neq \mathsf{LL}$ it follows $\tau' \sqcap \mathsf{HH} = \tau'' \neq \mathsf{LL}$ concluding the case.

$\mathsf{c}_1; \mathsf{c}_2$

This case follows directly by induction on c_1.

□

Bibliography

[1] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM (JACM)*, 46(5):749–786, September 1999.

[2] Martín Abadi and Jan Jurjens. Formal Eavesdropping and Its Computational Interpretation. In *Theoretical Aspects of Computer Software, 4th International Symposium (TACS)*, volume 2215 of *LNCS*, pages 82–94, Sendai, Japan, October 29-31 2001. Springer.

[3] Martín Abadi and Phillip Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). *JCRYPTOL: Journal of Cryptology*, 15(2):103–127, 2002.

[4] Pedro Adão, Gergei Bana, Jonathan Herzog, and Andre Scedrov. Soundness of formal encryption in the presence of key-cycles. In *10th European Symposium on Research in Computer Security (ESORICS)*, volume 3679 of *LNCS*, pages 374–396. Springer-Verlag, 2005.

[5] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP)*, pages 40–53, Boston, Massachusetts, USA, January 2000. ACM Press.

[6] Ross Anderson and Markus Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *Proceedings of the 5th International Workshop on Security Protocols*, volume 1361 of *LNCS*, pages 125–136. Springer, 1997.

[7] Alessandro Armando and Luca Compagna. SAT-based model-checking for security protocols analysis. *Int. J. Inf. Sec.*, 7(1):3–32, 2008. Software available at `http://www.ai-lab.it/satmc`. Currently developed under the AVANTSSAR project, `http://www.avantssar.eu`.

[8] Aslan Askarov, Daniel Hedin, and Aandrei Sabelfeld. Cryptographically-Masked Flows. In *Proceedings of the 13th International Static Analysis Symposium Static Analysis (SAS)*, volume 4134 of *LNCS*, pages 353–369, Seoul, Korea, August 2006. Springer.

[9] Aslan Askarov, Daniel Hedin, and Aandrei Sabelfeld. Cryptographically-Masked Flows. *Theoretical Computer Science*, 402(2-3):82–101, August 2008.

[10] Lorenzo Baloci and Andrea Vianello. Un sistema per lo studio della sicurezza. Baccalaureate Thesis, University of Venice, Italy, April 2010.

[11] Omer Berkman and Odelia Moshe Ostrovsky. The unbearable lightness of PIN cracking. In *11th International Conference, Financial Cryptography and Data Security (FC 2007)*, volume 4886 of *LNCS*, pages 224–238, Scarborough, Trinidad and Tobago, February 2007. Springer.

[12] Mike Bond. Attacks on Cryptoprocessor Transaction Sets. In *Cryptographic Hardware and Embedded Systems - CHES, Third International Workshop*, volume 2162 of *LNCS*, pages 220–234, Paris, France, May 2001. Springer.

[13] Mike Bond. *Understanding Security APIs*. PhD thesis, University of Cambridge Computer Laboratory, 2004.

[14] Mike Bond and Piotr Zieliński. Decimalization table attacks for PIN cracking. Technical Report UCAM-CL-TR-560, University of Cambridge, Computer Laboratory, 2003. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-560.pdf.

[15] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing PKCS#11 security tokens. In *Proocedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 260–269, Chicago, Illinois, USA, October 2010. ACM.

[16] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. CryptokiX: a cryptographic software token with security fixes. In *Proocedings of the 4th International Workshop on Analysis of Security APIs (ASA)*, Edinburgh, UK, July 2010.

[17] Gérard Boudol and Ilaria Castellani. Noninterference for Concurrent Programs. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of *LNCS*, pages 382–395, Crete, Greece, July 2001. Springer.

[18] Christian Cachin and Nishanth Chandran. A Secure Cryptographic Token Interface. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF)*, pages 141–153, Long Island, New York, 2009. IEEE Computer Society Press.

[19] Matteo Centenaro. Type-based Analysis of PKCS#11. Technical Report CS-2010-7, Computer Science Department, Ca' Foscari University, 2010. `http://www.unive.it/media/allegato/dipartimenti/informatica/Ricerca/RapportiTecnici/CS-2010-7.pdf`.

[20] Matteo Centenaro and Riccardo Focardi. Match It or Die: Proving Integrity by Equality. In *Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, volume 6186 of *LNCS*, pages 130–145, Paphos, Cyprus, March 2010. Springer Berlin / Heidelberg.

[21] Matteo Centenaro, Riccardo Focardi, Flaminia L. Luccio, and Graham Steel. Type-Based Analysis of PIN Processing APIs. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS)*, volume 5789 of *LNCS*, pages 53–68, Saint-Malo, France, September 2009. Springer.

[22] Tom Chothia, Dominic Duggan, and Jan Vitek. Type-based distributed access control. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW)*, pages 170–, Pacific Grove, CA, USA, July 2003. IEEE Computer Society.

[23] Jolyon Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779 of *LNCS*, pages 411–425, Cologne, Germany, September 2003. Springer.

[24] Jolyon Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.

