PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à Inria Paris

Games and Logic for the Verification of Cryptographic Protocols

Soutenue par
Justine Sauvage
Le 21 janvier 2026

École doctorale n°386
École doctorale de Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Composition du jury :

Ioana Boureanu Professor, University of Surrey	*Rapportrice*
Steve Kremer Directeur de recherche, INRIA	*Rapporteur*
Gilles Barthe Scientific director, MPI SP	*Examinateur*
David Monniaux Directeur de recherche, CNRS	*Examinateur*
Sabine Oechsner Assistant Professor, Vrije Universiteit Amsterdam	*Examinatrice*
François Pottier Directeur de recherche, INRIA	*Examinateur*
Bruno Blanchet Directeur de recherche, INRIA	*Directeur de thèse*
Adrien Koutsos Chargé de recherche, INRIA	*Co-Directeur de thèse*
David Baelde Professeur, ENS Rennes	*Invité*

Inria

ENS | PSL

Abstract

This thesis investigates cryptographic protocol verification in the CCSA framework, a formal verification approach based on a probabilistic logic for proving security properties in the computational model. This framework is implemented in the Squirrel proof assistant. The main focus of the thesis is the mechanization of cryptographic reductions — a core proof technique in cryptography in which the security of a protocol is reduced to a cryptographic hardness assumption via the construction of a simulator.

Prior to this thesis, the CCSA framework provided logical axioms whose soundness was established through manual, error-prone reductions to a fixed set of cryptographic hardness assumption (e.g., CCA, PRF, EUF-MAC). Each axiom also necessitated implementation efforts, which were prone to errors. Unfortunately, these tasks (designing, proving, and implementing the axioms) were inaccessible to typical users, thus limiting the scalability of the CCSA approach.

The main contribution of this thesis is a framework that captures reductions to arbitrary cryptographic games for the CCSA framework. We introduce a logic with a core predicate, the *bideduction predicate*, which express the existence of a simulator justifying a cryptographic reduction. We then provide a proof system to derive such predicates, implicitly inferring simulators. We further implement in Squirrel a proof search procedure that synthesize memoizing simulators and generates time-sensitive invariants to justify the inferred simulator's correctness. Our implementation significantly extends Squirrel's scope as it extends the set of supported cryptographic hardness assumptions. To validate our approach, we apply it to case studies, reproving existing Squirrel case studies and analysing new ones which were not provable in Squirrel before. This work culminates with the first mechanized proof of ballot privacy for the FOO e-voting protocol — the largest proof conducted in Squirrel to date.

Résumé

Cette thèse étudie la vérification des protocoles cryptographiques dans le cadre CCSA, une approche de vérification formelle basée sur une logique probabiliste pour prouver les propriétés de sécurité dans le modèle computationnel. Cette approche est implémentée dans l'assistant de preuve Squirrel. Cette thèse s'intéresse à la mécanisation des réductions cryptographiques, une technique de preuve centrale en cryptographie où la sécurité d'un protocole est réduite à une hypothèse calculatoire cryptographique par la construction d'un simulateur.

Avant cette thèse, le cadre CCSA fournissait des axiomes logiques dont la validité était établie manuellement par des réductions. Ces réductions sont une source possible d'erreurs et les axiomes logiques n'avait été conçus seulement pour un nombre restreint d'hypothèses calculatoires (par exemple, CCA, PRF, EUF-MAC). Chaque axiome nécessitait également un effort d'implémentation, lui aussi source d'erreurs. Malheureusement, ces tâches (conception, preuve et implémentation des axiomes) étaient inaccessibles aux utilisateurs typiques, limitant ainsi la capacité de l'approche CCSA à passer à l'échelle.

La contribution majeure de cette thèse est une approche permettant de capturer des réductions vers des jeux cryptographiques arbitraires dans le cadre de la logique CCSA. Nous introduisons une logique dont le prédicat central, le prédicat de bidéduction, formalise l'existence d'un simulateur justifiant une réduction cryptographique. Nous proposons ensuite un système de preuve pour dériver ces prédicats, qui infère implicitement les simulateurs. Nous avons en outre implémenté dans Squirrel une procédure de recherche de preuve qui synthétise des simulateurs qui mémoïsent et génèrent des invariants sensibles au temps pour justifier la correction des simulateurs inférés. Notre implémentation élargit significativement la portée des preuves dans Squirrel, en étendant l'ensemble des hypothèses calculatoires cryptographiques supportées. Pour valider notre approche, nous l'avons appliquée à des études de cas, en reproduisant des preuves existantes dans Squirrel et en traitant de nouveaux cas qui n'étaient pas prouvables auparavant. Ce travail culmine avec la première preuve mécanisée de la confidentialité des votes pour le protocole de vote électronique FOO — la plus grande preuve réalisée à ce jour dans Squirrel.

Contents

Abstract iii

Résumé v

1 Introduction 1
- 1.1 Protocols and attacks . . . 1
  - 1.1.1 Alice, Bob and Mallory . . . 1
  - 1.1.2 Taking a step back . . . 5
- 1.2 Modelling . . . 6
  - 1.2.1 Users, attacker model, and oracle programming language . . . 6
  - 1.2.2 Taking a step back . . . 8
- 1.3 Security proofs . . . 9
  - 1.3.1 Cryptographic hardness assumptions . . . 9
  - 1.3.2 Security properties . . . 10
  - 1.3.3 Reasoning: game hops and cryptographic reductions . . . 11
- 1.4 Mechanizing proofs . . . 13
  - 1.4.1 CCSA approaches . . . 14
  - 1.4.2 Computer-aided verification . . . 20
- 1.5 This thesis . . . 21
  - 1.5.1 Starting problem . . . 21
  - 1.5.2 Contributions . . . 21
  - 1.5.3 Related work . . . 23

2 The CCSA-HO Logic 27
- 2.1 Types and type structures . . . 28
- 2.2 Terms . . . 29
  - 2.2.1 Variables and typing environments . . . 30
  - 2.2.2 Term syntax and semantics . . . 31
- 2.3 Recursion . . . 34
  - 2.3.1 Environments and models . . . 34
- 2.4 Probabilistic logic . . . 34

3 Formal Model for Cryptographic Reduction 39
- 3.1 Overview and motivating example . . . 39
- 3.2 Syntax . . . 44
  - 3.2.1 Expressions and programs . . . 44
  - 3.2.2 Games . . . 45
  - 3.2.3 Simulators and adversaries . . . 46

3.3 Semantics . . . 47
3.3.1 Memories . . . 47
3.3.2 Program random tapes . . . 47
3.3.3 Expression and program semantics . . . 48
3.3.4 Cost model . . . 50
3.3.5 Adversaries . . . 51
3.3.6 Adversaries and security . . . 51

4 Bideduction **53**
4.1 Overview . . . 54
4.2 Bideduction judgement . . . 57
4.2.1 Name constraints . . . 57
4.2.2 Assertion logic . . . 59
4.2.3 Bideduction judgement . . . 60
4.3 Bideduce rule . . . 63
4.4 Chapter appendix: couplings . . . 64
4.4.1 Preliminaries: probability theory . . . 64
4.4.2 Couplings and lifting lemma . . . 65
4.4.3 Well-formedness of constraint systems . . . 67
4.4.4 Couplings arrays . . . 68
4.4.5 Constructing a coupling contained in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ . . . 71
4.4.6 Proof of Theorem 1 . . . 72

5 Bideduction Proof System **75**
5.1 Overview . . . 76
5.2 Proof system . . . 77
5.2.1 Preliminary definitions . . . 77
5.2.2 Inference rules . . . 79
5.2.3 Example . . . 82
5.3 Soundness . . . 84
5.3.1 Preliminary definitions . . . 84
5.3.2 Validity and well-formedness lemmas . . . 85
5.3.3 Memory flow lemmas . . . 89
5.3.4 Computation lemmas . . . 90
5.3.5 Adversary lemmas . . . 90
5.3.6 Footprint lemma . . . 91
5.3.7 Structural rules . . . 92
5.3.8 Adversarial rules . . . 96
5.3.9 Computational rules . . . 99

6 Automation **107**
6.1 Motivating example: an abstract mixnet . . . 108
6.2 Preliminary definitions . . . 111
6.2.1 Standard library . . . 111
6.2.2 Assertion logic . . . 113
6.3 Basic simulator synthesis . . . 118
6.3.1 Synthesis queries . . . 118

6.3.2 Synthesis query rules . . . 119
6.3.3 The basic simulator synthesis procedure . . . 122
6.4 Inductive simulator synthesis . . . 127
6.4.1 Invariant synthesis . . . 127
6.4.2 Example . . . 129
6.4.3 Soundness . . . 131

7 Implementation and Case Studies **137**
7.1 Preliminary on SQUIRREL syntax . . . 137
7.2 Implementation . . . 139
7.2.1 Inputs of the **crypto** tactic . . . 140
7.2.2 Outputs of the **crypto** tactic . . . 141
7.3 Case studies . . . 143
7.4 The FOO protocol . . . 144
7.4.1 Cryptographic primitives and assumptions . . . 145
7.4.2 The FOO protocol security . . . 153
7.4.3 Proof . . . 155

Conclusion **159**

List of Figures **162**

Bibliography **163**

Chapter 1

Introduction

Everyday life has become intertwined with technologies. Our contemporary societies rely on digital tools for daily usages (chat with friends, mail at work, storing photos, etc.), but these technologies also play a central role in our democratic life, our health or educational systems, etc. This is not inconsequential. In particular, these technologies have improved our communications (chat, clouds, numerical identities, etc.) but with that comes the questions: How to prevent private data leaks? How to protect people's privacy? How to prevent malicious interferences? The generalization of remote communications in every layer of society makes these questions crucial. Designing ways to communicate that answer these challenges is an old research interest. This is the core of the science called **cryptography**. This thesis lies in this large research area that tackles the security of remote communications.

You have arrived at the beginning of this thesis. In this part, you will find a description of its context. What are the problems that cryptography addresses and in particular how this science expresses and studies them. To illustrate this, we use an example: a staged problem, enacting two people Alice and Bob facing a security challenge. We will use this example to give an overview of the different directions taken by the research community and, more precisely, where this thesis fits. This beginning is designed to be understandable with little technical knowledge. We want the context and general problem to be understandable for anyone interested. Then, the introduction becomes more complicated as we progress to the end. We narrow to fellow scientists to explain the framework of this thesis and explain the contributions this thesis makes to the domain.

1.1 Protocols and attacks

1.1.1 Alice, Bob and Mallory

Context - Alice and Bob problem

Alice is a journalist. In her line of work, she often encounters situations where secret is important. Typically, she manages identities of whistleblowers, ongoing investigations' data, individual sources, and so on, which in some cases could even endanger her or concerned people if revealed.

Bob is one of her collaborators. They are working together on a sensitive case: say an investigation on exactions committed by a powerful state. They need to meet online to exchange sensitive information and their conversations must be kept hidden from anyone. That is, they want everything their computers send to one another to be confidential, when they both connect to the remote call.

> We can also add more examples of digital security requirements, such as protecting internet browsing privacy (which could reveal health or political opinions, for example), securing databases against leaks (such as emails and phone numbers), ensuring secrecy or authentication in internet voting, ...

Encryption and decryption

Luckily, Alice and Bob already know how to hide their messages' content: by encrypting them. An encryption mechanism is a **probabilistic** procedure that takes a message m, the **plaintext**, and transforms it into an encrypted message c, the **ciphertext**, such that no information about m can be retrieved from c[1]. In general, the asymmetric encryption mechanism works as follows: Alice samples a **secret key** $\mathsf{sk_a}$ and computes the associated **public key** pk_a, which is publicly associated to her. If Bob wants to send her a message m only she can read, he encrypts m into the ciphertext $\{m\}_{pk_a}$. Finally, a decryption mechanism is the inverse mechanism to decrypt ciphertexts. To retrieve m from $\{m\}_{pk_a}$, Alice needs the secret key $\mathsf{sk_a}$. Since this key is secret, only she should be able to decrypt. The decryption of c is noted $dec(c, \mathsf{sk_a})$, and assume that the decryption with the secret key $\mathsf{sk_a}$ of the encryption of any message m with the public key pk_a yield m, or:

$$dec(\{m\}_{pk_a}, \mathsf{sk_a}) = m.$$

Bob also owns a secret key $\mathsf{sk_b}$ that he sampled. Alice can use the associated public key pk_b to send messages that only he can read.

Unfortunately, asymmetric encryption is not well suited for video calls. Better-suited procedures are symmetric encryption, notably faster than their asymmetric counterpart. These procedures are also encryption procedures but where decryption and encryption use the same key. This means that, in that kind of procedure, a single key key is used to encrypt any message m into a ciphertext $\{m\}_{\mathsf{key}}$, and to decrypt c back to $dec(c, \mathsf{key})$. To use such procedures to encrypt their call, Alice and Bob must share a secret key key. So they need to establish a key known only by themselves. They decide that each of them is going to sample half of it, the final key will be the concatenation of the two halves. That is Alice samples a half key $\mathsf{n_a}$, Bob samples a half key $\mathsf{n_b}$ and the final key is $\mathsf{key} = \mathsf{n_a} \cdot \mathsf{n_b}$.

They both need to exchange their respective part: $\mathsf{n_a}$ and $\mathsf{n_b}$. In some ways, we are still at square one: Alice and Bob need to exchange confidential messages. But now Alice and Bob only need to exchange two half keys, which is a small amount of data. They can use an asymmetric encryption procedure for that. *The question is: how can they use this asymmetric encryption procedure to send each other their share of the symmetric key, using only their public key materials?*

The first thought one might have is that Alice can send her part encrypted with Bob's public key pk_b, that is $\{\mathsf{n_a}\}_{pk_b}$, and Bob can send his part encrypted with Alice's public

[1] Except the length the message, which can never be completely hidden.

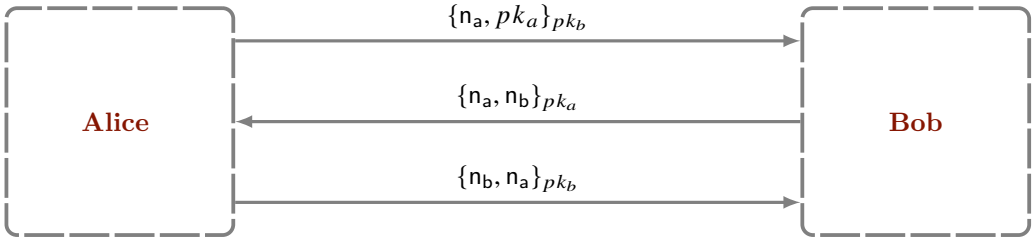


Figure 1.1: The Niedam-Schroder protocol.

key pk_a, that is $\{\mathsf{n_b}\}_{pk_a}$. But, in that case, Alice or Bob cannot be sure that what they received has indeed been sent by the other, and not someone else. Indeed, the keys pk_a and pk_b are public and anyone can encrypt messages destined to Alice or to Bob.

Protocol

Alice and Bob make a quick research, and find an academic research article solving their problem [1]. They decide to follow the Needham-Schroeder protocol. It has been proven secure by a pen and paper math proof.

It works as follows, and is described in Figure 1.1:

- First, Alice sends her share of the symmetric key, $\mathsf{n_a}$, to which she attaches her public key pk_a to identify her message. She encrypts the whole with Bob's public key pk_b, ensuring that only Bob can retrieve $\mathsf{n_a}$.
- Upon receiving this message, Bob can decrypt it with his secret key. The identity attached to $\mathsf{n_a}$ informs him that Alice wants to exchange her share with him. He samples his share $\mathsf{n_b}$, sends it back attached to $\mathsf{n_a}$ to authenticate it and encrypted with Alice's key.
- Finally, Alice decrypts this last message, and sends back $\mathsf{n_b}$ and $\mathsf{n_a}$ encrypted with Bob's key, to signal to Bob she has received the half key and so that Bob can check she received what he had sent.

The final key Alice and Bob will use is the concatenation of $\mathsf{n_a}$ and $\mathsf{n_b}$.

But Alice and Bob did not dig enough for the research, and missed that this protocol has a flaw.

Let us now say that Mallory is a state agent, from the state incriminated by Alice and Bob's investigation. They are tasked to interfere and retrieve the data Alice and Bob possess. Furthermore, we assume that the secrecy of protocol design does not participate in ensuring Alice and Bob's key is kept secret — the protocol's security must rely only on the secrecy of the encryption keys. In other words, the protocol should be secure even if Mallory knows exactly how it works.

> The principle that security protocols must ensure security assuming that attackers, like Mallory, knows exactly how it works in detail, is called Kerckhoffs's principle [2]. It states that the security of cryptographic systems must rely solely on the secrecy of secret values, and not on the secrecy of the mechanism itself. It has shaped the

contemporary approach to cryptographic research.

Let us take the worst-case scenario: Mallory works for very powerful state and has full control of the network Alice and Bob use to communicate. They can block messages and send their own message instead, without any way for Alice and Bob to notice. We also assume they control other actors on the network.

Mallory will communicate with Alice by corrupting another colleague of her, while using her answers to make Bob believe he is talking to Alice. This kinds of attacks is called a *man in the middle* attacks, and can be problematic: in our case Mallory is able to retrieve Bob's information by impersonating Alice. The attacks is presented in Figure 1.2.

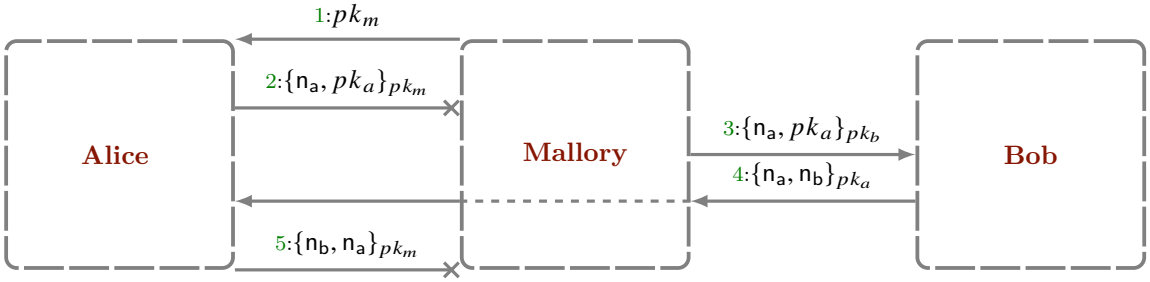


Figure 1.2: Mallory attack. Mallory signals to Alice they want to establish a symmetric encryption key with her (1). Alice sends to Mallory a half key $\mathsf{n_a}$ encrypted with Mallory's public key (2). Mallory can decrypt Alice's message and send to Bob the same message (3) but encrypted with Bob key. Bob believes Alice want to establish a symmetric key with him, and will sends back the answer dictated by the protocols (4). Mallory can forward it to Alice, who will decrypt it for them, and send back Bob's half key encrypted under *Mallory*'s key (5).

Luckily, Alice is made aware of this attack because by continuing her research, she found out that a researcher, Lowe, found the attack and suggested a correction to protect against it [3]. The key idea is that using $\mathsf{n_a}$ as proof of identity is not enough, so Bob must follow Alice's example, and also adds his public key to his message before encrypting with Alice's key. The protocol is then as follows in Section 1.1.1.

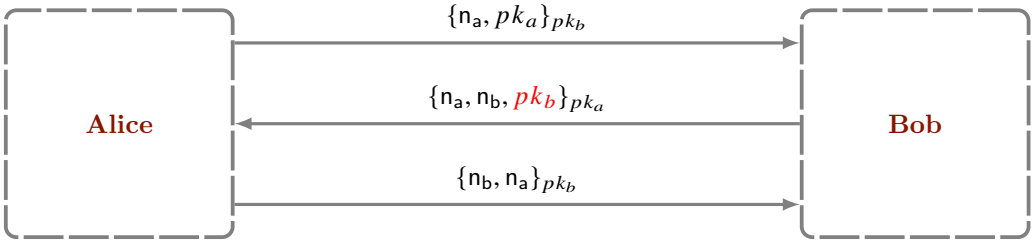


Figure 1.3: Needham-Schroder-Lowe protocol

Now, Mallory cannot mimic Bob response, because they cannot produce $\{\mathsf{n_a}, \mathsf{n_b}, pk_m\}_{pk_a}$ from only $\{\mathsf{n_a}, pk_a\}_{pk_m}$ and $\{\mathsf{n_a}, \mathsf{n_b}, pk_b\}_{pk_a}$, having no way to decrypt the latter message to build their own. Finally, Alice and Bob question this new protocol. The research article

by Needham and Schröder provided a proof, but appears to have a flaw. How can they be sure they are protected by this new protocol, and that Mallory would not be able to learn their secret key through another attack?

> The property capturing the fact that Bob knows the message he receives is indeed sent by Alice is called authentication (of Alice's identity in this case). There exist other cryptographic procedures that ensure authentication, called signatures. Also, in practice, such cryptographic procedures are implemented in code library, and embedded in a larger system that already provides the security such as the one Alice and Bob are looking for. We could quote Signal [4], that could be used by Alice and Bob to authenticate and make their messages private. Protocols are widely used for other purposes too. For example, the protocol TLS [5] is behind internet authenticate and confidentiality, the protocol Belenios [6] is used for e-voting, etc. Alice and Bob's example serves to introduce the reader to how the research community thinks through and tackles situations like the one of Bob and Alice. For educational purposes we consider that Alice and Bob will build their solution by themselves, using solely encryption.

1.1.2 Taking a step back

A protocol is a distributed system: a program whose parts run between several machines that communicate. A cryptographic protocol is a protocol build upon cryptographic mechanism, e.g. the encryption. These cryptographic functions are said to be the **primitives** of a protocol, because they serve as basic building blocks for it.

Cryptographic protocols are widely used to secure remote communication, especially in critical situations. In all these situations, users can legitimately ask whether the system that supposedly ensures the safety of their data is indeed working as intended. Hence, research tackles cryptographic protocol verification. There are two aspects for the verifications: the primitives, and the protocol while considering primitives as black boxes, and three layers of verifications for each. Each that could hide vulnerabilities.

- First, there can be vulnerabilities at design level. In our example, Mallory exploited a design flaw in Alice and Bob first protocol.
- Second, primitives and protocols still have to be implemented and compiled to be used. At this layer stands the verification that the actual programs ran by the computer do follow the intended design.
- Finally, the context in which the devices run their codes more generally yields vulnerabilities. The context includes specificities of the hardware but also extra channels of information (temperatures of the computer, time of executions, cash states, etc.) that can both be exploited to retrieve information. In the latter, these sources of information are called side channels.

It is unrealistic to verify all in one go. Each layer separately is an entire research area, with its specific difficulties. In this thesis, we consider the protocol dimension, at the level of design. For the rest of this part, we concentrate on how to ensure that no attack exploiting only the protocols design —ie that could be captured by this level of

abstraction— can occur, while abstracting the primitives as black boxes, that we assume safe.

> All layers can hide vulnerabilities. Importantly, abstracting one part of the protocol does not necessarily trivialize verification on the rest. In particular, abstracting primitives and implementation can find vulnerabilities, see [7] or [8].
> Still, keep in mind that some specific attacks will me missed at this level. Indeed, on a software and hardware levels, we can quote the famous attacks Heartbleed [9] that exploit out of bound behavior or Spectre [10], that exploits caches' behaviour. For works on vulnerabilities of primitives themselves, [11] is a good example.

1.2 Modelling

A mathematical proof can provide stronger guarantees to Alice and Bob that NSL can be used safely. For that, we need a mathematical framework to model the scenario. In this section, we first examine how to model the honest users (Alice and Bob) and the attacker (Mallory), and then how their interactions can be expressed using their mathematical descriptions.

1.2.1 Users, attacker model, and oracle programming language

Alice, Bob, and Mallory all three had computers to respectively execute the protocol steps or compute their attack.

Programming language

The key question is how do we model computation? Recall that we abstract away any hardware aspect of computers. So, very basically, computing something is executing a program. Let us consider an arbitrary probabilistic programming language; which provides basic imperative instruction: while loops, branching, sequences, basic mathematical operations... i.e. enough instructions to compute anything a computer can. We model Alice and Bob's behaviour with programs in this probabilistic language.

Attacker model

In the modelling there is an important difference between the users Alice, Bob and Mallory. Alice and Bob are honest users, meaning they will follow the protocol exactly as intended. Mallory is an *unknown* program, that can interfere in Alice and Bob's interaction. That means Mallory can:

- see everything passing through the network,
- block messages
- and mimic any other agent in the network (e.g. Alice and Bob's colleagues).

In other words, the entire network itself except for Alice and Bob is a part of Mallory, modelled as an arbitrary program.

In that case, we say that Mallory is an **active attacker**, and that only Alice and Bob are trusted or honest users. We sometimes reduce the attacker capabilities on the network to only being able to listen. In that case, we say that the attacker is **passive**.

Oracle programming language

Finally, let us reconcile the two aspects of our modelling. On one hand, we have Mallory managing the network, up to piloting when and what Alice and Bob receive. On the other, we have Alice and Bob who strictly follow the protocol course of action: they wait to receive a given message, run a given program defined by the protocol to compute an answer.

We group each programs into oracles. An oracle is a function, such that when given a message m as input, runs a program and outputs the results of the computations. We then have three main oracles, the oracles mimicking all of Alice and Bob behaviours.

- The oracle $\mathsf{Alice1}$ captures Alice's first message. Upon receiving as input a public key pk, the public key of someone who want to initiate the protocol with her, Alice stores pk and sends back the encryption of her half key $\mathsf{n_a}$ encrypted with this pk. This yields the oracle:

$$\begin{aligned}\mathsf{Alice1}(pk) \stackrel{\mathsf{def}}{=} \; & \mathsf{x}_{pk} \leftarrow pk \\ & \mathbf{return}\{\mathsf{n_a}, pk_a\}_{\mathsf{x}_{pk}}\end{aligned}$$

- The oracle $\mathsf{Alice2}$ captures Alice's second message. She receives a ciphertext c, that she decrypts with her key. She parses it as the tuple $(\mathsf{n'_a}, \mathsf{n'_b}, pk')$. Following NSL's specification, we should have $\mathsf{n'_a} = \mathsf{n_a}$ and $pk' = pk$. If this is not the case, Alice aborts; otherwise she sends back the encryption $\{\mathsf{n'_b}, \mathsf{n_a}\}_{pk}$. Note that, then, the key she derives is $\mathsf{n_a} \cdot \mathsf{n'_b}$. This yields the oracle:

$$\begin{aligned}\mathsf{Alice2}(c) \stackrel{\mathsf{def}}{=} \; & (\mathsf{n'_a}, \mathsf{n'_b}, pk'_b) \leftarrow \mathsf{dec}(c, \mathsf{sk_a}) \\ & \mathbf{if}\ \mathsf{n'_a} = \mathsf{n_a} \wedge pk'_b = \mathsf{x}_{pkb}\ \mathbf{then\ return}\ \{\mathsf{n'_b}, \mathsf{n_a}\}_{\mathsf{x}_{pkb}} \\ & //\text{Alice derives the key } \mathsf{n_a} \cdot \mathsf{n'_b}\end{aligned}$$

- The oracle Bob captures Bob's sole message. He receives a ciphertext c that he decrypts with his key. He parses it as the tuple $(\mathsf{n'_a}, pk'_a)$ and sends back the encryption $\{\mathsf{n'_a}, \mathsf{n_b}, pk_b\}_{pk'_a}$. Note that, then, the key he derives is $\mathsf{n'_a} \cdot \mathsf{n_b}$. This yields the following oracle:

$$\begin{aligned}\mathsf{Bob}(c) \stackrel{\mathsf{def}}{=} \; & (\mathsf{n'_a}, pk'_a) \leftarrow \mathsf{dec}(c, \mathsf{sk_b}) \\ & \mathbf{return}\ \{\mathsf{n'_a}, \mathsf{n_b}, pk_b\}_{pk'_a} \\ & //\text{Bob derives the key } \mathsf{n'_a} \cdot \mathsf{n_b}\end{aligned}$$

Also, we add an initialization oracle, that samples Alice and Bob's secret keys and key shares, and derives Alice and Bob's corresponding public keys. This is the oracle:

$$\begin{aligned}\mathsf{Init}() \stackrel{\mathsf{def}}{=}\ & \mathsf{sk_a} \xleftarrow{\$} \\ & \mathsf{sk_b} \xleftarrow{\$} \\ & pk_a \leftarrow pk(\mathsf{sk_a}) \\ & pk_b \leftarrow pk(\mathsf{sk_b}) \\ & \mathsf{n_a} \xleftarrow{\$} \\ & \mathsf{n_b} \xleftarrow{\$} \\ & \mathbf{return}(pk_a, pk_b)\end{aligned}$$

The protocol NSL is then characterized by the set of oracles:

$$\mathsf{NSL} \stackrel{\mathsf{def}}{=} \{\mathsf{Init}; \mathsf{Alice1}; \mathsf{Alice2}; \mathsf{Bob}\}$$

The Init oracle must always be called first —to initialize all data that Alice and Bob's oracles need— and Alice's second message oracle Alice2 can only be called after Alice's first message oracle Alice1.

Mallory is an unknown program with access to these oracles but without access to their internal variables. In their computations, Mallory can call the oracle whenever they want to get back what would be Alice and Bob's outputs.

Finally, for the sake of simplicity we consider only one round of the NSL protocol. That is, each oracle can be called only once.

1.2.2 Taking a step back

Actually, the model used here, that sees users and attacker as programs, is called the **computational model** [12]. In practice, this model represents users by probabilistic Turing Machines, one of the classical mathematical modelling of computation. Roughly, Turing Machines are machines with memory and basic instructions to read and write in atomic memory cells (i.e. generally one bit). Theoretically, it is enough to capture the complex behaviour that we know of from computers. However, this level of granularity in Turing Machines makes them very painful to work with. Programs, on the other hand, provide a more abstract and macroscopic but equivalent view, making them more suitable for our needs.

Another approach to this type of modelling exists, which differs in how the attacker is represented. The **symbolic model** [13] focuses on a logical description of the interaction between the protocol and the adversary, abstracting away probabilistic considerations. It models messages using a term algebra and random values as fresh symbols. The key difference with the computational model is that, in the symbolic, one must define the set of adversary's capabilities that must be captured by the algebra, while in the computational model the adversary has any capabilities of a probabilistic Turing Machine. In our example, a suitable term algebra should cover the encryption, decryption, pair functions to capture Mallory's behaviour, while in the computational model these capabilities are already covered, as Mallory can do anything a computer can, but Mallory has also access to many more capabilities, e.g. xor, loops, etc.

In a nutshell, the symbolic model focuses on *logical attacks*, at the cost of restricting the attacker capabilities more than it is in practice. The computational model goes the other way. It keeps a realistic attacker but at the cost of having a complex model in which protocols are expensively difficult to prove secure. Both approaches have been successfully applied in the literature. In the symbolic, we can notably cite major protocol proofs such as SPDM [14], EMV [15], MLS [16], etc. For the computational model, we can cite the verification of Signal [17] or of AWS key management [18].
The two approaches are complementary, the symbolic model is easier to use and can scale to larger protocols, the computational model provides stronger security guarantees. Also, the symbolic approach can miss attacks that the computational model would not. See for example [19].

1.3 Security proofs

Now, what is left is to formulate our security property in this model. In our approach, we leave the specifics of encryption design and implementation to other research areas, and crucially, the encryption security modelling. It will be easier for us if we formulate the protocol's security property using the same formalism as the cryptographic hypothesis.

1.3.1 Cryptographic hardness assumptions

In our example, the security property of the encryption scheme is that the encryption does not reveal anything about the encrypted message, besides its length. Cryptographers formalize this idea with the notion of indistinguishability between games.

We think of the hypothesis as an impossibility: the impossibility for any attacker to distinguish the encryption of two different messages.

Theoretically, we express it with **indistinguishability** of two games. A game is a set of oracles that an attacker can call. We challenge an adversary to guess whether it is interacting with game $\mathcal{G}_0$ or with game $\mathcal{G}_1$. In both games, the adversary has access to an initialization oracle, that returns a public key $pk(\mathsf{sk})$. In the first game, $\mathcal{G}_0$ $\mathcal{A}$ is also given access to an encryption oracle. This oracle must be called with two inputs m_0 and m_1, and returns the encryption of m_0 with the public key $pk(\mathsf{sk})$. In the second game, $\mathcal{G}_1$, $\mathcal{A}$ is given access to a similar oracle, but that outputs the encryption of the second input m_1. These oracles are:

$$\begin{aligned}\mathsf{Init}() \stackrel{\mathsf{def}}{=} \ & \mathsf{sk} \xleftarrow{\$}; \\ & pk \leftarrow pk(\mathsf{sk}) \\ & \textbf{return } pk(\mathsf{sk})\end{aligned}$$

$$\mathsf{Encrypt}_{\mathcal{G}_0}(m_0, m_1) \stackrel{\mathsf{def}}{=} \textbf{ if } \mathsf{len}(m_0) = \mathsf{len}(m_1) \textbf{ then return } \{m_0\}_{pk}$$

$$\mathsf{Encrypt}_{\mathcal{G}_1}(m_0, m_1) \stackrel{\mathsf{def}}{=} \textbf{ if } \mathsf{len}(m_0) = \mathsf{len}(m_1) \textbf{ then return } \{m_1\}_{pk}$$

Finally, the two games are defined as the following sets of oracles:

$$\mathcal{G}_0 \stackrel{\text{def}}{=} \{\mathsf{Init}; \mathsf{Challenge} \stackrel{\text{def}}{=} \mathsf{Encrypt}_{\mathcal{G}_0}\} \qquad \mathcal{G}_1 \stackrel{\text{def}}{=} \{\mathsf{Init}; \mathsf{Challenge} \stackrel{\text{def}}{=} \mathsf{Encrypt}_{\mathcal{G}_1}\}$$

We say that the encryption is secure if no attacker can distinguish between the two games $\mathcal{G}_0$ and $\mathcal{G}_1$. But, formulated like this, it can never exist encryption functions that are secure. Indeed, one winning strategy for the attacker would be to try to decrypt the oracle output values for any possible value of the secret key. The key sk is a bitstring of size η, then the attacker has 2^η possible keys to test. Furthermore, if it samples uniformly different keys to test, it has a probability of $\frac{t}{2^\eta}$ to find the key in t tries. Then, with enough time, it finds the key with probability 1. Hence, we define the security when we restrict t to be "small enough", so that the risk $\frac{t}{2^\eta}$ is negligible, and acceptable for users.

The real typical attackers that will challenge the encryption security are attackers like Mallory. The trick is to note that they only have a limited amount of time to guess the key before Alice and Bob call, otherwise they fail.

Hence, we can make the value of η big enough so that the chances they guess correctly in realistic time is negligible, which represents a risk Alice and Bob can agree to take.

Finally, we say the two games $\mathcal{G}_0$ and $\mathcal{G}_1$ are **indistinguishable** if and only if all attackers have negligibly the same behaviour in the two versions. In that case, we say that our encryption mechanism is **IND-CPA** secure. More formally, the advantage $Adv_{\mathcal{G}_0,\mathcal{G}_1}(\mathcal{A})$ of an attacker $\mathcal{A}$ against $\mathcal{G}_0, \mathcal{G}_1$ quantifies how much $\mathcal{A}$'s behaviour differs in the two versions, or

$$Adv_{\mathcal{G}_0,\mathcal{G}_1}(\mathcal{A}) = |\Pr(\mathcal{A}^{\mathcal{G}_1}(1^\eta) = 1) - \Pr(\mathcal{A}^{\mathcal{G}_0}(1^\eta) = 1)|$$

where $\mathcal{A}^{\mathcal{G}_1}(1^\eta)$ and $\mathcal{A}^{\mathcal{G}_1}(1^\eta)$ are the outputs made by $\mathcal{A}$ when interacting with, respectively, $\mathcal{G}_0$ and $\mathcal{G}_1$. Our encryption mechanism is IND-CPA secure if and only if the advantage of any attacker running in **polynomial time in** η is **negligible in** η. That is, the advantage approaches 0 faster than the inverse of any polynomial in η. This is summarized in Definition 1.

Definition 1. *An encryption mechanism is IND-CPA secure if for all probabilistic polynomial-time adversaries $\mathcal{A}$ against the indistinguishability of $\mathcal{G}_0$ and $\mathcal{G}_1$, its advantage is negligible in η.*

1.3.2 Security properties

Finally, we can express the protocol's security property in our model. We will mimic the way the cryptographic assumptions are formulated, i.e. we will use a formulation using indistinguishability of games. To that end, we need to find two games, $\mathcal{G}_0$ and $\mathcal{G}_1$, such that their indistinguishability implies that Mallory cannot learn anything about the secret key Alice and Bob derive. That property is called **key secrecy**. Going back to our running example, Mallory is the attacker, and it has access to oracles that mimic Alice and Bob's behaviour: the oracles Init, $\mathsf{Alice1}$, $\mathsf{Alice2}$ and Bob.

We need one last oracle, the $\mathsf{Challenge}$ oracle, to capture the security property. First, notice that *two* keys are actually derived. Indeed, Alice and Bob both derive a final key. Hence, we have two key secrecies: that of the key derived by Alice and that of the key derived by Bob. In the attack example, it was Bob's key secrecy that was broken, so let's

formulate Bob's key secrecy property. It makes sense to verify this property only when Bob thinks he is talking to Alice, otherwise, he will not use the key he had derived to talk to Alice afterward. Therefore, we want to be sure that Mallory cannot learn anything about the secret key $\mathsf{n}'_\mathsf{a} \cdot \mathsf{n}_\mathsf{b}$ when Bob thinks he has been talking to Alice and detects no disruption in the exchange.

> Checking that Alice and Bob derive the same key is another property of the protocol, the key agreement, that we do not tackle here.

Following the IND-CPA game intuition, we define two versions of a challenge oracle, to be added to NSL oracles. The attacker is supposed to call the oracle with Alice's last message. In the first version, the attacker receive the secret key $\mathsf{n}'_\mathsf{a} \cdot \mathsf{n}_\mathsf{b}$ derived by Bob. In the second version, it receives a freshly sampled key key, independent of Alice and Bob exchanges. In both cases, the challenge oracle outputs $\mathsf{n}'_\mathsf{a} \cdot \mathsf{n}_\mathsf{b}$ or key only under the following conditions:

- from Bob's point of view, he was taking to Alice, that is $pk'_a = pk_a$; and
- there was not be disruption in the protocol execution: the input of challenge oracle corresponds to Alice's last messages. That is $\mathsf{dec}(m, \mathsf{sk_b}) = (n_b, n'_a)$, with m the input of the oracle.

The two games are as follows.

$$
\begin{array}{l}
\mathcal{G}_0^{NSL} \stackrel{\mathsf{def}}{=} \{ \\
\quad \mathsf{include}\ NSL; \\
\quad \mathsf{Challenge}(m) \stackrel{\mathsf{def}}{=}\ \textbf{return if}\ pk'_a = pk_a \wedge \mathsf{dec}(m, \mathsf{sk_b}) = (n_b, n'_a)\ \textbf{then}\ \mathsf{n}'_\mathsf{a} \cdot n_b \\
\quad \}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}_1^{NSL} \stackrel{\mathsf{def}}{=} \{ \\
\quad \mathsf{include}\ NSL; \\
\quad \mathsf{Init} \stackrel{\mathsf{def}}{=}\ \mathsf{NSL.Init}(); \mathsf{key} \xleftarrow{\$} \\
\quad \mathsf{Challenge}(m) \stackrel{\mathsf{def}}{=}\ \textbf{return if}\ pk'_a = pk_a \wedge \mathsf{dec}(m, \mathsf{sk_b}) = (n_b, n'_a)\ \textbf{then}\ \mathsf{key} \\
\quad \}
\end{array}
$$

As it has been done earlier, we say that the protocol ensures the secrecy of Bob's key if for all attackers $\mathcal{A}$, the advantage of $\mathcal{A}$ against $\mathcal{G}_0^{NSL}$ vs. $\mathcal{G}_1^{NSL}$ is negligible in η.

1.3.3 Reasoning: game hops and cryptographic reductions

Finally, we have our hypothesis —the encryption games — and our proof goal. The last question is how do we use our hypothesis to prove our goal.

The usual method in cryptographic proof is to use game hopping [20]. The idea is as follows: on the left hand, we have the game $\mathcal{G}_0^{NSL}$ and on the right hand we have $\mathcal{G}_1^{NSL}$. Between the two, we add more games to gradually transform the left game into the right one. We then prove the indistinguishability between each transformation —**the game hops**.

For our example, we introduce two more games: $\mathcal{G}_0^{Ideal}$ and $\mathcal{G}_1^{Ideal}$. They are both an ideal version of NSL games, where any encryptions under Alice and Bob public keys encrypts dummy message rather than original plaintexts.

We then chain the game hops as follows:

$$\mathcal{G}_0^{NSL} \sim \mathcal{G}_0^{Ideal} \sim \mathcal{G}_1^{Ideal} \sim \mathcal{G}_1^{NSL}.$$

We thus have to prove three indistinguishabilities:

- $\mathcal{G}_0^{NSL}$ versus $\mathcal{G}_0^{Ideal}$;
- $\mathcal{G}_0^{Ideal}$ versus $\mathcal{G}_1^{Ideal}$. In these two games, the only things the adversary can see before calling the challenge oracle are encryption of dummy messages, that does not reveal anything on the symmetric key derived at the end.
- And $\mathcal{G}_1^{NSL}$ versus $\mathcal{G}_1^{Ideal}$.

Let us concentrate solely on the first proof. By contraposition of the indistinguishability property, let us assume there exists an attacker that breaks the indistinguishability $\mathcal{G}_0^{NSL} \sim \mathcal{G}_0^{Ideal}$ and let us call it $\mathcal{M}$. The advantages of $\mathcal{M}$ against the indistinguishability of $\mathcal{G}_0^{NSL}$ and $\mathcal{G}_0^{Ideal}$ is *not* negligible in η. Now we have an entire program $\mathcal{M}$ at our disposal to build other programs.

In particular, we can try to build a program $\mathcal{A}$ to break the *cryptographic assumption* indistinguishability. Should we succeed, then we would have contradicted our hypothesis. Hence, we would have shown that no such $\mathcal{M}$ can exist and so that no attacker breaks $\mathcal{G}_0^{NSL} \sim \mathcal{G}_0^{Ideal}$.