[25] Véronique Cortier and Graham Steel. A generic security API for symmetric key management on cryptographic devices. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS)*, volume 5789 of *LNCS*, pages 605–620, Saint Malo, France, September 2009. Springer.

[26] Judicaël Courant, Cristian Ene, and Yassine Lakhnech. Computationally Sound Typing for Non-interference: The Case of Deterministic Encryption. In *27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*, volume 4855 of *LNCS*, pages 364–375, New Delhi, India, December 2007. Springer.

[27] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal Analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.

[28] Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, November 2010.

[29] Delphine Demange and David Sands. All secrets great and small. In *Programming Languages and Systems, 18th European Symposium on Programming (ESOP)*, volume 5502 of *LNCS*, pages 207–221, York, UK, March 2009. Springer.

[30] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Commununications of the ACM*, 20(7):504–513, 1977.

[31] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions in Information Theory*, 2(29):198–208, March 1983.

[32] Riccardo Focardi and Matteo Centenaro. Information Flow Security of Multi-Threaded Distributed Programs. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 113–124, Tucson, AZ, USA, June 2008. ACM Press.

[33] Riccardo Focardi, Flaminia L. Luccio, and Graham Steel. Blunting Differential Attacks on PIN Processing APIs. In *Proceedings of the 14th Nordic Conference on Secure IT Systems (NordSec)*, volume 5838 of *LNCS*, pages 88–103, Oslo, Norway, October 2009. Springer.

[34] Cédric Fournet and Tamara 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, San Francisco, Ca, USA, January 2008. ACM Press.

[35] Sibylle Fröschle and Graham Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In *Revised Selected Papers of the Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, volume 5511 of *Lecture Notes in Computer Science*, pages 92–106, York, UK, March 2009. Springer.

[36] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, USA, April 1982. IEEE Computer Society Press.

[37] Andrew D. Gordon and Alan Jeffrey. Authenticity by Typing for Security Protocols. *Journal of Computer Security*, 11(4):451–520, 2003.

[38] Hackers crack cash machine PIN codes to steal millions. The Times online. `http://www.timesonline.co.uk/tol/money/consumer_affairs/article4259009.ece`.

[39] IBM Inc. *CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors*, 2006. Releases 2.53–3.27. Available at http://www-03.ibm.com/security/cryptocards/pcicc/library.shtml.

[40] Gavin Keighren, David Aspinall, and Graham Steel. Towards a Type System for Security APIs. In *Proceedings of Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, pages 173–192, York, UK, March 2009.

[41] Peeter Laud. Secrecy Types for a Simulatable Cryptographic Library. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pages 26–35, Alexandria, VA, USA, 2005. ACM.

[42] Peeter Laud. On the computational soundness of cryptographically masked flows. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 337–348, San Francisco, Ca, USA, January 2008. ACM Press.

[43] Dennis Longley and S. Rigby. An Automatic Search for Security Flaws in Key Management Schemes. *Computers and Security*, 11(1):75–89, March 1992.

[44] Mohammad Mannan and Paul C. van Oorschot. Reducing Threats from Flawed Security APIs: The Banking PIN Case. *Computers & Security*, 28(6):410–420, September 2009.

[45] Heiko Mantel and Andrei Sabelfeld. A Unifying Approach to the Security of Distributed and Multi-Threaded Programs. *Journal of Computer Security*, 11(4):615–676, 2003.

[46] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[47] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing Robust Declassification. In *Proceedings of 17th Computer Security Foundations Workshop (CSFW)*, pages 172–186. IEEE Computer Society, 2004.

[48] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing Robust Declassification and Qualified Robustness. *Journal of Computer Security*, 14(2):157–196, May 2006.

[49] openCryptoki. `http://sourceforge.net/projects/opencryptoki/`.

[50] PIN Crackers Nab Holy Grail of Bank Card Security. Wired Magazine Blog 'Threat Level'. `http://blog.wired.com/27bstroke6/2009/04/pins.html`.

[51] RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard*, June 2004.

[52] Andrei Sabelfeld and Heiko Mantel. Static Confidentiality Enforcement for Distributed Programs. In *Proceeding of the 9th International Static Analysis Symposium*, volume 2477, pages 17–20, Madrid, Spain, September 2002. Springer.

[53] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[54] Andrei Sabelfeld and David Sands. Probabilistic Noninterference for Multi-Threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, England, July 2000.

[55] Andrei Sabelfeld and David Sands. Declassification: Dimensions and Principles. *Journal of Computer Security*, 17(5):517–548, October 2009.

[56] Geoffrey Smith. A new type system for secure information flow. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 115–125, Cape Breton, Nova Scotia, June 2001. IEEE Computer Society.

[57] Geoffrey Smith and Rafael Alpìzar. Secure information flow with random assignment and encryption. In *Proceedings of the 2006 ACM workshop on Formal methods in security engineering (FMSE)*, pages 33–44, Alexandria, VA, USA, November 3 2006. ACM.

[58] Graham Steel. Formal Analysis of PIN Block Attacks. *Theoretical Computer Science*, 367(1-2):257–270, November 2006.