Let's proceed that way. Let's start running $\mathcal{M}$. It will do computation on its own, and might at some point, call on the oracles of $\mathcal{G}_0^{NSL}$ and $\mathcal{G}_0^{Ideal}$[2]. Let us say, it has called $\mathsf{Alice1}$ and is waiting for an answer. We need a program that simulates it, using the IND-CPA's oracles. Recall that this oracle is in the $\mathcal{G}_0^{NSL}$:

$$\begin{aligned}\mathsf{Alice1}(pk) \stackrel{\mathsf{def}}{=}\ & x_{pk} \leftarrow pk \\ & \textbf{return } \{\mathsf{n_a}, pk_a\}_{x_{pk}}\end{aligned}$$

Also, the $\mathcal{G}_0^{Ideal}$ is obtained by replacing the encrypted messages by a dummy one when Alice uses Bob's public key:

$$\begin{aligned}\mathsf{Alice1}(pk) \stackrel{\mathsf{def}}{=}\ & x_{pk} \leftarrow pk \\ & \textbf{if } x_{pk} = pk_b \textbf{ then return } \{\mathsf{dummy}_{A1}\}_{x_{pk}} \textbf{ else return } \{\mathsf{n_a}, pk_a\}_{x_{pk}}\end{aligned}$$

[2]Recall that they share the same API of oracles

Let $\mathcal{S}_{A1}$ be the following program

$$
\begin{aligned}
S_{A1}(pk) \stackrel{\text{def}}{=}\ & x_{left} \leftarrow (\mathsf{n_a}, pk_a);\\
& x_{righ} \leftarrow \mathsf{dummy}_{A1};\\
& \mathsf{res} \leftarrow \textbf{if } pk = pk_b \textbf{ then } \mathsf{call\ Challenge}(x_{left}, x_{right}) \textbf{ else } \{x_{left}\}_{pk}\\
& \textbf{return } \mathsf{res}
\end{aligned}
$$

Note that $\mathcal{S}_{A1}$ simulates Alice1's response, meaning that executing $\mathcal{S}_{A1}$ with game $\mathcal{G}_0$ (resp. $\mathcal{G}_1$) will yield the same output as calling Alice1 in the game $\mathcal{G}_0^{NSL}$ (resp. $\mathcal{G}_0^{Ideal}$). Our proof goal is thus to build such an $\mathcal{S}$, a **simulator**, that acts has an interface between $\mathcal{M}$ and the IND-CPA oracles, by mimicking the NSL oracles whenever $\mathcal{M}$ calls them. Then, our adversary $\mathcal{A}$ is the full program regrouping $\mathcal{M}$ interfaces with $\mathcal{S}$, that have the same advantage against IND-CPA that $\mathcal{M}$ has against $\mathcal{G}_0^{NSL} \sim \mathcal{G}_0^{Ideal}$. The advantage of $\mathcal{A}$ is not negligible, and so our IND-CPA assumption is violated.

However, if we look at the other oracles, there is a capability we do not have: decryption by the secret key. That means the IND-CPA assumption was too weak for our purpose. But we have chosen it that way to make this example easier. In practice, there are a variety of cryptographic assumptions on encryption mechanisms. One in particular has also a decryption oracle, the **CCA2 assumption**. Writing the reduction with it, however, is very tedious. Indeed, allowing decryption oracles requires adding memory conditions in the games: to make sure we do not decrypt ciphertext outputs by the challenge oracle, and following the memory evolution throughout the reduction is not necessarily straightforward.

> In practice, proving protocols relies on several cryptographic assumptions, and protocols are more distant to the assumptions.
> In a game hopping proof, cryptographic reductions are only one means among others to prove a game hop. Games indistinguishabilities can also rely on other technics like 'up-to-bad' argument [21], hybrid argument [22], etc. Cryptographic reductions will be crucial in this thesis, that is why we used the opportunity to give a first insight here.

1.4 Mechanizing proofs

Writing out the proofs for even this small example would take too long and be tedious. The proof for the actual protocol seems too painful. Moreover, note that the initial protocol that Alice and Bob were shown to be secure had a vulnerability that Mallory was able to exploit, which the handwritten proofs failed to detect.

For all these reasons,[3] it can be worthwhile to have computers make all the hard work for us — or at least assist us in doing it. And for that, we need to formalize and mechanize cryptographic proofs and make them understandable for a computer. The question is how?

We bring to the reader's attention that starting here the content becomes very technical. We give in this section an overview of the formal setting this thesis uses.

[3] And because a teacher once said to me that a good computer scientist is a lazy one, especially if their laziness leads to the development of an entire research area.

1.4.1 CCSA approaches

Let us concentrate on the CCSA —Computationally Complete Symbolic Attacker— approaches to the problem. The initial CCSA approach was formalized by Bana and Common [23] followed by other approaches either extending or revising the approach. The ground basis of these approaches is that we always represent the joint executions of the adversary and the honest users using **terms**, and the property of this joint execution using **formulae**. A proof system then describes logical reasoning on formulae, whose soundness derives from cryptographic reasoning. In this thesis, we use a **highter-order extension** [24] of the original CCSA approach. For the rest of the thesis, this approach (and its logic) will be called the CCSA-HO approach (and logic)—or solely the logic when clear from the context. In this section, we give an overview of the principal components and intuition of the CCSA-HO approach.

Let us first formalize the objects which we reason about. Following the previous guideline, let us first try the joint executions of programs. Programs appear in two forms in our model: under oracles for the honest computations of Alice and Bob, and as arbitrary programs — the attackers.

Terms.

We use **terms** to describe computations and in particular messages of the protocols. This messages can be cut into two categories: honest ones (returned by honest users, i.e. Alice and Bob in our running example), issued from honest computations, and adversarial ones, issued from unknown computation. For honest computation, CCSA approaches separate deterministic computations from probabilistic ones. We use **honest symbols** in terms to capture deterministic computations, while the honest randomness is regrouped under special symbols, **names**, representing the random samplings explicitly made during protocol executions. For adversarial messages, we use **adversarial symbols**. The only one we introduce in this section is att, a function symbol whose interpretation is any unknown adversarial computation (i.e. made by an unknown polynomial and probabilistic program). This is the symbol we use to represent all of Mallory's computation.

In this part we will describe a subset of terms, sufficient to understand the example. A full formal description is given in the next chapter. Terms are build upon basic bricks: names, honest symbols and att. This is summarized as follows.

$$
\begin{aligned}
t := \;& \mid \mathsf{n} \text{ with } n \text{ a name symbol} \\
& \mid f \text{ with } f \text{ a honest symbol} \\
& \mid \mathsf{att} \\
& \mid t\ t'
\end{aligned}
$$

Let us go back to our NSL games. Let's try to represent messages with terms. Let $\mathsf{sk_a}$, $\mathsf{sk_b}$ be names, to represent Alice and Bob's secret keys.

Then, the first message, sent during initialization, can be represented by the following term:

$$t_{\mathsf{init}} = (pk\ \mathsf{sk_a}, pk\ \mathsf{sk_b}).$$

Imagine now that after the initialization, the attacker calls Alice's first oracle Alice1. To compute an input, and send this messages to the challenger, the adversary has only retrieved the terms t_{init}. Hence, Alice's input from the adversary is

$$att\ [t_{\mathsf{init}}]$$

an arbitrary computation made from t_{init}. Then, let $\mathsf{n_a}$ be a name representing Alice's nonce, $\mathsf{r_{A1}}$ be a nonce representing the encryption randomness, and enc be an honest symbol, representing the encryption. The terms representing the output of Alice1 is

$$t_{A1} = \mathsf{enc}\ (\mathsf{n_a}, pk\ \mathsf{sk_a})\ (\mathsf{att}\ [t_{\mathsf{init}}])\ \mathsf{r_{A1}},$$

representing the encryption of $(\mathsf{n_a}, pk\ \mathsf{sk_a})$ under $(\mathsf{att}\ [t_{\mathsf{init}}])$ seen as a public key.

Continuing like that, we can define the term t that captures the interaction where Mallory gets the two public keys from the initialization, then called Alice first oracle and uses her responses to compute the input with which they called Bob, and finishes;

$$t = t_{\mathsf{init}}, t_{A1}, t_B \tag{1.1}$$

with

$$\begin{aligned} t_{in} &= \mathsf{dec}\ (\mathsf{att}\ [t_{\mathsf{init}}, t_{A1}])\ \mathsf{sk_b} \\ t_{na} &= \mathsf{fst}\ t_{in} \\ t_{pk} &= \mathsf{snd}\ t_{in} \\ t_B &= \mathsf{enc}\ (t_{na}, \mathsf{n_b}, pk(\mathsf{sk_b}))\ t_{pk}\ \mathsf{r_B} \end{aligned}$$

where we let $\mathsf{n_b}$ and $\mathsf{r_B}$ be names, and fst (resp. snd) be an honest symbol representing a function extracting the first (resp. second) element of a pair.

Frame encoding: recursion.

Terms can encode messages. Still, above we captured the sequence of messages seen by Mallory in one particular sequence of calls by a sequence of messages. This is not practical, since we would like to mechanize security proofs for any possible sequence of calls. So, the question is: is it possible to capture *arbitrary* sequences into terms in the CCSA-HO framework?

Let us go back to what an interaction with the NSL game $\mathcal{G}_0^{NSL}$ looks like. We have a sequence of messages — a **frame** of messages— with input messages computed by Mallory and output messages answered by either Alice or Bob. Each message can be represented by a term, but the complete frame is also determined by the order in which Mallory chose to call the oracles. Let us call a **trace** the data of which oracle is called when. When interacting with NSL, several traces are possibles. Mallory can respect the expected one for NSL, that is

$$(\mathsf{Init}, \mathsf{Alice1}, \mathsf{Bob}, \mathsf{Alice2}, \mathsf{Challenge}),$$

but they could choose other sequence of calls, like

$$(\mathsf{Init}, \mathsf{Alice1}, \mathsf{Bob}, \mathsf{Challenge})$$

where they do not call Alice's second oracle, or

$$(\mathsf{Init}, \mathsf{Bob}, \mathsf{Alice1}, \mathsf{Alice2}, \mathsf{Challenge})$$

where they call Bob before Alice.

To encode the frame of messages in terms, we choose to consider that traces are non-adaptative. That means that Mallory commits to follow a trace before their execution. This is a restriction: in practice, Mallory could choose whom to call based on their previous calls.

> This trace-centered point of view was inspired by the symbolic model, SQUIRREL aiming at first at bridging the gap between the computational model and the symbolic model. Still, this point of view restrict our model in a subtle way: we prove a result for all trace of an arbitrary, but fixed, length, while in the oracle setting, a reduction is for all possible interactions of polynomial size.

Let us annotate the calls in traces by time-points. We then add specific time-point symbols: let init be the initial time-point: the time when the initialization oracle is called, also let $\mathsf{A1}$, $\mathsf{A2}$, B and C be the symbols representing the time-points of the respective oracles $\mathsf{Alice1}$, $\mathsf{Alice2}$, Bob and $\mathsf{Challenge}$ — if called. Let τ, τ', etc. be time-point variables. Then a trace is the data needed to interpret the time-point symbols, i.e. associate the symbols to a time-point.

We have enough to define a frame of messages. We define first special function input_{NSL} and output_{NSL}, where for all τ, $\mathsf{input}_{NSL}(\tau)$ and $\mathsf{output}_{NSL}(\tau)$ are, respectively, the input message and the output message at time-point τ. Finally, the frame of messages is a special recursive function symbol frame_{NSL}, where $\mathsf{frame}_{NSL}(\tau)$ represents the frame of messages at time τ. It can be expressed as the sequence of $\mathsf{output}_{NSL}(\tau)$, $\mathsf{input}_{NSL}(\tau)$, and the frame at the preceding time-point $\mathsf{pred}(\tau)$. Formally, these three functions are mutually **recursively defined** as follows:

$$\mathsf{input}_{NSL}(\mathsf{init}) = \mathsf{output}_{NSL}(\mathsf{init}) = \mathsf{frame}_{NSL}(\mathsf{init}) \stackrel{\mathsf{def}}{=} \mathsf{empty}$$

and for all τ in the sequence of calls,

$$\mathsf{input}_{NSL}(\tau) \stackrel{\mathsf{def}}{=} \mathsf{att}(\mathsf{frame}_{NSL}(\mathsf{pred}(\tau))),$$

$$\mathsf{frame}_{NSL}(\tau) \stackrel{\mathsf{def}}{=} \langle \mathsf{output}_{NSL}(\tau), \mathsf{input}_{NSL}(\tau), \mathsf{frame}_{NSL}(\mathsf{pred}(\tau)) \rangle, \text{ and}$$

$$\begin{aligned}
\mathsf{output}_{NSL}(\tau) \stackrel{\mathsf{def}}{=}\ & \text{match } \tau \text{ with} \\
& \mid \mathsf{A1} \quad \to \mathsf{enc}\ (\mathsf{n_a}, (pk\ \mathsf{sk_a}))\ (\mathsf{input}_{NSL}(\mathsf{A1}))\ \mathsf{r_{A1}} \\
& \mid \mathsf{B} \quad \to \mathsf{let}\ \mathsf{n'_a}, pk'_a = \mathsf{dec}\ (\mathsf{input}_{NSL}(\mathsf{B}))\ \mathsf{sk_b}\ \mathsf{in}\ \mathsf{enc}\ (\mathsf{n'_a}, \mathsf{n_b}, (pk\ \mathsf{sk_b}))\ pk'_a\ \mathsf{r_B} \\
& \mid \mathsf{A2} \quad \to \mathsf{let}\ \mathsf{n'_a}, \mathsf{n'_b}, pk'_b = \mathsf{dec}\ (\mathsf{input}_{NSL}(\mathsf{A2}))\mathsf{sk_a}\ \mathsf{in} \\
& \qquad \mathsf{if}\ \mathsf{n'_a} = \mathsf{n_a} \wedge pk'_b = (\mathsf{input}_{NSL}(\mathsf{A1}))\ \mathsf{then}\ \mathsf{enc}\ (\mathsf{n'_b}, \mathsf{n_a})\ pk'_b\ \mathsf{r_{A2}} \\
& \mid \mathsf{C} \quad \to \mathsf{let}\ \mathsf{n'_a}, pk'_a = \mathsf{dec}\ (\mathsf{input}_{NSL}(\mathsf{B}))\ \mathsf{sk_b}\ \mathsf{in} \\
& \qquad \mathsf{let}\ \mathsf{n''_b}, \mathsf{n''_a} = \mathsf{dec}\ (\mathsf{input}_{NSL}(\mathsf{C}))\ \mathsf{sk_b}\ \mathsf{in} \\
& \qquad \mathsf{if}\ pk'_a = pk\ \mathsf{sk_a} \wedge n''_b = n_b \wedge n''_a = n'_a\ \mathsf{then}\ \mathsf{n'_a} \cdot \mathsf{n_b}.
\end{aligned}$$

Similarly, we can define frame_{Ideal}, output_{Ideal} and input_{Ideal}, corresponding to the functions describing an interaction with $\mathcal{G}_0^{Ideal}$. Their definitions differ from the corresponding function for $\mathcal{G}_0^{NSL}$ in the encryption made by Alice and Bob's oracles where dummy values ($\mathsf{dummy}_{A1}, \mathsf{dummy}_B$ and dummy_{A2}) are encrypted. That means that frame_{Ideal} and input_{Ideal} have the same definition as done earlier, and output_{Ideal} follows the same pattern: it is defined for each time-point A1, B, A2 and C. For example, we have:

$$\mathsf{output}_{Ideal}(\mathsf{A1}) \stackrel{\mathsf{def}}{=} \mathsf{if}\ (\mathsf{input}_{ideal}(\mathsf{A1})) = pk\ \mathsf{sk_b}\ \mathsf{then\ enc}\ \mathsf{dummy}_{A1}\ (\mathsf{input}_{ideal}(\mathsf{A1}))\ \mathsf{r_{A1}}$$
$$\mathsf{else\ enc}\ (\mathsf{n_a}, (pk\ \mathsf{sk_a}))\ (\mathsf{input}_{ideal}(\mathsf{A1}))\ \mathsf{r_{A1}}$$

However, one must be careful. In this ideal version of Alice's first message, the adversary gets the encryption of a dummy value. Hence, the decryption in Bob's first message would output the dummy value and not the values Alice has sent in the real version. The output for Bob's first message is defined as follows:

$$\begin{aligned}\mathsf{output}_{Ideal}(\mathsf{B}) \stackrel{\mathsf{def}}{=}\ & \mathsf{let}\ \mathsf{n'_a}, pk'_a = \\ & \quad \mathsf{if}\ (\mathsf{input}_{Ideal}(\mathsf{B})) = \mathsf{enc}\ \mathsf{dummy}_{A1}\ (\mathsf{input}_{ideal}(\mathsf{A1}))\ \mathsf{r_{A1}}\ \mathsf{then}\ (n_a, \mathsf{pub}\ \mathsf{sk_a}) \\ & \quad\ \mathsf{else\ dec}\ (\mathsf{input}_{Ideal}(\mathsf{B}))\ \mathsf{sk_b} \\ & \mathsf{in} \\ & \mathsf{if}\ pk'_a = pk\ \mathsf{sk_a}\ \mathsf{then\ enc}\ \mathsf{dummy}_B\ pk'_a\ \mathsf{r_B} \\ & \quad \mathsf{else\ enc}\ (\mathsf{n'_a}, \mathsf{n_b}, (pk\ \mathsf{sk_b}))\ pk'_a\ \mathsf{r_B}\end{aligned}$$

The other two values $\mathsf{output}_{Ideal}(\mathsf{A2})$ and $\mathsf{output}_{Ideal}(\mathsf{C})$ are defined similarly.

Term semantics

Notice that, the honest computations were separated between honest symbols — for *deterministic* behaviour — and names, for *randomness*. This is done because it is convenient, in the semantics, to make the randomness explicit — this helps to track shared randomness. We note $\rho = (\rho_h, \rho_a)$ a random source. It is a tuple of two random source structures: ρ_h, that gives values to names, and ρ_a that captures the randomness of adversarial computation.

Also, let us call a model $\mathbb{M}$ the structure that gives a trace and the interpretation of honest symbols and attacker symbols. For example, we expect that, for our case, our model interprets the honest symbol enc as the program that encrypts. We will see in the next section that a model is actually a richer structure; this approximation is sufficient in our explanation here.

The semantics of a term t, written $[\![t]\!]^{\rho}_{\mathbb{M}}$, is then, at a first approximation, the (deterministic) program, that outputs the interpretation of t where names and attacker randomness is given by ρ, and the attacker and honest symbols are interpreted as programs yielded by $\mathbb{M}$.

For example, let us take the trace $\mathcal{T} = (\mathsf{Init}, \mathsf{A1}, \mathsf{B})$, in a model that interprets fst, snd, if _ then _ else _, and enc as expected.

The interpretation of the term $\mathsf{frame}_{NSL}(\mathsf{B})$ with $\mathbb{M}$ and a random source ρ is exactly the one of t in Eq. (1.1) with $\mathbb{M}$ and ρ. Thus, the probabilistic joint execution where

Mallory calls the oracle Alice1 and then Bob before termination has the same distributions as the program

$$\rho \xleftarrow{\$}; \mathbf{return}\ [\![t]\!]^{\rho}_{\mathbb{M}}$$

> In practice, the CCSA-HO logic interprets a term as a family of random variables, indexed by η. So for a term t, the family of the random variables $\rho \mapsto [\![t]\!]^{\rho}_{\mathbb{M},\eta}$ for all η. It happens that in some cases, like in our examples, these random variables are also computable. That is that there exists a Turing Machines, or equivalently a program, that computes them. That is why in this section we simplify the semantics to interpret terms as programs.

Equivalence predicate and key secrecy

Finally, on top of terms, it is possible to build first-order formulae to express properties about program computations. In particular, we have a predicate to express indistinguishability. Let t and t' be terms, and $\mathcal{G}_0$ and $\mathcal{G}_1$ the two following games:

$$\mathcal{G}_0 = \{\mathsf{Init} := \rho \xleftarrow{\$}; \mathbf{return}\ [\![t]\!]^{\rho}_{\mathbb{M}}\}$$
$$\mathcal{G}_1 = \{\mathsf{Init} := \rho \xleftarrow{\$}; \mathbf{return}\ [\![t']\!]^{\rho}_{\mathbb{M}}\}.$$

We say that $\mathbb{M} \models t \sim t'$ is valid whenever for all distinguisher polynomial program $\mathcal{D}$,

$$\mid \Pr_{\rho}(\mathcal{D}^{\mathcal{G}_0}(1^{\eta}, \rho_a) = 1) - \Pr_{\rho}(\mathcal{D}^{\mathcal{G}_1}(1^{\eta}, \rho_a) = 1) \mid$$

is negligible in the security parameter η, where $\mathcal{D}^{\mathcal{G}_0}(1^{\eta}, \rho_a)$ (rep. $\mathcal{D}^{\mathcal{G}_0}(1^{\eta}, \rho_a)$)is the output of $\mathcal{D}$ on inputs 1^{η} and ρ_a when given access to the game $\mathcal{G}_0$ (resp. $\mathcal{G}_1$).

Then the first game step to show NSL's key privacy property can be captured by showing that for all model $\mathbb{M}$, in particular *for all traces*,

$$\mathbb{M} \models \forall \tau, \mathsf{frame}_{NSL}(\tau) \sim \mathsf{frame}_{Ideal}(\tau).$$

Cryptographic axioms

The CCSA-HO logic introduces reasoning rules, in the form of inference rules. In particular, if we follow the hand proof of NSL, we are going to need rules to capture cryptographic reasoning, especially rules to use cryptographic game indistinguishability like IND-CPA. Let us try to write an axiom capturing IND-CPA assumption. Intuitively, our axiom says that for all terms m and m', names sk and r the formula

$$\mathsf{enc}\ m\ (pk(\mathsf{sk}))\ \mathsf{r} \sim \mathsf{enc}\ m'\ (pk(\mathsf{sk}))\ \mathsf{r} \tag{1.2}$$

is valid for all models where enc is an IND-CPA encryption.

Still, one must be careful. Indeed, m and m' are terms, intuitively computations sent by the game adversary to the Challenge oracle. As such, they represent adversary computation obtained without calling the Challenge oracle. Hence, beware that the key sk is secret to the attacker, but the public key $pk(sk)$ is sent to the attacker at the beginning. The key cannot appear in the computation of m and m', except under the function $pk(\_)$. Also,

r represents a fresh sampling capturing the randomness used for encryption during the Challenge oracle call, so it must never appear in m or m'.

Capturing these restrictions can be made with syntactic conditions, and yield the following axioms' schema.

Definition 2 (IND-CPA axioms [23])**.** *Our axiom scheme is then: for all (closed) terms m and m', samplings sk, the following inference rule is valid: That is written*

$$\frac{len(m) = len(m')}{\mathsf{enc}\ m\ (pk(\mathsf{sk}))\ \mathsf{r} \sim \mathsf{enc}\ m'\ (pk(\mathsf{sk}))\ \mathsf{r}}\{\mathsf{sk} \not\in_{pk(\_)} m, m'; r \not\in m, m'\}$$

where

- *The condition $\mathsf{sk} \not\in_{pk(\_)} m, m'$ states that sk does not appear in m and m' except under $pk(\_)$. It is put as a condition of the rule as it cannot be expressed in the logic.*
- *The condition $\mathsf{r} \not\in m, m'$ state r that do not appear in m and m'.*
- *The formula $len(m) = len(m')$ states that m and m' must have the same length. That is accounting for the fact that the encryption cannot hide the length of the plaintext.*

Now, how do we show that this axiom schema is sound ? Let's take two arbitrary names sk and r, and two arbitrary terms m and m' respecting the condition of Definition 2, and show, by cryptographic reduction that the indistinguishability $\mathsf{enc}\ m\ (pk(\mathsf{sk}))\ \mathsf{r} \sim \mathsf{enc}\ m'\ (pk(\mathsf{sk}))\ \mathsf{r}$ reduced to the IND-CPA games indistinguishability.

First, these syntactic conditions ensure that there exist two programs $\mathcal{S}_m$ and $\mathcal{S}'_m$ such that for all ρ, $\mathbb{M}$, without calling the Challenge oracle.

$$\mathcal{S}_m(pk(\mathsf{sk})) = [\![m]\!]^{\rho}_{\mathbb{M}} \qquad \text{and} \qquad \mathcal{S}'_m(pk(\mathsf{sk})) = [\![m']\!]^{\rho}_{\mathbb{M}}.$$

The simulators $\mathcal{S}$ that retrieves $pk(sk)$ by the Init oracles, uses $\mathcal{S}_m$ and $\mathcal{S}'_m$ to compute m and m' and call the Challenge oracle on them is an adversary against the IND-CPA games such that

$$\mathcal{S}^{\mathcal{G}_0} = [\![\mathsf{enc}\ m\ pk(\mathsf{sk})\ \mathsf{r}]\!]^{\rho}_{\mathbb{M}} \qquad \text{and} \qquad \mathcal{S}^{\mathcal{G}_1} = [\![\mathsf{enc}\ m'\ pk(\mathsf{sk})\ \mathsf{r}]\!]^{\rho}_{\mathbb{M}}.$$

This simulator $\mathcal{S}$ then justifies the reduction of indistinguishability of games induced by the formula Eq. (1.2) to the IND-CPA indistinguishability, hence proving the soundness of our axioms in Definition 2.

In detail, let us suppose then there exist two terms m and m' respecting the conditions, $\mathbb{M}$, that break the indistinguishability of the axiom schema. That is, there exists a distinguished $\mathcal{D}$ against the associated games that has a non-negligible advantage. In that case, the program $\mathcal{D}(\mathcal{S})$, where we interface $\mathcal{D}$ with $\mathcal{S}$ is an adversary against IND-CPA. Besides, for all random source ρ:

$$\begin{aligned}(\mathcal{D}(\mathcal{S}))^{\mathcal{G}_0} = \mathcal{D}(\mathcal{S}^{\mathcal{G}_0}) = \mathcal{D}([\![\mathsf{enc}\ m\ pk(\mathsf{sk})\ \mathsf{r}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}) \quad \text{and;}\\ (\mathcal{D}(\mathcal{S}))^{\mathcal{G}_1} = \mathcal{D}(\mathcal{S}^{\mathcal{G}_1}) = \mathcal{D}([\![\mathsf{enc}\ m'\ pk(\mathsf{sk})\ \mathsf{r}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}})\end{aligned}$$

So $\mathcal{D}(\mathcal{S})$ has the same advantages against IND-CPA that $\mathcal{D}$ has against our axiom, which ends the reduction proof.

1.4.2 Computer-aided verification

In this part, we present first the proof assistant Squirrel which is based on the CCSA-HO formalism that will be central to the thesis. Then we give a quick overview of other approaches to mechanize cryptographic proof.

CCSA-HO and Squirrel

This approach has already proven successful in handwritten protocol proofs, as demonstrated in [25–28]. At this level of formalization, the system becomes amenable to mechanization. Central to this thesis is the Squirrel proof assistant, which mechanizes the CCSA-HO logic. In Squirrel, users can declare protocols as processes and apply tactics —each roughly corresponding to one or a few inference rules— to construct proofs. Notably, Squirrel includes cryptographic tactics, where each tactic roughly aligns with the application of one cryptographic axiom. This approach has been successfully applied to protocol verification in [24, 29–31].

Other approaches

In the CCSA framework. In the CCSA framework, CryptoVampire [32] is a recent tool designed to prove trace properties of security protocols using the CCSA framework. Unlike Squirrel, CryptoVampire is fully automated: it proceeds by encoding the security of a protocol as a first-order logic task, that is then discharged to first-order theorem provers (e.g. Vampire [33]).

CCSA and related logic are, obviously, not the only approaches to cryptographic protocol proof automation. Different techniques have been used to obtain formal mechanized proofs of cryptographic arguments.

Program logics. Techniques [34–36] relying on imperative program logics, the most prominent one being the probabilistic relational Hoare Logic (pRHL), encode the cryptographic design and security property under study as a stateful and sequential imperative program. Then, the cryptographic arguments proving this program's security can be captured by program logics.

Often, these approaches embed their program logic in an expressive ambient logic, e.g. SSProve [36] is a Rocq framework, and EasyCrypt [34] implements a higher-order ambient logic. While Squirrel's local logic is also a higher-order logic, its (current) global logic is less expressive than, e.g., Easycrypt's ambient logic, because it relies on asymptotic rather than concrete security — though recent work [37] blurs this demarcation. This is deliberate: Squirrel aims to capture higher-level arguments, with a focus on protocols, which are notoriously laborious to analyse in pRHL-based tools. Because of this, past Squirrel developments have required less mathematical libraries than proofs dealing with crypto primitives.

Game transformations. CryptoVerif [38] directly manipulates cryptographic games which are iteratively modified using an ad hoc set of game transformations implemented in the tool.

Property-specific approaches. There has been some number of works which aim at automating cryptographic proofs for a fixed target security property and a restricted class of programs, e.g. to show that padding-based encryption schemes are IND-CCA$_2$ [39], to prove that block-cipher modes are IND-CPA [40] or AEAD [41], or to analyze the EUF-MAC security of structure preserving signatures [42]. Another similar previous work is Owl [43], which uses a type-based approach to prove reachability properties under a fixed set of cryptographic assumptions (IND-CPA, RO, ...). The restrictions on the class of programs, assumptions and target security properties allow these approaches to be highly automated and efficient but restricted.

1.5 This thesis

1.5.1 Starting problem

The CCSA-HO logic creates a distance with the cryptographic games' formalism. This distance helps to support sometimes complex high-level reasoning into inference rules, which lead to the implementation of the proof assistant Squirrel.

The drawback to that idea is that each new cryptographic argument has to be carefully added. The legacy method for adding a cryptographic assumption in the logic and the tool requires to design an axioms' schema in the logic. Then proving that it actually derives for the cryptographic assumption by cryptographic reduction, i.e. proving its soundness. And finally, implementing that axiom schema in the proof assistant.

All the steps from design to implementation can be source of human mistakes. Also, each step requires specific technical knowledge: good understanding of the logic for the design, knowledge on cryptographic reductions for the proof, and developer mastering level of Squirrel for the implementation. Adding one cryptographic assumption is time-consuming, non-user-friendly, and error-prone. This has consequences for the Squirrel tool. For example, because of such reasons, axioms like CCA2 were not added, due to the complexity of its formulation.

From these problems arises the following end goal: find a systematic way to do this work once and for all for any arbitrary cryptographic assumptions.

1.5.2 Contributions

In this thesis we explore one possible answer to that problem: develop a framework to support cryptographic reductions to arbitrary games in the logics. Rapidly, in this thesis we present the following contributions:

- We extend in the logic a predicate, the bideduction predicate, to capture the existence of simulators, and design a proof system for it.

- This new proof system is used to design and prove sound a simulator synthesis procedure, that we implemented in the Squirrel tool. It leads to the addition of a single tactic in the tool, that try to reduce an equivalence to an arbitrary cryptographic assumption expressed by two games' indistinguishability given by the user.

- Finally, to test this work on several cases studies. This shows that our implementation was able to replace legacy cryptographic SQUIRREL axioms but also support new cryptographic assumptions. Most notably, we prove the e-voting protocol FOO [44] using our work to support blinding and commitment hiding cryptographic hardness assumption. It is the largest SQUIRREL proof to date.

These contributions are detailed below.

Bideduction and cryptographic assumption

The idea behind the bideduction was to have a formal way to derive the existence of a simulator against two arbitrary games. If we go back to our example, we want a predicate of the form

$$\vdash (\emptyset, \emptyset) \rhd_{\mathcal{G}_0,\mathcal{G}_1} (\mathsf{frame}_0(t), \mathsf{frame}_1(t))$$

which translates as there exists a simulator $\mathcal{S}$ such that $\mathcal{S}$ computes $\mathsf{frame}_0(t)$ from the empty term, i.e. no inputs, when interacting with game $\mathcal{G}^0$ and computes $\mathsf{frame}_1(t)$ from the empty term when interacting with game $\mathcal{G}^1$. The existence of such a simulator would then witness the validity of the formula

$$\mathsf{frame}_0(t) \sim \mathsf{frame}_1(t).$$

Interestingly, the bideduction already existed in the CCSA-HO logics, but not for simulators. The legacy bideduction predicate translates the existence of a *deterministic* Turing Machine that computed the terms. In particular, these machines can do no oracles, and no random samplings. Hence, these two aspects are at the core of this work.

- We allow for *probabilistic* Turing Machines. This requires careful handling of randomness, that is tracking — using the **symbolic constraints** extra structure— randomness ownership: which names belong to the simulator and which ones to the game.

- These probabilistic Turing Machines are also granted access to *oracles* of an arbitrary cryptographic game. We extend the predicate with Hoare-style **pre- and post-conditions** to track the internal persistent state of a game, that is how the game memory evolves upon each oracle's calls.

We prove that this improved notion is expressive enough to derive sound axioms in the CCSA-HO logic, and provide a proof system to derive bideducibility. The proof system follows the structures of the legacy ones: the legacy inference rules to build atomic deterministic behaviour are still valid. Notably, we extended it to extra behaviour — oracle calls and random samplings and modify the existing rules to account for the fact simulators are not inherently compositionable. Indeed, if $\mathcal{S}_1$ and $\mathcal{S}_2$ are simulators, then $\mathcal{S}_1(\mathcal{S}_2)$ is not necessarily one. This endorses modification for the inductions and transitivity rules.

Simulator synthesis

Having established our predicate and proof system, it seems that we have the necessary formalization to tackle proof mechanization. In the second part of this thesis, we present

the simulator synthesis procedure that we design and implement to automatically establish bideduction. The end goal of this works is its implementation in the proof assistant Squirrel. As a result, our design is guided by the need for practicality in the tool, and reaching the scope of legacy-implemented axioms. In particular, this procedure does not provide completeness: we allow our implementation to fail, relying on user pre-work.

This procedure takes as inputs a partial bideduction predicate, and tries to fill the holes to make it valid. The procedure's core heuristically applies the proof rules based on the structure of the terms and one particularity of this procedure is that it is also guided by symbolic constraints. In the final procedure, this core is called in three successive phases that derive induction invariants and memoized terms, extra terms to deduce that will help with the bideduction, to handle inductions.

Case studies in Squirrel

The procedure was implemented into a single tactic —crypto— in Squirrel, that automatically proves indistinguishability goals by bideduction. This tactic is used in case studies to replace legacy tactics with crypto in existing examples of Squirrel, but also to provide proofs using new cryptographic assumptions, not supported by the tools before, notably, the CCA2 assumption.

We develop a formal proof of vote privacy for the FOO [44] e-voting protocol in Squirrel. Our synthesis procedure is critical to deal with the complexity of the proof. Our security proof is based on the CCSA pen-and-paper proof of [27], which we generalize to an arbitrary number of voters. Our proof is the most complex Squirrel proof to date, both in terms of the diversity of the cryptographic assumptions, and in lines of code. Further, our proof is the first computational mechanized proof of ballot privacy for FOO.

1.5.3 Related work

We compare our work to different approaches in the area of programming languages and formal methods for cryptography.

Mechanized cryptographic reductions

Different techniques have been used to obtain formal mechanized proofs of cryptographic arguments. In this section, we compare our work to the formal approaches presented in Section 1.4.2.

CryptoVampire. The fully automated tool, CryptoVampire [32], relies on the standard CCSA crypto axioms, it suffers from the issues we address in this thesis.

Program logic. Program logic approaches [34–36] are very expressive, but current tools only support the manual application of cryptographic games: to reduce the security of a design Π to a game $\mathcal{G}$, one has to explicitly write a simulator $\mathcal{S}$ such that $\Pi = \mathcal{S}^{\mathcal{G}}$. We do not have this limitation: in our approaches the simulators are implicitly infered by the proof system.

Game transformations. CryptoVerif [38] is the only tool that automatically finds cryptographic reductions without being restricted to a fixed set of built-in assumptions. However, because of its lack of logical foundations, CryptoVerif does not support generic mathematical reasoning. In particular, proof obligations resulting from the application of a cryptographic assumption cannot be discharged to the user as we do, limiting the tool's expressiveness. Moreover, CryptoVerif can only handle assumptions of the form $(\mathcal{G}_0, \mathcal{G}_1)$ where $\mathcal{G}_0$ is a *stateless* game and $\mathcal{G}_1$ features monotonous state (in the form of global write-once tables). Our approach does not suffer from such a restriction from a theoretical point-of-view: arbitrary stateful operations can be handled by using a suitable assertion logic.

Property-specific approaches. All the works we presented ealier which aim at automating cryptographic proofs for a fixed target security property and a restricted class of programs ([39], [40] or [41], [42]. or [43]) are unsuitable as general-purpose frameworks to mechanize cryptographic reductions.

Deduction problem

The deduction problem has been extensively studied in the literature, albeit in different settings. E.g. [45–47] study this problem in Dolev-Yao models, hence they only consider adversaries with very restricted computing capabilities and which do not have access to any oracles. In [48], the authors rely on a deduction predicate with a computational semantics, which they use to prove some security properties. However, this work is mostly interested in non-deducibility rather than deducibility, and they only consider adversaries without access to any oracles.

Component-based synthesis

Component-based synthesis consists in automatically generating code implementing a given target API, starting from a source API. While our problem could be reformulated in this setting (the target API is the protocol under study, the source API is the cryptographic game), existing CBS techniques are (to the best of our knowledge) unsuitable for our setting, either because the code they can synthesize is too simple for our simulators (e.g. [49, 50] only support loop-free programs) or because they are test-driven and do not provide formal guarantees on the produced code (e.g. [51, 52]). More generally, while the problem of program synthesis has been extensively explored by the programming language community, it is usually done with different goals in mind, and under different design constraints. We are not aware of any work allowing to synthesize recursive and probabilistic programs interacting with stateful APIs in an automated fashion, which is what we need here.

Security of e-voting protocols.

There exists a pen-and-paper computational proof for FOO [27] in the CCSA framework. This proof however is restricted to a three-voters model (Alice, Bob and protocol adversary). We authorize our protocol adversary to simulate several voters, which introduce the need

for inductive reasoning, significantly complicating the proof. There exists several tool-assisted proofs of security for FOO [53–55], but all of them are in the symbolic model. To our knowledge, our proof is the first mechanized *computational* cryptographic proof for FOO. There have been a few mechanized computational cryptographic proofs of other e-voting protocols, for Helios [56], Belenios [6], and Selene [57]. All these proofs have been carried-out in EasyCrypt [58].

Chapter 2

The CCSA-HO Logic

Contents

2.1 Types and type structures **28**
2.2 Terms **29**
2.2.1 Variables and typing environments 30
2.2.2 Term syntax and semantics 31
2.3 Recursion **34**
2.3.1 Environments and models 34
2.4 Probabilistic logic **34**

We recall the main features of the probabilistic logic of [24], which we call CCSA-HO in this thesis.

Let us begin by clarifying what we aim to formalize. We want to express the joint computation of probabilistic Turing machines as terms, to then formalize reasoning about protocols. The CCSA-HO logic interprets terms as random variables, which is a more general setting than PPTM. The transition from one setting to another is made as follows. A probabilistic Turing machine is a Turing machine with an additional tape — a random tape — sampled beforehand, which provides the necessary randomness. In other words, given an input and a random source, such machines produce outputs by deterministically computing over both.

The probabilistic logic models PPTM as random variables that maps a sample space, the set of random tapes, to an outcome space, the computation function from input tapes to outputs.

We used types to describe outcome spaces. For example, the type bool is the boolean type, which describes the set $\{0, 1\}$. And the type message $\rightarrow$ bool is the set of functions that go from messages (i.e. bitstrings) to booleans.

Furthermore, in our specific case, all Probabilistic Turing Machines takes as input the bitstring 1^η, with η the security parameter, an arbitrary number in $\mathbb{N}$. So, for each η, the Probabilistic Turing Machine might have a different behaviour, depening on this fixed input 1^η. In summery, terms are interpreted as families of random variables *indexed by* η whose outcome spaces are defined by the term's type.

In this chapter, we define the syntax and semantics of terms in two steps. First, the CCSA-HO logics build terms upon variables that represent, for example, key generation algorithms, and black box functions, like the encryption algorithm. Then, we add recursive definitions to the logics, like, for example, the definition of frames of messages in our previous chapter.

After we declare outcome spaces (types, Section 2.1), and build higher-order terms (Section 2.2 and Section 2.3), our end goal is to reason on (families of) random variables. For example, one key notion we need for protocol verification is the notion of indistinguishability of two random variables. The CCSA-HO logic is a first-order logic build upon terms. We describe in Section 2.4 the predicates of this logic, in particular the indistinguishability predicate, and give their semantics. And in the last section, Section 2.4 we define the logic's sequent.

2.1 Types and type structures

Recall that our goal is to represent random variables using terms. In particular, we want to semantically describe **outcome spaces**. In this logic, this is done by typing. A term of type τ is interpreted as a random variable over the interpretation of τ. The structure defining types' interpretation is called a **type structure**.

Furthermore, in the running example of Chapter 1, in which we used keys, defined as bit strings of length η. This highlights that our type interpretation must be parameterized by η too: a program sampling a key has its output in $\{0,1\}^\eta$. Thus, a type structure is a structure giving the outcome states represented by each type **for any** η.

In this part, we introduce syntax of type and the definition of type structure, along with the semantics of a type relatively to a type structure and a security parameter.

Type syntax and semantics

We assume a set of base types $\mathbb{B}$. A **type**, denoted by τ, is either a base type $\tau_b \in \mathbb{B}$ or $\tau_1 \to \tau_2$ where τ_1 and τ_2 are types. That is:

$$\tau := \tau_b \in \mathbb{B} \mid \tau_1 \to \tau_2$$

For example, we could define the base type key for keys, which represents the bitstrings of size η, and the base type pkey for public keys. Then, the function that builds public keys from secret key can be typed $\mathsf{key} \to \mathsf{pkey}$.

A type τ is interpreted in a **type structure** $\mathbb{M}$, which associates to each type and each value of η a set $(\llbracket\tau\rrbracket^\eta_{\mathbb{M}})$. In other words, each type is associated to a collection of sets $(\llbracket\tau\rrbracket^\eta_{\mathbb{M}})_{\eta\in\mathbb{N}}$, one set for each value of η. Furthermore, this type structure is coherent with the type constructor, meaning that for any types τ_1 and τ_2, $\llbracket\tau_1 \to \tau_2\rrbracket^\eta_{\mathbb{M}} = \llbracket\tau_1\rrbracket^\eta_{\mathbb{M}} \to \llbracket\tau_2\rrbracket^\eta_{\mathbb{M}}$ the set of all functions from $\llbracket\tau_1\rrbracket^\eta_{\mathbb{M}}$ to $\llbracket\tau_2\rrbracket^\eta_{\mathbb{M}}$.

Also, it would be crucial to link the logic semantics with PPTM framework. As such, we also assume that all base types are serialized, that is, to exist a binary representation of the elements in the set they represent.

Finally, we require that

- for convenience, the type unit is a base type and represents the singleton set, e.g. for type structure $\mathbb{M}$ all security parameter η, $[\![\mathsf{unit}]\!]^{\eta}_{\mathbb{M}} = \{\emptyset\}$, and
- the type bool, for booleans, is always a type of $\mathbb{B}$ and that for any type of structure $\mathbb{M}$ and security parameter η, $[\![\mathsf{bool}]\!]^{\eta}_{\mathbb{M}} = \{0,1\}$.

The latter will be useful to embed a logic into terms.

Labels on types

Later, it will be convenient to express properties of types. In particular, we attach to each type a set of labels; which restrict the type's interpretation.

In this thesis, we use the labels:

- $\mathsf{finite}(\tau)$ when τ must be interpreted as a finite set for all η;
- $\mathsf{fixed}(\tau)$, when τ must be interpreted the same way for all security parameters η;
- $\mathsf{enum}(\tau)$, when τ must be a set enumerable by the same PPTM in polynomial time in η, for all η;
- $\mathsf{large}(\tau)$, when τ must be interpreted as a set such that sampling [1] in $[\![\tau]\!]^{\eta}_{\mathbb{M}}$ ensures the probability of collision between two independent samplings to be negligible. In other words, the type is suitable for sampling secret values, like keys. This needs further formalism to be clearly expressed, and we will come back to it later.

Remark. Note that for all type τ, $\mathsf{finite}(\tau)$ and $\mathsf{fixed}(\tau)$ implies $\mathsf{enum}(\tau)$ but also that τ is enumerable in *constant* time, which is stronger.

Example 1. *The type* bool *is labelled* fixed *and* finite*. Indeed, for all* η *and* $\mathbb{M}$*, we enforced that the interpretation of* bool *is* $\{0,1\}$*.*

Example 2. *The type* key *is interpreted as* $\{0,1\}^{\eta}$*. It is then not* fixed *but it is* finite*. Furthermore, if we want the probability of sampling a given key to be negligible, this type has to be labelled by* large*.*

In the thesis, we will often use the types index and timestamp, respectively, to represent indices (to index family of keys, set of agents, etc.) and timestamps (time-points in a protocol execution). They are both labelled finite.

Also, messages in CCSA-HO framework, are represented by the type message, always interpreted as the set of arbitrary bitstring (i.e. $\{0,1\}^*$). Thus, we have $\mathsf{fixed}(\mathsf{message})$ but not $\mathsf{finite}(\mathsf{message})$.

2.2 Terms

This section introduces the terms related definitions. We define the syntax and semantics of terms, which interprets standard lambda-calculus into (families) of random variables. Particular emphasis is placed on the treatment of random sources, represented as finite tapes, which is a key mechanism for modelling probabilistic behaviour.

[1] Samplings are defined in term structures, an extension of type structure for terms, see later.

2.2.1 Variables and typing environments

We assume a set of **variables** $\mathcal{X}$. These variables are the basic bricks on which we build terms. To these variables, we associate **typing environments**. A typing environment $\mathcal{E}$ is a partial map from variables to types.

Each variable will represent a random variable that samples in the type interpretations, i.e. a function whose image sets in the type interpretations. For example, a variable b of type bool will be a random variable that samples in $\{0,1\}$ for all η.

We assume a particular subset $\mathcal{N} \subset \mathcal{X}$ of variables: **names**. They are the symbols that will be of used in representing the basic random variables used in our protocol (e.g. for samplings keys). A name $\mathsf{n} \in \mathcal{N}$ must have a type of the form $\tau_0 \to \tau$ where the *index* type τ_0 must be finite, i.e. we have $\mathsf{finite}(\tau_0)$. For convenience, a name n of type $\mathsf{unit} \to \tau_\mathsf{b}$ with τ_b a base type will be also said to be of type τ_b.

> Names' input types are finite because, as wee will see it later in this chapter, the random source to interpret names is finite in the CCSA-HO logic.
> The originally CCSA logic [23] was based on infinite random tapes, but in practice it was always used finitely. Partly because it was only used for discrete probabilistic reasoning and does not allow for non-terminating sampling semantics. For technical reasons, adding highter-order [24] led to restricting the random sources to be finite. While this finiteness might initially appear restrictive, it imposes few substantive limitations in practice. Indeed, logical proofs are carried out uniformly across all models. That is, when a cryptographic proof is carried on in the logic, the results is roughly: for all models, under certain unrelated restrictions, the protocol is secure. Then, it suffices to consider the result for a model that provides enough randomness, i.e. a model that provides a sufficiently long random tape. This can actually be determined. Indeed, the honest tapes' usage can be bounded by the number of samplings of a pre-declared protocol. The case of the adversary tape is more subtle, however. Since the protocol attacker is modelled as a polynomial-time Turing machine, its runtime is bounded by a known polynomial. This, in turn, allows us to derive an upper bound on the length of the adversarial random tape's usage.

Example 3. *In order to model an asymmetric encryption function, we would typically use types* message, skey, pkey, *and* seed, *for respectively the messages to encrypt, the secret keys, the public keys and the randomness sources for the encryption itself, and two function symbols* $\mathsf{pub} : \mathsf{skey} \to \mathsf{pkey}$ *and* $\mathsf{enc} : \mathsf{message} \to \mathsf{pkey} \to \mathsf{seed} \to \mathsf{message}$.

The keys and seeds are represented by names. For example, let $\mathsf{k} : \mathsf{skey}$ *be a name to represent a secret key, and* $\mathsf{r} : \mathsf{seed}$ *a name to represent a seed.*

2.2.2 Term syntax and semantics

Syntax

Let's tackle the terms' syntax. The idea behind the syntax is to capture how we build random variables upon basic ones. The CCSA-HO terms are λ-term, that is:

$$\begin{aligned} t ::= &\mid x \\ &\mid (t\ t) \\ &\mid \lambda(x : \tau).\, t \qquad \text{(where } x \in \mathcal{X}\text{)} \end{aligned}$$

As usual, terms are taken modulo α-renaming.

In this thesis, we only consider well-typed terms. Terms are typed in a typing environment $\mathcal{E}$, following usual rules which we omit.

Example 4. *Continuing Example 3, the term*

$$\mathit{enc}\ m\ (\mathit{pub}\ \mathsf{k})\ \mathsf{r}$$

represents the encryption of m.

Later on in this chapter, we define how the symbols enc, pub, *etc. can be interpreted. We will expect that the interpretation of the above term actually represents the encryption of m, providing the symbols are interpreted as expected. Notably,* enc *will be interpreted as an encryption.*

Random sources

One specificity of terms semantics is that all terms are interpreted as random variables which sharer their random sources.

We require that the random sources is given by the model. We extend type structures into *randomness structures* $\mathbb{M}$ which for any security parameter η, provides a set $\mathbb{T}_{\mathbb{M},\eta}$ for the random source.

The nature of $\mathbb{T}_{\mathbb{M},\eta}$ is tailored by the need we have for the logic. We have seen in the introduction that we aim at using terms to represent computations of PPTM (or programs), and both honest and adversarial computations respectively share one random source for all probabilistic behaviours. In Chapter 1, we identified two disjoint sources of randomness. The source ρ is a tuple (ρ_h, ρ_a) where ρ_h provide the randomness for honest computation and ρ_a the randomness for adversarial computation. So an element of $\mathbb{T}_{\mathbb{M},\eta}$ is a tuple ρ_h, ρ_a where ρ_h and ρ_a are both *finite* tapes of bits.

Also, our randomness structure must provide for each type — which can be sampled from — how to encode them in bits. That is, let n be an arbitrary name with type $\tau_0 \to \tau_1$ in $\mathcal{E}$. Then, we require that any randomness structure $\mathbb{M}$ provides

- A natural number $\mathsf{R}_{\mathbb{M},\eta}(\tau_1)$, the numbers of bits needed to encode elements in $[\![\tau_1]\!]^{\eta}_{\mathbb{M}}$;
- a machine w_{n} such that, for every $\eta \in \mathbb{N}$ and $a \in [\![\tau_0]\!]^{\eta}_{\mathbb{M}}$, $w_{\mathsf{n}}(\eta, a, \rho_{\mathsf{h}})$ extracts, in time polynomial in η, $\mathsf{R}_{\mathbb{M},\eta}(\tau_1)$ consecutive random bits from the honest tape ρ_{h}; and

- a sampling machine, a machine $[\![\tau_1]\!]^{\$}_{\mathbb{M}}$ such that for every η and bitstring w of length $\mathsf{R}_{\mathbb{M},\eta}(\tau_1)$, $[\![\tau_1]\!]^{\$}_{\mathbb{M}}(1^\eta, w)$ computes a value in $[\![\tau_1]\!]^{\eta}_{\mathbb{M}}$ in polynomial time in η.

Then, the interpretation of the name n by a randomness structure $\mathbb{M}$ with respect to a typing environment $\mathcal{E}$, is the η-indexed family of random variables, such that for any η, the η^{th} element is a random variable that for any tape ρ yields $[\![\mathsf{n}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$ obtained by feeding the random bits extracted by w_{n} to the sampling machine $[\![\tau_1]\!]^{\$}_{\mathbb{M}}$. That is, we have that:

$$[\![\mathsf{n}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} \stackrel{\mathsf{def}}{=} \begin{cases} [\![\tau_0]\!]^{\eta}_{\mathbb{M}} & \to \; [\![\tau_1]\!]^{\eta}_{\mathbb{M}} \\ a & \mapsto \; [\![\tau_1]\!]^{\$}_{\mathbb{M}}(\eta, w_{\mathsf{n}}(\eta, a, \rho_{\mathsf{h}})) \end{cases}$$

Furthermore, we require that $w_{\mathsf{n}}(\eta, a, \rho_{\mathsf{h}})$ and $w_{\mathsf{n}'}(\eta, a', \rho_{\mathsf{h}})$ extract disjoint parts of ρ_{h} when either the names n, n' or the indices a, a' differ. Hence, we ensure that by construction, if $\mathsf{n}_1 : \tau_1 \to \tau$ and $\mathsf{n}_2 : \tau_2 \to \tau$ are distinct names and $a_1 \in [\![\tau_1]\!]^{\eta}_{\mathbb{M}}, a_2 \in [\![\tau_2]\!]^{\eta}_{\mathbb{M}}$, the random variables $\rho \mapsto [\![\mathsf{n}_1]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}(a_1)$ and $\rho \mapsto [\![\mathsf{n}_2]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}(a_2)$ are independent. The same holds for $[\![\mathsf{n}_1]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}(a_1)$ and $[\![\mathsf{n}_1]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}(a_1')$ whenever $a_1' \neq a_1$.

Example 5. *This example explains the sampling procedures for the types* bool *and* message.

- *For the type* bool*, we assume* $\mathsf{R}_{\mathbb{M},\eta}(\mathsf{bool}) = 1$ *for all randomness structure* $\mathbb{M}$ *and security parameter* η*. The sampling machine* $\mathsf{R}_{\mathbb{M},\eta}(\mathsf{bool})$ *is the identity machine. That is, for all* η*, and* $w \in \{0,1\}$*,* $[\![\mathsf{bool}]\!]^{\$}_{\mathbb{M}}(1^\eta, w)$ *is the machines that return its input* w*.*

- *For the type* message*, recall that we assume that for all randomness structure* $\mathbb{M}$ *and security parameter* η*, we have* $[\![\mathsf{message}]\!]^{\eta}_{\mathbb{M}} = \{0,1\}^*$*. We need an upper bound on the number of bits we need to sample in this type. In CCSA-HO logic, we assume* $\mathsf{R}_{\mathbb{M},\eta}(\mathsf{message}) = \eta$*, meaning we can only sample messages of size* η*.*

Term structure

A *term structure* $\mathbb{M}$ is the data that contains everything we need to translate terms into random variables. It extends a randomness structure with a mapping from variables in $\mathcal{X}$ to random variables. More precisely, for a type τ, let $\mathbb{RV}_{\mathbb{M}}(\tau)$ be the set of η-indexed families of random variables from $\mathbb{T}_{\mathbb{M},\eta}$ to $[\![\tau]\!]^{\eta}_{\mathbb{M}}$:

$$\mathbb{RV}_{\mathbb{M}}(\tau) \stackrel{\mathsf{def}}{=} \left\{(X_\eta)_{\eta\in\mathbb{N}} \mid X_\eta : \mathbb{T}_{\mathbb{M},\eta} \to [\![\tau]\!]^{\eta}_{\mathbb{M}} \text{ for every } \eta \text{ in } \mathbb{N}\right\}.$$

A term structure $\mathbb{M}$ *with respect to a typing environment* $\mathcal{E}$ is the extension of a random structure $\mathbb{M}$ that maps any variable $(x : \tau)$ in $\mathcal{E}$ to an element $\mathbb{M}(x)$ in $\mathbb{RV}_{\mathbb{M}}(\tau)$.

This is then lifted naturally to interpret any term t of type τ in $\mathcal{E}$ into $[\![t]\!]_{\mathbb{M}:\mathcal{E}} \in \mathbb{RV}_{\mathbb{M}}(\tau)$ in the following way. For a term t, we write $[\![t]\!]^{\eta}_{\mathbb{M}:\mathcal{E}}$ the η^{th} element of $[\![t]\!]_{\mathbb{M}:\mathcal{E}}$, and $[\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$ the element $[\![t]\!]^{\eta}_{\mathbb{M}:\mathcal{E}}(\rho)$. Then,

$$\begin{aligned} [\![x]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} &= \mathbb{M}(x)(\eta)(\rho) && \text{(when } (x:\tau) \in \mathcal{E}) \\ [\![t'\ t'']\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} &= [\![t']\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}([\![t'']\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}) \\ [\![\lambda(x:\tau).t']\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} &= a \in [\![\tau]\!]^{\eta}_{\mathbb{M}} \mapsto [\![t']\!]^{\eta,\rho}_{\mathbb{M}(x\mapsto \mathbb{1}^{\eta}_{a}):\mathcal{E},(x:\tau)} \end{aligned}$$

where

- $(\mathcal{E}, (x : \tau))$ is the typing environment $\mathcal{E}$ extended with the declaration $(x : \tau)$. Note that x is a fresh variable in $\mathcal{E}$.
- $\mathbb{1}_a^\eta$ is a family $(\mathbb{1}_a^\eta(\eta'))_{\eta' \in \mathbb{N}}$ of random variables such that
  - for all ρ, $\mathbb{1}_a^\eta(\eta)(\rho) = a$.
  - for all ρ and $\eta' \neq \eta$, $\mathbb{1}_a^\eta(\eta')(\rho)$ is an arbitrary value in $[\![\tau]\!]_{\mathbb{M}}^{\eta'}$.
- $\mathbb{M}[x \mapsto \mathbb{1}_a^\eta]$ is the term structure $\mathbb{M}$ extended with the association of x to $\mathbb{1}_a^\eta$.

The last two points are one valid way to express that we want the semantics of $\lambda(x : \tau).t'$ to be a function. Its semantics has to be a family of random variables such that for each η, the η^{th} random variable outputs for any tape ρ, the function that to a value a in $[\![\tau]\!]_{\mathbb{M}}^\eta$ associates the value $[\![t'']\!]_{\mathbb{M}:\mathcal{E}}^{\eta,\rho}$ with t'' equal to t' where "x is replaced by a". Then one problem arises. The value a in a semantic value and has no counterpart in the syntax, which means that a term like $t'[x \mapsto a]$ cannot be written. So we extend the model to associates x to a and modify the typing environment accordingly in $\mathcal{E}'$. But then, we are not finished. Indeed, a term structure may only map x to a family of random variables. So, we extend $\mathbb{M}$ into $\mathbb{M}'$ that associates the variable x to $\mathbb{1}_a^\eta$, that takes value a in relevant cases.

Builtins and local formulas

For convenience, we assume that environments declare some builtins symbols, and we restrict ourselves to models where they are interpreted as expected. Builtins notably include

$$\wedge, \vee, \Rightarrow : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \qquad \neg : \mathsf{bool} \to \mathsf{bool}$$
$$=_\tau : \tau \to \tau \to \mathsf{bool} \quad \forall_\tau, \exists_\tau : (\tau \to \mathsf{bool}) \to \mathsf{bool} \quad \text{(for each } \tau\text{)}$$

and their interpretation notably satisfies:

$$[\![=_\tau]\!]_{\mathbb{M}:\mathcal{E}}^{\eta,\rho}(a, a') = 1 \in [\![\mathsf{bool}]\!]_{\mathbb{M}}^\eta \text{ iff. } a = a' \qquad (a, a' \in [\![\tau]\!]_{\mathbb{M}}^\eta)$$
$$[\![\forall_\tau]\!]_{\mathbb{M}:\mathcal{E}}^{\eta,\rho}(f) = 1 \in [\![\mathsf{bool}]\!]_{\mathbb{M}}^\eta \text{ iff. } f(a) = 1 \text{ for all } a \in [\![\tau]\!]_{\mathbb{M}}^\eta$$

We generally omit type subscripts and use standard infix notations: we may write, for example, $\forall x : \tau.\ \ f(x) \Rightarrow x = g(x)$ when $f : \tau \to \mathsf{bool}$ and $g : \tau \to \tau$. Hence terms of type bool can be seen as formulas, which we call *local formulas*. The semantics of a local formula is a family of boolean random variables in $\mathbb{RV}_{\mathbb{M}}(\mathsf{bool})$.

Furthermore, $\mathsf{well\text{-}founded}_\tau(<)$ is an additional atom of the logic which requires that the interpretation of the binary function symbol $<$ is deterministic (i.e. $[\![<]\!]_{\mathbb{M}:\mathcal{E}}^{\eta,\rho}$ does not depend on ρ) and that $([\![\tau]\!]_{\mathbb{M}}^\eta, [\![<]\!]_{\mathbb{M},\mathcal{E}}^\eta)$ is a well-founded set for every η.

Example 6. *Let's go back to Example 4.*

Then, the following local formula

$$\exists m.\exists m'.\neg\ (m = m') \wedge \mathit{enc}\ m\ (\mathit{pub}\ \mathsf{sk})\ \mathsf{r} = \mathit{enc}\ m'\ (\mathit{pub}\ \mathsf{sk})\ \mathsf{r}$$

states the existence of two different terms of type message *such that their encryption is equal (which is generally impossible).*

2.3 Recursion

We have everything we need to properly wrap up the CCSA-HO local logic. All that remains is to add recursion.

2.3.1 Environments and models

We extend typing environments by adding *definitions*. A definition is an element of the form $(x : \tau = t)$ that to a variable x associates a type τ and a term t of type τ. Such a term t can refer to other variables either declared or defined in the environment, which allow defining mutually recursive functions.

We want to extend our framework to support definitions. A model $\mathbb{M}$ with respect to an environment $\mathcal{E}$, noted $\mathbb{M} : \mathcal{E}$, is a restriction of a term structure $\mathbb{M}$ where for any definition $(x : \tau = t)$, we have that

$$[\![x]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = [\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}.$$

The difficulty lies in dealing with recursive definitions, e.g. when x appears in its definition t.

Detailing how a model can be build with respect to an environment with possibly recursive definitions is out of space to this thesis (see [24] for details). Here, it is sufficient to assume that to do so we need the definitions in the environment to be well-founded, which is guaranteed by respecting certain conditions we won't detail. More interestingly, to rigorously define models, the authors of [24] enforces in environments and models the existence of a symbol $\leq$ in the logic itself, whose semantic is a well-founded order such that for all models $\mathbb{M}$, with respect to $\mathcal{E}$, and for all η and tape ρ, the order $[\![\leq]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = \leq^{\eta}$ is deterministic, i.e. its definition does not dependent on ρ; and is well-founded on

$$\{(x, a) \text{ for } (x : \tau \to \tau' = \lambda y.t) \in \mathcal{E}, a \in [\![\tau]\!]^{\eta}_{\mathbb{M}}\}.$$

Example 7. *Let's define two mutually recursive functions inspired from the frames and outputs of Chapter 1, but where we abstract away any unnecessary details.*

Let $h_o : \mathsf{int} \to \mathsf{message}$, $h_f : \mathsf{int} \to \mathsf{message}$, $t_f : \mathsf{message}$ *and* $t_o : \mathsf{message}$ *be abstract symbols. We define* h_f *and* h_o *by*

$$h_f : \mathsf{int} \to \mathsf{message} = \lambda\, i.\mathsf{if}\ i = 0\ \mathsf{then}\ t_f\ \mathsf{else}\ \langle h_o\ i, h_f(i-1)\rangle$$
$$h_o : \mathsf{int} \to \mathsf{message} = \lambda\, i.\mathsf{if}\ i = 0\ \mathsf{then}\ t_o\ \mathsf{else}\ h_f(i-1)$$

In this example, for all integer n, h_f *applied to* n *relies on the values of* h_f *applied to* $n-1$ *and* h_o *applied to* n. *Expressing* h_o *applied to* n *requires the value of* h_f *applied to* $n-1$. *Then, the order justifying the well-foundedness of this definition is the following:*

$$(h_f, 0) \leq (h_o, 0) \leq (h_f, 1) \leq (h_o, 1) \cdots$$

2.4 Probabilistic logic

In CCSA-HO framework, we aim to capture properties of the random variables. To express such properties, a layer of logic is added above the terms, which we refer to as global

logic or CCSA-HO logic. In this section, we present the predicates that will be used in this thesis, as well as their semantics. At the heart of its semantics lies the notion of an adversary, which we define first.

PPTM, adversaries and cost model

The semantics of some global formula requires us to precisely define what is an adversary in the semantics. Indeed, recall that one of our end goals is to express indistinguishability, which states the impossibility for an adversary to distinguish two situations.

An adversary $\mathcal{A}$ is a *probabilistic polynomial turing machine* (PPTM), i.e. a Turing machine which runs in polynomial time relatively to the security parameter η and the size of its inputs and which has a dedicated read-only tape for randomness.

In the semantics, we restrict adversaries' inputs to elements of order 0 or 1, which corresponds respectively to base type elements or function over base type elements (i.e. of type $\tau_1 \to \tau_2$ where τ_1 and τ_2 are base types). In the latter case, we let the adversaries call a function f on any input x for a cost of 1 plus the size of x to retrieve $f(x)$.

Global formulae

Syntax

To avoid confusion with the local formulae, the formulae of the logic are called *global formulae* and uses different symbols for logical connectives and quantifiers, e.g. global conjunction is noted $\tilde{\wedge}$.

Global formulae are defined by the syntax:

$$F \quad ::= \quad \tilde{\bot} \mid F \mathrel{\tilde{\Rightarrow}} F \mid \tilde{\forall}(x : \tau).\, F \mid \mathsf{adv}(t) \mid [t]_{\mathrm{e}} \mid [t] \mid t_1, \ldots, t_n \sim t'_1, \ldots, t'_n$$

Other connectives and quantifiers ($\tilde{\neg}, \tilde{\vee}, \tilde{\wedge}, \tilde{\exists}$) are defined from $\tilde{\bot}, \tilde{\Rightarrow}, \tilde{\forall}$ as usual. We only consider well-types formulae (we omit the typing rules, which are standard).

Semantics

The semantics of a formula, is defined recursively relatively to a model and environment. Crucially, the key formula for this thesis is the equivalence between vectors of terms, whose semantics relies on the notion of being negligible in η, that we define first.

Definition 3 (Negligible)**.** *A function $f : \mathbb{N} \to \mathbb{N}$ is negligible in $\eta \in \mathbb{N}$ if and only if the quantity $f(\eta)$ is asymptotically smaller than η^k for all $k \geq 0$.*

Let $\mathbb{M} : \mathcal{E}$ be a model with respect to an environment. For any formula F, we say that F is satisfied in $\mathbb{M} : \mathcal{E}$ which we write

$$\mathbb{M} : \mathcal{E} \models F.$$

It is defined as follows:

- For any term t of base type, $\mathsf{adv}(t)$ states that the interpretation of t can be computed by a PPTM which can only access the adversarial random tape: $\mathbb{M} : \mathcal{E} \models \mathsf{adv}(t)$ iff

there exists a PPTM $\mathcal{M}$ s.t. $\mathcal{M}(1^\eta, \rho_\mathsf{a}) = [\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$ for all $\eta \in \mathbb{N}$ and $\rho = (\rho_\mathsf{a}, \rho_\mathsf{h}) \in \mathbb{T}_{\mathbb{M},\eta}$. The definition is extended to terms g of type $\tau^1_\mathsf{b} \to \ldots \to \tau^k_\mathsf{b} \to \tau_\mathsf{b}$ in the following way: $\mathbb{M} : \mathcal{E} \models \mathsf{adv}(g)$ iff there exists a PPTM $\mathcal{M}$ s.t. for any input $\vec{\mathsf{in}} \in \Pi_{i\in[1,k]}[\![\tau^i_\mathsf{b}]\!]^\eta_\mathbb{M}$, $\mathcal{M}(1^\eta, \rho_\mathsf{a}, \vec{\mathsf{in}}) = [\![g]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}(\vec{\mathsf{in}})$ for all $\eta \in \mathbb{N}$ and $\rho = (\rho_\mathsf{a}, \rho_\mathsf{h}) \in \mathbb{T}_{\mathbb{M},\eta}$.

- For any boolean term t, the atoms $[t]_\mathsf{e}$ and $[t]$ state, respectively, that t is exactly true and *overwhelmingly* true:

$$\begin{array}{lll} \mathbb{M} \models [t]_\mathsf{e} & \text{iff} & [\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = 1 \text{ for all } \eta \in \mathbb{N}, \rho \in \mathbb{T}_{\mathbb{M},\eta} \\ \mathbb{M} \models [t] & \text{iff} & \Pr_{\rho\in\mathbb{T}_{\mathbb{M},\eta}}([\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = 0) \text{ is negligible in } \eta \end{array}$$

- For $\vec{t} = t_1, \ldots, t_n$ and $\vec{t'} = t'_1, \ldots, t'_n$ such that t_i and t'_i have the same type for every i, the atom $\vec{t} \sim \vec{t'}$ states that $\vec{t}$ and $\vec{t'}$ are computationally indistinguishable. More precisely, $\mathbb{M} \models \vec{t} \sim \vec{t'}$ holds when, for any PPTM $\mathcal{D}$, the following quantity is negligible in η:

$$\left| \Pr_{\rho\in\mathbb{T}_{\mathbb{M},\eta}}(\mathcal{D}(1^\eta, [\![\vec{t}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \rho_\mathsf{a}) = 1) - \Pr_{\rho\in\mathbb{T}_{\mathbb{M},\eta}}(\mathcal{D}(1^\eta, [\![\vec{t'}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \rho_\mathsf{a}) = 1) \right|$$

- $\tilde{\bot}$ can never be satisfied.
- For two formulae F and F', $\mathbb{M} : \mathcal{E} \models F \mathrel{\tilde{\Rightarrow}} F'$ when if $\mathbb{M} : \mathcal{E} \models F$ then $\mathbb{M} : \mathcal{E} \models F'$.
- For a formula F, $\mathbb{M} : \mathcal{E} \models \tilde{\forall}(x : \tau), F$ when for all random variable $X \in \mathbb{RV}_\mathbb{M}(\tau)$, then $\mathbb{M}[x \mapsto X] : \mathcal{E}(x : \tau) \models F$.

Example 8. $\mathsf{adv}(\mathsf{n}_\mathsf{fresh})$ *never holds since names are sampled from the honest random tape. We require that* $\mathsf{adv}(\mathsf{att})$ *holds in any model.*

Example 9. *For any* t : bool *we have* $\mathbb{M} \models [t]_\mathsf{e} \mathrel{\tilde{\Rightarrow}} [t]$ *for any* $\mathbb{M}$*, i.e. that formula is valid. The converse implication is not valid. Moreover,* $[t]$ *is logically equivalent to* $t \sim \mathsf{true}$*, i.e.* $\mathbb{M} : \mathcal{E} \models [t]$ *for all model* $\mathbb{M} : \mathcal{E}$ *if and only if* $\mathbb{M} : \mathcal{E} \models t \sim \mathsf{true}$ *for all model* $\mathbb{M} : \mathcal{E}$*.*

Example 10. *The global formulas* $[\varphi] \mathrel{\tilde{\wedge}} [\psi]$ *and* $[\varphi \wedge \psi]$ *are logically equivalent, but this does not hold with disjunctions.*

Indeed, $\mathbb{M} : \mathcal{E} \models [\varphi] \mathrel{\tilde{\vee}} [\psi]$ *states that at least* $\mathbb{M} : \mathcal{E} \models [\varphi]$ *or* $\mathbb{M} : \mathcal{E} \models [\psi]$*. On the contrary,* $\mathbb{M} : \mathcal{E} \models [\varphi \vee \psi]$*, states that the formula* $\varphi \vee \psi$ *is overwhelming true, but it could because for some* ρ*,* φ *holds and* ψ *holds for other tapes. So it might not imply that* $[\varphi] \mathrel{\tilde{\vee}} [\psi]$ *is satisfied.*

Example 11. *Finally, the following axiom scheme is valid, for any terms* $\vec{u}, \vec{v}$*:* $\tilde{\forall} x \tilde{\forall} y.\ [x = y] \mathrel{\tilde{\Rightarrow}} (\vec{u} \sim \vec{v}) \mathrel{\tilde{\Rightarrow}} (\vec{u}[x \mapsto y] \sim \vec{v}[x \mapsto y])$.

Notice that the notion of models does not force the sampling of keys to be uniform, i.e. we can reason over arbitrary key generation algorithms. However, it is reasonable to require that freshly sampled keys cannot be guessed by the attacker. Note that it is the case in terms structure such that the (global) formulae $[\mathsf{k} \neq t]$ for any name k : key, and term t that does not contain the name k, i.e. in which k does not recursively appear and does not contain free undefined variables These conditions ensure that the random variables $[\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$ and $[\![\mathsf{k}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$ are independent; the probability that they coincide is thus negligible in models where keys are sampled uniformly enough in a large enough sample space. This is exactly how we define the label large.

Global and local sequent

Finally, we define the sequent of this logic and give its semantics. The full sequent calculus of the logic will not be defined in this thesis, as it is out of scope.

Sequents are divided into two categories: local and global, which reflects the two kind of formulae: local and global.

Definition 4 (Global sequent)**.** *A global sequent* $\mathcal{E}; \Theta \vdash F$ *is formed from an environment* $\mathcal{E}$*, a set of global formulae* Θ *and a global formula* F*. It is valid if and only if* $\tilde{\wedge}\Theta \tilde{\Rightarrow} F$ *is satisfied by all models of* $\mathcal{E}$*.*

Definition 5 (Local sequent)**.** *A local sequent* $\mathcal{E}; \Theta; \Gamma \vdash f$ *is formed from an environment* $\mathcal{E}$*, a set of global formulae* Θ *and local formulae* Γ *and a local formula* f*. It is valid if and only if* $\tilde{\wedge}\Theta \tilde{\Rightarrow} [\wedge\Gamma \Rightarrow f]$ *is satisfied by all models of* $\mathcal{E}$*.*

Chapter 3

Formal Model for Cryptographic Reduction

Contents

3.1 Overview and motivating example **39**
3.2 Syntax **44**
3.2.1 Expressions and programs 44
3.2.2 Games 45
3.2.3 Simulators and adversaries 46
3.3 Semantics **47**
3.3.1 Memories 47
3.3.2 Program random tapes 47
3.3.3 Expression and program semantics 48
3.3.4 Cost model 50
3.3.5 Adversaries 51
3.3.6 Adversaries and security 51

The next three chapters introduce the formalism of bideduction designed during this thesis, and published in [59]. This chapter introduces our formalism for cryptographic reductions. Chapter 4 will build upon it to define the bideduction judgement, its semantics, and the rule capturing cryptographic reduction of an indistinguishability of terms to a game. Finally, Chapter 5 introduces a proof system to derive bideduction judgements and proves the soundness of that system. This chapter contains the definitions of syntax, Section 3.2, and semantics, Section 3.3, for games and adversaries. They rely on our specific notion of programs, tailored for the need to control randomness usages. To begin, we provide an overview, Section 3.1, of the main concepts introduced in this chapter, using a simple protocol as an example.

3.1 Overview and motivating example

Since the initial example in our introduction chapter is too complex for this early stage of formalism development, we will use a simpler example: the Hash Lock Protocol [60]. In this

section, we introduce this protocol, and illustrate our formalism on it. This serves as an opportunity to present the intuition behind the formalism and its key aspects before fully defining it in the next sections. This section starts with modelling the Hash Lock protocol and Key Secrecy property in the CCSA-HO formalism, then defines a cryptographic game, the PRF game, in our syntax, and provides a proof of reduction of Hash Lock Key Secrecy property to the game by exhibiting a simulator.

The Hash Lock protocol

The Hash Lock protocol relies on a keyed hash function $\mathsf{h}(\_,\_)$, and involves participants $T_1, T_2, \ldots$ where each T_i owns a secret hashing key k_i to be used across an unbounded number of sessions. For its j^{th} session, participant T_i inputs x and outputs $\langle \mathsf{n}_{i,j}, \mathsf{h}(\langle \mathsf{n}_{i,j}, x\rangle, \mathsf{k}_i)\rangle$, where $\langle \mathsf{n}_{i,j}, x\rangle$ is a pair combining a session-specific nonce, i.e. a fresh random sampling, and input x. It is schematized in Figure 3.1.

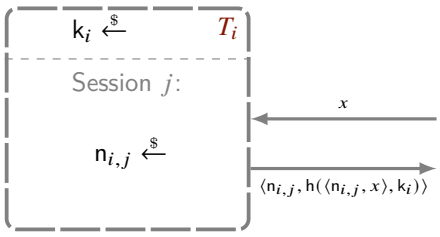


Figure 3.1: The Hash-Lock protocol

Modelling in the CCSA-HO formalism

As seen in Chapter 1, we model an execution of the protocol along an arbitrary execution trace, given as a finite sequence of *timestamps*. Each timestamp in the trace corresponds to an elementary interaction between the adversary and some participant, where some T_i inputs a message from the adversary and outputs its answer. The timestamp where participant T_i plays its session j will be represented by $\mathsf{T}(i, j)$. All timestamps must be of this form, except for the initial timestamp, noted init, and the special value undef used to represent timestamps not present in the trace. We let pred be the predecessor function on timestamps that are present in the trace. We finally model the execution using three mutually recursive functions:

- $\mathsf{input}(t)$ represents the input provided to the protocol at time t;
- $\mathsf{output}(t)$ the protocol output at that time; and
- $\mathsf{frame}(t)$ the sequence of all outputs up to time t, included.

As usual, we consider an active adversary that fully controls the network: it can read, intercept and even modify all messages exchanged by honest participants. The inputs of the protocol are then always the result of an adversarial computation, which we represent with the function $\mathsf{att}(\_)$. Our functions can then be defined as follows for any $t \neq \mathsf{undef}$:

$$\mathsf{frame}(\mathsf{init}) = \mathsf{input}(\mathsf{init}) = \mathsf{output}(\mathsf{init}) = \mathsf{empty}$$

$$\begin{aligned} \mathsf{frame}(\mathsf{t}) &= \langle \mathsf{frame}(\mathsf{pred}(\mathsf{t})), \mathsf{output}(\mathsf{t}) \rangle \\ \mathsf{input}(\mathsf{t}) &= \mathsf{att}(\mathsf{frame}(\mathsf{pred}(\mathsf{t}))) \qquad (\text{when } \mathsf{init} < \mathsf{t}) \\ \mathsf{output}(\mathsf{T}(\mathsf{i},\mathsf{j})) &= \langle \mathsf{n}_{i,j}, \mathsf{h}(\langle \mathsf{n}_{i,j}, \mathsf{input}(\mathsf{T}(\mathsf{i},\mathsf{j})) \rangle, \mathsf{k}_i) \rangle \end{aligned}$$

where undef is a special value of type timestamp representing time-points that have not been scheduled in the execution trace.

PRF assumption and Key Secrecy

The PRF cryptographic assumption [61] on a keyed hash function h roughly states that hashes $\mathsf{h}(\_,\mathsf{k})$ using a secret key k are indistinguishable from random values. More precisely, it can be expressed as the indistinguishability of two games $\mathcal{G}_0$ and $\mathcal{G}_1$, where the key k is initially sampled, and the adversary is provided with two oracles:

- the *hashing* oracle allows computing hashes of chosen messages in both $\mathcal{G}_0$ and $\mathcal{G}_1$;
- the *challenge* oracle returns the hash of its input in $\mathcal{G}_0$, but a fresh sampling in $\mathcal{G}_1$.

To avoid irrelevant distinguishing attacks, both oracles reject inputs that have already been used. The games are presented in Figure 3.2.

Now, we are interested in proving that Hash Lock ensures a form of key secrecy: *when outputting the hash keyed by key k the protocol doesn't reveal any information about k to the adversary.* More formally, we would like to show that, at any point of an interaction of the adversary with the protocol, the adversary cannot distinguish a new hash $\mathsf{h}(\langle \mathsf{n}_{i_0,j_0}, \mathsf{input}(\mathsf{T}(i_0,j_0)) \rangle, \mathsf{k}_{i_0})$ from a randomly sampled value. We seek to verify, for an arbitrary $t_0 = \mathsf{T}(i_0,j_0) \neq \mathsf{undef}$ and for a fresh name $\mathsf{n}_{\mathsf{fresh}}$:

$$\mathsf{frame}(\mathsf{pred}(t_0)), \mathsf{output}(t_0) \sim \mathsf{frame}(\mathsf{pred}(t_0)), \langle \mathsf{n}_{i_0,j_0}, \mathsf{n}_{\mathsf{fresh}} \rangle \tag{3.1}$$

To prove Eq. (3.1), we need a pseudo-randomness assumption on h: we rely on the PRF assumption. The function h is said to be a PRF when the *advantage* of any PPTM in distinguishing $\mathcal{G}_0$ and $\mathcal{G}_1$ is negligible, i.e. for any PPTM $\mathcal{A}$, the probability

$$\left| Pr(\mathcal{A}^{\mathcal{G}_0} = 1) - Pr(\mathcal{A}^{\mathcal{G}_1} = 1) \right| \text{ is negligible in } \eta.$$

Cryptographic reductions

We are going to prove our security property using a *cryptographic reduction* to the PRF game. More precisely, assuming a PTIME adversary $\mathcal{A}$ against the target indistinguishability of Eq. (3.1) we build a PTIME adversary $\mathcal{B}$ against the PRF game $(\mathcal{G}_0, \mathcal{G}_1)$ of Figure 3.2 such that $\mathcal{B}$ is the composition $\mathcal{A} \circ \mathcal{S}$ of the adversary $\mathcal{A}$ with a *simulator* $\mathcal{S}$ computing the terms appearing on the left or right side of the security formula in Eq. (3.1) — depending on whether $\mathcal{S}$ has access to the oracles $\mathcal{G}_0$ or $\mathcal{G}_1$. Roughly, $\mathcal{S}$ satisfies:

$$\begin{aligned} \mathcal{S}^{\mathcal{G}_0} &= \left(\mathsf{frame}(\mathsf{pred}(t_0)), \mathsf{output}(t_0)\right) \\ \mathcal{S}^{\mathcal{G}_1} &= \left(\mathsf{frame}(\mathsf{pred}(t_0)), \langle \mathsf{n}_{i_0,j_0}, \mathsf{n}_{\mathsf{fresh}} \rangle\right) \end{aligned}$$

Thus, $\mathcal{A}$'s advantage against Eq. (3.1) is exactly $\mathcal{B}$'s advantage against the PRF game $(\mathcal{G}_0, \mathcal{G}_1)$, which we assumed negligible.

Initialization ($\mathcal{G}_0$ and $\mathcal{G}_1$):

$$\mathsf{k} \xleftarrow{\$}; \ \ell_{\mathbf{hash}} \leftarrow []; \ \ell_{\mathbf{challenge}} \leftarrow [];$$

Hash and challenge oracles for game $\mathcal{G}_\mathsf{b}$ ($\mathsf{b} \in \{0;1\}$):

$$\begin{aligned}
&\textbf{oracle } \mathbf{hash}(x) := \{\ \ell_{\mathbf{hash}} \leftarrow x :: \ell_{\mathbf{hash}}; \\
&\qquad\qquad \textbf{return } (\mathsf{if}\ x \notin \ell_{\mathbf{challenge}}\ \mathsf{then}\ \mathsf{h}(x,\mathsf{k})\ \mathsf{else}\ \mathsf{zero})\ \} \\
&\textbf{oracle } \mathbf{challenge}(x) := \{\ r \xleftarrow{\$}; \\
&\qquad\qquad v \leftarrow \mathsf{if}\ x \notin \ell_{\mathbf{hash}} \cup \ell_{\mathbf{challenge}}\ \mathsf{then} \\
&\qquad\qquad\qquad \mathsf{if}\ \mathsf{b}\ \mathsf{then}\ \mathsf{h}(x,\mathsf{k})\ \mathsf{else}\ r \\
&\qquad\qquad\quad \mathsf{else}\ \mathsf{zero}\ ; \\
&\qquad\qquad \ell_{\mathbf{challenge}} \leftarrow x :: \ell_{\mathbf{challenge}}; \\
&\qquad\qquad \textbf{return } v\ \}
\end{aligned}$$

Remark: Queries to both oracles are logged in the lists $\ell_{\mathbf{hash}}$ and $\ell_{\mathbf{challenge}}$ to avoid repeated queries that would make the assumption trivially unfeasible.
Remark: This formulation of PRF assumption is an equivalent variant of the standard one for PRF [61], which will facilitate its translation into a CCSA axiom.

Figure 3.2: Games for the PRF cryptographic assumption.

The simulator $\mathcal{S}$ is described in a slightly beautified and simplified imperative language in Figure 3.3. On lines 2–16, $\mathcal{S}$ computes the term $\mathsf{frame}(\mathsf{pred}(t_0))$.

Since frame is defined mutually recursively with input and output, $\mathcal{S}$ computes simultaneously all three functions for all timestamps in $\{\mathsf{init}; \ldots; \mathsf{pred}(t_0)\}$. Concretely, $\mathcal{S}$ uses three identically named arrays input, output, and frame indexed by timestamps, which are being filled by the **for** loop starting on line 2, following the recursive definition of input, frame and output.

To fit with the probabilistic handling of names in the logic, the simulator's and game's randomness is *early-sampled*: the simulator and game both have access to tagged random tapes from which they extract their random values; these tapes are implicitly sampled before the execution starts. On line 7, the simulator performs a random sampling $\mathsf{n} \xleftarrow{\$} \mathsf{T_S}[\mathsf{offset_n}(\mathsf{i},\mathsf{j})]$: this actually reads an early-sampled random tape at a position determined by the tag $\mathsf{T_S}$ (indicating a simulator sampling) and an offset associated to the name $\mathsf{n}_{i,j}$ being simulated.