[59] Jeffrey A. Vaughan and Steve Zdancewic. A Cryptographic Decentralized Label Model. In *IEEE Symposium on Security and Privacy*, pages 192–206, Oakland, California, USA, May 2007. IEEE Computer Society.

[60] Dennis Volpano. Secure introduction of one-way functions. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 246–254, Cambridge, England, July 2000. IEEE.

[61] Dennis Volpano and Geoffrey Smith. Eliminating Covert Flows with Minimum Typings. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW)*, pages 156–169, Rockport, Massachusetts, USA, June 1997. IEEE Computer Society.

[62] Dennis M. Volpano, Geoffrey Smith, and Cynthia E. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[63] Paul Youn, Ben Adida, Mike Bond, Jolyon Clulow, Jonathan Herzog, Amerson Lin, Ronald L. Rivest, and Ross Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, August 2005.

Estratto per riassunto della tesi di dottorato

Studente: Matteo Centenaro matricola: 955482

Dottorato: Informatica

Ciclo: XXIII

Titolo della tesi :Type-based Analysis of Security APIs

Abstract:

A Security API is an interface between processes running at different levels of trust with the aim of assuring that a specific security policy holds. It allows an untrusted system to access the functionalities offered by a trusted secure resource assuring that no matter what sequence of the API commands are invoked, the intended security policy is satisfied.

This kind of API is often developed having in mind a target application and how it will typically use the available services. It is thus easy to miss the fact that some functionalities could be used in a malicious way to break the intended security policy. In fact, a number of attacks to existing security APIs have been found in the last years.

This thesis proposes a type-based analysis to verify the security of these critical components. Language-based analysis is, in fact, a powerful tool to formally prove security and, at the same time, helps API developers to understand the root-causes of known vulnerabilities affecting APIs and guides them in programming secure code.

A security API which slowly leaks secret data to an attacker capable to spot interferences between input parameters and a command output can be secured by a noninterference policy. The thesis extends the setting of language-based information flow security to account for cryptographic expressions (both randomized and deterministic ones) and applies the obtained results to analyse the ATM PIN verification API. A possible fix to it is also proposed and shown to be secure by typing.

A security API which, instead, directly releases a secret value as the result of a sequence of legal commands will be analysed with a type system ensuring that data secrecy is preserved at run-time. The thesis presents the case of programs implementing key management functionalities and proposes a type system to reason on the security of RSA PKCS#11 API and verify the correctness of a novel patch to it.

Una security API è un 'interfaccia che regola la comunicazione tra due processi, che vengono eseguiti con diversi livelli di affidabilità, allo scopo di assicurare che sia soddisfatta una determinata proprietà di sicurezza. In questo modo, un sistema potenzialmente non fidato può fruire delle funzionalità offerte da una risorsa sicura senza mettere a rischio la sicurezza dei dati in essa contenuti: la API deve infatti impedire che una delle possibili sequenze di suoi comandi permetta di usare in maniera inappropriata la risorsa fidata. Molto spesso una API di questo tipo viene progettata e sviluppata considerando quella che sarà la sua applicazione più comune e come questa utilizzerà i servizi messi a disposizione. Una mancata visione di insieme dei comandi esposti dalla API apre la strada ad un suo possibile utilizzo malevolo atto a sovvertire la sicurezza dei dati che dovrebbe proteggere. Negli ultimi anni, infatti, sono stati scoperti numerosi attacchi contro le security API esistenti.

La tesi propone un'analisi, basata sui sistemi di tipi, per verificare la sicurezza di queste cruciali componenti. L'analisi formale dei linguaggi è, infatti, uno strumento molto potente per dimostrare che un dato programma soddisfa le proprietà di sicurezza richieste. Inoltre, fornisce anche un aiuto agli sviluppatori delle API per capire a fondo le cause delle vulnerabilità che le affliggono e li guida ad una programmazione consapevolmente sicura.

Un attaccante che è in grado di osservare differenze nei risultati ottenuti invocando la stessa funzione di una API con diversi parametri di input, può utilizzarle per derivare informazioni sui segreti custoditi nella parte fidata del sistema. Una security API soggetta a questo genere di attacchi può essere resa sicura utilizzando una proprietà di non-interferenza. La tesi estende la teoria esistente nel campo dell'information flow security per analizzare la sicurezza di programmi che fanno uso di primitive crittografiche (sia randomizzate che deterministiche) e applica i risultati ottenuti per studiare la API impiegata per la verifica dei PIN nella rete degli sportelli ATM (bancomat). Utilizzando il sistema di tipi proposto, è stato possibile proporre e verificare una soluzione che rende sicura tale API.

Una security API che, come risultato di una inaspettata sequenza di comandi, rivela una informazione che dovrebbe rimanere segreta, può essere invece analizzata con un sistema di tipi atto a controllare che la segretezza dei dati sia preservata durante tutto il tempo di esecuzione dei programmi. La tesi presenta il caso di programmi che offrono servizi per la gestione delle chiavi crittografiche, introducendo un sistema di tipi in grado di ragionare sulla sicurezza dello standard RSA PKCS\#11 e di verificare la correttezza di una nuova patch che lo rende sicuro.

Firma dello studente

____