Our notion of oracle call is also adapted to fit with the logic. On line 9 the simulator computes $\mathsf{h}(\langle \mathsf{n}, \mathsf{input}[\,\mathsf{t}\,]\rangle, \mathsf{k}_{i_0})$ using the oracle call

$$\mathcal{G}.\mathbf{hash}(\langle \mathsf{n}, \mathsf{input}[\,\mathsf{t}\,]\rangle)[\mathsf{offset_k}(\mathsf{i}_0)].$$

In addition to passing the message to be hashed as an argument, the simulator specifies here where the oracle should read the key: although this value is usually understood as being sampled when the game initializes, this is irrelevant in our early-sampled semantics; of course, our model will forbid that the simulator calls oracles with inconsistent values for the key's offset, and the simulator cannot read the random tape at this position. This unusual setup will be useful, again, to help track the relationship between the samplings and the names that are being simulated.

```
1  (* Recursively compute input, output and frame using the hash oracle. *)
2  for each t ∈ [ init; t0[ {
3    match t with
4    | init → input[t] ← empty; output[t] ← empty; frame[t] ← empty
5    | T(i,j) →
6      input[t] ← att(frame[pred t])
7      n ←$ TS[offsetn(i, j)]   (* pre−sampled value access *)
8      if i = i0 then {
9        x ← G.hash(⟨ n, input[t] ⟩)[offsetk(i0)]   (* oracle call *)
10     } else {
11       k ←$ TS[offsetk(i)]   (* pre−sampled value access *)
12       x ← h(⟨ n, input[t] ⟩, k)
13     }
14     output[t] ← ⟨ n, x ⟩
15     frame[t] ← ⟨ frame[pred t], output[t] ⟩
16 }
17
18 (* Use the challenge oracle to compute output(t0) or ⟨n_{i0,j0}, n_fresh⟩. *)
19 input[t0] ← att(frame[pred t0])
20 n ←$ TS[offsetn(i0, j0)] (* pre−sampled value access *)
21 output' ← ⟨ n, G.challenge(⟨ n, input[t0] ⟩)[offsetk(i0); offset_{n_fresh}()] ⟩
22 return (frame[pred t0], output')
```

Remark: The test that the input t is not undef is implicitly made by the test : $t \in [\mathsf{init}; t_0[$. The segment contains only defined timestamps.

Figure 3.3: Reduction to the PRF assumption.

On line 18, $\mathsf{frame}[\mathsf{pred}(t_0)]$ has been properly computed and can be used to compute in output' a value which will be $\mathsf{output}(t_0)$ on the left and $\langle \mathsf{n}_{i_0,j_0}, \mathsf{n}_{\mathsf{fresh}} \rangle$ on the right. This is done using the challenge oracle $\mathcal{G}.\mathbf{challenge}$, passing the offsets for the key and the fresh sampling. Crucially, the call to the challenge oracle returns the expected value because the input $\langle \mathsf{n}_{i_0,j_0}, \mathsf{input}(t_0) \rangle$ has never been queried to the hash oracle, except with negligible probability. Indeed, the hash oracle $\mathcal{G}.\mathbf{hash}$ has only been queried on the values in:

$$\mathcal{H} \stackrel{\text{def}}{=} \left\{ \langle \mathsf{n}_{\mathsf{i},\mathsf{j}}, \mathsf{input}(t) \rangle \mid t = \mathsf{T}(\mathsf{i},\mathsf{j}) < t_0 \right\}$$

and the probability that a collision occurs between $\langle \mathsf{n}_{i_0,j_0}, \mathsf{input}(t_0) \rangle$ and a value in $\mathcal{H}$ is bounded by:

$$|\mathcal{H}| \times \Pr(\mathsf{n}_{i_0,j_0} = \mathsf{n}_{i,j}) \qquad (\text{for } (\mathsf{i},\mathsf{j}) \neq (\mathsf{i}_0,\mathsf{j}_0))$$

This is negligible since $\mathcal{H}$ is a set of constant size w.r.t. η, and since the names n_{i_0,j_0} and $\mathsf{n}_{i,j}$ are independent uniform random samplings in an exponentially large set.

3.2 Syntax

We will now present our formal definitions of cryptographic games and adversaries. The two concepts are based on the concepts of expression and programs. We will define these first. As we have seen, we use a slight modification of the standard cryptographic concepts to facilitate the relationship between logical and computational sampling.

3.2.1 Expressions and programs

Both games and their adversaries rely on the notion of programs. Programs will be used to define oracles in games and adversary code for adversaries.

Expressions

We assume a set of program variables $\mathcal{X}_\mathsf{p}$, and an intrinsic typing associating to each variable $v \in \mathcal{X}_\mathsf{p}$ a base type — we do not need a higher-order programming language. The *library* of our language, denoted by $\mathcal{L}_\mathsf{p}$, is a set of typed function symbols disjoint from $\mathcal{X}_\mathsf{p}$ representing built-in functions shared with the logic — i.e. we will have $\mathcal{L}_\mathsf{p} \subseteq \mathcal{E}$. We assume that $\mathcal{L}_\mathsf{p}$ contains at least the standard arithmetic and boolean operations (e.g. $0, \cdot + \cdot, \mathsf{if}\ \cdot\ \mathsf{then}\ \cdot\ \mathsf{else}\ \cdot, \mathsf{true}, \mathsf{false}$).

Also, we assume that these built-in functions are of order 0 or 1 and have PPTM representation. That means that for any model with respect to $\mathcal{L}_\mathsf{p}$,

- for any symbol s of type τ_b in $\mathcal{L}_\mathsf{p}$, there exists a PPTM $\mathcal{M}$ such that $\mathcal{M}(1^\eta, \rho_\mathsf{a}) = [\![s]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$; and
- for any symbol s of type $\tau_1 \to \cdots \tau_n \to \tau$, there exists a PPTM $\mathcal{M}$ such that for all inputs $m_1, \cdots, m_n$ in the serialized set of $[\![\tau_1]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} \times \cdots \times [\![\tau_n]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$,

$$\mathcal{M}(1^\eta, m_1, \cdots, m_n, \rho_\mathsf{a}) = [\![g]\!]^{\eta,\rho}_{\mathbb{M}[x_1 \mapsto \mathbb{1}^\eta_{m_1} \cdots x_n \mapsto \mathbb{1}^\eta_{m_n}]:\mathcal{E}(x_1:\tau_1)\cdots(x_n:\tau_n)};$$

for all η, for all model $\mathbb{M}$ and all logical tapes $\rho = (\rho_\mathsf{a}, \rho_\mathsf{h})$

Definition 6 (Expression syntax)**.** *We form well-typed expressions from $\mathcal{X}_\mathsf{p}$, $\mathcal{L}_\mathsf{p}$, and a special constant* b *of type* bool*. This constant will denote the side we are in when describing a left or right cryptographic game. Formally:*

$$e_1, \ldots, e_n \in \mathsf{Expr} ::= e_1\ e_2 \mid v \mid f \mid \mathsf{b} \qquad (v \in \mathcal{X}_\mathsf{p}, f \in \mathcal{L}_\mathsf{p})$$

Randomness tagging

Following the logical semantics, it will also be useful in programs to keep track of randomness origin. To this end, we use *tagged random samplings* from eagerly sampled tapes. This translates in the program's syntax by the usage of *tags*. The programs' syntax is then defined relatively to an arbitrary set of tags `Tag`.

In practice, we use a specific set of tags, designed for game adversaries, defined in the following section.

$$
\begin{array}{lll}
\mathsf{p}_1, \ldots, \mathsf{p}_n ::= & v \leftarrow e & \mid \textbf{skip} \\
& \mid v \xleftarrow{\$} \mathtt{T}[e] & \mid \mathsf{p}_1; \mathsf{p}_2 \\
& \mid v \leftarrow O_f(\vec{e})[\vec{e}_g; \vec{e}_l] & \mid \textbf{if } e \textbf{ then } \mathsf{p}_1 \textbf{ else } \mathsf{p}_2 \\
& \mid \textbf{abort} & \mid \textbf{while } e \textbf{ do } \mathsf{p}
\end{array}
$$

Figure 3.4: Syntax of programs.

$$
\begin{array}{lll}
\mathsf{decl\_var} & ::= v \leftarrow e & (v \in \mathcal{X}_\mathsf{p}, e \in \mathsf{Expr}) \\
\mathsf{decl\_sample} & ::= v \xleftarrow{\$} & (v \in \mathcal{X}_\mathsf{p}) \\
\mathsf{decl\_oracle} & ::= \textbf{oracle } f(\vec{v}) := \{\mathsf{decl\_sample}^*; \mathsf{p}; \textbf{return } e\} & \\
 & & (f \in \mathcal{O},\ \vec{v} \in \mathcal{X}_\mathsf{p}^*,\ e \in \mathsf{Expr},\ \mathsf{p} \text{ a program}) \\
\mathsf{decls} & ::= \mathsf{decl\_sample}^*; \mathsf{decl\_var}^*; \mathsf{decl\_oracle}^* &
\end{array}
$$

Figure 3.5: Syntax of games defined over oracle names $\mathcal{O}$.

Programs

In this thesis, we use a standard While language for programs. However, to model game adversaries, we need specific operations that do not usually appear in a While language: random samplings and oracle calls. As illustrated with the simulator of Figure 3.3, we follow a style that fits well with our logic.

The full syntax is given in Figure 3.4.

The instruction $v \xleftarrow{\$} \mathtt{T}[e]$, where $v : \tau \in \mathcal{X}_\mathsf{p}$, $\mathtt{T} \in \mathsf{Tag}$, and e is of type int, samples a value of type τ using the randomness from random source $\mathtt{T}$ read at offset e, and stores it into v. The instruction $v \leftarrow O_f(\vec{e})[\vec{e}_g; \vec{e}_l]$ is an oracle call, where the variable v receives the call's result, f is the oracle being called, $\vec{e}$ are the oracle inputs, and $\vec{e}_g, \vec{e}_l$ have type int. These integers let the program control the offsets at which the oracle reads its randomness for, respectively, global samplings and local samplings.

For both expressions and programs, we assume a standard type system (whose rules we omit) and only consider well-typed expressions and programs w.r.t. $\mathcal{X}_\mathsf{p}$ and $\mathcal{L}_\mathsf{p}$.

3.2.2 Games

Cryptographic games set up some data (e.g. randomly sample keys) and provide functionalities through *oracles* to compute over this data, possibly changing it. Computations performed by oracles are described by *simple programs*, which is a program without the adversary's specific features — random sampling and oracle calls.

As explained before, we are interested in pairs of games $(\mathcal{G}_0, \mathcal{G}_1)$ that are assumed to be indistinguishable; such pairs will be described by a single game $\mathcal{G}$ using the special variable b, i.e. $\mathcal{G}_i$ is obtained from $\mathcal{G}$ when $\mathsf{b} = i$.

Definition 7. *A* game $\mathcal{G} = (\mathcal{O}, \mathsf{decls})$ *is a finite set of oracle names* $\mathcal{O}$*, and a sequence of declarations* decls *according to the syntax given in Figure 3.5. Declarations contain, in order, sequences of:*

1) initialization of variables, either:

- $v_g \xleftarrow{\$}$, *which initializes the* global *random variable* v_g*;*
- *or* $v \leftarrow e$*, which assigns the evaluation of expression* e *to* v*.*

We let $\mathcal{G}.\mathsf{gs}$ *be the set of all initialized random variables.*

2) **oracle** $f(\vec{v}) := \{v_1^l \xleftarrow{\$}; \ldots; v_k^l \xleftarrow{\$}; \mathsf{p}; \text{return } e\}$*, which defines an oracle* $f \in \mathcal{O}$ *with inputs* $\vec{v}$ *that initializes a sequence of* local *random variables* $v_1^l, \ldots, v_k^l$*, then executes a simple program* p *and finally returns* e*. We assume that* p *never modifies the values of the* random *global variables (e.g.* v_g *above), and the* random *local variables of any oracle (e.g.* $v_1^l, \ldots, v_k^l$ *above).* [1]

We require that a game provides a single definition for each oracle name $f \in \mathcal{O}$*. Given one such definition, we let:*

$$f.\text{args} \stackrel{\text{def}}{=} \vec{v} \quad f.\text{loc}_\$ \stackrel{\text{def}}{=} (v_1^l, \ldots, v_k^l) \quad f.\text{prog} \stackrel{\text{def}}{=} \mathsf{p} \quad f.\text{expr} \stackrel{\text{def}}{=} e$$

We also let $f.\text{glob}_\$$ *be the vector of all global random variables that are used in the oracle* f*.*

The PRF game shown in Figure 3.2 is an instance of this notion of game.

> The restriction to simple programs in the oracle body — those that do not perform random sampling or oracle calls within — is not very limiting in practice. To our knowledge, most standard cryptographic games conform to this structure.

3.2.3 Simulators and adversaries

A game simulator $\mathcal{S}$ is syntactically a program, built for a specific set of tags: $\mathtt{Tag} = \{\mathtt{T_A}, \mathtt{T_S}, \mathtt{T_G}\}$. All tags represent a source of randomness that can be used during the execution of $\mathcal{S}$:

- $\mathtt{T_A}$ is for random samplings performed by $\mathcal{S}$ that will correspond to samplings in ρ_a;
- $\mathtt{T_S}$ is for random samplings by $\mathcal{S}$ that will correspond to samplings in ρ_h;
- $\mathtt{T_G}$ is for the game (oracle) randomness that $\mathcal{S}$ cannot directly access, and which will correspond to samplings in ρ_h.

An adversary is a simulator which also has restrictions. Indeed, it also needs to handle randomness "correctly", and to be polynomial-time. Notice that in our syntax, the program itself — here, the adversary — controls the randomness offsets, but should not be able to read nor write these random bits itself. In particular, we distinguish the *global* offsets $\vec{e}_g$ used for the global random variables of the game from the *local* offsets $\vec{e}_l$ used for the local random variables of the oracles. As each oracle call must use fresh randomness for local samplings, our semantics will forbid the adversary from re-using local integer offsets. Similarly, global offsets will have to be consistent from one call to the next, as the game's global variables must be sampled only once.

Hence, the notion of adversaries is a *semantical* notion that we define later in section Section 3.3.5.

[1] The emphases are on "random" in this part: the program can modify global variables that are not random, such as logging list, for example.

$$\mu^{i\ \eta,\mathfrak{p}}_{\mathsf{init}_{\mathbb{M}}}(\cdot) \stackrel{\mathsf{def}}{=} \{\mathsf{eta} \mapsto \eta\}$$
$$\mu^{i\ \eta,\mathfrak{p}}_{\mathsf{init}_{\mathbb{M}}}(\mathcal{G}) \stackrel{\mathsf{def}}{=} \mu^{i\ \eta,\mathfrak{p}}_{\mathsf{init}_{\mathbb{M}}}(\mathsf{decl\_vars}_{\mathcal{G}})$$
$$\mu^{i\ \eta,\mathfrak{p}}_{\mathsf{init}_{\mathbb{M}}}(\mathsf{decls}; v \leftarrow e) \stackrel{\mathsf{def}}{=} (\!|v \leftarrow e|\!)^{\eta,\mathfrak{p}}_{\mu} \text{ where } \mu = \mu^{i\ \eta,\mathfrak{p}}_{\mathsf{init}_{\mathbb{M}}}(\mathsf{decls})$$

Figure 3.6: Initial memory of a game $\mathcal{G}$ w.r.t. $\mathbb{M}$ and side bit i.

3.3 Semantics

The semantics of a program interacting with a game will be parameterized by a model $\mathbb{M}$ of $\mathcal{L}_{\mathsf{p}}$ specifying the semantics of types and library functions, the values of the security parameter η and the side b, a program random tape, and a memory. We start this section by defining memories.

3.3.1 Memories

A *memory* $\mu \in \mathsf{Mem}_{\mathbb{M},\eta}$ w.r.t. a type structure $\mathbb{M}$ and η is a function that associates to any variable $v \in \mathcal{X}_{\mathsf{p}}$ of type τ a value $\mu(v) \in [\![\tau]\!]^{\eta}_{\mathbb{M}}$. As usual, $\mu[v \mapsto a]$ is the memory such that $(\mu[v \mapsto a])(v) = a$ and $(\mu[v \mapsto a])(v') = \mu(v')$ for any variable $v' \neq v$.

Definition 8 (Initial Memory)**.** *The initial memory* $\mu^{i\ \eta,\mathfrak{p}}_{init_{\mathbb{M}}}(\mathcal{G})$ *of a game* $\mathcal{G}$ *for the security parameter* η*, program random tape* $\mathfrak{p}$*, model* $\mathbb{M}$ *and side bit* $i \in \{0,1\}$ *is defined in Figure 3.6. It is obtained by evaluating the deterministic global variable assignments. Moreover, the value of the security parameter is made available to the game and the simulator through the variable* eta*. Global random variables are not in this initial memory; they will be sampled by directly reading the random tape when needed.*

3.3.2 Program random tapes

To fit with the logic, all the randomness of our programs is eagerly sampled and passed to the program using read-only random tapes. We limit ourselves to sampling base types.

To sample a value of base type τ_{b}, we retrieve a vector $w_{\$}$ of $\mathsf{R}_{\mathbb{M},\eta}(\tau_{\mathsf{b}})$ bits from the random tapes, and then use the sampling algorithm $[\![\tau_{\mathsf{b}}]\!]^{\$}_{\mathbb{M}}(\eta, w_{\$})$ provided by the model to obtain a value in $[\![\tau_{\mathsf{b}}]\!]^{\eta}_{\mathbb{M}}$. To simplify the presentation and analysis of the bideduction logic in the following chapters, we use a different random tape for each usage: we will use a family of random tapes, one for each pair $(\mathtt{T}, \tau_{\mathsf{b}})$ of randomness source (i.e. in our particular case, $\mathtt{T} \in \{\mathtt{T_A}, \mathtt{T_G}, \mathtt{T_S}\}$) and base type $\tau_{\mathsf{b}} \in \mathbb{B}$ we are sampling from. However, we only consider bool for $\mathtt{T_A}$ since adversarial randomness is only needed for the adversarial function symbols in $\mathcal{L}_{\mathsf{p}}$.

Definition 9. *A* program random tape $\mathfrak{p}$ *is a family* $(\mathfrak{p}|_l)_{l \in L}$ *of infinite sequences of bits indexed by the set of* labels*:*

$$L \stackrel{\mathsf{def}}{=} \{(\mathtt{T_A}, \mathsf{bool})\} \cup \bigcup_{\tau_b \in \mathbb{B}} \{(\mathtt{T_G}, \tau_b)\} \cup \{(\mathtt{T_S}, \tau_b)\}.$$

For any tag T*, we split* $\mathfrak{p}|_{T,\tau}$ *into blocks of* $\mathsf{R}_{\mathbb{M},\eta}(\tau)$ *bits, and for any* $k \in \mathbb{N}$*, we let* $\mathfrak{p}|^{\eta,\mathbb{M}}_{T,\tau}[k]$ *be the* k^{th} *such block. We may omit* $\mathbb{M}$ *and* τ *when they are clear from the context.*

$$[\mathsf{b}]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu} \stackrel{\text{def}}{=} i \qquad [v]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu} \stackrel{\text{def}}{=} \mu(v) \quad \text{when } v \in \mathcal{X}_{\mathsf{p}}$$

$$[f]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu} \stackrel{\text{def}}{=} [\![f]\!]^{\eta,(\mathfrak{p}|_{\mathtt{T_A},\mathsf{bool}},\rho_0)}_{\mathbb{M}:\mathcal{E}} \quad \text{when } f \in \mathcal{L}_{\mathsf{p}} \qquad [e_1\ e_2]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu} \stackrel{\text{def}}{=} [e_1]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu}([e_2]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu})$$

Figure 3.7: Semantics of expressions w.r.t. a model $\mathbb{M} : \mathcal{E}$.

Finally, we let $\mathfrak{P}$ be the set of all program random tapes.

3.3.3 Expression and program semantics

We have defined all the key elements that make up our semantics. We start with the semantics of expressions, then move on to programs.

Expression semantics

We say that a logical environment $\mathcal{E}$ is *compatible* with the set of program variables $\mathcal{X}_{\mathsf{p}}$ and library $\mathcal{L}_{\mathsf{p}}$ if $\mathcal{L}_{\mathsf{p}} \subseteq \mathcal{E}$ and the set of variables defined or declared in $\mathcal{E}$ is disjoint from $\mathcal{X}_{\mathsf{p}}$.

The semantics $[e]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu}$ of an expression e of type τ is a value in $[\![\tau]\!]^{\eta}_{\mathbb{M}}$. This semantics is evaluated relatively to a memory μ, a model $\mathbb{M} : \mathcal{E}$ such that $\mathcal{X}_{\mathsf{p}}, \mathcal{L}_{\mathsf{p}}$ and $\mathcal{E}$ are compatible, a security parameter η, a program random tape $\mathfrak{p}$, and a bit $i \in \{0,1\}$ stating on which side the expression is evaluated. The semantics of expressions, defined in Figure 3.7, uses the bit i to interpret the special boolean term b, and the memory μ to evaluate program variables in $\mathcal{X}_{\mathsf{p}}$. Moreover, the semantics of a library function $f \in \mathcal{L}_{\mathsf{p}}$ is

$$[f]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu} \stackrel{\text{def}}{=} [\![f]\!]^{\eta,(\mathfrak{p}|_{\mathtt{T_A},\mathsf{bool}},\rho_0)}_{\mathbb{M}:\mathcal{E}}$$

i.e. the (logical) semantics of f in the model $\mathbb{M}$, using $\mathfrak{p}|_{\mathtt{T_A},\mathsf{bool}}$ as adversarial (logical) random tape, and the all-zero random tape ρ_0 as honest random tape — indeed any library function $\mathcal{L}_{\mathsf{p}}$ will be assumed to be adversarial, and therefore does not need *honest* randomness.

We omit $\mathbb{M}$ and i when they are clear from context, and write $[e]^{\eta,\mathfrak{p}}_{\mu}$ instead of $[e]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu}$.

Program semantics

The semantics of a program is parameterized by the game $\mathcal{G}$ that the program can interact with, a model $\mathbb{M} : \mathcal{E}$ (such that $\mathcal{X}_{\mathsf{p}}, \mathcal{L}_{\mathsf{p}}$ and $\mathcal{E}$ are compatible) used to interpret library function symbols, the side bit $i \in \{0,1\}$, and the security parameter η. The evaluation $(\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathcal{G},\mathbb{M},i,\mu} \in \mathsf{Mem}_{\mathbb{M},\eta} \cup \{\bot\}$ of a program p in memory μ and using the program random tape $\mathfrak{p}$ is either the memory obtained by executing p, or $\bot$ if the execution does not terminate. Its definition, given in Figure 3.8, is mostly standard; we describe next the treatment of oracle calls and samplings.

If v is a variable of type τ_{b}, then the evaluation of the random sampling $v \xleftarrow{\$} \mathtt{T}[e]$ w.r.t. memory μ and program random tape $\mathfrak{p}$ evaluates the integer e as an offset $k \in \mathbb{N}$, retrieves the k-th block of random bits $\mathfrak{p}|^{\eta}_{(\mathtt{T},\tau_{\mathsf{b}})}[k]$ from the random tape labelled by $(\mathtt{T},\tau_{\mathsf{b}})$, and uses it to run the sampling algorithm $[\![\tau_{\mathsf{b}}]\!]^{\$}_{\mathbb{M}}$ provided by the model $\mathbb{M}$.

$$
\begin{aligned}
(\!|v \leftarrow e|\!)^{\eta,\mathfrak{p}}_{\mu} &\stackrel{\text{def}}{=} \mu\left[v \mapsto [e]^{\eta,\mathfrak{p}}_{\mu}\right] \qquad (\!|\textbf{abort}|\!)^{\eta,\mathfrak{p}}_{\mu} \stackrel{\text{def}}{=} \bot \qquad (\!|\textbf{skip}|\!)^{\eta,\mathfrak{p}}_{\mu} \stackrel{\text{def}}{=} \mu \\
(\!|\mathsf{p}_0; \mathsf{p}_1|\!)^{\eta,\mathfrak{p}}_{\mu} &\stackrel{\text{def}}{=} \begin{cases} (\!|\mathsf{p}_1|\!)^{\eta,\mathfrak{p}}_{\mu'} & \text{if } (\!|\mathsf{p}_0|\!)^{\eta,\mathfrak{p}}_{\mu} = \mu' \\ \bot & \text{if } (\!|\mathsf{p}_0|\!)^{\eta,\mathfrak{p}}_{\mu} = \bot \end{cases} \\
(\!|\textbf{if } e \textbf{ then } \mathsf{p}_0 \textbf{ else } \mathsf{p}_1|\!)^{\eta,\mathfrak{p}}_{\mu} &\stackrel{\text{def}}{=} \begin{cases} (\!|\mathsf{p}_0|\!)^{\eta,\mathfrak{p}}_{\mu} & \text{if } [e]^{\eta,\mathfrak{p}}_{\mu} = \mathsf{true} \\ (\!|\mathsf{p}_1|\!)^{\eta,\mathfrak{p}}_{\mu} & \text{if } [e]^{\eta,\mathfrak{p}}_{\mu} = \mathsf{false} \end{cases} \\
(\!|\textbf{while } e \textbf{ do } \mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mu} &\stackrel{\text{def}}{=} \lim_{n\to\infty} (\!|\textbf{loop}_n|\!)^{\eta,\mathfrak{p}}_{\mu} \\
&\qquad \text{where } \textbf{loop}_n = \begin{pmatrix} (\textbf{if } e \textbf{ then } \mathsf{p} \textbf{ else skip})^n; \\ \textbf{if } e \textbf{ then abort else skip} \end{pmatrix} \\
(\!|v \xleftarrow{\$} \mathtt{T}[e]|\!)^{\eta,\mathfrak{p}}_{\mu} &\stackrel{\text{def}}{=} \mu\left[v \mapsto [\![\tau]\!]^{\$}_{\mathbb{M}}(\eta, \mathfrak{p}|^{\eta}_{(\mathtt{T},\tau)}[k])\right] \text{ where } k = [e]^{\eta,\mathfrak{p}}_{\mu} \text{ and } v \text{ has type } \tau \\
(\!|v \leftarrow O_f(\vec{e})[\vec{e}_g; \vec{e}_l]|\!)^{\eta,\mathfrak{p}}_{\mu} &\stackrel{\text{def}}{=} \text{let } \mu' = \mu \begin{bmatrix} f.\text{args} & \mapsto [\vec{e}]^{\eta,\mathfrak{p}}_{\mu} \\ f.\text{glob}_{\$} & \mapsto \mathfrak{p}|^{\eta}_{\mathtt{T_G}}[\,[\vec{e}_g]^{\eta,\mathfrak{p}}_{\mu}\,] \\ f.\text{loc}_{\$} & \mapsto \mathfrak{p}|^{\eta}_{\mathtt{T_G}}[\,[\vec{e}_l]^{\eta,\mathfrak{p}}_{\mu}\,] \end{bmatrix} \text{ in} \\
&\qquad \text{let } \mu'' = (\!|f.\text{prog}|\!)^{\eta,\mathfrak{p}}_{\mu'} \text{ in} \\
&\qquad \mu''\left[v \mapsto [f.\text{expr}]^{\eta,\mathfrak{p}}_{\mu''}\right]
\end{aligned}
$$

Figure 3.8: Program semantics w.r.t. a model $\mathbb{M} : \mathcal{E}$, a side $i \in \{0, 1\}$ and a game $\mathcal{G}$.

To evaluate an oracle call instruction $v \leftarrow O_f(\vec{e})[\vec{e}_g; \vec{e}_l]$, we first evaluate the arguments $\vec{e}$, the global randomness offsets $\vec{e}_g$ and the local randomness offsets $\vec{e}_l$, and store the results in, resp., f.args, $\mathcal{G}$.glob$_{\$}$ and f.loc$_{\$}$; then, we execute the oracle body f.prog; and finally, we store the result of the evaluation of the return expression f.expr in v.

Turing completeness

Let us finish on one remark about our programming language. It will be essential later to bridge the gap between two formulations of indistinguishability: the one based on adversaries expressed with programs (in our setting, see later on), and the other based on PPTMs in the equivalence semantics, define in Section 2.4.

For that, we made sure our syntax defines a Turing-complete programming language. This means that it can express any computation that a Turing machine can perform, assuming the model and environment provides certain minimal features: a library with bit-wise operations and a dedicated type for bits, interpreted as binary strings. So we assume the library does provide bit-wise operations, and notice that the type message already defines bitstrings. Then, all Turing machines can be expressed as a program in our setting.

In particular, for any PPTM $\mathcal{D}$, there exists a program p that only accesses the program tape $(\mathtt{T_A}, \mathsf{bool})$, such that for any infinite random tape ρ, and bitstring w, the PPTM $\mathcal{D}$ and program p perform the same computation, that is

$$\mathcal{D}(w, \rho) = (\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathbb{M}:\mathcal{E},[x\mapsto w]}[\mathsf{res}]$$

where $\mathfrak{p}$ is any family of tapes such that $\mathfrak{p}|_{(\mathtt{T_A},\mathsf{bool})} = \rho$.

3.3.4 Cost model

To keep our approach generic and abstract, we assume that our program semantics is endowed with a time-cost model satisfying some standard and expected properties.

More precisely, we assume a cost function C parameterized by the model $\mathbb{M}$ which associates to each program p, the security parameter η and memory μ a worst-case execution time $C_{\mathbb{M}}(\mathsf{p}, \eta, \mu) \in \mathbb{N} \cup \{+\infty\}$ which bounds execution times of p for all possible program tapes — this cost must be $+\infty$ if some execution does not terminate. Also, in the following part, it will be useful to restrict the cost to subset of μ. Thus, we say that p has inputs $\vec{X}$ when all variables that the program read before defining them are in $\vec{X}$. Then for any η and memory μ, the worst case execution time $C_{\mathbb{M}}(\mathsf{p}, \eta, \mu)$ is exactly the worst case execution time $C_{\mathbb{M}}(\mathsf{p}, \eta, [\vec{X} \mapsto \mu(\vec{X})]]$[2].

We say that a program p is PTIME w.r.t. $\mathbb{M}$ when $C_{\mathbb{M}}(\mathsf{p}, \eta, \mu)$ is bounded by a polynomial in η and $|\mu|$ (the sum of the sizes of all values stored in μ). We will assume only a few basic properties of this cost model:

- all expressions are PTIME, which is reasonable, as sampling procedures provided by the model are PTIME, and since library functions are assumed to be adversarial;
- the memory after executing a PTIME program is of polynomial size in η and the size of the initial memory;
- an oracle call is PTIME, which is both a constraint on the cost model and the game;
- if both p and q are PTIME programs, then so is $(p; q)$;
- **while** $l \neq []$ **do** $(\mathsf{p};\ l \leftarrow \mathsf{tail}\ l)$ is PTIME provided that p is a PTIME program that does not modify variable l, in all models where tail induces a well-founded ordering on the semantic values of type list and when
  - either the size of l and of its element is bounded by a constant;
  - or for any memory μ, the cost of p is bounded by a polynomial in only η and the size of l and its elements in μ.

> We distinguish two cases for PTIME while loops. The first case is very restrictive: we will use it when we build while loop where the program p can reuse any previous computation. This is particularly useful for induction. The second case applies when the program does not reuse previous computations. It will be useful to lift some restrictions on the list when possible. See Chapter 5 for details.

[2] All our programs may also uses special variables b and eta that we omit in the cost.

3.3.5 Adversaries

A program p has an adversarial behaviour against $\mathcal{G}$ with respect to a memory μ, a tape $\mathfrak{p}$, iff. it only calls the oracles of $\mathcal{G}$, respecting their type. Moreover, an adversary must not read the special side constant b, and must not read or write the game variables. Finally, the program must properly use random samplings, when executing in $\mathbb{M}$, with the security parameter η, the memory μ and the tape $\mathfrak{p}$:

- We forbid the adversary from directly sampling from the $\mathtt{T_G}$-labelled random tapes, which are reserved for the game's random samplings.
- We require that local offsets in oracle calls are fresh: an integer used as a local offset may not be used anywhere else as an offset, in this oracle or in a past or future call.
- We require that global offsets are consistent across all oracle calls: each of the game's global samplings must correspond to a unique global offset.

Also, we extend this notion to inputs, and say that p with input variables $\vec{X}$ has adversarial behaviour w.r.t. $\mathbb{M}, \eta, \mu, \vec{a}, \mathfrak{p}$ when it has adversarial behaviour w.r.t $\mathbb{M}, \eta, \mu[\vec{X} \mapsto \vec{a}], \mathfrak{p}$.

An adversary against $\mathcal{G}$ (or $\mathcal{G}$-adversary) with respect to $\mathbb{M}$ and η is a program which has an adversarial behaviours with respect to $\mathbb{M}, \eta, \mathfrak{p}, \mu^{i}_{\mathsf{init}_{\mathbb{M}}^{\eta,\mathfrak{p}}}$ for all tape $\mathfrak{p}$ and side i.

3.3.6 Adversaries and security

Finally, it is possible to define the security of a cryptographic game.

Definition 10. *We use a special variable* res *to store the return value of a program. A game* $\mathcal{G}$ *is secure in a compatible model* $\mathbb{M}$ *if for any PTIME adversary* p*, the following quantity is negligible in* η*:*

$$\left| \Pr_{\mathfrak{p}} \left((\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathcal{G},\mathbb{M},0,\mu_0} [\mathit{res}] = 1 \right) - \Pr_{\mathfrak{p}} \left((\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathcal{G},\mathbb{M},1,\mu_1} [\mathit{res}] = 1 \right) \right|$$

where $\mu_i = \mu^{i}_{\mathit{init}_{\mathbb{M}}^{\eta,\mathfrak{p}}}$ *for any* $i \in \{0, 1\}$ *is the initial memory of the game.*

Chapter 4

Bideduction

Contents

4.1 Overview **54**
4.2 Bideduction judgement **57**
4.2.1 Name constraints 57
4.2.2 Assertion logic 59
4.2.3 Bideduction judgement 60
4.3 Bideduce rule **63**
4.4 Chapter appendix: couplings **64**
4.4.1 Preliminaries: probability theory 64
4.4.2 Couplings and lifting lemma 65
4.4.3 Well-formedness of constraint systems 67
4.4.4 Couplings arrays 68
4.4.5 Constructing a coupling contained in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ 71
4.4.6 Proof of Theorem 1 72

In the previous chapter, we introduced a formal framework for expressing cryptographic reductions. We defined programs, games, and adversaries with precise specifications. In this chapter, we define the bideduction judgment, where these latter elements form the basis for its semantics. More precisely, this chapter formally describes how our approach allows to synthesize a simulator by significantly extending the notion of *bideduction* introduced in [30].

We start the chapter with an overview, following up on the overview in Section 3.1, to help build intuition on the deduction judgement and its key ingredients. In Section 4.2, we then define the key notions of *constraint systems*, *assertions* and related notions, and use these definitions to define the bideduction judgment. However, for the sake of clarity, we defer the rigorous definition of *well-formedness of constraint systems* and associated proofs in Section 4.4 and use only an intuitive understanding in Section 4.2. In a second section, Section 4.3, we provide a full definition of the inference rule linking biduction and CCSA-HO equivalence, the BIDEDUCE. Its soundness proof is also deferred in Section 4.4.

4.1 Overview

We begin this chapter with an overview to provide intuitions and first insights into the formal definitions and notions that follow. In the previous chapter, we introduced an example (see Section 3.1) to illustrate how our formalism represents games, simulators, adversaries, and cryptographic reductions. First, let us present what is a (mono-)deduction: we say that terms $\vec{v}$ can be deduced from terms $\vec{u}$ if there exists a polynomial-time simulator $\mathcal{S}$ such that $\mathcal{S}(\vec{u}) = \vec{v}$.

The goal of this overview is to provide an intuition for the key extension to mono-deductions: bi-objects, constraint systems and pre-and post-conditions, explaining what they represent, why they are needed, etc. The section ends with the bideduction rule, which captures how a term indistinguishability can be justified by a cryptographic reduction to a game indistinguishability.

Bideduction. In bideduction, the initial knowledge $\vec{u}$ and the target $\vec{v}$ are replaced by pairs of vectors of terms, respectively $\#(\vec{u}_0; \vec{u}_1)$ and $\#(\vec{v}_0; \vec{v}_1)$, called *bi-terms*, which typically represent messages in the left and right scenarios of an indistinguishability. We use a dash # to distinguish the pairing of the left and right scenarios from the standard pairing that can appear in the terms $\vec{u}_0, \vec{u}_1, \vec{v}_0, \vec{v}_1$. For example, the indistinguishability in Section 3.1:

$$\mathsf{frame}(\mathsf{pred}(t_0)), \mathsf{output}(t_0) \sim \mathsf{frame}(\mathsf{pred}(t_0)), \langle \mathsf{n}_{i_0,j_0}, \mathsf{n}_{\mathsf{fresh}} \rangle \tag{4.1}$$

can be represented by

$$\#(\mathsf{frame}(\mathsf{pred}(t_0)), \mathsf{output}(t_0); \quad \mathsf{frame}(\mathsf{pred}(t_0)), \langle \mathsf{n}_{i_0,j_0}, \mathsf{n}_{\mathsf{fresh}} \rangle)$$

or, alternatively,

$$\mathsf{frame}(\mathsf{pred}(t_0)), \#(\mathsf{output}(t_0); \langle \mathsf{n}_{i_0,j_0}, \mathsf{n}_{\mathsf{fresh}} \rangle)$$

by factorizing common parts of the left and right terms. Informally, we say that $\#(\vec{u}_0; \vec{u}_1)$ bideduces $\#(\vec{v}_0; \vec{v}_1)$ with access to the pair of games $(\mathcal{G}_0, \mathcal{G}_1)$, which we write

$$\#(\vec{u}_0; \vec{u}_1) \rhd_{(\mathcal{G}_0,\mathcal{G}_1)} \#(\vec{v}_0; \vec{v}_1)$$

if there exists a *single* polynomial-time simulator $\mathcal{S}$ such that $\mathcal{S}^{\mathcal{G}_0}(\vec{u}_0) = \vec{v}_0$ and $\mathcal{S}^{\mathcal{G}_1}(\vec{u}_1) = \vec{v}_1$. The cryptographic reduction argument of Section 3.1 is captured by the rule:

$$\frac{\emptyset \rhd_{(\mathcal{G}_0,\mathcal{G}_1)} \#(\vec{v_0}; \vec{v_1})}{\vec{v_0} \sim \vec{v_1}} \textsc{Bideduce} \tag{4.2}$$

To prove the soundness of this rule, assume that its conclusion is false. Then the adversary $\mathcal{A}$ against $\vec{v_0} \sim \vec{v_1}$ can be composed with the simulator $\mathcal{S}$ witnessing $\emptyset \rhd_{(\mathcal{G}_0,\mathcal{G}_1)} \#(\vec{v_0}; \vec{v_1})$ to obtain an adversary $\mathcal{B} = \mathcal{A} \circ \mathcal{S}$ against the games $(\mathcal{G}_0, \mathcal{G}_1)$. Thus, $\mathcal{S}$ is a sub-procedure of the adversary $\mathcal{B}$ against the cryptographic game $(\mathcal{G}_0, \mathcal{G}_1)$ we are reducing to — this is why it is crucial that the same simulator $\mathcal{S}$ is used for both sides of the bideduction judgment. Keep in mind that the bideduction premise requires an empty input: non-empty inputs $\#(\vec{u}_0; \vec{u}_1)$ will be useful later, e.g. to support transitivity and inductive reasoning steps in our proof system for bideduction.

```
1  (* Recursively compute input, output and frame using the hash oracle. *)
2  for each t ∈ [ init; t0[ {
3    match t with
4    | init → input[t] ← empty; output[t] ← empty; frame[t] ← empty
5    | T(i,j) →
6      input[t] ← att(frame[pred t])
7      n ←$ TS[offsetn(i, j)]   (* pre–sampled value access *)
8      if i = i0 then {
9        x ← G.hash(⟨ n, input[t] ⟩)[offsetk(i0)]   (* oracle call *)
10     } else {
11       k ←$ TS[offsetk(i)]   (* pre–sampled value access *)
12       x ← h(⟨ n, input[t] ⟩, k)
13     }
14     output[t] ← ⟨ n, x ⟩
15     frame[t] ← ⟨ frame[pred t], output[t] ⟩
16 }
17
18 (* Use the challenge oracle to compute output(t0) or ⟨n_{i0,j0}, n_fresh⟩. *)
19 input[t0] ← att(frame[pred t0])
20 n ←$ TS[offsetn(i0, j0)] (* pre–sampled value access *)
21 output' ← ⟨ n, G.challenge(⟨ n, input[t0] ⟩)[offsetk(i0), offset_{n_fresh}()] ⟩
22 return (frame[pred t0], output')
```

Figure 4.1: Reduction to the PRF assumption (copy of Figure 3.3).

We anticipate some elements of the next chapter on the proof system, as some components of the bideduction judgment are designed to allow to build a proof system. In the following, we present a selected set of rules, where we omit the games and write $\triangleright$ instead of $\triangleright_{(\mathcal{G}_0,\mathcal{G}_1)}$, and where bi-terms will be written with bold fonts e.g. $\vec{\boldsymbol{u}}$, $\vec{\boldsymbol{v}}$ and $\vec{\boldsymbol{w}}$ are all bi-terms.

A key goal while designing the bideduction judgment is to ensure that the judgment is *composable*, which is reflected in the proof system we aim at. Our proof system is a collection of rules capturing atomic simulators (samplings, oracle calls, assignments, etc.) and rules to compose them. Concretely, composition of simulators takes the form of *transitivity rule* that allows to transform a bideduction $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}, \vec{\boldsymbol{w}}$ into a bideduction of $\vec{\boldsymbol{v}}$ followed by a bideduction of $\vec{\boldsymbol{w}}$ with $\vec{\boldsymbol{v}}$ as additional input:

$$\frac{\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}} \qquad \vec{\boldsymbol{u}}, \vec{\boldsymbol{v}} \triangleright \vec{\boldsymbol{w}}}{\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}, \vec{\boldsymbol{w}}}$$

Coming back to Figure 3.3, reproduced here as Figure 4.1 for the reader's convenience, such rules will allow combining the simulator $\mathcal{S}_1$ (lines 2–16) of $\emptyset \triangleright \mathsf{frame}(\mathsf{pred}(t_0))$ with the simulator $\mathcal{S}_2$ (lines 19–21) of $\mathsf{frame}(\mathsf{pred}(t_0)) \triangleright \#(\mathsf{output}(t_0); \langle \mathsf{n}_{i_0,j_0}, \mathsf{n}_{\mathsf{fresh}} \rangle)$ to obtain the full simulator of Figure 4.1 which proves the equivalence in Eq. (4.1).

We must characterize the conditions under which the composition of $\mathcal{S}_1$ and $\mathcal{S}_2$ is an adversary against the game. This introduces two challenges:

- managing randomness consistently across composed simulations, and
- tracking the game's state accurately through sequences of oracle calls.

Randomness usage. The atomic simulators we want to capture in the proof system include random sampling and oracle calls. First performing a random sampling for a name n_i, is captured by the following rule:

$$\frac{}{\vec{u} \rhd \mathsf{n}_i}$$

For example, this rule allows us to infer the three pieces of simulator line 7, line 11 and line 20 in Figure 3.3.

Furthermore, the *oracle rule* allows the simulator to call any oracle of the game $\mathcal{G} \in \{\mathcal{G}_0, \mathcal{G}_1\}$ on any input it can compute. For instance, in the case of the hash oracle of the PRF game of Figure 3.2

$$\frac{\vec{u} \rhd \vec{v}}{\vec{u} \rhd \mathsf{h}(\vec{v}, \mathsf{k_i})}$$

which intuitively says that if one can simulate an input $\vec{v}$, one can simulate the output of an oracle calls on $\vec{v}$. This is the rule justifying the oracle call on line 9 of Figure 3.3.

The addition of these rules creates two issues:

i) We should not be able to use the oracle rule twice on two different keys $\mathsf{k_i}$ and $\mathsf{k_j}$, since the PRF game only supports hashing with a single and fixed key.

ii) We should not be allowed to sample a name in the simulator (through the name rule) and use this name to represent a sampling in the game (through the oracle rule).

We solve these issues by equipping our bideduction judgment with a system of *name constraints* that allow us to track the usage of randomness by the simulator. Our constraint systems notably allow us to express: *consistency* conditions on the secret keys of the game, ensuring that oracle calls are always asked to use the same offset for the same secret key; *ownership* of random samplings, preventing the simulator from directly accessing random values that are used as secret keys by the game. Concretely, constraint systems are recorded in bideduction judgments, and their validity is deferred until the end of the bideduction derivation. Roughly, the name and bideduction rule BIDEDUCE roughly look like this:

$$\frac{}{\{(\mathsf{n}_i, \mathsf{T_S})\} \vdash \vec{u} \rhd \mathsf{n}_i} \qquad \frac{\vdash \mathrm{Valid}(\mathcal{C}) \qquad \mathcal{C}; \emptyset \rhd \#(\vec{v_0}; \vec{v_1})}{\vec{v_0} \sim \vec{v_1}}$$

Here, $(\mathsf{n}_i, \mathsf{T_S})$ records that n_i has been sampled by the simulator, and $\mathrm{Valid}(\mathcal{C})$ is a standard SQUIRREL formula ensuring that the constraint system $\mathcal{C}$ is valid.

Stateful games. Recall that the oracles of the PRF game are guarded by conditions involving the logs $\ell_{\mathbf{hash}}$ and $\ell_{\mathbf{challenge}}$. E.g., the value $\langle \mathsf{n}_{\mathsf{i}_0,\mathsf{j}_0}, \mathsf{input}(\mathsf{T}(\mathsf{i}_0, \mathsf{j}_0)) \rangle$ sent to the challenge oracle at the end of the simulator (line 21 of Figure 3.3) must not have been already queried to the hash oracle **hash**. As discussed in the previous section, proving this requires establishing a property on the game's internal memory, namely that the set of previously hashed values is of the form:

$$\left\{ \langle \mathsf{n}_{\mathsf{i},\mathsf{j}}, \mathsf{input}(t) \rangle \mid t = \mathsf{T}(\mathsf{i}, \mathsf{j}) < \mathsf{T}(\mathsf{i}_0, \mathsf{j}_0) \right\}$$

To account for that, we equip our bideduction judgment with a Hoare-style pre-condition φ and post-condition ψ to track the memory state of the game. Putting everything together, we shall use bideduction judgments that are roughly of the form:

$$C, (\varphi, \psi) \vdash \#(\vec{u}_0; \vec{u}_1) \rhd_{(\mathcal{G}_0, \mathcal{G}_1)} \#(\vec{v}_0; \vec{v}_1)$$

Informally, this judgement states that there exists a simulator $\mathcal{S}$ using randomness as prescribed by the name constraints C such that, for any $i \in \{0; 1\}$, the execution of $\mathcal{S}$ on input $\vec{u}_i$ starting from a game $\mathcal{G}_i$ in the initial state satisfying φ computes $\vec{v}_i$ and leaves the game $\mathcal{G}_i$ in a final state satisfying ψ.

> The notion of bideduction we introduce in this chapter supports simulators whose capacities go beyond what was possible using the basic bideduction of [30]. In particular, we consider simulators that are *probabilistic* programs with access to *stateful oracles*. As shown in the example presented above, supporting these features requires extending bideduction in two non-trivial ways: we record the randomness usage of the simulator using name constraints (see Section 4.2.1) and we use Hoare-style pre- and post-condition to track the state of the game's internal memory (see Section 4.2.2). While the latter extension is standard in program logic, the former is a novel contribution of this thesis.

4.2 Bideduction judgement

We now develop our central concept: bideduction in presence of a cryptographic game. We will deal with several pairs of objects where each component is involved in the deduction on one side $i \in \{0, 1\}$ of the games. We introduce special notations for such pairs, following the style of [38].

Definition 11 (Bi-objects, $\boldsymbol{u}$, $\#(\_;\_)$)**.** *We call* bi-term *a pair of terms* $\boldsymbol{u} = \#(u_0; u_1)$*. We will similarly define and manipulate several kinds of bi-objects: for instance, we call (local) bi-formula a pair of local formulas* $\boldsymbol{f} = \#(f_0; f_1)$*. We allow ourselves to factorize common parts of a bi-term (or any bi-object) by pushing the # downwards: e.g.* $f(\#(u; v), g(\#(s; t)))$ *denotes* $\#(f(u, g(s)); f(v, g(t)))$*.*

Finally, for a bi-element $\boldsymbol{e} = \#(e_0; e_1)$*,* e_0 *is called the left element of* $\boldsymbol{e}$*, and* e_1 *the right element of* $\boldsymbol{e}$ *and for any bi-element* $\boldsymbol{e}$*,* e_0 *will always refer to its left element, and* e_1 *to its right element.*

We shall follow the intuitions given in Section 4.1, and derive a formal definition of bideduction for which we can prove that bideducibility entails indistinguishability. We begin by introducing two necessary preliminary ingredients: constraints on the use of random tapes, and assertions for describing the game's memory at a point in the simulator's computation.

4.2.1 Name constraints

Our simulators can perform random samplings, either directly or indirectly through oracle calls. Names in the bideduction judgement will be used in both cases to represent such

computations. For example, in the PRF game using key k, a simulator may compute $\mathsf{h}(m, \mathsf{k})$ through an oracle call (assuming that m is computable) but it may also compute $\mathsf{h}(m, \mathsf{s})$ explicitly when s and k are distinct names (by drawing s and computing the application of h itself). The two situations must be distinguished, as a simulator is forbidden from accessing the game's random samplings directly.

We introduce *name constraints* to keep track of how names are used in bideduction. We will make use of the following set of *tags*:

$$\mathsf{TAG}_{\mathsf{constr}} = \{\mathtt{T}_S, \mathtt{T}^{\mathsf{loc}}_{\mathsf{G}}\} \cup \{\mathtt{T}^{\mathsf{glob}}_{\mathsf{G},v} \mid v \in \mathcal{G}.\mathsf{gs}\}$$

Tag $\mathtt{T}_S$ indicates that a name corresponds to a random sampling of the simulator; $\mathtt{T}^{\mathsf{loc}}_{\mathsf{G}}$ corresponds to an oracle's local sampling; finally, $\mathtt{T}^{\mathsf{glob}}_{\mathsf{G},v}$ corresponds to the global sampling of variable v in the game.

Definition 12. *A* name constraint *is a tuple* $c = (\vec{\alpha}, \mathsf{n}, t, T, f)$ *where* $\vec{\alpha}$ *are variables in* $\mathcal{X}$ *whose types are tagged* finite, n *is a name,* t *is a term,* $T \in \mathsf{TAG}_{constr}$, *and* f *is a local formula. A* constraint system $\mathcal{C}$ *is a list of name constraints.*

Intuitively, a constraint expresses that, for any arbitrary instantiation of the variables $\vec{\alpha}$ such that f holds, the name n is used at index t as specified by tag T. Variables $\vec{\alpha}$ are bound in the constraint. Accordingly, constraints are considered modulo renaming of these variables and, when we consider several constraints jointly, we implicitly assume that their bound variables are disjoint. We do *not* require that free variables of t and f are all bound by $\vec{\alpha}$.

We formally define the *multiset* $\mathcal{N}^{\eta,\rho}_{\mathcal{C},\mathbb{M}} \stackrel{\mathsf{def}}{=} \bigcup_{c\in\mathcal{C}} \mathcal{N}^{\eta,\rho}_{c,\mathbb{M}}$ where:

$$\mathcal{N}^{\eta,\rho}_{(\vec{\alpha},\mathsf{n},t,T,f),\mathbb{M}} \stackrel{\mathsf{def}}{=} \{\langle \mathsf{n}, [\![t]\!]^{\eta,\rho}_{\mathbb{M}\sigma}, T\rangle \mid \mathsf{dom}(\sigma) = \vec{\alpha},\ [\![f]\!]^{\eta,\rho}_{\mathbb{M}\sigma} = \mathsf{true}\}$$

This interpretation of constraint systems supports a natural notion of constraint subsumption: we write $\mathcal{E}, \Theta \models \mathcal{C} \subseteq \mathcal{C}'$ when for any $\mathbb{M}$ such that $\mathbb{M} : \mathcal{E} \models \Theta$, for any η and ρ, we have the multiset inclusion $\mathcal{N}^{\eta,\rho}_{\mathcal{C},\mathbb{M}} \subseteq \mathcal{N}^{\eta,\rho}_{\mathcal{C}',\mathbb{M}}$.

Example 12. *The system* $[(\{i\}, \mathsf{n}, i, \mathtt{T}^{\mathsf{glob}}_{\mathsf{G},v}, f),\ (\{i\}, \mathsf{n}, i, \mathtt{T}_S, f')]$ *expresses that: for every value of* i *for which* f *holds,* $(\mathsf{n}\ i)$ *represents the global sampling of the variable* v *of the game; and for every value of* i *for which* f' *holds,* $(\mathsf{n}\ i)$ *represents a sampling performed by the simulator.*

For this to make sense, we expect the formulas f *and* f' *to be mutually exclusive, i.e.* $[\forall i.\neg(f \wedge f')]_{\mathrm{e}}$ *should be valid. Otherwise, there would exist a valuation* v *of* i *such that the index* v *of the name* n *would be tagged both as a simulator's and a game's sampling, which cannot happen in a valid interaction between the simulator and the game.*

Validity

We define a validity criterion for constraint systems that captures when the usage of names is consistent. First, name-tag associations must be *functional*: no name is associated to two different tags. Second, the local samplings must be *fresh*: the associated names do not occur anywhere else. Third, a globally sampled variable must be associated to a *unique*

$$\mathrm{Fun}(c_1, c_2) \stackrel{\mathsf{def}}{=} \begin{cases} \forall \vec{\alpha}_1 \forall \vec{\alpha}_2.\ f_1 \wedge f_2 \Rightarrow t_1 \neq t_2 \text{ when } T_1 \neq T_2, \mathsf{n}_1 = \mathsf{n}_2 \\ \top \text{ otherwise} \end{cases}$$

$$\mathrm{Fresh}(c_1, c_2) \stackrel{\mathsf{def}}{=} \begin{cases} \forall \vec{\alpha}_1 \forall \vec{\alpha}_2.\ f_1 \wedge f_2 \Rightarrow c_1(\vec{\alpha}_1) \neq c_2(\vec{\alpha}_2) \Rightarrow t_1 \neq t_2 \text{ when } T_1 = T_2 = \mathsf{T}^{\mathsf{loc}}_{\mathsf{G}},\ \mathsf{n}_1 = \mathsf{n}_2 \\ \top \text{ otherwise} \end{cases}$$

$$\mathrm{Unique}(c_1, c_2) \stackrel{\mathsf{def}}{=} \begin{cases} \forall \vec{\alpha}_1 \forall \vec{\alpha}_2.\ f_1 \wedge f_2 \Rightarrow t_1 = t_2 & \text{when } T_1 = T_2 \in \{\mathsf{T}^{\mathsf{glob}}_{\mathsf{G},\nu} \mid \nu \in \mathcal{G}.\mathsf{gs}\},\ \mathsf{n}_1 = \mathsf{n}_2 \\ \forall \vec{\alpha}_1 \forall \vec{\alpha}_2.\ f_1 \wedge f_2 \Rightarrow \bot & \text{when } T_1 = T_2 \in \{\mathsf{T}^{\mathsf{glob}}_{\mathsf{G},\nu} \mid \nu \in \mathcal{G}.\mathsf{gs}\},\ \mathsf{n}_1 \neq \mathsf{n}_2 \\ \top & \text{otherwise} \end{cases}$$

Figure 4.2: Constraint validity conditions, with $c_i = (\vec{\alpha}_i, \mathsf{n}_i, t_i, T_i, f_i)$ for $i \in \{1; 2\}$; and we let $c_1(\vec{\alpha}_1) \neq c_2(\vec{\alpha}_2)$ be a shorthand for $\top$ if c_1 and c_2 are distinct occurrences, and $\vec{\alpha}_1 \neq \vec{\alpha}_2$ otherwise.

name. These three conditions must hold whenever condition f holds, and are defined formally as local formulas in Figure 4.2. We finally define the *validity* of a constraint system $\mathcal{C}$ as the exact truth of all conditions on all pairs of constraint occurrences:

$$\mathrm{Valid}(\mathcal{C}) \stackrel{\mathsf{def}}{=} [\ \bigwedge_{c_1, c_2 \in \mathcal{C}} \mathrm{Fun}(c_1, c_2) \wedge \mathrm{Fresh}(c_1, c_2) \wedge \mathrm{Unique}(c_1, c_2)\ ]_{\mathrm{e}}$$

As expected, $\Theta \models \mathrm{Valid}(\mathcal{C}')$ and $\Theta \models \mathcal{C} \subseteq \mathcal{C}'$ imply $\Theta \models \mathrm{Valid}(\mathcal{C})$. The validity condition relies on the exact truth predicate, in other words, we require our simulator to always behave correctly w.r.t. randomness usage. Importantly, we never require that names are distinct but only that their indices are distinct. The former would be too strong: we certainly do not rule out the possibility that a simulator, performing a random sampling by itself, happens to obtain the same value as a game's random sampling.

We will make use of bi-systems of constraints $\boldsymbol{\mathcal{C}}$. In practice, they will be pairs of lists of the same length, so we view them as lists of bi-constraints. We define $\mathrm{Valid}(\boldsymbol{\mathcal{C}})$ as $\mathrm{Valid}(\mathcal{C}_1)\ \tilde{\wedge}\ \mathrm{Valid}(\mathcal{C}_2)$.

4.2.2 Assertion logic

As explained in Section 4.1, we need to keep track of the game's memory during the simulator's computation. We shall thus equip our bideduction judgement with pre- and post-conditions, relying on an abstract assertion language — a concrete instance of it will be taken in our implementation and in Chapter 6.

We thus assume an arbitrary language of assertions, with a notion of well-typedness w.r.t. environments, and a notion of satisfaction: given some environment $\mathcal{E}$, an assertion φ that is well-typed w.r.t. $\mathcal{E}$, a model $\mathbb{M} : \mathcal{E}$, a security parameter η, a random tape ρ and a memory μ, we write $\mathbb{M}, \eta, \rho, \mu \models^A \varphi$ to denote that φ is satisfied by the left-hand side elements. Note that an assertion φ can specify properties of both the game's memory μ and logical values, including names, thanks to ρ. This allows, e.g., to have an assertion expressing that the value of a particular name $[\![\mathsf{n}\ t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$ does not belong to some list stored in the game's memory μ.

We do not necessarily ask for the assertion logic to support usual logical constructor, we write $\mathbb{M}, \eta, \rho, \mu \models^A \varphi \Rightarrow \psi$ for $\mathbb{M}, \eta, \rho, \mu \models^A \varphi$ implies $\mathbb{M}, \eta, \rho, \mu \models^A \psi$. This is a

notation, the element $\varphi \Rightarrow \psi$ might not be an assertion formula. Also, when $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ are bi-assertions, we write $\mathbb{M}, \eta, \rho, \mu \models^A \boldsymbol{\varphi} \Rightarrow \boldsymbol{\psi}$ when $\mathbb{M}, \eta, \rho, \mu \models^A \varphi_0 \Rightarrow \psi_0$ and $\mathbb{M}, \eta, \rho, \mu \models^A \varphi_1 \Rightarrow \psi_1$.

4.2.3 Bideduction judgement

We now have all the ingredients to form the syntax of our bideduction judgement. Defining its semantics, though, requires a little more work.

Definition 13. *Let $\boldsymbol{u}^1, \ldots, \boldsymbol{u}^m, \boldsymbol{v}$ be a sequence of bi-terms of base types, and $m \in \mathbb{N}$. We say that a program p with distinguished variables $X_1, \ldots, X_m$ and* res *computes $\vec{\boldsymbol{u}} \triangleright \boldsymbol{v}$ in time $n \in \mathbb{R}$ w.r.t. $\mathbb{M}, \eta, \mathfrak{p}, \rho, \mu$ and side $i \in \{0,1\}$ when:*

$$\mu'[\mathsf{res}] = [\![v_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} \text{ with } \mu' = (\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu[X_k \mapsto [\![u_i^k]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}]_{1 \le k \le m}}$$

and the computation cost $C_{\mathbb{M}}(\mathsf{p}, \eta, \mu[X_k \mapsto [\![u_i^k]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}])$ is bounded by n. In this context, μ' is the final memory *of the computation.*

This notion is naturally lifted to vectors of bi-terms $\vec{\boldsymbol{v}}$.

Naively, one may then say that a bideduction $\boldsymbol{u} \triangleright \boldsymbol{v}$ holds w.r.t. a game $\mathcal{G}$ when there exists a simulator p against $\mathcal{G}$ which computes $\boldsymbol{u} \triangleright \boldsymbol{v}$ w.r.t. any $\mathbb{M}, \eta, \rho, \mathfrak{p}, \mu$ and i. While it makes sense to quantify universally over $\mathbb{M}, \eta, \mu$ and i, doing the same for $\mathfrak{p}$ and ρ would be meaningless, resulting in an unfeasible notion of bideduction. Intuitively, we can only expect the semantics of program p and v_i to coincide *if they agree on the parts of the tapes that are read*. Crucially, these parts will be described by the constraint system associated to the considered bideduction. For example, if we need a name k to correspond to the PRF game's (globally sampled) key key, it is necessary that the tapes ρ and $\mathfrak{p}$ coincide on positions corresponding to, resp., k (for ρ) and key (for $\mathfrak{p}$).

In order to define this relation between logical and program random tapes, we assume a mapping from (semantic) names to offsets in program random tapes: for each environment $\mathcal{E}$, for each name symbol $\mathsf{n} : \tau' \to \tau$ declared in $\mathcal{E}$, for each $\mathbb{M} : \mathcal{E}$, $\eta \in \mathbb{N}$ and $a \in [\![\tau']\!]^{\eta}_{\mathbb{M}}$, we assume an offset $O_{\mathbb{M},\eta}(\mathsf{n}, a) \in \mathbb{N}$, such that $(1^\eta, a) \mapsto O_{\mathbb{M},\eta}(\mathsf{n}, a)$ is injective and PTIME computable — this actually corresponds to the $\mathsf{offset}_{\mathsf{n}}(a)$ library function in the simulator of Figure 3.3.

Definition 14. *Let C be a constraint system and $\mathbb{M}$ a model, both w.r.t. $\mathcal{E}$. For any $\eta \in \mathbb{N}$, we define $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ as the relation between $\mathbb{T}_{\mathbb{M},\eta}$ and program random tapes $\mathfrak{P}$ such that $\rho\ \mathcal{R}^{\eta}_{C,\mathbb{M}}\ \mathfrak{p}$ holds when ρ_{a} is a prefix of $\mathfrak{p}[\mathtt{T}_{\mathtt{A}}, \mathsf{bool}]$ and for all $(\mathsf{n}, a, \mathtt{T}) \in \mathcal{N}^{\eta,\rho}_{C}$, $[\![\mathsf{n}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}(a) = \mathfrak{p}|^{\eta}_{\mathtt{T}}[O_{\mathbb{M},\eta}(\mathsf{n}, a)]$.*

Couplings between logical and program tapes. Constraining the bideduction $\emptyset \triangleright \boldsymbol{v}$ by C will guarantee that there exists a program p which computes $\emptyset \triangleright \boldsymbol{v}$ w.r.t. any tapes $\mathfrak{p}, \rho$ that are related by $\mathcal{R}^{\eta}_{C,\mathbb{M}}$, i.e. (omitting the initial memory):

$$\text{for all } i \in \{0,1\} \text{ and } \rho\ \mathcal{R}^{\eta}_{C,\mathbb{M}}\ \mathfrak{p},\ (\!|p|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},i}[\mathsf{res}] = [\![v_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}.$$

In order to be able to *lift* the equality above to an equality over *distributions* (required in computational indistinguishability), i.e. to show that for any possible value x,

$$\Pr_{\mathfrak{p} \in \mathfrak{P}}\left((\!|p|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},i}[\mathsf{res}] = x\right) = \Pr_{\rho \in \mathbb{T}_{\mathbb{M},\eta}}\left([\![v_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = x\right) \tag{4.3}$$

we rely on the standard notions of *probabilistic coupling* and *lifting* (as in, e.g., [62]). We only present the main intuitions here, deferring full details in Section 4.4.3.

Consider some distribution $\mathbb{C}$ over *pairs* of logical and program tapes in $\mathbb{T}_{\mathbb{M},\eta} \times \mathfrak{P}$. The *left marginal* of $\mathbb{C}$ is the distribution over $\mathbb{T}_{\mathbb{M},\eta}$ obtained by extracting the logical tape ρ from a pair of tapes $(\rho, \mathfrak{p})$ sampled according to $\mathbb{C}$. The *right marginal* of $\mathbb{C}$ is similar, except that it extracts the program tape $\mathfrak{p}$. A distribution $\mathbb{C}$ is said to be a *probabilistic coupling* of $\mathbb{T}_{\mathbb{M},\eta}$ and $\mathfrak{P}$, which we write $\mathbb{C} : \mathbb{T}_{\mathbb{M},\eta} \bowtie \mathfrak{P}$, if its left and right marginals follow the same distributions as the distributions endowing, resp., $\mathbb{T}_{\mathbb{M},\eta}$ and $\mathfrak{P}$. When $\mathbb{C} : \mathbb{T}_{\mathbb{M},\eta} \bowtie \mathfrak{P}$, we thus have, for any x:

$$\Pr_{\rho \in \mathbb{T}_{\mathbb{M},\eta}} \left([\![v_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = x \right) = \Pr_{(\rho,\mathfrak{p}) \in \mathbb{C}} \left([\![v_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = x \right) \tag{4.4}$$

$$\Pr_{\mathfrak{p} \in \mathfrak{P}} \left((\!| p |\!)^{\eta,\mathfrak{p}}_{\mathbb{M},i}[\mathsf{res}] = x \right) = \Pr_{(\rho,\mathfrak{p}) \in \mathbb{C}} \left((\!| p |\!)^{\eta,\mathfrak{p}}_{\mathbb{M},i}[\mathsf{res}] = x \right) \tag{4.5}$$

where the top (resp. bottom) equation follows from the left (resp. right) marginal property of $\mathbb{C}$.

Assume that we can build a coupling $\mathbb{C} : \mathbb{T}_{\mathbb{M},\eta} \bowtie \mathfrak{P}$ contained in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ (this roughly means that $\mathbb{C}$ only samples pairs of tapes related by $\mathcal{R}^{\eta}_{C,\mathbb{M}}$). Then Eq. (4.3) holds. Indeed:

$$\begin{aligned}
&\Pr_{\rho \in \mathbb{T}_{\mathbb{M},\eta}} \left([\![v_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = x \right) \\
&= \Pr_{(\rho,\mathfrak{p}) \in \mathbb{C}} \left([\![v_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = x \right) && \text{(by Eq. (4.4))} \\
&= \Pr_{(\rho,\mathfrak{p}) \in \mathbb{C}} \left((\!| p |\!)^{\eta,\mathfrak{p}}_{\mathbb{M},i}[\mathsf{res}] = x \right) && (\mathbb{C} \text{ contained in } \mathcal{R}^{\eta}_{C,\mathbb{M}}) \\
&= \Pr_{\mathfrak{p} \in \mathfrak{P}} \left((\!| p |\!)^{\eta,\mathfrak{p}}_{\mathbb{M},i}[\mathsf{res}] = x \right) && \text{(by Eq. (4.5))}
\end{aligned}$$

Couplings from constraint systems. Given a constraint system C, we would thus like to build a coupling that is contained in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$. It turns out that this cannot always be achieved: counter-examples, like Example 13 shown next, arise when a constraint $c = (\vec{\alpha}, \mathsf{n}, t, \mathtt{T}, f)$ is cyclic, e.g. because it features a condition f or an index t that depends on the name n t introduced in the constraint.

Example 13. *Consider a name* $\mathsf{n} : \mathsf{unit} \to \mathsf{bool}$ *and let* $C = \{c_0, c_1\}$ *with:*

$$\begin{aligned}
c_0 &= (\emptyset, \mathsf{n}, \langle\rangle, \mathtt{T}_S, \mathsf{n}\ \langle\rangle = 0) \\
c_1 &= (\emptyset, \mathsf{n}, \langle\rangle, \mathtt{T}^{\mathsf{loc}}_{\mathsf{G}}, \mathsf{n}\ \langle\rangle = 1)
\end{aligned}$$

In words, $\mathsf{n}\ \langle\rangle$ *must be seen as a simulator name when it is 0, and a local sampling of the game when it is 1. But, to know in which case we are, we must already have sampled* $\mathsf{n}\ \langle\rangle$*!*

Let us show that a coupling cannot be included in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$*. First observe that* $\rho\ \mathcal{R}^{\eta}_{C,\mathbb{M}}\ \mathfrak{p}$ *imposes that* $\rho_{\mathtt{a}}$ *is a prefix of* $\mathfrak{p}[\mathtt{T}_{\mathtt{A}}, \mathsf{bool}]$ *and:*

- *either* $[\![\mathsf{n}\ \langle\rangle]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = 0$ *and* $\mathfrak{p}|^{\eta}_{\mathtt{T}_S}[O_{\mathbb{M},\eta}(\mathsf{n}, \langle\rangle)] = 0$*;*
- *or* $[\![\mathsf{n}\ \langle\rangle]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = 1$ *and* $\mathfrak{p}|^{\eta}_{\mathtt{T}^{\mathsf{loc}}_{\mathsf{G}}}[O_{\mathbb{M},\eta}(\mathsf{n}, \langle\rangle)] = 1$*.*

Less formally, the logical tape must coincide with the simulator tape on $\mathsf{n}\ \langle\rangle$ *when this sampling is zero; otherwise it must coincide with the local sampling tape for that name. Thus, the program tape* $\mathfrak{p}$ *such that* $\mathfrak{p}|^{\eta}_{\mathtt{T}_S}[O_{\mathbb{M},\eta}(\mathsf{n}, \langle\rangle)] = 1$ *and* $\mathfrak{p}|^{\eta}_{\mathtt{T}^{\mathsf{loc}}_{\mathsf{G}}}[O_{\mathbb{M},\eta}(\mathsf{n}, \langle\rangle)] = 0$ *is not*

related to any logical tape in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ *— for any* ρ*, we do not have* $\rho\ \mathcal{R}^{\eta}_{C,\mathbb{M}}\ \mathfrak{p}$*. Hence the right marginal of a coupling included in* $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ *would never sample such tapes. This missing set of tapes has non-zero measure (in fact it has measure* $\frac{1}{4}$*) hence the right marginal of our coupling would not coincide with the standard distribution over program tapes, which is a contradiction.*

Such pathological cases are, however, irrelevant for our use of constraint systems, and we rule them out by introducing the notion of *well-formed* constraint system. Details have been postponed to Section 4.4.3. Given a valid and well-formed constraint system, we are then able to build the desired coupling. Roughly, this is done step by step: well-formedness ensures that there exists an order in which to sample the names corresponding to constraints such that, when processing a constraint c, we are able to compute f and t using the already sampled parts of the tape; then, if f holds, we sample the segments of the logical and program tapes determined by n, t and $\mathtt{T}$ (validity ensures that these segments are not yet sampled). Once all constraints are processed, the rest of the tapes are sampled using the relevant probability distributions.

The following key lemma establishes that any well-formed and valid constraint system C can be used to build a coupling contained in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ (see proof in Section 4.4.5).

Lemma 1. *Let* C *be a well-formed constraint system w.r.t.* $\mathbb{M}, \eta$ *such that* $\mathbb{M} \models \mathrm{Valid}(C)$*. Then, there exists a coupling* $\mathbb{C} : \mathbb{T}_{\mathbb{M},\eta} \bowtie \mathfrak{P}$ *contained in* $\mathcal{R}^{\eta}_{C,\mathbb{M}}$*.*

This lemma will be key to justifying our BIDEDUCE rule, which involves a bideduction judgement with empty inputs. However, the notion of well-formedness needs to be adapted to arbitrary bideductions: the general notion of well-formedness is relative to the input terms the input memory. This will be crucial to soundly chain well-formedness. Again details are postponed to section Section 4.4.

Bideduction. Finally, we define a *library model* of $\mathcal{L}_{\mathsf{p}}$ to be a model with respect to $\mathcal{L}_{\mathsf{p}}$, that is not defined on any variable that is not in $\mathcal{L}_{\mathsf{p}}$.

We finally define our intricate notion of bideduction.

Definition 15. *A bideduction judgement is of the form:*

$$\mathcal{E}, \Theta, \boldsymbol{C}, (\boldsymbol{\varphi}, \boldsymbol{\psi}) \vdash \vec{\boldsymbol{u}} \rhd_{\mathcal{G}} \vec{\boldsymbol{v}}$$

where $\mathcal{G}$ *is a game,* $\mathcal{E}$ *is an environment,* Θ *is a set of global formulas,* $\boldsymbol{C}$ *is a constraint bi-system, the pre-condition* $\boldsymbol{\varphi}$ *and post-condition* $\boldsymbol{\psi}$ *are bi-assertions, the inputs* $\vec{\boldsymbol{u}}$ *and output* $\vec{\boldsymbol{v}}$ *are vectors of bi-terms.*

It is valid *when, for any library model* $\mathbb{M}_0$*, there exists a program* p *and a polynomial* P *such that for any model* $\mathbb{M} : \mathcal{E}$ *extending* $\mathbb{M}_0$ *in such a way that* $\mathbb{M} \models \Theta \tilde{\wedge} \mathrm{Valid}(\boldsymbol{C})$*, for any* $\eta \in \mathbb{N}$*,* $i \in \{0, 1\}$*,* $\boldsymbol{C}$ *is well-formed w.r.t.* $\mathbb{M}, \eta$ *relatively to* $\vec{\boldsymbol{u}}$ *and* $\boldsymbol{\varphi}$ *and for any tapes* $\rho\ \mathcal{R}^{\eta}_{C_i,\mathbb{M}}\ \mathfrak{p}$*, and for any* μ *such that* $\mathbb{M}, \eta, \rho, \mu \models^{A} \varphi_i$*,*

- p *has an adversarial behaviours against* $\mathcal{G}$ *w.r.t.* $\mathbb{M}; \eta, \mu, [\![u]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \mathfrak{p}$*,*
- p *computes* $\vec{\boldsymbol{u}} \rhd \vec{\boldsymbol{v}}$ *in time* $P(\eta, |\ [\![u]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}\ |)$ *w.r.t.* $\mathbb{M}, \eta, \mathfrak{p}, \rho, \mu, i$ *and, the corresponding final memory* μ' *is such that* $\mathbb{M}, \eta, \rho, \mu' \models^{A} \psi_i$*.*

- *Moreover, we require that the computation of* p *when* $\mathsf{b} = i$ *relies on global samplings* $G_\$$ *and local samplings* $L_\$$ *such that*

$$G_\$ \subseteq \{ O_{\mathbb{M},\eta}(\mathsf{n}, a) \mid \langle \mathsf{n}, a, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},v} \rangle \in \mathcal{N}^{\eta,\rho}_{c,\mathbb{M}},\ c \in C_i \}$$
$$L_\$ \subseteq \{ O_{\mathbb{M},\eta}(\mathsf{n}, a) \mid \langle \mathsf{n}, a, \mathsf{T}^{\mathsf{loc}}_{\mathsf{G}} \rangle \in \mathcal{N}^{\eta,\rho}_{c,\mathbb{M}},\ c \in C_i \}$$

Note that, while the general structure of the previous definition is guided by the need to derive indistinguishabilities from bideducibilities (as proved formally in the next theorem), some aspects of the definition are not necessary for this goal but ease compositional proofs of bideduction through our proof system. This is the case for the conditions on local and global samplings, which make it easy to compose programs while preserving the fact that they are adversaries against $\mathcal{G}$. That will appear in Chapter 5.

> At a first sight, the fact that when $\boldsymbol{C}$ is valid and that the randomness of p is entirely covered by the names of $\boldsymbol{C}$ might seem redundant with the fact that p has an adversarial behaviour.
> Indeed, we designed the validity of $\boldsymbol{C}$ to be a formula that ensures a good handling of randomness, i.e. compatible with an adversarial handling. Further, the inclusion of the sets $G_\$$ and $L_\$$ in the names of $\boldsymbol{C}$ ensures that all samplings made by p are constraint by $\boldsymbol{C}$. So why do these two conditions not imply that p is an adversary? The key observation here is that $L_\$$ is a set. Thus, if p is sampling twice at the same offsets, it would not be visible. In particular, assume a simple constraint system that only ensures that a name n is fresh. It is valid. Now, imagine that p uses twice the offset corresponding to n for a fresh sampling in an oracle call. Then, the sampling of p will be in the set $L_\$$, but p does not respect the freshness constraint.

4.3 Bideduce rule

Having formally defined bideduction, we can now incorporate it into the CCSA-HO logical framework. For this purpose, we introduce a new inference rule that captures cryptographic reductions from a game indistinguishability to a CCSA-HO equivalence. First, we present the rule and its soundness Theorem 1.

In essence, the bideduction judgement states the existence of a simulator. At the beginning of this chapter, we started with a rule of the form:

$$\frac{\emptyset \rhd_{(\mathcal{G}_0,\mathcal{G}_1)} \#(\vec{v_0}; \vec{v_1})}{\vec{v_0} \sim \vec{v_1}}\ \textsc{Bideduce} \tag{4.6}$$

We enriched the bideduction judgement with two ingredients: a constraint system and pre- and post-conditions. Adding these ingredients, the rule becomes:

$$\frac{\boldsymbol{C}, (\boldsymbol{\varphi}, \boldsymbol{\psi}) \vdash \emptyset \rhd_{(\mathcal{G}_0,\mathcal{G}_1)} \#(\vec{v_0}; \vec{v_1})}{\vec{v_0} \sim \vec{v_1}}\ \textsc{Bideduce} \tag{4.7}$$

The question is: what should we require on $\boldsymbol{C}$, $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ for this rule to be sound? Coming back to the bideduction judgement, the premise implies that, intuitively, there

exists a simulator $\mathcal{S}$ that computes $\#(\vec{v}_0;\vec{v}_1)$ when interacting with $(\mathcal{G}_0,\mathcal{G}_1)$ such that starting in a memory satisfying $\boldsymbol{\varphi}$, it yields a memory satisfying $\boldsymbol{\psi}$, and the constraint system $\boldsymbol{C}$ captures the randomness usage of $\mathcal{S}$.

We prove the conclusion of the rule by cryptographic reduction to the game, *using* $\mathcal{S}$. As such, when $\mathcal{S}$ starts, the game was just initialized. Thus, $\boldsymbol{\varphi}$ must include the initial memory of the game, and $\boldsymbol{\psi}$ is arbitrary.

Also, we need $\mathcal{S}$ to be an adversary, which is ensured by the bideduction judgement only when $\boldsymbol{C}$ is valid, which we check using an additional premise.

We formalize this intuition in Theorem 1, which introduces the full rule and its soundness theorem.

Theorem 1. *Let $\mathcal{E}$ be an environment, Θ a set of global formulas, and $\boldsymbol{\varphi}$ be a bi-assertion such that, for all $\mathbb{M} : \mathcal{E}$ satisfying Θ, for all $i \in \{0,1\}$, η, ρ, and $\mathfrak{p}$ such that ρ_a is a prefix of $\mathfrak{p}[\mathtt{T_A} \times \mathsf{bool}]$, we have $\mathbb{M},\eta,\rho,\mu^{i}_{init\mathbb{M}}{}^{\eta,\mathfrak{p}}(\mathcal{G}) \models^{A} \varphi_i$. The following rule is sound w.r.t. models where $\mathcal{G}$ is secure, for any $\boldsymbol{C}$, $\#(\vec{v}_0;\vec{v}_1)$ and $\boldsymbol{\psi}$:*

$$\textsc{Bideduce}\quad \frac{\mathcal{E},\Theta \vdash \mathrm{Valid}(\boldsymbol{C}) \qquad \mathcal{E},\Theta,\boldsymbol{C},(\boldsymbol{\varphi},\boldsymbol{\psi}) \vdash \emptyset \rhd_{\mathcal{G}} \#(\vec{v}_0;\vec{v}_1)}{\mathcal{E},\Theta \vdash \vec{v}_0 \sim \vec{v}_1}$$

4.4 Chapter appendix: couplings

In this appendix we go back to Section 4.2.3, where we intuitively introduced the notion of well-formedness for constraint systems, coming from the need to lift semantical equalities to probabilistic equalities. We define formally the well-formnedness condition, and prove Lemma 1.

We first introduce necessary notions of probability theory in Section 4.4.1 and then define in Section 4.4.2 and Section 4.4.3 couplings and well-formedness notions. The two next sections aim at proving Lemma 1, i.e. that a probabilistic coupling contained in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ can be constructed from any well-formed and valid constraint systems C. First, we prove a preliminary result showing how to build a coupling between two distributions over arrays of independent and identically distributed (i.i.d. for short) values in Section 4.4.4, and we then use this result to prove Lemma 1 in Section 4.4.5. Finally, the section goes back to Theorem 1 and provides its proof in Section 4.4.6.

4.4.1 Preliminaries: probability theory

We first recall some standard definitions from measure and probability theory.

Definitions. For any set $\mathbb{S}$, we let $\mathcal{P}(\mathbb{S})$ be the *power-set* of $\mathbb{S}$. A *σ-algebra* $\mathcal{F}$ over a set $\mathbb{S}$ is a non-empty subset of $\mathcal{P}(\mathbb{S})$ closed under: i) complement; and ii), countable union and intersection. An element E of a σ-algebra is called an *event*. A *measurable space* $(\mathbb{S},\mathcal{F})$ is a set $\mathbb{S}$ equipped with a σ-algebra $\mathcal{F}$. A *measure space* $(\mathbb{S},\mathcal{F},\mu)$ is a measurable set $(\mathbb{S},\mathcal{F})$ together with a function $\mu : \mathcal{F} \to [0;1]$ — called a *measure* — such that i) $\mu(\emptyset) = 0$; ii) μ is non-negative (i.e. $\forall E \in \mathcal{F}$, $\mu(E) \geq 0$); iii) μ is σ-additive, i.e. for any countable sequences $(E_i)_{i\in\mathbb{N}}$ of disjoint elements of $\mathcal{F}$, $\mu(\bigcup_i E_i) = \sum_i \mu(E_i)$. A *probability space* $(\mathbb{S},\mathcal{F},\mu)$ is a measure space whose total mass is 1, i.e. $\mu(\mathbb{S}) = 1$. A *distribution* D

over a measurable space $(\mathbb{S}, \mathcal{F})$ is a function such that $(\mathbb{S}, \mathcal{F}, D)$ is a probability space. Two distributions D_1 and D_2 over $(\mathbb{S}, \mathcal{F})$ are said to be of the *same law* if $D_1(E) = D_2(E)$ for any $E \in \mathcal{F}$. Finally, a *random variable* $X : \Omega \to \mathbb{S}$ from a probability space $(\Omega, \mathcal{F}_\Omega, \mu_\Omega)$ to a measurable space $(\mathbb{S}, \mathcal{F}_\mathbb{S})$ is any function such that $\forall E \in \mathcal{F}_\mathbb{S}$, $X^{-1}(E) \in \mathcal{F}_\Omega$.

Notations. If $(\mathbb{S}, \mathcal{F}, \mu)$ is a measure space and E an event of $\mathcal{F}$, then the probability $\Pr(E)$ of E is simply $\mu(E)$. Similarly, if D is a distribution over $(\mathbb{S}, \mathcal{F})$ and E an event of $\mathcal{F}$, then $\Pr(D \in E) \stackrel{\text{def}}{=} D(E)$. If X is a random variable from the probability space $(\Omega, \mathcal{F}_\Omega, \mu_\Omega)$ to $(\mathbb{S}, \mathcal{F}_\mathbb{S})$ and E an event of $\mathcal{F}_\mathbb{S}$, then $\Pr(X \in E) \stackrel{\text{def}}{=} \mu(X^{-1}(E))$.

Distributions as programs. We will describe some distributions using programs written in pseudo-code, e.g. if D is a distribution, then the program $\mathsf{x} \xleftarrow{\$} D; \mathsf{y} \xleftarrow{\$} D;$ **return** $(\mathsf{x}, \mathsf{x}+\mathsf{y})$ defines a distribution over pair of values. Given a program p, we write $\Pr_p(E)$ the probability of event E w.r.t. the distribution defined by p.

π and λ systems. Let $\mathbb{S}$ be a set and $X \subseteq \mathcal{P}(\mathbb{S})$, then:

- $\sigma(X)$ is the smallest σ-algebra containing X — we say that X generates $\sigma(X)$.
- X is a *π-system* if X is closed under finite intersections.
- X is a *λ-system* if $\emptyset \in X$ and X is closed under complement and countable disjoint unions.

We recall the following standard result:

Proposition 1 (Dynkin (π, λ)-Theorem)**.** *Let P be a π-system and L a λ-system. If $P \subseteq L$ then $\sigma(P) \subseteq L$.*

To show that two distributions coincide, it is sufficient to show that they coincide on a generating π-system B.

Proposition 2. *Let $(\mathbb{S}, \mathcal{F})$ be a measurable set and D_1, D_2 be two distributions over $\mathbb{S}$. Let B by a π-system such that $\sigma(B) = \mathcal{F}$. If D_1 and D_2 agree on B then D_1 and D_2 agree on $\mathcal{F}$, i.e.*

$$\text{if } \forall E \in B, D_1(E) = D_2(E) \text{ then } \forall E \in \mathcal{F}, D_1(E) = D_2(E)$$

Proof. Let $L \stackrel{\text{def}}{=} \{E \in \mathcal{F} \mid D_1(E) = D_2(E)\}$. We can check that L is a λ-system. By hypothesis, $B \subseteq L$. Hence, by Dynkin (π, λ)-theorem, $\sigma(B) \subseteq L$, which, since B generates $\mathcal{F}$, means that $\mathcal{F} \subseteq L$. Moreover, we trivially have from the definition of L that $L \subseteq \mathcal{F}$. Hence $\mathcal{F} = L$, and thus that D_1 and D_2 coincides on $\mathcal{F}$. □

4.4.2 Couplings and lifting lemma

Recall that, in section Section 4.2.3, in order to be able to *lift* equalities over tapes in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ to equalities over probabilities, we relied on the standard notion of a probabilistic coupling and lifting (as in [62]). In this section, we give the definition of probabilistic coupling, before defining containment and a general lifting lemma.

Definition 16 (Probabilistic coupling)**.** *Let* $(\mathbb{S}_1, \mathcal{F}_1, \mu_1)$ *and* $(\mathbb{S}_2, \mathcal{F}_2, \mu_2)$ *be two probabilistic spaces. A coupling* $\mathbb{C}$ *of* μ_1 *and* μ_2*, written* $\mathbb{C} : \mu_1 \bowtie \mu_2$*, is a random variable* $\mathbb{C} : \Omega \to \mathbb{S}_1 \times \mathbb{S}_2$ *from some probabilistic space* Ω *to* $\mathbb{S}_1 \times \mathbb{S}_2$ *such that:*

- μ_1 *and* $\mathbb{C}$*'s* left marginal *follow the same law, i.e.:*

$$\forall E_1 \in \mathcal{F}_1.\ \Pr_{\mu_1}(E_1) = \Pr(\mathbb{C} \in E_1 \times \mathbb{S}_1).$$

- *similarly,* μ_2 *and* $\mathbb{C}$*'s* right marginal *follow the same law.*

The coupling we build will be contained in the relation $\mathcal{R}^{\eta}_{C,\mathbb{M}}$, ensuring that only related tapes are coupled.

Definition 17 (Probabilistic containement)**.** *Let* $(\mathbb{S}, \mathcal{F}, \mu)$ *be a probabilistic space and* $E \in \mathcal{F}$ *an event. We say that the measure* μ *is* contained *in* E*, when for all* $F \in \mathcal{F}$*,* $\mu(F) = \mu(F \cap E)$*.*

The following lemma allows to lift an equality over elements related by a relation R to a equality over probabilities, as long as there exists a probabilistic coupling contained in R.

Lemma 2. *Let* $(\mathbb{S}_1, \mathcal{F}_1, \mu_1)$ *and* $(\mathbb{S}_2, \mathcal{F}_2, \mu_2)$ *be two probabilistic spaces,* $R \subseteq \mathbb{S}_1 \times \mathbb{S}_2$ *a relation between* $\mathbb{S}_1$ *and* $\mathbb{S}_1$ *and* $E_1 \in \mathcal{F}_1$ *and* $E_2 \in \mathcal{F}_2$ *be events such that:*

$$\text{for all } x\ R\ y,\ \ x \in E_1 \text{ iff. } y \in E_2. \tag{4.8}$$

Then $\Pr_{\mu_1}(E_1) = \Pr_{\mu_2}(E_2)$ *if there exists a coupling* $\mu : \mu_1 \bowtie \mu_2$ *contained in* R*.*

Proof. First, notice that by Eq. (4.8):

$$(E_1 \times \mathbb{S}_2) \cap R = (E_1 \times E_2) \cap R \tag{4.9}$$

and

$$(\mathbb{S}_2 \times E_2) \cap R = (E_1 \times E_2) \cap R. \tag{4.10}$$

Now, let $\mu : \mu_1 \bowtie \mu_2$ be a coupling contained in R. Then:

$$\begin{aligned}
\Pr_{\mu_1}(E_1) &= \Pr_{\mu}(E_1 \times \mathbb{S}_2) && \text{(left marginal property)}\\
&= \Pr_{\mu}((E_1 \times \mathbb{S}_2) \cap R) && \text{(by containement)}\\
&= \Pr_{\mu}((E_1 \times E_2) \cap R) && \text{(by Eq. (4.9))}\\
&= \Pr_{\mu}((\mathbb{S}_1 \times E_2) \cap R) && \text{(by Eq. (4.10))}\\
&= \Pr_{\mu}(\mathbb{S}_1 \times E_2) && \text{(by containement)}\\
&= \Pr_{\mu_2}(E_2) && \text{(right marginal property)}
\end{aligned}$$

which concludes this proof. □

4.4.3 Well-formedness of constraint systems

The goal of this section is to define the notion of well-formedness of a constraint system used in Lemma 1.

Doing so requires us to first introduce what are constraint instances.

Definition 18 (Constraint instance)**.** *Let $\vec{\alpha} = (\alpha_0, \dots, \alpha_j)$ be a sequence of variables of type $\vec{\tau} = \tau_0, \dots, \tau_j$. An instance of a constraint $c = (\vec{\alpha}, \mathsf{n}, t, T, f)$ w.r.t. a type structure $\mathbb{M}_0$ and $\eta \in \mathbb{N}$ is an element $(\vec{a}, c)$ where $\vec{a} = (a_0, \dots, a_j)$ and for any $i \in \{0, \dots, j\}$, $a_i \in [\![\tau_i]\!]^{\eta}_{\mathbb{M}_0}$.*

Given a model $\mathbb{M}$ and a random tape ρ, we can interpret a constraint instance as a multi-set in a similar way to what we did with constraints:

$$\mathcal{N}^{\eta,\rho}_{(\vec{a},(\vec{\alpha},\mathsf{n},t,T,f)),\mathbb{M}} \stackrel{\text{def}}{=} \left\{ \langle \mathsf{n}, [\![t]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto \mathbb{1}^{\eta}_{\vec{a}}]}, T\rangle \;\middle|\; [\![f]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto \mathbb{1}^{\eta}_{\vec{a}}]} = \mathsf{true} \right\}$$

We lift this to any sequence l_C of constraint instances as follows:

$$\mathcal{N}^{\eta,\rho}_{l_C,\mathbb{M}} \stackrel{\text{def}}{=} \bigcup_{(\vec{a},c)\in l_C} \mathcal{N}^{\eta,\rho}_{(\vec{a},c),\mathbb{M}}$$

where, in the equation above, $\bigcup$ must be understood as multi-set union in the equation above.

We are now ready to explain what is a well-formed constraint system. Roughly, a constraint system C is well-formed if there exists an ordering $c_1, \dots, c_n$ of the concrete instances it represents that verifies the property that for any i, the instance $c_i = (\vec{a}, (\vec{\alpha}, \mathsf{n}, t, T, f))$ is such that the index t and condition f can be computed using only the names defined by the previous constraint instances $c_1, \dots, c_{j-1}$.

Definition 19 (Restriction of a random tape)**.** *The restriction $\rho_{|\mathbb{M},\eta,l_C}$ of a random tape ρ by a sequence of constraint instances l_C w.r.t. a model $\mathbb{M}$ and $\eta \in \mathbb{N}$ is the random tape obtained from ρ by zeroing all random bits that corresponds to names that are **not** in $\mathcal{N}^{\eta,\rho}_{l_C,\mathbb{M}}$.*

Definition 20 (Relative well-formedness of constraint instances)**.** *A finite sequence $l_C = (c_1, \dots, c_K)$ of constraint instances is well-formed w.r.t. a model $\mathbb{M} : \mathcal{E}$ and $\eta \in \mathbb{N}$ relatively to the terms $\vec{u}$ and an assertion φ when for any $k \le K$, if $c_k = (\vec{a}, (\vec{\alpha}, n, t, T, f))$ then there exists a function g such that for all tape ρ and memory μ such that $\mathbb{M} : \mathcal{E}, \eta, \rho, \mu \models \varphi$ then*

$$g(\rho_{|\mathbb{M},\eta,l^k_C}, \mu, [\![\vec{u}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}) = [\![(t \mid f)]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto \mathbb{1}^{\eta}_{\vec{a}}]:(\mathcal{E},\vec{\alpha})}$$

where $l^k_C = (c_0, \dots, c_{k-1})$.

We can now define the well-formedness of a constraint system.

Definition 21 (Relative Well-formedness of constraint systems)**.** *A constraint system C is well-formed w.r.t. a model $\mathbb{M}$ and $\eta \in \mathbb{N}$ relatively to a vector of input terms $\vec{u}$ and an assertion φ when there exists a sequence l_C of constraints instances such that for any tape ρ, $\mathcal{N}^{\eta,\rho}_C = \mathcal{N}^{\eta,\rho}_{l_C,\mathbb{M}}$, and l_C is well-formed w.r.t. $\mathbb{M}, \eta$ relatively to $\vec{u}, \varphi$.*

In that case, we say that l_C witnesses the well-formedness of C.

We write $\mathcal{E}, \Theta \models_{\mathsf{WF}(\vec{u},\varphi)} \mathcal{C}$ if $\mathcal{C}$ is well-formed w.r.t. $\mathbb{M}, \eta$ relatively to $\vec{u}, \varphi$ for any η and any $\mathbb{M} : \mathcal{E}$ such that $\mathcal{E}, \Theta \models \mathbb{M}$. For pairs $\mathcal{C} = \#(C_0; C_1)$ of constraint systems, $\mathcal{E}, \Theta \models_{\mathsf{WF}(\vec{u},\varphi)} \mathcal{C}$ stands for well-formedness of both C_0 relatively to $\vec{u_0}, \varphi_0$ and C_1 relatively to $\vec{u_1}, \varphi_1$. The fact that the notion is relatively to term and memory comes from the need to compose the well-formedness of constraints system in the way we do for adversaries in the bideduction inference rules later on. Still, the well-formedness we need in Lemma 1 must be absolute, in the sense that the functions that computes step by step the components of the constraint system takes no inputs (i.e. no inputs terms and no memory). The notion of well-formnedness is defined below.

Definition 22 (Well-formedness of constraint systems)**.** *A constraint system $\mathcal{C}$ is said well-formed w.r.t. a model $\mathbb{M}$ and $\eta \in \mathbb{N}$ if and only if it is well-formed relatively to*

- *the empty term, and*
- *the assertion $\bot$, a special assertion formula such that only the empty memory can satisfy.*

Note that, if $\mathcal{C}$ is well-formed w.r.t. a model $\mathbb{M} : \mathcal{E}$ and $\eta \in \mathbb{N}$, then there exists a list of constraint instances $l_{\mathcal{C}} = (c_1, \ldots, c_K)$ of constraint such that for any $k \leq K$, if $c_k = (\vec{a}, (\vec{\alpha}, n, t, T, f))$ then there exists a function g such that for all tape ρ

$$g(\rho_{|\mathbb{M},\eta,l_{\mathcal{C}}^k}) = [\![(t \mid f)]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto \mathbb{1}_{\vec{a}}^{\eta}]:(\mathcal{E},\vec{\alpha})}$$

where $l_{\mathcal{C}}^k = (c_0, \ldots, c_{k-1})$.

Then, the following lemma close the gap between relative well-formedness and well-formedness.

Lemma 3. *Let $\mathcal{C}$ be well-formed w.r.t. a model $\mathbb{M}$ and $\eta \in \mathbb{N}$ relatively to $\emptyset, \varphi$ and let us assume there exists a function s such that, for all ρ, $s(\rho_a)$ produces a memory μ such that $\mathbb{M} : \mathcal{E}, \eta, \rho, \mu \models \varphi$. Then $\mathcal{C}$ is well-formed w.r.t. a model $\mathbb{M}$ and $\eta \in \mathbb{N}$.*

4.4.4 Couplings arrays

We prove some preliminary results showing how to build couplings of arrays of values.

I.i.d. sampling of arrays. Let $\mathbb{I}$ be a finite set, and let $D_{\mathbb{S}}$ be a fixed but arbitrary distribution over some measurable space $(\mathbb{S}, \mathcal{F})$. We identify the set $\mathbb{S}^{\mathbb{I}}$ with arrays indexed by $\mathbb{I}$ of values in $\mathbb{S}$.

Definition 23. *We let $D_{\mathbb{S}}^{\mathbb{I}}$ be the distribution over $\mathbb{S}^{\mathbb{I}}$ (equipped with the product σ-algebra) where all cells are* independently *sampled according to $D_{\mathbb{S}}$, i.e. the distribution defined by the program (in pseudo-code):*

$$\begin{array}{l} \mathsf{a} \leftarrow [\bot \ \textit{for}\ \_ \in \mathbb{I}]; \\ \textbf{\textit{for}}\ (j \in \mathbb{I})\ \textbf{\textit{do}}\ \{\, \mathsf{a}[j] \xleftarrow{\$} D_{\mathbb{S}};\, \} \\ \textbf{return}\ \mathsf{a}; \end{array} \tag{4.11}$$

where $\bot$ is a special element (s.t. $\bot \notin \mathbb{S}$) used to denote a cell that is yet to be sampled.

Proposition 3. *Let $\mathbb{I}$ be a finite set, and p be any program of the form:*

$$\begin{array}{l}\mathsf{a} \leftarrow [\bot \text{ for } \_ \in \mathbb{I}];\\ \mathsf{s} \leftarrow \mathsf{s}_{\mathsf{init}};\\ \textbf{for } (\_ \in |\mathbb{I}|) \textbf{ do } \{\, i \leftarrow \mathsf{f}(\mathsf{s});\ \mathsf{a}[i] \xleftarrow{\$} D_{\mathbb{S}};\ \mathsf{s} \leftarrow \mathsf{g}(\mathsf{s},\mathsf{a});\ \}\\ \textbf{return } \mathsf{a};\end{array} \tag{4.12}$$

where $\mathsf{s}_{\mathsf{init}}$, f and g are arbitrary mathematical deterministic functions such that at the end of the execution of the above program, all cells in $\mathbb{I}$ are sampled.

Then p defines a distribution over $\mathbb{S}^{\mathbb{I}}$ of law $D_{\mathbb{S}}^{\mathbb{I}}$.

Proof. Let $n = |\mathbb{I}|$ and $E_1, \ldots, E_n \in \mathcal{F}$ be events of $(\mathbb{S}, \mathcal{F})$. First, let us prove that:

$$\Pr_p(\mathsf{a} \in \textstyle\prod_i E_i) = \Pr_{p_0}(\mathsf{a} \in \prod_i E_i) \tag{4.13}$$

where p_0 is the program sampling the array in an i.i.d. fashion as described in Eq. (4.11) (hence $\Pr_{p_0}(\mathsf{a} \in \prod_i E_i) = \prod_i \Pr(D_{\mathbb{S}} \in E_i)$). We start be splitting the sum:

$$\Pr_p(\mathsf{a} \in \textstyle\prod_i E_i) \;=\; \sum_\sigma \Pr_p((\mathsf{a} \in \prod_i E_i) \mid A_\sigma) \cdot \Pr_p(A_\sigma)$$

where the sum is over all permutations of $\{1, \ldots, n\}$ and A_σ is the event: "p sampled values in the array in the order σ". Conditioned by A_σ, the probability that p samples an array in $\prod_i E_i$ is the probability that the program:

$$\begin{array}{l}\mathsf{a} \leftarrow [\bot \text{ for } \_ \in \mathbb{I}];\\ \textbf{for } (j \in \mathbb{I}) \textbf{ do } \{\, \mathsf{a}[\sigma(j)] \xleftarrow{\$} D_{\mathbb{S}};\ \}\\ \textbf{return } \mathsf{a};\end{array}$$

samples an array in $\prod_i E_i$, i.e. $\prod_i \Pr(D_S \in E_{\sigma^{-1}(i)})$. Hence:

$$\begin{aligned} & \textstyle\sum_\sigma \Pr_p((\mathsf{a} \in \prod_i E_i) \mid A_\sigma) \cdot \Pr_p(A_\sigma)\\ =\ & \textstyle\sum_\sigma \prod_i \Pr(D_S \in E_{\sigma^{-1}(i)}) \cdot \Pr_p(A_\sigma)\\ =\ & \textstyle\prod_i \Pr(D_S \in E_i) \cdot \sum_\sigma \Pr_p(A_\sigma)\\ =\ & \textstyle\prod_i \Pr(D_S \in E_i)\end{aligned}$$

This concludes the proof of Eq. (4.13).

To finish the proof, we must show that $\Pr_p(\mathsf{a} \in E) = \Pr_{p_0}(\mathsf{a} \in E)$ for any event E in the product σ-algebra $\prod_{1 \le i \le n} \mathcal{F}$. Let B be the set:

$$B \stackrel{\text{def}}{=} \{E_1 \times \cdots \times E_n \mid E_1, \ldots, E_n \in \mathcal{F}\}$$

We know that p and $D_{\mathbb{S}}^{\mathbb{I}}$ coincide on B (by Eq. (4.13)). Moreover, we can check that B is a π-system. By Proposition 2, p and $D_{\mathbb{S}}^{\mathbb{I}}$ agree on the σ-algebra generated by B, which is the product σ-algebra over $\mathbb{S}^{\mathbb{I}}$. Consequently, p is of law $D_{\mathbb{S}}^{\mathbb{I}}$. □

Couplings i.i.d. arrays from selection functions. Let $\mathbb{I}_1$ and $\mathbb{I}_2$ be two finite sets, and let $D_{\mathbb{S}}$ be fixed by arbitrary distribution over a measurable space $(\mathbb{S}, \mathcal{F})$ (the sample space).

Assume that we have a function select such that, for any two partially sampled arrays $\mathsf{a}_1 : \mathbb{I}_1 \to \mathbb{S} \cup \{\bot\}$ and $\mathsf{a}_2 : \mathbb{I}_2 \to \mathbb{S} \cup \{\bot\}$, $(\mathsf{select}\ \mathsf{a}_1\ \mathsf{a}_2)$ either selects a pair of $\bot$-valued indices of a_1 and a_2, or returns a special value done. More precisely:

$$\begin{gathered}\forall \mathsf{a}_1, \mathsf{a}_2.\ \mathsf{select}\ \mathsf{a}_1\ \mathsf{a}_2 \in (\mathbb{I}_1 \times \mathbb{I}_2) \cup \{\mathsf{done}\} \\ \text{and } \mathsf{select}\ \mathsf{a}_1\ \mathsf{a}_2 = (i_1, i_2) \Rightarrow \mathsf{a}_1[i_1] = \bot \wedge \mathsf{a}_2[i_2] = \bot\end{gathered} \tag{4.14}$$

Let $p_c(\mathsf{select})$ be the distribution over $D_{\mathbb{S}}^{\mathbb{I}_1} \times D_{\mathbb{S}}^{\mathbb{I}_2}$ defined by the program (in pseudo-code):

$\mathsf{a}_1 \leftarrow [\bot \text{ for } \_ \in \mathbb{I}_1];$
$\mathsf{a}_2 \leftarrow [\bot \text{ for } \_ \in \mathbb{I}_2];$
while $(\mathsf{select}\ \mathsf{a}_1\ \mathsf{a}_2 \neq \mathsf{done})$ **do** {
  $(i_1, i_2) \leftarrow \mathsf{select}\ \mathsf{a}_1\ \mathsf{a}_2;$
  $\mathsf{v} \xleftarrow{\$} D_{\mathbb{S}};$
  $\mathsf{a}_1[i_1] \xleftarrow{\$} \mathsf{v};$
  $\mathsf{a}_2[i_2] \xleftarrow{\$} \mathsf{v};$
}
for $(i \in \mathbb{I}_1)$ **do** {**if** $(\mathsf{a}_1[i] = \bot)$ **then** $\mathsf{a}_1[i] \xleftarrow{\$} D_{\mathbb{S}}$; **else skip**}
for $(i \in \mathbb{I}_2)$ **do** {**if** $(\mathsf{a}_2[i] = \bot)$ **then** $\mathsf{a}_2[i] \xleftarrow{\$} D_{\mathbb{S}}$; **else skip**}
return $(\mathsf{a}_1, \mathsf{a}_2);$

Proposition 4. *For any selection function* select *satisfying Eq. (4.14), we have that:*

$$p_c(\mathsf{select}) : D_{\mathbb{S}}^{\mathbb{I}_1} \bowtie D_{\mathbb{S}}^{\mathbb{I}_2}.$$

Proof. It is clear that $p_c(\mathsf{select})$'s left marginal follows the same distribution as:

$\mathsf{a}_1 \leftarrow [\bot \text{ for } \_ \in \mathbb{I}_1];$
$\mathsf{a}_2 \leftarrow [\bot \text{ for } \_ \in \mathbb{I}_2];$
while $(\mathsf{select}\ \mathsf{a}_1\ \mathsf{a}_2 \neq \mathsf{done})$ **do** {
  $(i_1, i_2) \leftarrow \mathsf{select}\ \mathsf{a}_1\ \mathsf{a}_2;$
  $\mathsf{a}_1[i_1] \xleftarrow{\$} D_{\mathbb{S}};$
  $\mathsf{a}_2[i_2] \leftarrow \mathsf{a}_1[i_1];$
}
for $(i \in \mathbb{I}_1)$ **do** {**if** $(\mathsf{a}_1[i] = \bot)$ **then** $\mathsf{a}_1[i] \xleftarrow{\$} D_{\mathbb{S}}$; **else skip**}
return $\mathsf{a}_1;$

This program samples all the cells of a_1 independently according to the distribution $D_{\mathbb{S}}$, in some particular order. By Proposition 3, we know that the order in which we sample cells does not matter, and that the distribution defined by this program is of law $D_{\mathbb{S}}^{\mathbb{I}_1}$.

Repeating the same reasoning on the right, we get that right marginal of $p_c(\mathsf{select})$ follows the distribution $D_{\mathbb{S}}^{\mathbb{I}_2}$, which concludes this proof. □

4.4.5 Constructing a coupling contained in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$

We now recall and prove Lemma 1.

Lemma 1. *Let C be a well-formed constraint system w.r.t. $\mathbb{M}, \eta$ such that $\mathbb{M} \models \mathrm{Valid}(C)$. Then, there exists a coupling $\mathbb{C} : \mathbb{T}_{\mathbb{M},\eta} \bowtie \mathfrak{P}$ contained in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$.*

Proof. Let $\mathbb{M}$ be a model, η a value of the security parameter and C a constraint system such that C is both valid and well-formed w.r.t. $\mathbb{M}$. We are going to build, for any $\eta \in \mathbb{N}$, a coupling that is contained in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$.

We use the framework of Proposition 4 for building couplings. We instantiate it such that a_1 represents a (partially defined) logical tape, which will be noted ρ, and a_2 represents the relevant finite portion of a partial computational tape, noted $\mathfrak{p}$. Given a partial logical tape ρ, mapping each type and index in $\mathsf{R}_{\mathbb{M},\eta}(\tau)$ to a value in $\{0, 1, \bot\}$, we say that a term t is well-defined w.r.t. ρ when $[\![t]\!]^{\eta,\rho_1}_{\mathbb{M}:\mathcal{E}} = [\![t]\!]^{\eta,\rho_2}_{\mathbb{M}:\mathcal{E}}$ for all tapes ρ_1 and ρ_2 that coincide with ρ where it is defined. When it is the case, we allow ourselves to simply write $[\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$ for this unique value.

We now describe the selection function $(\mathsf{select}\ \rho\ \mathfrak{p})$ with which we instantiate the framework. Let l_C be a sequence of instances witnessing the well-formedness of C w.r.t. $\mathbb{M}, \eta$. At each iteration, the function select chooses, if it exists, the smallest integer k, such that the k^{th} element in l_C is a constraint instance $(\vec{a}, c)$ with $c = (\vec{\alpha}, \mathsf{n}, t, T, f) \in C$ such that:

(a) both $[\![f]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto\vec{a}]:\mathcal{E},\vec{\alpha}}$ and $[\![t]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto\vec{a}]:\mathcal{E},\vec{\alpha}}$ are defined w.r.t. ρ, and the former is true;

(b) ρ still contains $\bot$ in the segment corresponding to name n and index $[\![t]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto\vec{a}]:\mathcal{E},\vec{\alpha}}$;

(c) $\mathfrak{p}$ still contains $\bot$ in the segment corresponding to name n, index $[\![t]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto\vec{a}]:\mathcal{E},\vec{\alpha}}$ and tag T,

and returns the corresponding indices in the logical and computational tapes. Otherwise, it returns done.

It should be noted that select does not simply consider offsets in the order prescribed by l_C. To explain why this is necessary, consider two equivalent constraint instances $(\vec{a}, c)$ and $(\vec{a'}, c')$, i.e. such that they both satisfy (a) and their indices are the same:

$$[\![t]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto\vec{a}]:\mathcal{E},\vec{\alpha}} = [\![t']\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto\vec{a'}]:\mathcal{E},\vec{\alpha}}.$$

Then, $(\vec{a}, c)$ and $(\vec{a'}, c')$ refers to the same offsets, and the function select should not return twice the same offsets.

We now show that our coupling is contained in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$. To do so, consider an arbitrary run of $p_c(\mathsf{select})$. We say that an instance $(\vec{a}, c)$ is addressed at some point in this run if satisfies (a) but neither (b) nor (c). Once an instance is addressed, the value of the corresponding name will have been set in the tapes. Note, though that an instance needs not be selected to be addressed: it suffices that an equivalent constraint instances is selected.

We observe that, at every step of our run, and for every instance for which condition (a) holds, conditions (b) and (c) are equivalent. Indeed, if only one kind of tape is defined

for our name, it must have been set due to the previous selection of another constraint instance, but validity imposes that distinct instances address distinct names.

Then, we note that, for every instance $(\vec{a}, c)$ in the sequence l_C, if all instances preceding $(\vec{a}, c)$ in l_C have been addressed then condition (a) holds. This is a consequence of well-foundedness. Indeed, let $c = (\vec{\alpha}, \mathsf{n}, t, T, f)$ and $l' = \big((\vec{a}_0, c_0), \ldots, (\vec{a}_k, c_k)\big)$ be the instances strictly before the position of the instance $(\vec{a}, c)$ we consider in l_C. We have that for any $\hat{\rho}$, the semantics of $(t \mid f)$ is a function of $\hat{\rho}_{|\mathbb{M},\eta,l'}$. We assumed that all the instances of l' have been addressed. Then for any tapes $\hat{\rho}$ and $\tilde{\rho}$ that coincide with the partial tape ρ, we have that $\hat{\rho}_{|\mathbb{M},\eta,l'} = \tilde{\rho}_{|\mathbb{M},\eta,l'}$, and thus $(t \mid f)$ is well-defined w.r.t. ρ.

To conclude, every instance in l_C will eventually be addressed. Hence, for any $(n, v, \mathtt{T}) \in \mathcal{N}^{\eta,\rho}_{l_C,\mathbb{M}} = \mathcal{N}^{\eta,\rho}_{C,\mathbb{M}}$, we have $[\![n]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}(v) = \mathfrak{p}|^{\eta}_{\mathtt{T}}[O_{\mathbb{M},\eta}(n, v)]$. The rest of $\mathcal{R}^{\eta}_{C,\mathbb{M}}$, concerning ρ_a and $\mathfrak{p}[\mathtt{T}_S, \mathsf{bool}]$ is obvious. □

4.4.6 Proof of Theorem 1

We recall Theorem 1 and proved it.

Theorem 1. *Let $\mathcal{E}$ be an environment, Θ a set of global formulas, and $\boldsymbol{\varphi}$ be a bi-assertion such that, for all $\mathbb{M} : \mathcal{E}$ satisfying Θ, for all $i \in \{0, 1\}$, η, ρ, and $\mathfrak{p}$ such that ρ_a is a prefix of $\mathfrak{p}[\mathtt{T}_\mathtt{A} \times \mathsf{bool}]$, we have $\mathbb{M}, \eta, \rho, \mu^{i\ \eta,\mathfrak{p}}_{init\mathbb{M}}(\mathcal{G}) \models^A \varphi_i$. The following rule is sound w.r.t. models where $\mathcal{G}$ is secure, for any $\boldsymbol{C}$, $\#(\vec{v_0}; \vec{v_1})$ and $\boldsymbol{\psi}$:*

$$\frac{\mathcal{E}, \Theta \vdash \mathrm{Valid}(\boldsymbol{C}) \qquad \mathcal{E}, \Theta, \boldsymbol{C}, (\boldsymbol{\varphi}, \boldsymbol{\psi}) \vdash \emptyset \rhd_{\mathcal{G}} \#(\vec{v_0}; \vec{v_1})}{\mathcal{E}, \Theta \vdash \vec{v_0} \sim \vec{v_1}}\ \textsc{Bideduce}$$

Proof. Let $\mathcal{E}$ be an environment, Θ a set of global formulas, and $\boldsymbol{\varphi}$ be a bi-assertion such that, for all $\mathbb{M} : \mathcal{E}$ satisfying Θ, for all $i \in \{0, 1\}$, η, ρ and $\mathfrak{p}$ such that ρ_a is a prefix of $\mathfrak{p}[\mathtt{T}_\mathtt{A} \times \mathsf{bool}]$, we have $\mathbb{M}, \eta, \rho, \mu^{i\ \eta,\mathfrak{p}}_{\mathsf{init}\mathbb{M}}(\mathcal{G}) \models^A \varphi_i$.

Let $\vec{v_0}$ and $\vec{v_1}$ be terms. Let assume that the two following judgements are valid :

$$\mathcal{E}, \Theta \vdash \mathrm{Valid}(\boldsymbol{C}), \tag{4.15}$$

$$\mathcal{E}, \Theta, \boldsymbol{C}, (\boldsymbol{\varphi}, \boldsymbol{\psi}) \vdash \emptyset \rhd_{\mathcal{G}} \#(\vec{v_0}; \vec{v_1}). \tag{4.16}$$

Let us show that the following is valid:

$$\mathcal{E}, \Theta \vdash \vec{v_0} \sim \vec{v_1}.$$

That is, let us show that for any model $\mathbb{M}$ with respect to $\mathcal{E}$

$$\mathbb{M} : \mathcal{E}, \Theta \models \vec{v_0} \sim \vec{v_1}$$

Then, let $\mathbb{M} : \mathcal{E}$ be a model satisfying Θ. Let $\mathcal{D}$ be a PPTM, and we must show that the following quantity is negligible in η:

$$\left| \Pr_{\rho \in \mathbb{T}_{\mathbb{M},\eta}} \left(\mathcal{D}(1^\eta, [\![\vec{v_0}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \rho_\mathsf{a}) = 1\right) - \Pr_{\rho \in \mathbb{T}_{\mathbb{M},\eta}} \left(\mathcal{D}(1^\eta, [\![\vec{v_1}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \rho_\mathsf{a}) = 1\right) \right|$$

We do this by cryptographic reduction to the indistinguishability of $\mathcal{G}$.

By validity of $\boldsymbol{C}$ given by Eq. (4.15), the judgement of Eq. (4.16) yields that there exists a program p and a polynomial P such that

(h1) for any tapes ρ $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ $\mathfrak{p}$, and for any $\eta \in \mathbb{N}$, $i \in \{0,1\}$, and for any μ such that $\mathbb{M}, \eta, \rho, \mu \models^{A} \varphi_i$, p has adversarial behaviour w.r.t. $\mathbb{M}, \eta, \mu, [\![\emptyset]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \mathfrak{p}$

(h2) C is well-formed w.r.t. $\mathbb{M}, \eta$ relatively to $\emptyset, \boldsymbol{\varphi}$, for any $\eta \in \mathbb{N}$,

(h3) for any tapes ρ $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ $\mathfrak{p}$, and for any $\eta \in \mathbb{N}$, $i \in \{0,1\}$, and for any μ such that $\mathbb{M}, \eta, \rho, \mu \models^{A} \varphi_i$, p computes $\emptyset \triangleright \boldsymbol{v}$ in time $P(\eta + | [\![\emptyset]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} |)$ w.r.t. $\mathbb{M}, \eta, \mathfrak{p}, \rho, \mu, i$.

We now construct a PTIME-adversary that wins against $\mathcal{G}$ with non-negligible probability. First, $\mathcal{D}$ is a PPTM, hence, by Turing completeness (see Section 3.3.3), let p' be a PTIME program that does the same computation as $\mathcal{D}$. Let $\mathsf{p}'' = \mathsf{p}; \mathsf{p}'$ assuming $\vec{X_{res}}$ are input variables for p', where $\vec{X_{res}}$ are the program variables storing p output, then for all $\mathfrak{p} \in \mathfrak{P}$, and $i \in \{0,1\}$

$$\mathcal{D}(1^{\eta}, (\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu_i}[\vec{\mathsf{res}}], \rho_{\mathsf{a}}) = (\!|\mathsf{p}''|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},i\mu_i}[\mathsf{res}]$$

Then, let $\rho, \mathfrak{p}$ be tapes such that ρ $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ $\mathfrak{p}$. Let $\mu_i = \mu^{i\ \eta,\mathfrak{p}}_{\mathsf{init}\mathbb{M}}$ for any $i \in \{0,1\}$, the initial memory of the game $\mathcal{G}$. By hypothesis, and definition of $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ we have $\mathbb{M}, \eta, \rho, \mu_i \models^{A} \varphi_i$, for $i \in \{0,1\}$. Hence, by (h3), we have, for any tapes ρ $\mathcal{R}^{\eta}_{C,\mathbb{M}}$ $\mathfrak{p}$,

$$(\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},0,\mu_0}[\vec{X_{res}}] = [\![\vec{v_0}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} \text{ and } (\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},1,\mu_1}[\vec{X_{res}}] = [\![\vec{v_1}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}.$$

and thus,

$$(\!|\mathsf{p}''|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},0,\mu_0}[\vec{\mathsf{res}}] = \mathcal{D}(1^{\eta}, [\![\vec{v_0}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \rho_{\mathsf{a}}) \text{ and } (\!|\mathsf{p}''|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},1,\mu_1}[\vec{\mathsf{res}}] = \mathcal{D}(1^{\eta}, [\![\vec{v_1}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \rho_{\mathsf{a}}). \tag{4.17}$$

Let $\tau_0, \cdots, \tau_k$ be the types of $\vec{v_0}$ (and $\vec{v_1}$), and for all security parameter η, let $\mathcal{T}^{\eta} = [\![\tau_0, \cdots, \tau_k]\!]^{\eta}_{\mathbb{M}}$.

Given ρ, let $\hat{\mathfrak{p}}$ be a program tape where ρ_a is a pre-fix of $\hat{\mathfrak{p}}[\mathsf{T_A} \times \mathsf{bool}]$ and all other bits of $\hat{\mathfrak{p}}$ are zeroes. Note that he program tape $\hat{\mathfrak{p}}$ and the memory $\mu^{i\ \eta,\hat{\mathfrak{p}}}_{\mathsf{init}}$ can be full computed by a function. We can then lift (h2) to the (un-relative) well-formedness of C by Lemma 3.

Then, by Lemma 1, we know that there exists a coupling $\mathbb{C} : \mathbb{T}_{\mathbb{M},\eta} \bowtie \mathfrak{P}$ contained in $\mathcal{R}^{\eta}_{C,\mathbb{M}}$.

Hence, using Eq. (4.17) and Lemma 2, we have that :

$$\Pr_{\rho \in \mathbb{T}_{\mathbb{M},\eta}}\left(\mathcal{D}(1^{\eta}, [\![\vec{v_0}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \rho_{\mathsf{a}}) = 1\right) = \Pr_{\mathfrak{p} \in \mathfrak{P}}\left((\!|\mathsf{p}''|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},0,\mu_0}[\vec{\mathsf{res}}], \rho_{\mathsf{a}}) = 1\right) \text{ and}$$
$$\Pr_{\rho \in \mathbb{T}_{\mathbb{M},\eta}}\left(\mathcal{D}(1^{\eta}, [\![\vec{v_1}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \rho_{\mathsf{a}}) = 1\right) = \Pr_{\mathfrak{p} \in \mathfrak{P}}\left((\!|\mathsf{p}''|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},1,\mu_1}[\vec{\mathsf{res}}], \rho_{\mathsf{a}}) = 1\right).$$

Hence, we have:

$$\left|\Pr_{\rho \in \mathbb{T}_{\mathbb{M},\eta}}\left(\mathcal{D}(1^{\eta}, [\![\vec{v_0}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \rho_{\mathsf{a}}) = 1\right) - \Pr_{\rho \in \mathbb{T}_{\mathbb{M},\eta}}\left(\mathcal{D}(1^{\eta}, [\![\vec{v_1}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \rho_{\mathsf{a}}) = 1\right)\right|$$
$$= \left|\Pr_{\mathfrak{p} \in \mathfrak{P}}\left((\!|\mathsf{p}''|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},0,\mu_0}[\vec{\mathsf{res}}], \rho_{\mathsf{a}}) = 1\right) - \Pr_{\mathfrak{p} \in \mathfrak{P}}\left((\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathbb{M},1;\mu_1}[\vec{\mathsf{res}}], \rho_{\mathsf{a}}) = 1\right)\right|.$$

It remains to show that p'' is a PTIME adversary.

First, notice that, through Lemma 1, we have that for every program tape $\mathfrak{p}$, there exists a logical tape ρ such that ρ $\mathcal{R}^{\eta}_{C_i,\mathbb{M}}$ $\mathfrak{p}$ for any side i. We can then lift the hypotheses

(h1) and (h3) to any tape $\mathfrak{p}$ and thus show that for every tape $\mathfrak{p}$ and side i, p has an adversarial behaviours w.r.t. $\mathbb{M}, \eta, \mu^{i\ \mathfrak{p},\eta}_{\mathsf{init}\mathbb{M}}, \mathfrak{p}$ and $C_{\mathbb{M}}(\mathsf{p}, \eta, \mu^{i\ \mathfrak{p},\eta}_{\mathsf{init}\mathbb{M}})$ in bounded by $P(\eta)$. Hence, p is a PTIME adversary against $\mathcal{G}$.

Furthermore, the program p' does not do any random samplings, except for the one in $\mathtt{T_A}$, and no oracles calls. Also, it is PTIME. Thus, the program p" is also an adversary against $\mathcal{G}$, by (h1). Hence, by cryptographic reduction to $\mathcal{G}$ the following quantity is negligible in η:

$$\left|\Pr_{\mathfrak{p}\in\mathfrak{P}}\left(\llparenthesis \mathsf{p}"\rrparenthesis^{\eta,\mathfrak{p}}_{\mathcal{G},\mathbb{M},0,\mu_0}[\mathsf{res}] = 1\right) - \Pr_{\mathfrak{p}\in\mathfrak{P}}\left(\llparenthesis \mathsf{p}"\rrparenthesis^{\eta,\mathfrak{p}}_{\mathcal{G},\mathbb{M},1,\mu_1}[\mathsf{res}] = 1\right)\right| \tag{4.18}$$

which ends the proof. □

Chapter 5

Bideduction Proof System

Contents

5.1 Overview **76**

5.2 Proof system **77**

5.2.1 Preliminary definitions 77

5.2.2 Inference rules 79

5.2.3 Example 82

5.3 Soundness **84**

5.3.1 Preliminary definitions 84

5.3.2 Validity and well-formedness lemmas 85

5.3.3 Memory flow lemmas 89

5.3.4 Computation lemmas 90

5.3.5 Adversary lemmas 90

5.3.6 Footprint lemma 91

5.3.7 Structural rules 92

5.3.8 Adversarial rules 96

5.3.9 Computational rules 99

The previous two chapters introduced the framework for games and adversaries, bideduction judgments and the BIDEDUCE inference rule. This chapter has two main objectives. First, it presents a proof system for deriving bideduction judgments. A key design goal of this system is composability — ensuring that proofs can be modularly combined. Second, the chapter establishes the soundness of the proof system, showing that any derivable judgment is valid. To help develop intuition for both the design of the proof system and the soundness results, the chapter begins with an overview following-up on the overviews' of Chapter 3 and Chapter 4.

5.1 Overview

The bideduction rules of our proof-system allow building simulators piece-by-piece in a compositional way, using the simulators provided by the premises of a rule as sub-procedures of the simulator being built to justify the rule's conclusion. To give some intuition about rules design, we come back to the rules introduced in Section 4.1.

The simplified rules presented in Chapter 4 mostly ignored constraints systems and pre- and post-conditions consideration.

We show in this overview how to enrich the rules with these features. Further, we will associate to each rule an informal program showing how the simulators provided by the premises are composed to obtain the simulator for the conclusion, to help build an intuition for the rule soundness.

Transitivity rule. The *transitivity rule* introduced in Chapter 4 was:

$$\frac{\vec{u} \rhd \vec{v} \qquad \vec{u}, \vec{v} \rhd \vec{w}}{\vec{u} \rhd \vec{v}, \vec{w}}$$

The simulators behind this rule is put in sequence the two simulators coming from the premises :

$$\underline{\mathcal{S}}(\vec{u}) := \begin{array}{l} \vec{v} \leftarrow \mathcal{S}_1(\vec{u}); \\ \vec{w} \leftarrow \mathcal{S}_2(\vec{u}, \vec{v}) \end{array}$$

To enrich this rules with pre- and post-conditions, we notice that the game state in which $\mathcal{S}_2$ executes is the state at the end of $\mathcal{S}_1$'s execution, and that the pre-condition (resp. post-condition) for $\mathcal{S}$ is the pre-condition of $\mathcal{S}_1$ (resp. post-condition of $\mathcal{S}_2$). Hence, in our judgement, the pre- and post-condition must be chained.

Also, the randomness used by $\mathcal{S}$ is the union of the randomness used by $\mathcal{S}_1$ and the randomness used by $\mathcal{S}_2$. We note $\mathcal{C}^1 \cdot \mathcal{C}^2$ the concatenation of the constraint system $\mathcal{C}_1$ with the constraint system $\mathcal{C}_2$, and generalize it to bi-constraint systems concatenation, where $\boldsymbol{C}^1 \cdot \boldsymbol{C}^2$ is the bi-constraint system where the right constraint system is the concatenation of the right constraint system of $\mathcal{C}_1$ and the right constraint system of $\mathcal{C}_2$ (and similarly on the left).

In conclusion, the rule is enriched like this:

$$\frac{\boldsymbol{C}^1, (\boldsymbol{\varphi}, \boldsymbol{\varphi}') \vdash \vec{\boldsymbol{u}} \rhd \vec{\boldsymbol{v}} \qquad \boldsymbol{C}^2, (\boldsymbol{\varphi}', \boldsymbol{\psi}) \vdash \vec{\boldsymbol{u}}, \vec{\boldsymbol{v}} \rhd \vec{\boldsymbol{w}}}{\boldsymbol{C}^1 \cdot \boldsymbol{C}^2, (\boldsymbol{\varphi}, \boldsymbol{\psi}) \vdash \vec{\boldsymbol{u}} \rhd \vec{\boldsymbol{v}}, \vec{\boldsymbol{w}}}$$

The name rule. The *name rule* was already enriched with the constraint system in Section 4.1. Also, the simulator captured by this rule only performs a sampling, which hence let the game state unchanged. So the enriched rule is:

$$\frac{}{\{(\mathsf{n}_i, \mathsf{T_S})\}, (\boldsymbol{\varphi}, \boldsymbol{\varphi}) \vdash \vec{\boldsymbol{u}} \rhd \mathsf{n}_i}$$

with a *simplified* version of constraint system. The following simulators justify the rule soundness:

$$\underline{\mathcal{S}}(\vec{u}) := \mathsf{x_n} \xleftarrow{\$} \mathsf{T_S}[\mathsf{offset_n(i)}]$$

The oracle rule. Consider the oracle rule for a hash oracle of game $\mathcal{G}$ as sketched in Chapter 4.

$$\frac{\vec{u} \rhd v}{\vec{u} \rhd \mathsf{h}(v, \mathsf{k_i})}$$

The underlying simulator is the following:

$$\underline{\mathcal{S}}(\vec{u}) := \begin{array}{l} v \leftarrow \mathcal{S}_1(\vec{u}); \\ \quad \mathsf{x_h} \leftarrow \mathsf{G.hash}(\vec{v})[\mathsf{offset_k(i)}] \end{array}$$

First, this rule must add new constraints. Indeed, we must register that $\mathsf{k_i}$ is the key of the game. So the rule becomes:

$$\frac{C \vdash \vec{u} \rhd v}{C \cdot \{(\mathsf{k_i}, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},k})\} \vdash \vec{u} \rhd \mathsf{h}(v, \mathsf{k_i})}$$

Now, we want to extend the rule to integrate pre- and post-conditions. Recall that, in the game memory, the list $\ell_{\mathbf{hash}}$ is updated when the hash oracle is called (Figure 3.2).

We must enrich the rule to enforce that, when starting from a memory respecting φ, the hash oracle on v leads to a memory respecting φ' where φ' ensures that v is in the list.

Since the pre-and post-conditions are left abstract in the proof system, we assume there exists a logic that can express such properties through *oracle triples*, a specific judgement capturing pre-and post conditions update during an oracle call.

More precisely, we assume the existence of a judgement

$$\{\varphi\} t \leftarrow O_{\mathbf{hash}}(v)[\mathsf{k_i}]\{\psi\}$$

that says that whenever the **hash** oracle is called with a state respecting φ, on input v, with key $\mathsf{k_i}$, it reaches a state respecting ψ and returns t — we leave the verification of such triples is left later.

Then, the oracle rule becomes:

$$\frac{C, (\varphi, \varphi') \vdash \vec{u} \rhd v \qquad \{\varphi'\} t \leftarrow O_{\mathbf{hash}}(v)[\mathsf{k_i}]\{\psi\}}{C, (\varphi, \psi) \cdot \{(\mathsf{k_i}, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},k})\} \vdash \vec{u} \rhd t}$$

5.2 Proof system

We now present the proof system we designed for bideduction. Our proof rules are guided by the structure of the term to be bideduced. This section provides the full table of inference rules, along with intuitions on the proofs of soundness for some key rules. The full proofs of soundness are postponed to Section 5.3.

5.2.1 Preliminary definitions

Conditional terms. To enable expressive rules, it is useful to consider vectors of *conditional* terms. Such a vector $\vec{t}$ is an element $((f_1, \mathsf{if}\ f_1\ \mathsf{then}\ t_1), \ldots, (f_n, \mathsf{if}\ f_n\ \mathsf{then}\ t_n))$[1] or more conveniently noted $((t_1 \mid f_1), \ldots, (t_n \mid f_n))$, where $f_1, t_1, \cdots f_n, t_n$ are terms.

[1] Here, $(\mathsf{if}\ f\ \mathsf{then}\ t)$ is syntactic sugar for $(\mathsf{if}\ f\ \mathsf{then}\ t\ \mathsf{else}\ \mathsf{witness}_\tau)$ where τ is the type of t and $\mathsf{witness}_\tau$ is an arbitrary symbol of type τ (whose existence is guaranteed, as all types are inhabited).

We will thus consider bideductions of the form $\vec{u} \triangleright \vec{t}$, with $\vec{t}$ a vector of conditional bi-terms.

Operation on constraints systems. We shall use two operations on constraint systems. First, the concatenation of bi-constraint systems is defined as $\#(C_0^1; C_1^1) \cdot \#(C_0^2; C_1^2) = \#(C_0^1 \cdot C_0^2; C_1^1 \cdot C_1^2)$. Second, we define the generalization $\prod x.\boldsymbol{C}$ of $\boldsymbol{C}$ over x as the system $\boldsymbol{C}$ where x is added to the vector of bound variables in all basic constraints of C_0 and C_1. The validity of $\prod x.\boldsymbol{C}$ implies that of $\boldsymbol{C}$.

Oracle Hoare triples. Oracle Hoare triples are used to capture the evolution of the memory during an oracle call. Going back to Section 5.1, we assumed the existence of a dedicated judgement. Still, the one presented earlier is incomplete. Indeed, as shown in Example 14 it is convenient to take into account the branching condition under which the oracle call is made.

Example 14. *Using the PRF game of Figure 3.2, we should have that:*

$$(\emptyset, \emptyset) \triangleright h(n, \mathsf{k}), \#(\mathsf{h}(m, \mathsf{k}); \mathsf{n}_{\mathsf{fresh}})$$

for any adversarial messages n, m such that n and m are always distinct, i.e. $[n \neq m]_{\mathrm{e}}$ (ignoring constraint systems and pre-and -post conditions for now).

If n and m are two names n *and* m*, we cannot guarantee that they are* always *distinct. However, we have the following:*

$$(\emptyset, \emptyset) \triangleright \mathsf{h}(\mathsf{n}, \mathsf{k}), \mathsf{if}\ \mathsf{n} \neq \mathsf{m}\ \mathsf{then}\ \#(\mathsf{h}(\mathsf{m}, \mathsf{k}); \mathsf{n}_{\mathsf{fresh}})$$

That is, the challenge oracle is only called in the then *branch when* $\mathsf{n} \neq \mathsf{m}$ *does hold. The ability to propagate information from term-level conditionals in oracle calls is crucial to verify such bideductions.*

We define the Oracle Hoare triples below.

Definition 24 (Oracle Hoare triples)**.** *An* oracle triple *for an oracle f, written*

$$\{\boldsymbol{\varphi}; \boldsymbol{b}\}\boldsymbol{v} \leftarrow O_f(\vec{t})[\vec{k}; \vec{r}]\{\boldsymbol{\psi}\}$$

is formed from: assertions $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ for the pre- and post-conditions, a boolean term $\boldsymbol{b}$, an output term $\boldsymbol{v}$, input terms $\vec{t}$, and terms $\vec{k}$ and $\vec{r}$ for the global and local randomness offsets of the oracle. We require that the offsets are of the form $\vec{k} = (\mathsf{k}_v\ o_{v\sharp})_{v \in f.\mathrm{glob}_\$}$ and $\vec{r} = (\mathsf{r}_v\ s_{v\sharp})_{v \in f.\mathrm{loc}_\$}$, where k_v and r_v are names.

An oracle triple must characterize correctly the execution of an oracle call in a program. We give here the definition of an oracle triple validity.

Definition 25 (Oracle Hoare triples validity)**.** *Consider an oracle triple for an oracle f:*

$$\{\boldsymbol{\varphi}; \boldsymbol{b}\}\boldsymbol{v} \leftarrow O_f(\vec{t})[\vec{k}; \vec{r}]\{\boldsymbol{\psi}\}$$

whose offsets are of the form

$$\vec{k} = (k_v\ o_{v\sharp})_{v \in f.\mathrm{glob}_\$} \qquad \text{and} \qquad \vec{r} = (r_v\ s_{v\sharp})_{v \in f.\mathrm{loc}_\$}.$$

This triple is valid, which we write:

$$\Theta \models \{\varphi; b\} v \leftarrow O_f(\vec{t})[\vec{k}; \vec{s}]\{\psi\},$$

when, for any $\mathbb{M}$ *such that* $\mathbb{M} \models \Theta$*, for any* η*,* ρ*,* μ*,* $i \in \{0, 1\}$*, and any fresh variable* X*, if* $\mathbb{M}, \eta, \rho, \mu \models^A \varphi_i$ *then*

- *if* $[\![b_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = 0$ *then* $\mathbb{M}, \eta, \rho, \mu \models^A \psi_i$ *else,*
- *if* $[\![b_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = 1$*, then* $\mathbb{M}, \eta, \rho, \mu' \models^A \psi_i$ *and* $[\![v_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = \mu'(X)[f.\mathrm{expr}]^{\eta,\mathfrak{p}}_{\mathbb{M}:\mathcal{E},i,\mu'}$*, where:*

$$\begin{aligned} \mu' &= (\!| X \leftarrow O_f([\![\vec{t}_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}})[\vec{e}_k ; \vec{e}_s] |\!)^{\eta,\mathfrak{p}}_{\mu} \\ \vec{e}_k &= \left(O_{\mathbb{M}:\mathcal{E},\eta}(k_v, [\![o_{v,i}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}})\right)_{v \in f.\mathrm{glob}_\$} \text{ and} \\ \vec{e}_s &= \left(O_{\mathbb{M}:\mathcal{E},\eta}(r_v, [\![s_{v,i}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}})\right)_{v \in f.\mathrm{loc}_\$} \end{aligned}$$

with $\mathfrak{p}$ *an arbitrary program tape s.t.* ρ_a *is a prefix of its adversarial tape.*

5.2.2 Inference rules

Summary. First, our proof system features rules for basic simulator constructions, like duping output element, **if then else** program instructions, etc. More interestingly, a central rule of our proof system features transitivity, which corresponds to composing simulators. Furthermore, in order to represent unbounded collections of objects to bideduce, we extend the bideduction judgement beyond terms of base type, allowing order-1 types when the argument types are enumerable — this is captured by the type restriction enum, see Section 2.1. This does not change the semantics of bideduction: we simply view these functions as an explicit representation of their graph. This extension notably brings rules to support λ-terms and induction though while loops.

Inference rules of our proof system, given in Figure 5.1, Figure 5.2 and Figure 5.3 are organized in three categories:

- First, the *structural* rules. This includes weakening rules (of hypotheses, pre- and post-conditions, constraints ...), re-ordering of the terms, and rewriting.
- Second, the *computational* rules. They capture the computations that do not require random samplings or oracle calls from the simulator. This comprises function applications, transitivity, computation of adversarial terms, conditional if then else , computing a function's graphs, and induction.
- Finally, *adversarial* rules capture adversarial capabilities: random samplings and oracle calls.

Structural Rules

The common point of all structural rules (Figure 5.1) is that they do not radically change the simulator obtained from the premise.

$$\textsc{Weak.Constr}\ \frac{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{w} \qquad \Theta \models C \subseteq C' \qquad \mathcal{E}, \Theta \models_{\mathsf{WF}(\vec{u}, \varphi)} C'}{\mathcal{E}, \Theta, C', (\varphi, \psi) \vdash \vec{u} \rhd \vec{w}}$$

$$\textsc{Weak.Cond}\ \frac{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd f, (v \mid f'), \vec{w} \qquad \mathcal{E}, \Theta \vdash [f \Rightarrow f']_{\mathrm{e}}}{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd (v \mid f), \vec{w}}$$

$$\textsc{Weak.Mem}\ \frac{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{v} \qquad \mathcal{E}, \Theta \models^A \varphi' \Rightarrow \varphi \qquad \mathcal{E}, \Theta \models^A \psi \Rightarrow \psi'}{\mathcal{E}, \Theta, C, (\varphi', \psi') \vdash \vec{u} \rhd \vec{v}}$$

$$\textsc{Weak.Hyps}\ \frac{\mathcal{E}, \Theta', C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{w} \qquad \Theta \models \Theta'}{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{w}}$$

$$\textsc{Refl}\ \frac{}{\mathcal{E}, \Theta, \emptyset, (\varphi, \varphi) \vdash \vec{u}, t \rhd t}$$

$$\textsc{Permute}\ \frac{\sigma, \sigma' \text{ are permutations} \qquad \mathcal{E}, \Theta, C, (\varphi, \psi) \vdash u^1, \cdots, u^m \rhd v^1, \ldots, v^n}{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash u^{\sigma(1)}, \ldots, u^{\sigma(m)} \rhd v^{\sigma'(1)}, \ldots, v^{\sigma'(n)}}$$

$$\textsc{Drop}\ \frac{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{v}, \vec{t}}{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{v}}$$

$$\textsc{Rewrite-L}\ \frac{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \#(\vec{w}_0; \vec{w}_1) \rhd v \qquad \mathcal{E}, \Theta \vdash [\vec{u}_0 = \vec{w}_0]_{\mathrm{e}} \tilde{\wedge} [\vec{u}_1 = \vec{w}_1]_{\mathrm{e}}}{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \#(\vec{u}_0; \vec{u}_1) \rhd v}$$

$$\textsc{Rewrite-R}\ \frac{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \#(\vec{w}_0; \vec{w}_1) \qquad \mathcal{E}, \Theta \vdash [\vec{v}_0 = \vec{w}_1]_{\mathrm{e}} \tilde{\wedge} [\vec{v}_0 = \vec{w}_1]_{\mathrm{e}}}{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \#(\vec{v}_0; \vec{v}_1)}$$

$$\textsc{Dup}\ \frac{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{v}, \vec{t}}{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{v}, \vec{t}, \vec{t}}$$

Figure 5.1: Structural bideduction rules

First, notice that a simulator that computes a term t, also computes any term t' exactly equal to t'. This also holds for input terms: using a term u or a term u' exactly equal does not change the simulator's result. This is captured by rules Rewrite-L and Rewrite-R.

Rule Drop holds because, given a simulator corresponding to the premise, we obtain a simulator for the conclusion by executing the premise simulation and then dropping some of its outputs. Similarly, Permute corresponds to re-ordering inputs and outputs, Refl to copying an input, Dup to duplicating an output.

Then, we have four weakening rules. The rule Weak.Hyps and Weak.Mem for hypothesis and pre- and post-conditions weakening, designed as expected. The rule Weak.Constr for constraints weakening on the same ideas, based on the previous remark in Chapter 4 that when $C \subseteq C'$ then $\mathrm{Valid}(C') \Rightarrow \mathrm{Valid}(C)$. Finally, the rule Weak.Cond weaken the local formula attached to term. For any term $(t \mid f)$ and a formula f' such that $[f \Rightarrow f']_{\mathrm{e}}$ then an adversary that computes $(v \mid f')$ and f can also compute $(v \mid f)$: intuitively, such adversary compute v more "often" than when f is true and at least every time that f is true.

$$\textsc{Library}\quad \frac{t \in \mathcal{L}_{\mathsf{p}},}{\mathcal{E}, \Theta, \emptyset, (\varphi, \varphi) \vdash \vec{u} \rhd t} \qquad \textsc{FA}\quad \frac{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{v}, (t^1 \mid f), \ldots, (t^n \mid f) \quad g \in \mathcal{L}_{\mathsf{p}}}{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{v}, (g\ t^1\ \ldots\ t^n \mid f)}$$

$$\textsc{If-then-else}\quad \frac{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{v}, (b \mid f), (t \mid f \wedge b), (t' \mid f \wedge \neg b)}{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd \vec{v}, (\mathsf{if}\ b\ \mathsf{then}\ t\ \mathsf{else}\ t' \mid f)} \qquad \textsc{Transitivity}\quad \frac{\mathcal{E}, \Theta, C^1, (\varphi, \varphi') \vdash \vec{u} \rhd \vec{t} \quad \mathcal{E}, \Theta, C^2, (\varphi', \psi) \vdash \vec{u}, \vec{t} \rhd \vec{v}}{\mathcal{E}, \Theta, C^1 \cdot C^2, (\varphi, \psi) \vdash \vec{u} \rhd \vec{t}, \vec{v}}$$

$$\textsc{Lambda-App}\quad \frac{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd (g, t \mid f) \quad \mathcal{E} \vdash \vec{t} : \tau \qquad \mathsf{enum}(\tau)}{\mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \rhd (g\ t \mid f)} \qquad \textsc{Lambda}\quad \frac{(\mathcal{E}, x : \tau), \Theta, C, (\varphi, \varphi) \vdash \vec{u}, x \rhd (t \mid f) \quad \mathcal{E}, x : \tau \vdash t : \tau_{\mathsf{b}} \qquad \tau_{\mathsf{b}} \in \mathbb{B} \qquad \mathsf{enum}(\tau)}{\mathcal{E}, \Theta, \prod_{(x:\tau)} . C, (\varphi, \varphi) \vdash \vec{u} \rhd (\lambda(x : \tau).t \mid f)}$$

$$\textsc{Quantificator-}O \in \{\forall, \exists\}\quad \frac{(\mathcal{E}, x : \tau), \Theta, C, (\varphi, \varphi) \vdash \vec{u}, x \rhd (t \mid f) \qquad \mathsf{enum}(\tau)}{\mathcal{E}, \Theta, \prod_{(x:\tau)} C, (\varphi, \varphi) \vdash \vec{u} \rhd (O(x : \tau).t \mid f)}$$

$$\textsc{Induction}\quad \frac{\begin{array}{c}\mathcal{E}, \Theta, C^0, (\varphi, \mathcal{I}_<(x_0)) \vdash \vec{u} \rhd (t \mid f) \\ (\mathcal{E}, x : \tau_{\mathsf{b}}), \Theta, C, (\mathcal{I}_<(x), \mathcal{I}_\leq(x)) \vdash \vec{u}, x, (\lambda(y : \tau_{\mathsf{b}}).(v\ y \mid y < x \wedge x \leq t) \mid f) \rhd (v\ x \mid f \wedge x \leq t) \\ (\mathcal{E}, x : \tau_{\mathsf{b}}), \Theta[x_0 < x]_{\mathsf{e}}[x \leq t]_{\mathsf{e}} \models^A \mathcal{I}_\leq(\mathsf{pred}(x)) \Rightarrow \mathcal{I}_<(x) \\ \mathsf{finite}(\tau_{\mathsf{b}}) \quad \mathsf{fixed}(\tau_{\mathsf{b}}) \quad \mathcal{E}, \Theta \vdash \mathsf{well\text{-}founded}_{\tau_{\mathsf{b}}}(<) <\in \mathcal{L}_{\mathsf{p}}\end{array}}{\mathcal{E}, \Theta, C^0 \cdot \prod_{(x:\tau_{\mathsf{b}})} C, (\varphi, \mathcal{I}_\leq(t)) \vdash \vec{u} \rhd (v\ t \mid f)}$$

Figure 5.2: Computational bideduction rules

Computational Rules

We call computational rules the rules that do not require random samplings or oracle calls from the simulator. The are describes in Figure 5.2. They include:

- the rule Library that builds a program that compute a library term (which is immediately an adversary);
- the rules that compute functions (FA for library function, Lambda-App for computed function, If-then-else for specific handling of if then else);
- the rules that chain programs: either with sequence of two programs (Transitivity) or under while loops (Lambda to compute a function graph, Quantificator-$O \in \{\forall, \exists\}$ for specific handling of quantifiers, Induction to compute recursively a function graph). These rules must ensure that the final program is polynomial. This is done through restrictions on the size of the while loop of the underlying simulator (we restrict types of lambda quantifier to be over enumerable types or fixed and finite types for induction).

$$\textsc{Name}\quad \frac{\mathcal{E}, \Theta, \boldsymbol{C}, (\boldsymbol{\varphi}, \boldsymbol{\psi}) \vdash \vec{\boldsymbol{u}} \rhd (\boldsymbol{t} \mid \boldsymbol{f})}{\mathcal{E}, \Theta, \boldsymbol{C} \cdot \{(\emptyset, n, \boldsymbol{t}, \mathtt{T}_S, \boldsymbol{f})\}, (\boldsymbol{\varphi}, \boldsymbol{\psi}) \vdash \vec{\boldsymbol{u}} \rhd (n\ \boldsymbol{t} \mid \boldsymbol{f})}$$

$$\textsc{Oracle}_f\quad \frac{\begin{array}{c}\mathcal{E}, \Theta, \boldsymbol{C}, (\boldsymbol{\varphi}, \boldsymbol{\psi}) \vdash \vec{\boldsymbol{u}} \rhd_{\mathcal{G}} \vec{\boldsymbol{w}}, (\vec{\boldsymbol{t}} \mid \boldsymbol{F}), (\vec{\boldsymbol{o}} \mid \boldsymbol{F}), (\vec{\boldsymbol{s}} \mid \boldsymbol{F}) \\ \Theta \models \{\boldsymbol{\psi}; \boldsymbol{F}\}\boldsymbol{v} \leftarrow O_f(\vec{\boldsymbol{t}})[\vec{\boldsymbol{k}}; \vec{\boldsymbol{r}}]\{\boldsymbol{\theta}\}\end{array}}{\mathcal{E}, \Theta, \boldsymbol{C}', (\boldsymbol{\varphi}, \boldsymbol{\theta}) \vdash \vec{\boldsymbol{u}} \rhd_{\mathcal{G}} \vec{\boldsymbol{w}}, (\boldsymbol{v} \mid \boldsymbol{F})}$$

$$\text{with } \boldsymbol{C}' = \boldsymbol{C} \cdot \prod_{v \in f.\mathrm{glob}_\$} (\emptyset, \mathsf{k}_v, o_{v\sharp}, \mathtt{T}^{\mathsf{glob}}_{\mathsf{G},v}, \boldsymbol{F}) \cdot \prod_{v \in f.\mathrm{loc}_\$} (\emptyset, \mathsf{r}_v, s_{v\sharp}, \mathtt{T}^{\mathsf{loc}}_{\mathsf{G}}, \boldsymbol{F});$$
$$\vec{\boldsymbol{o}} = (o_{v\sharp})_{v \in f.\mathrm{glob}_\$} \text{ and } \vec{\boldsymbol{s}} = (s_{v\sharp})_{v \in f.\mathrm{loc}_\$}$$

Figure 5.3: Adversarial bideduction rules

The rule Induction might be difficult to parse, so let us unpack its definition more precisely here. Roughly, it states that to bideduce $(\boldsymbol{v}\ \boldsymbol{t})$, it is sufficient to bideduce $\boldsymbol{t}$ and then $(\boldsymbol{v}\ x)$ for any $x \leq \boldsymbol{t}$, assuming by induction that we already computed $(\boldsymbol{v}\ y)$ for any $y < x$. The rule assumes the existence of $\boldsymbol{\mathcal{I}}$ a *memory invariant* describing the evolution of the game's state during the recursive evaluation of $\boldsymbol{v}$. Concretely, $\boldsymbol{\mathcal{I}}_{\bowtie}(x)$ is an abstract memory parameterized by a relation $\bowtie$ over τ and a value x of type τ_{b}, where $\boldsymbol{\mathcal{I}}_{<}(x)$ and $\boldsymbol{\mathcal{I}}_{\leq}(x)$ represents the memory resp. *before* and *after* the deduction of $\boldsymbol{v}\ x$. The premise $\boldsymbol{\mathcal{I}}_{\leq}(\mathsf{pred}(t)) \Rightarrow \boldsymbol{\mathcal{I}}_{<}(x)$ ensures that the assertion are chained correctly, that is the post-condition of one deduction implies the pre-condition of the following deduction. The conclusion of the induction rule states that at the end of the recursive computation, the memory satisfies $\boldsymbol{\mathcal{I}}$ up-to t.

Adversarial Rules

Finally, there are two adversarial rules, described in Figure 5.3, which captures two specific capabilities of adversaries: Name corresponds to random samplings; and, $\textsc{Oracle}_f$ to oracle calls.

In particular, the oracle rule soundness relies on the validity of the oracle triple. More precisely, from the bideduction premise, we get a simulator p that computes inputs and offsets for the oracle call. The final simulator p' is p followed by an oracle call. The new program is polynomial if p is. Furthermore, the validity of $\boldsymbol{C}$ ensures the freshness of local offsets and uniqueness of global offsets. The equality between the result of p' and the semantics of the output terms follows from the validity of the oracle triple.

5.2.3 Example

We illustrate how our proof system operates using examples. We present an example that encapsulates keys steps in the complete proof for the example introduced in Section 3.1, In Example 15, we show how one can prove the indistinguishability of a vector of three

non-recursive terms. This example illustrate the oracle calls rules, for the PRF game of Figure 3.2.

Example 15. *We are going to show the indistinguishability*

$$\mathsf{h}(\mathsf{n},\mathsf{k}),\mathsf{h}(\mathsf{m},\mathsf{s}),\mathsf{h}(\mathsf{m},\mathsf{k}) \sim \mathsf{h}(\mathsf{n},\mathsf{k}),\mathsf{h}(\mathsf{m},\mathsf{s}),\mathsf{n}_{\mathsf{fresh}} \tag{5.1}$$

using bideduction w.r.t. the PRF game of Figure 3.2, where $\mathsf{n},\mathsf{k},\mathsf{m},\mathsf{s}$ *are distinct names of type* τ*. We are going to show that:*

1. *the first and last terms can be computed using oracle calls;*
2. *and that the middle term is just a function application.*

For Item 2, we require that h *is an adversarial function symbol. To be able to carry out the simulation strategy of Item 1, we need to ensure that* $\mathsf{n} \neq \mathsf{m}$*, which we do using the trick of Example 14. First, we assume that the type* τ *is a* large type*: essentially, this means that independent names with output type* τ *have a negligible probability of collision. This assumption is captured by the* $\mathsf{large}(\tau)$ *hypothesis introduced in Chapter 2. Under this hypothesis, it can be shown that* $\mathsf{large}(\tau) \models [\mathsf{n} \neq \mathsf{m}]$ *and thus, using the rewriting rule of [29], that the indistinguishability in Eq. (5.1) is implied by the formula:*

$$\begin{aligned} &\mathsf{h}(\mathsf{n},\mathsf{k}),\mathsf{h}(\mathsf{m},\mathsf{s}),\mathsf{if}\ \mathsf{n}\neq\mathsf{m}\ \mathsf{then}\ \mathsf{h}(\mathsf{m},\mathsf{k}) \\ \sim\ &\mathsf{h}(\mathsf{n},\mathsf{k}),\mathsf{h}(\mathsf{m},\mathsf{s}),\mathsf{if}\ \mathsf{n}\neq\mathsf{m}\ \mathsf{then}\ \mathsf{n}_{\mathsf{fresh}} \end{aligned}$$

Let's take the hypotheses $\Theta = \mathsf{adv}(\mathsf{h}),\mathsf{large}(\tau)$*.*

For the sake of simplicity, we instantiate assertions by sets of memories: thus, roughly, satisfiability is set membership and implication is set inclusion. Consider the following assertions:

$$\begin{aligned} \varphi_0 &= \{(l_{\mathit{hash}} \mapsto [] \ ; l_{\mathit{challenge}} \mapsto [])\} \\ \varphi &= \{(l_{\mathit{hash}} \mapsto [\mathsf{n}] \ ; l_{\mathit{challenge}} \mapsto [])\} \end{aligned}$$

where $[]$ *is the empty list and* $[\mathsf{n}]$ *is the list containing a single element* n*. We have:*

$$\Theta \models \{\varphi_0\}\,\mathsf{h}(\mathsf{n},\mathsf{k}) \leftarrow O_{\mathit{hash}}(\mathsf{n})[\mathsf{k};.]\,\{\varphi\},$$

and, using Name *to compute* n*, we get the bideduction judgment:*

$$\Theta, \big((\mathsf{n},\mathtt{T}_S,\top),(\mathsf{k},\mathtt{T}^{\mathsf{glob}}_{\mathsf{G},\mathsf{k}},\top)\big),(\varphi_0,\varphi) \vdash \emptyset \triangleright \mathsf{h}(\mathsf{n},\mathsf{k}).$$

All the name constraints in this example have no bound variables and are for names $\mathsf{n},\mathsf{k},\mathsf{s},\mathsf{m}$ *which are not indexed. Thus, we use a shorter syntax for constraints, and write* $(\mathsf{name},\mathtt{T},f)$ *instead of* $(\emptyset,\mathsf{name},\emptyset,\mathtt{T},f)$*.*

Then, using Name*,* Dup *and* FA*, we also get that:*

$$\Theta, \big((\mathsf{m},\mathtt{T}_S,\top),(\mathsf{s},\mathtt{T}_S,\top)\big),(\varphi,\varphi) \vdash \emptyset \triangleright \mathsf{h}(\mathsf{m},\mathsf{s}).$$

Finally, we have:

$$\{\varphi \wedge \mathsf{n}\neq\mathsf{m}\}\ \ \#(\mathsf{h}(\mathsf{m},\mathsf{k});\mathsf{n}_{\mathsf{fresh}}) \leftarrow O_{\mathit{challenge}}(\mathsf{m})[\mathsf{k};\mathsf{n}_{\mathsf{fresh}}]\ \ \{\psi\}$$

for a certain ψ. As before, using the rules If-then-else, Oracle$_f$ *for* $f =$ ***challenge*** *and* Name*, we have that:*

$$\Theta, \big((\mathsf{n}, \mathsf{T}_S, \top), (\mathsf{m}, \mathsf{T}_S, \top), (\mathsf{m}, \mathsf{T}_S, \mathsf{n} \neq \mathsf{m}) \\ (\mathsf{k}, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},\mathsf{k}}, \mathsf{n} \neq \mathsf{m}), (\mathsf{n}_{\mathsf{fresh}}, \mathsf{T}^{\mathsf{loc}}_{\mathsf{G}}, \mathsf{n} \neq \mathsf{m})\big), (\varphi, \psi) \vdash \\ \emptyset \rhd \mathsf{if}\ \mathsf{n} \neq \mathsf{m}\ \mathsf{then}\ \#(\mathsf{h}(\mathsf{m}, \mathsf{k}); \mathsf{n}_{\mathsf{fresh}}).$$

By transitivity, we get the final judgement:

$$\mathcal{E}, \Theta, \mathcal{C}, (\varphi_0, \psi) \vdash \emptyset \rhd \mathsf{h}(\mathsf{n}, \mathsf{k}), \mathsf{h}(\mathsf{m}, \mathsf{s}), \mathsf{if}\ \mathsf{n} \neq \mathsf{m}\ \mathsf{then}\ \#(\mathsf{h}(\mathsf{m}, \mathsf{k}); \mathsf{n}_{\mathsf{fresh}})$$

where $\mathcal{C}$ is the following constraint system:

$$\{\ (\mathsf{n}, \mathsf{T}_S, \top) \quad , (\mathsf{k}, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},\mathsf{k}}, \top) \quad , (\mathsf{m}, \mathsf{T}_S, \top), \\ (\mathsf{s}, \mathsf{T}_S, \top) \quad , (\mathsf{n}, \mathsf{T}_S, \top) \quad , (\mathsf{m}, \mathsf{T}_S, \top), \\ (\mathsf{m}, \mathsf{T}_S, \mathsf{n} \neq \mathsf{m}), (\mathsf{k}, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},\mathsf{k}}, \mathsf{n} \neq \mathsf{m}), (\mathsf{n}_{\mathsf{fresh}}, \mathsf{T}^{\mathsf{loc}}_{\mathsf{G}}, \mathsf{n} \neq \mathsf{m})\ \}$$

Then Valid($\mathcal{C}$) *is easily verified, and the proof is done.*

5.3 Soundness

Theorem 2 states that our proof system is sound.

Theorem 2 (Proof system soundness)**.** *Any bideduction judgement derivable by the proof system is valid.*

In this section, you will find preliminary definitions and lemmas, followed by the soundness proofs for each inference rules.

For the rest of this section, we fix an arbitrary library model $\mathbb{M}_0$, an arbitrary game $\mathcal{G}$, and an arbitrary environment $\mathcal{E}$. All mention of models $\mathbb{M}$ will implicitly be with respect to $\mathcal{E}$, extending $\mathbb{M}_0$.

5.3.1 Preliminary definitions

For the sake of readability of the proofs, it is useful to decompose bideduction semantics into simpler properties. This section provides these simpler properties and the Theorem 3, which provides an equivalent characterization of bideduction judgement validity, that will be the one used for all soundness proofs afterward.

Definition 26 (Relative PTIME adversary)**.** *We way that a program* p *is a (relative) PTIME adversary w.r.t.* $\mathbb{M}, \eta, \boldsymbol{\mathcal{C}}, \boldsymbol{\varphi}, \vec{\boldsymbol{u}}$ *when there exists a polynomial P such that for any $i \in \{0, 1\}$, for any tapes ρ $\mathcal{R}^{\eta}_{C_i,\mathbb{M}}$ $\mathfrak{p}$, and any μ such that $\mathbb{M}, \eta, \rho, \mu \models^{A} \varphi_i$,* p *has an adversarial behaviour w.r.t. $\mathbb{M}, \eta, \mu, [\![\vec{u}_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \mathfrak{p}$, and the computation of* p *on input $[\![\vec{u}_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$ is bounded by $P(\eta+ \mid [\![\vec{u}_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} \mid)$*

Definition 27 (Memory flow)**.** *We say that* p flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\psi}$ *with respect to $\mathbb{M} : \mathcal{E}, \boldsymbol{\mathcal{C}}, \vec{\boldsymbol{u}}$ when for all $i \in \{0, 1\}$, for all pairs of tape ρ $\mathcal{R}^{\eta}_{C_i,\mathbb{M}}$ $\mathfrak{p}$, and memory μ such that $\mathbb{M}, \eta, \rho, \mu \models \varphi_i$, then $\mathbb{M}, \eta, \rho, \mu' \models \psi_i$, where $\mu' = (\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathbb{M}:\mathcal{E},i,\mu[\vec{X} \mapsto [\![\vec{u}_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}]}$.*

Definition 28 (Computation)**.** *We say that* p computes $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}$ *with respect to* $\mathbb{M} : \mathcal{E}, \boldsymbol{C}, \boldsymbol{\varphi}$ *when for all* $i \in \{0,1\}$*, for all pairs of tape* $\rho\ \mathcal{R}^{\eta}_{C_i,\mathbb{M}}\ \mathfrak{p}$*, and memory* μ *such that* $\mathbb{M}, \eta, \rho, \mu \models \varphi_i$*, then* p *computes* $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}$ *with respect to* $\mathbb{M}, \eta, \mathfrak{p}, \rho, \mu, i$ *in any time.*

Definition 29 (Randomness footprint)**.** *We say that* $\boldsymbol{C}$ captures p randomness *with respect to* $\mathbb{M} : \mathcal{E}, \boldsymbol{\varphi}, \vec{\boldsymbol{u}}$*, when for all* $i \in \{0,1\}$*, all memory* μ *such that* $\mathbb{M}, \eta, \rho, \mu \models \varphi_i$*, and all tapes* $\rho\ \mathcal{R}^{\eta}_{C_i,\mathbb{M}}\ \mathfrak{p}$*, the computations of* $(\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathbb{M}:\mathcal{E},i,\mu[\vec{X}\mapsto[\![\vec{u}_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}]}$ *relies on global samplings* $G_{\$}$ *and local samplings* $L_{\$}$ *and*

$$\begin{aligned} G_{\$} &\subseteq \{\ O_{\mathbb{M},\eta}(\mathsf{n}, a) \mid \langle \mathsf{n}, a, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},v} \rangle \in \mathcal{N}^{\eta,\rho}_{c,\mathbb{M}},\ c \in C_i\ \} \\ L_{\$} &\subseteq \{\ O_{\mathbb{M},\eta}(\mathsf{n}, a) \mid \langle \mathsf{n}, a, \mathsf{T}^{\mathsf{loc}}_{\mathsf{G}}\ \rangle \in \mathcal{N}^{\eta,\rho}_{c,\mathbb{M}},\ c \in C_i\ \} \end{aligned}$$

When it is clear from context, the elements such as model, side constraint systems, etc. will be omitted. With the above definitions, the validity of a bideduction judgement can be re-written as follows.

Theorem 3 (Bideduction judgement validity characterization)**.** *The bideduction judgement*

$$\mathcal{E}, \Theta, \boldsymbol{C}, (\boldsymbol{\varphi}, \boldsymbol{\psi}) \vdash \vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}} \tag{5.2}$$

is valid in $\mathbb{M}_0$ *if and only if there exists a program* p *such that for all* $\mathbb{M}$ *extending* $\mathbb{M}_0$ *such that* $\mathbb{M} \models \Theta \tilde{\wedge} \mathrm{Valid}(\boldsymbol{C})$*,*

1. p *is a PTIME adversary against* $\mathcal{G}$*;*
2. $\boldsymbol{C}$ *is well-formed relatively to* $\vec{\boldsymbol{u}}, \boldsymbol{\varphi}$*.*
3. p *computes* $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}$*.*
4. p *flows from* $\boldsymbol{\varphi}$ *to* $\boldsymbol{\psi}$*; and*
5. $\boldsymbol{C}$ *captures* p *randomness.*

In that case, we say that p witnesses the validity *of Eq. (5.2).*
Finally, Eq. (5.2) is valid if it is valid for any library model.

The following definition identifies classes of programs for which the properties of this section holds (almost) trivially.

Definition 30 (Basic program)**.** *We call a basic program a program that does no oracle calls, does not access the special variable* b *nor game variables, and samples only in the tape labelled* $\mathtt{T}_{\mathtt{A}} \times \mathsf{bool}$*.*

5.3.2 Validity and well-formedness lemmas

First, let us start with lemmas to propagate validity and well-formedness properties.

Lemma 4 (Constraint inclusion)**.** *Let* C *and* C' *be two constraint system such that* $C \subseteq C'$*. Then for any model* $\mathbb{M}$*,*

1. $\mathrm{Valid}(\mathcal{C}') \models \mathrm{Valid}(\mathcal{C})$,
2. *for any random tape* ρ : $\mathcal{N}_{\mathcal{C}}^{\eta,\rho} \subseteq \mathcal{N}_{\mathcal{C}'}^{\eta,\rho}$
3. $\mathcal{R}_{\mathcal{C}',\mathbb{M}}^{\eta} \subseteq \mathcal{R}_{\mathcal{C},\mathbb{M}}^{\eta}$, *and*

Proof. The validity of $\mathrm{Valid}(\mathcal{C}') \models \mathrm{Valid}(\mathcal{C})$ directly comes from the fact that for any constraint c_1 and c_2 in $\mathcal{C}$, we have that c_1 and c_2 are also in $\mathcal{C}'$, and by validity of $\mathcal{C}'$, we get $[\mathrm{Fun}(c_1, c_2) \wedge \mathrm{Fresh}(c_1, c_2) \wedge \mathrm{Unique}(c_1, c_2)]_{\mathrm{e}}$. The fact (2) is immediate with $\mathcal{N}_{\mathcal{C}}^{\eta,\rho}$ and $\mathcal{N}_{\mathcal{C}'}^{\eta,\rho}$'s definitions and the fact (3) directly follows from the fact 2), that is $\mathcal{N}_{\mathcal{C}}^{\eta,\rho} \subseteq \mathcal{N}_{\mathcal{C}'}^{\eta,\rho}$. □

Similarly, we have the following lemma

Lemma 5 (Cosntraints composition)**.** *For all* $\mathbb{M}$ *and constraints system* $\mathcal{C}^1$ *and* $\mathcal{C}^2$*, we have:*

- $\mathrm{Valid}(\mathcal{C}^1 \cdot \mathcal{C}^2) \models \mathrm{Valid}(\mathcal{C}^1) \wedge \mathrm{Valid}(\mathcal{C}^2)$,
- *for all* $j \in \{1, 2\}$, $\mathcal{R}_{\mathcal{C}^1 \cdot \mathcal{C}^2,\mathbb{M}}^{\eta} \subseteq \mathcal{R}_{\mathcal{C}^j,\mathbb{M}}^{\eta}$, *and*
- *for all* $j \in \{1, 2\}$ *and random tape* ρ : $\mathcal{N}_{\mathcal{C}^j,\mathbb{M}}^{\eta,\rho} \subseteq \mathcal{N}_{\mathcal{C}^1 \cdot \mathcal{C}^2,\mathbb{M}}^{\eta,\rho}$

We have similar properties for constraint generalization:

- $\mathrm{Valid}(\prod(x : \tau).\, \mathcal{C}) \models \tilde{\forall}(x : \tau).\, \mathrm{Valid}(\mathcal{C})$
- *for all* $a \in [\![\tau]\!]_{\mathbb{M}}^{\eta}$, $\mathcal{R}_{\prod(x:\tau).\mathcal{C},\mathbb{M}}^{\eta} \subseteq \mathcal{R}_{\mathcal{C},\mathbb{M}[x \mapsto a]}^{\eta}$
- *for all* $a \in [\![\tau]\!]_{\mathbb{M}}^{\eta}$ *and random tape* ρ*:*

$$\mathcal{N}_{\mathcal{C},\mathbb{M}[x \mapsto a]}^{\eta,\rho} \subseteq \mathcal{N}_{\prod(x:\tau).\mathcal{C},\mathbb{M}}^{\eta,\rho}$$

The proof is similar to the proof of Lemma 4.

We have similar results for well-formedness properties. In order to prove such properties, we need to be able to compose the inherent functions coming from the well-formedness definition of different constraints systems, and, as such, we need helper functions. Especially, we need functions to restrict random tapes (see Lemma 6 below).

For the rest of this subsection, we fix an arbitrary set of formulae Θ. All mention of models $\mathbb{M}$ will implicitly be such that $\mathbb{M} : \mathcal{E} \models \Theta$.

The following lemma enable to re-restrict random tape, which is the key lemma for the two main lemmas of this subsection: Lemma 7 and Lemma 9.

Lemma 6. *For any model* $\mathbb{M}$ *and security parameter* η*, for any vector of terms* $\vec{u}$ *and assertion* φ *for any sequences of constraint instances* l *and* l' *well-formed relatively to* $\vec{u}, \varphi$ *such that* $l' \subseteq l$*, there exists a function* g *such that, for all random tape* ρ *and memory* μ *such that* $\mathbb{M} : \mathcal{E}, \eta, \rho, \mu \models \varphi$*,*

$$g(\rho_{|\mathbb{M},\eta,l}, [\![\vec{u}]\!]_{\mathbb{M}:\mathcal{E}}^{\eta,\rho}, \mu) = \rho_{|\mathbb{M},\eta,l'}.$$

Proof sketch. The offsets used by l are all computed step by step using well-formedness propert, then we keep only the offsets share by l' and zeroes all the other offset in ρ. □

Next, we introduce the lemmas allowing to compose constraints systems.

Lemma 7. *For any term $\vec{u}$ and $\vec{v}$, assertions φ^1 and φ^2, for all constraints system C^1 and C^2 such that*

$$\mathcal{E}, \Theta \models_{WF(\vec{u}, \varphi^1)} C^1 \quad \text{and} \quad \mathcal{E}, \Theta \models_{WF(\vec{u}, \vec{v}, \varphi^2)} C^2$$

then whenever there exists a function s such that for all logical random tape ρ, memory μ such that $\mathbb{M} : \mathcal{E}, \eta, \rho, \mu \models \varphi^1$, and l_{C^1} witnessing the well-formedness of C^1,

$$s(\rho_{|\mathbb{M},\eta,l_{C^1}}, [\![\vec{u}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \mu) = \mu', [\![\vec{v}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$$

with $\mathbb{M} : \mathcal{E}, \eta, \rho, \mu' \models \varphi^2$ then

$$\mathcal{E}, \Theta \models_{WF(\vec{u}, \varphi^1)} C^1 \cdot C^2.$$

Proof. Let $\mathbb{M}$ be a model. By hypothesis, C^1 is well-formed w.r.t. $\mathbb{M}, \eta$ relatively to $\vec{u}, \varphi^1$. Let l_{C^1} be a sequence of constraint instances witnessing its well-formedness relatively to $\vec{u}, \varphi^1$. Similarly, let l_{C^2} be the sequence of constraint instances witnessing the well-formedness of C^2 relatively to $\vec{u}, \vec{v}, \varphi^2$.

Then, let us show that the concatenation of the two sequences $l_{C^1} \cdot l_{C^2}$ witnesses the well-formedness of $C^1 \cdot C^2$.

Let

$$l_{C^1} \cdot l_{C^2} = \big((\vec{a}_1, c_1), \ldots, (\vec{a}_K, c_K)\big)$$

and $(\vec{a}_k, c_k) = (\vec{a}_k, (\vec{\alpha}, n, t, T, f))$ be the k^{th} element of $l_{C^1} \cdot l_{C^2}$.

We must show that:

$$\exists g, \forall \rho, \mu \text{ such that } \mathbb{M} : \mathcal{E}, \eta, \rho, \mu \models \varphi^1 \quad \text{then } g(\rho_{|\mathbb{M},\eta,l^k}, [\![\vec{u}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \mu) = [\![(t \mid f)]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha} \mapsto \mathbb{1}^{\eta}_{\vec{a}_k}]:\mathcal{E}} \tag{5.3}$$

with $l^k = ((\vec{a}_1, c_1), \ldots, (\vec{a_{k-1}}, c_{k-1}))$.

If $(\vec{a}_k, c_k)$ is an element of l_{C^1}, then this is immediate by well-formedness of C^1. Thus, assume that $(\vec{a}_k, c_k)$ is an element of l_{C^2}. Let then K^1 be the length of l_{C^1},

Let ρ be an arbitrary random tape, and μ a memory such that $\mathbb{M} : \mathcal{E}, \eta, \rho, \mu \models \varphi^1$. Furthermore, let μ' be the memory output by s, i.e. such that

$$s(\rho_{|\mathbb{M},\eta,l_{C^1}}, [\![\vec{u}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \mu) = \mu', [\![\vec{v}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}.$$

By hypothesis, we have that $\mathbb{M} : \mathcal{E}, \eta, \rho, \mu' \models \varphi^2$, hence, by well-formedness of C^2, we have that there exists g_c such that:

$$g_c(\rho_{|\mathbb{M},\eta,l^k_{C^2}}, [\![\vec{u}, \vec{v}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \mu') = [\![(t \mid f)]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha} \mapsto \mathbb{1}^{\eta}_{\vec{a}_k}]:\mathcal{E}} \tag{5.4}$$

with $l^k_{C^2} = [(a_{\vec{K^1+1}}, c_{K^1+1}) \ldots (\vec{a_{k-1}}, c_{k-1})]$.

So finally, let g' be the following function

$$g'(\rho_{|\mathbb{M},\eta,l^k_{C^2}}, \rho_{|\mathbb{M},\eta,l_{C^1}}, [\![\vec{u}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \mu) := \mu', V \leftarrow s(\rho_{|\mathbb{M},\eta,l_{C^1}}, [\![\vec{u}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \mu) \tag{5.5}$$

$$g_c(\rho_{|\mathbb{M},\eta,l^k_{C^2}}, ([\![\vec{u}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, V), \mu'). \tag{5.6}$$

The function g' is then almost the target function g of Eq. (5.3). We are left to show there exists functions that return the argument for g', and composing these function with g' will end the proof. Hence, we are left to show that

- there exists a function that return $\rho_{|\mathbb{M},\eta,l^k_{C^2}}$ when given $\rho_{|\mathbb{M},\eta,l^k}$;
- there exists a function that return $\rho_{|\mathbb{M},\eta,l_{C^1}}$ when given $\rho_{|\mathbb{M},\eta,l^k}$.

These two points are consequences of Lemma 6. □

Lemma 8 (Well-formedness transitivity)**.** *Let constraints system C^1 and C^2 be such that*

$$\mathcal{E},\Theta \models_{WF(\vec{u},\varphi^1)} C^1 \quad \textit{and} \quad \mathcal{E},\Theta \models_{WF(\vec{u},\vec{v},\varphi^2)} C^2.$$

Assume there exists p *a program and φ an arbitrary assertion such that*

1. p *computes* $\vec{u} \triangleright \vec{v}$ *w.r.t.* C^1, φ^1
2. p *flows from* φ^1 *to* φ^2 *w.r.t.* $C^1, \vec{u}$
3. C^1 *captures* p*'s randomness w.r.t.* $\varphi, \vec{u}$

Then $\mathcal{E},\Theta \models_{WF(\vec{u},\varphi^1)} C^1 \cdot C^2$.

Proof. Proof sketch. This is a consequence of Lemma 7: we build the function s using the program p. □

Lemma 9. *For all $\mathbb{M} : \mathcal{E}$ and η, for any term $\vec{u}$, assertion φ any fresh variable x of type τ tagged* enum*, for all constraint system C,*

$$\textit{if} \quad \mathcal{E},(x:\tau),\Theta \models_{WF((\vec{u},x),\varphi)} C \quad \textit{then} \quad \mathcal{E},\Theta \models_{WF(\vec{u},\varphi)} \textstyle\prod_{(x:\tau)} C.$$

Proof. Let $\mathbb{M} : \mathcal{E}$ be a model, and η be a security parameter. For all v in $[\![\tau]\!]^\eta_{\mathbb{M}}$, let l^v_C be the sequence witnessing the well-formedness of C w.r.t. $\mathbb{M}_v \overset{\text{def}}{=} \mathbb{M}[x \mapsto \mathbb{1}^\eta_v]$. We write l^v the sequence obtained from l^v_C by replacing each constraint instance

$$\big(\vec{a},(\vec{\alpha},\mathsf{n},t,T,f)\big) \quad \text{by} \quad \big((v,\vec{a}),((x,\vec{\alpha}),\mathsf{n},t,T,f)\big).$$

The type τ is enumerable. Then let $(v_1,\dots,v_J)$ be a sequence of all elements of $[\![\tau]\!]^\eta_{\mathbb{M}}$. Finally, let $l$the concatenation of sequences $l^{v_1}\cdots l^{v_J}$. We are going to show that l witnesses the well-formedness of $\prod_{(x:\tau)} C$.

Let j be an integer in $\{1\dots J\}$, let then $l^{v_j}_C = ((\vec{a_1},c_1),\dots,(\vec{a_K},c_K))$, and $l^{v_j} = ((v_j,\vec{a_1},c^x_1),\dots,(v_j,\vec{a_K},c^x_K))$, and let k be an integer in $\{1,\dots K\}$. Let us show that there exists a function g, such that for all random tape ρ and memory μ such that $\mathbb{M} : \mathcal{E},\eta,\rho,\mu \models \varphi$,

$$g(\rho_{|\mathbb{M},\eta,l^{j,k}},[\![\vec{u}]\!]^{\eta,\rho}_{\mathbb{M},\mathcal{E}},\mu) = [\![(t \mid f)]\!]^{\eta,\rho}_{\mathbb{M}[x\mapsto\mathbb{1}^\eta_{v_j};\vec{\alpha_k}\mapsto\mathbb{1}^\eta_{\vec{a}}]}$$

with $l^{j,k} = l^{v_1}\cdots l^{v_{j-1}}\cdot l^{v_j,k}$ and

$$l^{v_j,k} = \big((v_j,\vec{a_1},c^x_1),\dots,(v_j,\vec{a_{k-1}},c^x_{k-1})\big).$$

By well-formedness of C relatively to $(\vec{u},x),\varphi$, w.r.t. $\mathbb{M}_{v_j},\eta$, there exists g_v such that for all random tape ρ and memory μ such that $\mathbb{M}_{v_j} : \mathcal{E},\eta,\rho,\mu \models \varphi$,

$$g_v(\rho_{|\mathbb{M}_{v_j},\eta,l^{v_j,k}_C},[\![\vec{u},x]\!]^{\eta,\rho}_{\mathbb{M}_{v_j}},\mu) = [\![(t \mid f)]\!]^{\eta,\rho}_{\mathbb{M}_{v_j}[\vec{\alpha_k}\mapsto\mathbb{1}^\eta_{\vec{a_k}}]}.$$

with $l_C^{v_j,k} = \big((\vec{a}_1, c_1), \ldots, (\vec{a_{k-1}}, c_{k-1})\big)$. Notice that $\mathbb{M}_{v_j}[\vec{\alpha_k} \mapsto \mathbb{1}^{\eta}_{\vec{a_k}}] = \mathbb{M}[x \mapsto \mathbb{1}^{\eta}_{v_j}; \vec{\alpha_k} \mapsto \mathbb{1}^{\eta}_{\vec{a_k}}]$, and $[\![\vec{u}, x]\!]^{\eta,\rho}_{\mathbb{M}_v} = ([\![\vec{u}]\!]^{\eta,\rho}_{\mathbb{M},\mathcal{E}}, v_j)$. Then, it remains to show that there exists a function that outputs $\rho_{|\mathbb{M}_{v_j},\eta,l_C^{v_j,k}}$ from $\rho_{|\mathbb{M},\eta,l^{j,k}}$, $[\![\vec{u}]\!]^{\eta,\rho}_{\mathbb{M},\mathcal{E}},v_j$ and μ. First, notice that in $l^{v_j,k}$, all instances bind the variable x to v_j, then $\mathcal{N}^{\eta,\rho}_{\mathbb{M}_{v_j},l_C^{v_j,k}} = \mathcal{N}^{\eta,\rho}_{\mathbb{M},l^{v_j,k}}$ and thus $\rho_{|\mathbb{M}_{v_j},\eta,l_C^{v_j,k}} = \rho_{|\mathbb{M},\eta,l^{j,k}}$ and we conclude using Lemma 6. □

5.3.3 Memory flow lemmas

Lemma 10 (Constant memory flow)**.** *Let $\mathbb{M} : \mathcal{E}$ be a model. Any basic program* p *flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\varphi}$ w.r.t. $\mathbb{M} : \mathcal{E}, \boldsymbol{C}, \vec{\boldsymbol{u}}$, for all assertion $\boldsymbol{\varphi}$, constraint system $\boldsymbol{C}$, and input vector $\vec{\boldsymbol{u}}$ such that the assigned variables in* p *are not in $\boldsymbol{\varphi}$.*

Proof sketch. The program p does not make any oracle calls and, as a result, leaves the game memory unchanged. Additionally, it does not modify any variables that are in $\boldsymbol{\varphi}$, so any memory μ that satisfies $\boldsymbol{\varphi}$ before p is executed will continue to satisfy it after. □

Lemma 11 (Memory flow weakening)**.** *Let* p *be a program, such that* p *flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\psi}$ w.r.t $\mathbb{M} : \mathcal{E}, \boldsymbol{C}, \vec{\boldsymbol{u}}$, for a given $\mathbb{M} : \mathcal{E}$, $\boldsymbol{\varphi}$, $\boldsymbol{\psi}$ and $\vec{\boldsymbol{u}}$,*

Let $\boldsymbol{\varphi}'$ $\boldsymbol{\psi}'$ be assertions, let $\boldsymbol{C}$ be a constraint system such that

$$\mathbb{M} : \mathcal{E} \models \boldsymbol{C} \subseteq \boldsymbol{C}' \tag{5.7}$$

$$\mathbb{M} : \mathcal{E} \models^{A} \boldsymbol{\varphi}' \Rightarrow \boldsymbol{\varphi} \tag{5.8}$$

$$\mathbb{M} : \mathcal{E} \models^{A} \boldsymbol{\psi} \Rightarrow \boldsymbol{\psi}' \tag{5.9}$$

Then p *flows from $\boldsymbol{\varphi}'$ to $\boldsymbol{\psi}'$ w.r.t $\mathbb{M} : \mathcal{E}, \boldsymbol{C}', \vec{\boldsymbol{u}}$*

Proof. Let $\vec{X}$ be a sequence of input variables for p. Let $i \in \{0, 1\}$, and ρ, $\mathfrak{p}$ a pair of tapes such that $\rho\ \mathcal{R}^{\eta}_{C_i',\mathbb{M}}\ \mathfrak{p}$, and μ a memory such that $\mathbb{M}, \eta, \rho, \mu \models \varphi_i'$.
By Eq. (5.7) and Lemma 4, we have $\mathcal{R}^{\eta}_{C_i',\mathbb{M}} \subset \mathcal{R}^{\eta}_{C_i,\mathbb{M}}$, hence $\rho\ \mathcal{R}^{\eta}_{C,\mathbb{M}}\ \mathfrak{p}$.
By Eq. (5.8), we have that $\mathbb{M}, \eta, \rho, \mu \models \varphi_i$.
Then, since p flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\psi}$, we have that $\mathbb{M}, \eta, \rho, \mu' \models \psi_i$, where $\mu' = (\!|\mathsf{p}|\!)^{\eta,\mathfrak{p}}_{\mathbb{M}:\mathcal{E},\mu[\vec{X}\mapsto[\![\vec{u_i}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}]}$.

Hence, by Eq. (5.9), we have that $\mathbb{M}, \eta, \rho, \mu' \models \psi_i'$, which ends the proof. □

Lemma 12 (Memory flow composition)**.** *Let* p *and* p′ *be programs such that*

- p *flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\varphi}'$ w.r.t $\mathbb{M} : \mathcal{E}, \boldsymbol{C}, \vec{\boldsymbol{u}}$,*
- p *compute $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}$ w.r.t. $\mathbb{M} : \mathcal{E}, \boldsymbol{C}, \boldsymbol{\varphi}$, and*
- p′ *flows from $\boldsymbol{\varphi}'$ to $\boldsymbol{\psi}$ w.r.t $\mathbb{M} : \mathcal{E}, \boldsymbol{C}, \vec{\boldsymbol{v}}$*

for $\mathbb{M} : \mathcal{E}$ a model, $\boldsymbol{\varphi}$ $\boldsymbol{\varphi}'$ $\boldsymbol{\psi}$ assertions, $\boldsymbol{C}$ a constraint system, $\vec{\boldsymbol{u}}$ and $\vec{\boldsymbol{v}}$ vectors of bi-terms. Then p; p′ *flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\psi}$ w.r.t $\mathbb{M} : \mathcal{E}, \boldsymbol{C}, \vec{\boldsymbol{u}}$.*

Proof sketch. Program p_1 flows to the pre-condition of p_2. The additional hypothesis ensures that the hypothis on the flow of p_2 can be used to conclude. □

5.3.4 Computation lemmas

Lemma 13 (Computation weakening)**.** *Let* p *be a program such that* p *computes* $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}$ *w.r.t* $\mathbb{M} : \mathcal{E}, \boldsymbol{C}, \boldsymbol{\varphi}$ *for some arbitrary* $\mathbb{M}, \boldsymbol{C}$ *and* $\boldsymbol{\varphi}$*.*

Let $\boldsymbol{C}'$ *and* $\boldsymbol{\varphi}'$ *be such that*

- $\mathbb{M} : \mathcal{E} \models \boldsymbol{C} \subseteq \boldsymbol{C}'$
- $\mathbb{M} : \mathcal{E} \models^{A} \boldsymbol{\varphi}' \Rightarrow \boldsymbol{\varphi}$

Then, p *computes* $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}$ *w.r.t* $\mathbb{M} : \mathcal{E}, \boldsymbol{C}', \boldsymbol{\varphi}'$*.*

Proof sketch. We have $\mathcal{R}^{\eta}_{\boldsymbol{C}',\mathbb{M}} \subseteq \mathcal{R}^{\eta}_{\boldsymbol{C},\mathbb{M}}$ and all memory μ that verifies $\boldsymbol{\varphi}$ verifies $\boldsymbol{\varphi}'$. □

Lemma 14 (Computation composition)**.** *Let* p_1 *and* p_2 *be programs such that*

- p_1 *computes* $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}$ *w.r.t.* $\boldsymbol{C}^1, \boldsymbol{\varphi}$
- p_1 *flow from* $\boldsymbol{\varphi}$ *to* $\boldsymbol{\psi}$ *w.r.t.* $\boldsymbol{C}^1, \vec{\boldsymbol{u}}$
- p_2 *computes* $\vec{\boldsymbol{v}} \triangleright \vec{\boldsymbol{w}}$ *w.r.t.* $\boldsymbol{C}^2, \boldsymbol{\psi}$

Then $\mathsf{p}_1; \mathsf{p}_2$ *computes* $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{w}}$ *w.r.t.* $\boldsymbol{C}^1 \cdot \boldsymbol{C}^2, \boldsymbol{\varphi}$*.*

Proof sketch. Program p_1 computes the input for p_2, which computes the output $\vec{\boldsymbol{w}}$ from it. The additional condition in the flow of p_1 ensures that the computation hypothesis for p_2 can be used for any memory reached after executing p_1. □

5.3.5 Adversary lemmas

Lemma 15 (Basic PTIME adversary)**.** *Let* p *be a PTIME basic program. Then,* p *is a PTIME adversary against* $\mathcal{G}$*.*

Proof sketch. Program p is PTIME and does no oracle call. □

Lemma 16 (Adversaries basic composition)**.** *Let* p *be an adversary against* $\mathcal{G}$*. Let* p' *be a basic program. Then* $\mathsf{p}; \mathsf{p}'$ *(resp.* $\mathsf{p}'; \mathsf{p}$ *) is an adversary against* $\mathcal{G}$*. Furthermore, if* p *is PTIME, then* $\mathsf{p}; \mathsf{p}$ *(resp.* $\mathsf{p}'; \mathsf{p}$*) is also PTIME.*

Proof sketch. We compose an adversary with a basic program: this does not change local and global randomness usage. □

Lemma 17 (Adversaries composition)**.** *Let* p_1 *and* p_2 *be PTIME-adversaries against* $\mathcal{G}$*, such that:*

- $\boldsymbol{C}^1$ *captures* p_1 *randomness w.r.t.* $\boldsymbol{\varphi}, \vec{\boldsymbol{u}}$ *,*
- $\boldsymbol{C}^2$ *captures* p_2 *randomness w.r.t.* $\boldsymbol{\psi}, \vec{\boldsymbol{v}}$ *,*
- p_1 *computes* $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}$ *w.r.t.* $C^1, \boldsymbol{\varphi}$*,*
- p_1 *flows from* $\boldsymbol{\varphi}$ *to* $\boldsymbol{\psi}$ *w.r.t.* $\boldsymbol{C}^1, \vec{\boldsymbol{u}}$*,*
- $\mathbb{M} : \mathcal{E} \models \mathrm{Valid}(C^1 \cdot C^2)$*,*

Then $\mathsf{p}_1;\mathsf{p}_2$ *is an adversary against* $\mathcal{G}$.

Proof. Let $\mathbb{M}$ be a model, let $i \in \{0,1\}$.

Let $\rho\ \mathcal{R}^{\eta}_{C,\mathbb{M}}\ \mathfrak{p}$ be tapes and let then μ be a memory such that $\mathbb{M} : \mathcal{E}, \eta, \rho, \mu \models^{A} \varphi_i$. Suppose that the execution of $\mathsf{p}_1;\mathsf{p}_2$ violate the conditions to be an adversarial behaviour.

Let assume that it violates the freshness constraints. That means that there are two moments during the execution where the same offset is used for and oracle samplings that should be fresh. If this happens both in p_1's (resp. p_2's) execution, then this contradicts the fact that p_1 (resp. p_2) is an adversary. Otherwise, by footprint properties, we have that this offset is of the form $O^{\eta}_{\mathbb{M}}(n,a)$ and that $\langle n, a, \mathsf{T}^{\mathsf{loc}}_{\mathsf{G}}\rangle$ is in $\mathcal{N}^{\eta,\rho}_{C^1_i}$ and $\mathcal{N}^{\eta,\rho}_{C^2_i}$. In that case, this violtes the validity of $C^1_i \cdot C^2_i$.

The same reasoning works if the consistency of global sampling is violated.

Hence, $\mathsf{p}_1;\mathsf{p}_2$ is an adversary. □

Lemma 18 (Adversaries composition)**.** *Let* p *be an adversary against* $\mathcal{G}$, $(x:\tau)$ *a variable, such that,* $\mathbb{M} : \mathcal{E} \models \mathrm{Valid}(\prod_{(x:\tau)} \boldsymbol{C})$. *For all model* $\mathbb{M}$ *and element* $a \in [\![\tau]\!]^{\eta}_{\mathbb{M}}$, $\boldsymbol{C}$ *captures* p *randomness w.r.t.* $\mathbb{M}[x \mapsto \mathbb{1}^{\eta}_{a}], \vec{\boldsymbol{u}}, x$ *and* $\mathbb{M} : \mathcal{E} \models \mathrm{Valid}(\prod_{(x:\tau)} \boldsymbol{C})$.

Then for any list l *of elements in* $[\![\tau]\!]^{\eta}_{\mathbb{M}}$, *assuming* p *does not modify* l *the following program is an adversary:*

```
1 while l ≠ [] do
2   p;
3   l ← tail l;
```

Proof sketch. We use the same reasoning as in the proof of Lemma 17. □

5.3.6 Footprint lemma

Lemma 19 (Footprint basic)**.** *Let* p *be a basic program. Then* $\emptyset$ *captures* p *randomness with respect to* $\boldsymbol{\varphi}, \vec{\boldsymbol{u}}$ *for any* $\boldsymbol{\varphi}$ *and* $\vec{\boldsymbol{u}}$.

Proof. Immediate: p does no oracle calls, so the global and local samplings set are empty. □

Lemma 20 (Footprint weakening)**.** *Let* p *be a program such that* $\boldsymbol{C}$ *captures* p *randomness w.r.t.* $\boldsymbol{\varphi}, \vec{\boldsymbol{u}}$. *Let* $\boldsymbol{C}'$ *and* $\boldsymbol{\varphi}'$ *be such that*

- $\mathbb{M} : \mathcal{E} \models \boldsymbol{C} \subseteq \boldsymbol{C}'$
- $\mathbb{M} : \mathcal{E} \models^{A} \boldsymbol{\varphi}' \Rightarrow \boldsymbol{\varphi}$

Then, $\boldsymbol{C}'$ *captures* p *randomness w.r.t.* $\boldsymbol{\varphi}', \vec{\boldsymbol{u}}$.

Lemma 21 (Footprint composition)**.** *Let* p_1 *be a program such that* $\boldsymbol{C}^1$ *captures* p_1 *randomness w.r.t.* $\boldsymbol{\varphi}, \vec{\boldsymbol{u}}$ *and* p_2 *be a program such that* $\boldsymbol{C}^2$ *captures* p_2 *randomness w.r.t.* $\boldsymbol{\varphi}', \vec{\boldsymbol{v}}$. *Furthermore, assume that*

- p_1 *computes* $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}$ *w.r.t.* $\boldsymbol{C}, \boldsymbol{\varphi}$ *and*
- p_1 *flows from* $\boldsymbol{\varphi}$ *to* $\boldsymbol{\varphi}'$ *w.r.t.* $\boldsymbol{C}, \vec{\boldsymbol{u}}$.

Then $\boldsymbol{C}^1 \cdot \boldsymbol{C}^2$ *capture* $\mathsf{p}_1;\mathsf{p}_2$ *randomness w.r.t.* $\boldsymbol{\varphi}, \vec{\boldsymbol{u}}$.

5.3.7 Structural rules

The next sections, and last, are dedicated to the soundness proof of the proof system's rules. This section concentrates on structural rules.

The structural rules have the particularity that the program that comes from the premise witnesses the validity of the conclusion, up to basic operation on the return values of the program. Roughly, the witness program of the conclusion only restructures the outputs. All the rules are in Figure 5.1.

Rule Refl

Proof. Let p be the following program, with $\vec{X}, X_t$ its input variables:

```
1 return X_t
```

Let us show that p witnesses the validity of $\mathcal{E}, \Theta, \emptyset, (\varphi, \varphi) \vdash \vec{u}, t \triangleright t$.

1. p is a PTIME adversary against $\mathcal{G}$;
2. p flows from φ to φ w.r.t. $\emptyset$;
3. p computes $\vec{u}, t \triangleright t$;
4. $\emptyset$ captures p randomness; and
5. $\emptyset$ is well-formed relatively to $\vec{u}, t, \varphi$.

Proof of (1) Immediately, p is PTIME and a basic program, so, by lemma Lemma 15, p is an adversary.

Proof of (2) By lemma Lemma 10, p flows from φ to φ.

Proof of (3) For all memory μ, the execution of p yields the memory $\mu[\mathsf{res} \mapsto \mu(X_t)]$. In particular, for any side $i \in \{0,1\}$ and any tapes ρ, when executed with $\mu[\vec{X} \mapsto [\![u_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, X_t \mapsto [\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}]$, p yields the memory $\mu[\vec{X} \mapsto [\![u_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, X_t \mapsto [\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}, \mathsf{res} \mapsto [\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}]$, which concludes the proof.

Proof of (4) p does no random samplings. Hence, any execution of p relies on empty global and locals samplings, that is : $G_\$ = \emptyset$ and $L_\$ = \emptyset$. Furthermore, for all ρ, we have $\mathcal{N}^{\eta,\rho}_{C} = \#(\emptyset;\emptyset)$. In conclusion, C captures p randomness.

Proof of (5) The empty constraint system is always well-formed, see Definition 22. □

Common proof sketch

All structural rules, except Refl are of the form

$$\frac{\begin{array}{c}\textsc{Generic.Structural-rule}\\ \mathcal{E}, \Theta, C, (\varphi, \psi) \vdash \vec{u} \triangleright \vec{v}\\ \mathcal{P}\end{array}}{\mathcal{E}, \Theta', C', (\varphi', \psi') \vdash \vec{u'} \triangleright \vec{v'}}$$

With $\mathcal{P}$ a premise that is not a bideduction judgement such that $\mathcal{P}$ implies

($\mathcal{P}$.1) $\Theta \models \boldsymbol{C} \subseteq \boldsymbol{C}'$

($\mathcal{P}$.2) $\Theta \models \Theta'$

($\mathcal{P}$.3) $\mathcal{E}, \Theta \models^{A} \boldsymbol{\varphi}' \Rightarrow \boldsymbol{\varphi}$

($\mathcal{P}$.4) $\mathcal{E}, \Theta \models^{A} \boldsymbol{\psi} \Rightarrow \boldsymbol{\psi}'$.

Indeed, for all rules, these statements are either directly elements of $\mathcal{P}$ or $\boldsymbol{C}' = \boldsymbol{C}$, $\boldsymbol{\varphi}' = \boldsymbol{\varphi}$ and/or $\boldsymbol{\varphi} = \boldsymbol{\psi}'$, which trivializes them.

In this section we present a generic gap-fill proof of the rule Generic.Structural-rule. This proofs will rely on extra-hypothesis (noted **(H1)**,**(H2)**, etc.) that we define and assume along the way (the gaps in the proof) and definitions. Also, each proof will consist in exhibiting a witness program p' for the conclusion. In our generic gap-fill proof, p' is left unspecified.

Each proof for any structural rules will then consist in providing a program p' and poving the hypothesis for their specific cases.

Proof. Let p be a program that witnesses the validity of the bideduction judgement in the premises of the rule.

Target proof. We want to prove that there exists a program p', such that for all $\mathbb{M}$ such that $\mathbb{M} \models \Theta \tilde{\wedge} \mathrm{Valid}(\boldsymbol{C})$ we have :

1. p' is a PTIME adversary against $\mathcal{G}$;
2. p' flows from $\boldsymbol{\varphi}'$ to $\boldsymbol{\psi}'$ w.r.t. $\boldsymbol{C}', \vec{\boldsymbol{u}}'$;
3. p' computes $\vec{\boldsymbol{u}}' \triangleright \vec{\boldsymbol{v}}'$ w.r.t. $C', \boldsymbol{\varphi}'$;
4. $\boldsymbol{C}'$ captures p' randomness; and
5. $\boldsymbol{C}'$ is well-formed relatively to $\vec{\boldsymbol{u}}', \boldsymbol{\varphi}'$.

Let $\mathbb{M}$ be a model, such that $\mathbb{M} \models \Theta' \tilde{\wedge} \mathrm{Valid}(\boldsymbol{C}')$. By $\mathcal{P}$ and Lemma 4, we then have $\mathbb{M} \models \Theta \tilde{\wedge} \mathrm{Valid}(\boldsymbol{C})$.

Hypothesis. Using bideduction premises, by **(H1)** we have that

(Hadv) p is a PTIME adversary against $\mathcal{G}$;

(Hflw) p flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\psi}$ of the premises w.r.t. $\boldsymbol{C}, \vec{\boldsymbol{u}}$;

(Hcomp) p computes $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{v}}$ w.r.t. $\boldsymbol{C}, \boldsymbol{\varphi}$;

(Hrnd) $\boldsymbol{C}$ captures p randomness; and

(Hwf) $\boldsymbol{C}$ is well-formed relatively to $\vec{\boldsymbol{u}}, \boldsymbol{\varphi}$.

Let us define **(H1)** $\stackrel{\text{def}}{=}$ The program p' is of the form $\mathsf{p}_1;\mathsf{p};\mathsf{p}_2$ where p_1 and p_2 only do operation assignments on fresh variables, without any random sampling, oracle call or while loops. Let us assume **(H1)**.

Furthermore, let $\vec{X}$ be input variables of p, $\vec{X}_1$ be input variables of p_1, and $\vec{X}_2$ be input variables of p_2. Let us assume the following two hypotheses:

- **(H2)** $\stackrel{\text{def}}{=}$ p_1 computes $\vec{\boldsymbol{u}'} \triangleright \vec{\boldsymbol{u}}$ w.r.t $\emptyset, \boldsymbol{\varphi}'$ and its output variables are input variables of p.
- **(H3)** $\stackrel{\text{def}}{=}$ p_2 computes $\vec{\boldsymbol{v}} \triangleright \vec{\boldsymbol{v}'}$ w.r.t $\emptyset, \boldsymbol{\psi}$ and its input variables are output variables of p.

Proof of (1) By **(H1)** we have that p' is of the form $\mathsf{p}_1;\mathsf{p};\mathsf{p}_2$ with p_1 and p_2 basic programs without while loops. Furthermore, by (Hadv), p is a PTIME-adversary against $\mathcal{G}$. Hence, by Lemma 15 p' is a PTIME-adversary against $\mathcal{G}$ which ends the proof.

Proof of (2) By the consequence of $\mathcal{P}$, we can weaken the constraint system, pre- and post-conditions by Lemma 11, and we are left to show that

$$\mathsf{p}' \text{ flows from } \boldsymbol{\varphi} \text{ to } \boldsymbol{\psi} \text{ w.r.t. } \boldsymbol{C}, \vec{\boldsymbol{u}}.$$

Then, by **(H1)** and Lemma 10 we have that

$$\mathsf{p}_1 \text{ flows from } \boldsymbol{\varphi} \text{ to } \boldsymbol{\varphi} \text{ w.r.t. } \boldsymbol{C}, \vec{\boldsymbol{u}'}$$
$$\mathsf{p}_2 \text{ flows from } \boldsymbol{\psi} \text{ to } \boldsymbol{\psi} \text{ w.r.t. } \boldsymbol{C}, \vec{\boldsymbol{v}}$$

Also, by **(H2)** and by weakening with Lemma 13 and $\mathcal{P}$

$$\mathsf{p}_1 \text{ computes } \vec{\boldsymbol{u}'} \triangleright \vec{\boldsymbol{u}} \text{ w.r.t. } \boldsymbol{C}, \boldsymbol{\varphi}$$

We end then the proof by composing memory flows through Lemma 12, using above results and (Hflw).

Proof of (3) By the consequence of $\mathcal{P}$, we can weaken the constraint system, pre- and post-conditions by Lemma 13, and we are left to show that

$$\mathsf{p}' \text{ computes } \vec{\boldsymbol{u}'} \triangleright \vec{\boldsymbol{v}'} \text{ w.r.t. } \boldsymbol{C}, \boldsymbol{\varphi}.$$

By **(H2)** and by weakening with Lemma 13

$$\mathsf{p}_1 \text{ computes } \vec{\boldsymbol{u}'} \triangleright \vec{\boldsymbol{u}} \text{ w.r.t. } \emptyset, \boldsymbol{\varphi}$$

By **(H1)** and Lemma 10 we have that

$$\mathsf{p}_1 \text{ flows from } \boldsymbol{\varphi} \text{ to } \boldsymbol{\varphi} \text{ w.r.t. } \emptyset, \vec{\boldsymbol{u}'}$$

By **(H3)**

$$\mathsf{p}_2 \text{ computes } \vec{\boldsymbol{v}} \triangleright \vec{\boldsymbol{v}'} \text{ w.r.t. } \emptyset, \boldsymbol{\psi}$$

We end then the proof by composing computation through Lemma 14, using above results, (Hflw) and (Hcomp), and observing $\emptyset \cdot \boldsymbol{C} \cdot \emptyset = \boldsymbol{C}$.

Proof of (4) By weakening Lemma 20 on footprints we are left to show that

$$\boldsymbol{C} \text{ captures } \mathsf{p}' \text{ randomness w.r.t. } \boldsymbol{\varphi}, \vec{\boldsymbol{u}}$$

By **(H1)** and Lemma 19, we have that $\emptyset$ captures p_1 randomness w.r.t $\boldsymbol{\varphi}, \vec{\boldsymbol{u}'}$ and p_2 randomness w.r.t $\boldsymbol{\psi}, \vec{\boldsymbol{v}}$.

By **(H2)** and by weakening with Lemma 13

$$\mathsf{p}_1 \text{ computes } \vec{\boldsymbol{u}'} \triangleright \vec{\boldsymbol{u}} \text{ w.r.t. } \emptyset, \boldsymbol{\varphi}$$

By **(H1)** and Lemma 10 we have that

$$\mathsf{p}_1 \text{ flows from } \boldsymbol{\varphi} \text{ to } \boldsymbol{\varphi} \text{ w.r.t. } \emptyset, \vec{\boldsymbol{u}'}$$

We end then the proof by composing footprint through Lemma 21, using above results, (Hflw) and (Hcomp).

Proof of (5) Let **(H4)** be that (5) directly follows from $\mathcal{P}$.

□

Proofs' specific parts

Now, for each rule, we give the program p', and the proof of the hypotheses **(H1)**, **(H2)**, **(H3)** and **(H4)** when they are not immediate.

Rule Weak.Constr

1 p;

Rule Weak.Cond Let $X_f, (X'_f, X_v), \vec{X_w}$ be p's input variables.

1 p;
2 $X_{res} \leftarrow$ if X_f then X_v
3 **return** $(X_f, X_{res}), \vec{X_w}$

The hypothesis **(H3)** comes from the fact that when $\mathcal{E}, \Theta \vdash [\boldsymbol{f} \Rightarrow \boldsymbol{f'}]_{\mathrm{e}}$, then $\mathcal{E}, \Theta \vdash [\mathsf{if}\ \boldsymbol{f}\ \mathsf{then}\ (\mathsf{if}\ \boldsymbol{f'}\ \mathsf{then}\ \boldsymbol{v}) = \mathsf{if}\ \boldsymbol{f}\ \mathsf{then}\ \boldsymbol{v}]_{\mathrm{e}}$.

Rule Weak.Mem

1 p;

Rule Weak.Hyps

1 p;

Rule Permute Let $X_1, \cdots X_m$ be p's input variables and let $Y_1, \cdots Y_n$ be p's output variables. Similarly, we let $X'_1, \cdots X'_m$ be p''s input variables, taken fresh, and let $Y'_1, \cdots Y'_n$ be p''s output variables.

```
1 X_1 ← X'_{σ^{-1}(1)};
2 ⋮
3 X_m ← X'_{σ^{-1}(m)};
4 p;
5 Y'_1 ← X'_{σ'(1)};
6 ⋮
7 Y'_n ← X'_{σ'(n)};
8 return Y'_1, ··· , Y'_n
```

Rule DROP Let $\vec{X}_u$ be $prog'$ and p's input variables and let $\vec{X}_v, \vec{X}_t$ be p's output variables.

```
1 p;
2 return X⃗_v
```

Rule REWRITE-L

```
1 p;
```

Rule REWRITE-R

```
1 p;
```

Rule DUP Let $\vec{X}_u$ be $prog'$ and p's input variables and let $\vec{X}_v, \vec{X}_t$ be p's output variables.

```
1 p;
2 return X⃗_v, X⃗_t, X⃗_t
```

5.3.8 Adversarial rules

In this section, we present the soundness proofs for our adversarial rules. These two rules are justified by simulators performing samplings and oracle calls. The main challenges here are ensuring that the witness program of the conclusion behaves as an adversary and that we maintain the validity and well-formedness of the constraint systems.

Rule NAME

Proof. Let p be a program that witnesses the validity of the bideduction judgement in the premises of NAME, and let X_f, X_t be its two return variables.

Let p' be the following program:

```
1 p;
2 if X_f
3  then
4   X_n ←$ T_S[offset_n(X_t)];
5 return X_f , X_n
```

Let show that p' witnesses the bideduction judgement in the conclusion of NAME. Let $\mathbb{M}$ be a model such that $\mathbb{M} \models \Theta \tilde{\wedge} \mathrm{Valid}(\boldsymbol{C} \cdot (\emptyset, n, \boldsymbol{t}, \mathtt{T}_\mathsf{S}, \boldsymbol{f}))$.

1. p' is a PTIME-adversary against $\mathcal{G}$;

2. p' flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\psi}$ w.r.t. $\boldsymbol{C} \cdot (\emptyset, n, \boldsymbol{t}, \mathtt{T_S}, \boldsymbol{f})$;

3. p' computes $\vec{\boldsymbol{u}} \triangleright (\mathsf{n}\ \boldsymbol{t} \mid \boldsymbol{f})$ with respect to $\boldsymbol{C} \cdot (\emptyset, n, \boldsymbol{t}, \mathtt{T_S}, \boldsymbol{f})$;

4. $\boldsymbol{C} \cdot (\emptyset, n, \boldsymbol{t}, \mathtt{T_S}, \boldsymbol{f}))$ captures p' randomness; and

5. $\boldsymbol{C} \cdot (\emptyset, n, \boldsymbol{t}, \mathtt{T_S}, \boldsymbol{f}))$ is well-formed relatively to $\vec{\boldsymbol{u}}, \boldsymbol{\varphi}$.

By constraint system inclusion Lemma 4, we have that $\mathbb{M} \models \Theta \tilde{\wedge} \mathrm{Valid}(\boldsymbol{C})$. Hence :

(Hadv) p is an adversary against $\mathcal{G}$;

(Hflw) p flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\psi}$ w.r.t. $\boldsymbol{C}$;

(Hcomp) p computes $\vec{\boldsymbol{u}} \triangleright (\boldsymbol{t} \mid \boldsymbol{f})$ with respect to $\boldsymbol{C}$;

(Hrnd) $\boldsymbol{C}$ captures p randomness; and

(Hwf) $\boldsymbol{C}$ is well-formed relatively to $\vec{\boldsymbol{u}}, \boldsymbol{\varphi}$.

Proof of (1) By composition Lemma 17, and (Hadv), p' is an PTIME adversary against $\mathcal{G}$.

Proof of (2) Immediately we have that $\mathbb{M} : \mathcal{E} \models \boldsymbol{C} \subseteq \boldsymbol{C} \cdot (\emptyset, n, \boldsymbol{t}, \mathtt{T_S}, \boldsymbol{f})$. By Lemma 11, we have that p flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\psi}$ with respect to $\boldsymbol{C} \cdot (\emptyset, n, \boldsymbol{t}, \mathtt{T_S}, \boldsymbol{f})$. Let us call p'' the program such that $\mathsf{p}' = \mathsf{p}; \mathsf{p}''$. Then p'' flows from $\boldsymbol{\psi}$ to $\boldsymbol{\psi}$ with respect to $\boldsymbol{C} \cdot (\emptyset, n, \boldsymbol{t}, \mathtt{T_S}, \boldsymbol{f}), (\boldsymbol{t} \mid \boldsymbol{f})$ by Lemma 10. We conclude by Lemma 12 and (Hcomp), that shows that $\mathsf{p}; \mathsf{p}''$ flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\psi}$ with respect to $\boldsymbol{C} \cdot (\emptyset, n, \boldsymbol{t}, \mathtt{T_S}, \boldsymbol{f})$.

Proof of (3) Let $i \in \{0, 1\}$. Let ρ and $\mathfrak{p}$ tapes such that $\rho\ \mathcal{R}^{\eta}_{C_i \cdot (\emptyset, n, t_i, \mathtt{T_S}, f_i), \mathbb{M}}\ \mathfrak{p}$ and μ such that $\mathbb{M} : \mathcal{E}, \eta, \rho, \mu \models \varphi_i$.

Let $\vec{X}$ be the inputs variables of p, and (X_f, X_t) be its output variable and p' input variable.

Let $\mu' \stackrel{\mathsf{def}}{=} (\!|\mathsf{p}|\!)^{\eta, \mathfrak{p}}_{\mu[\vec{x} \mapsto [\![u_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}]}$. By Lemma 4 we have $\rho\ \mathcal{R}^{\eta}_{C_i, \mathbb{M}}\ \mathfrak{p}$ and by (Hcomp) we have

$$\mu'[X_t] = [\![\mathsf{if}\ f\ \mathsf{then}\ t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$$
$$\mu'[X_f] = [\![f]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$$

Let τ be the type of t. We have that

$$
\begin{aligned}
(\!|\mathsf{p};\mathsf{p}''|\!)^{\eta,\mathfrak{p}}_{\mu[\vec{x}\mapsto[\![u_i]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}]} &= (\!|\mathsf{p}''|\!)^{\eta,\mathfrak{p}}_{\mu'} \\
&= \begin{cases} (\!|X_n \xleftarrow{\$} \mathsf{T_S}[\mathsf{offset_n}(X_t)]|\!)^{\eta,\mathfrak{p}}_{\mu'} & \text{if } [X_f]^{\eta,\mathfrak{p}}_{\mu'} = \mathsf{true} \\ (\!|\mathbf{skip}|\!)^{\eta,\mathfrak{p}}_{\mu'} & \text{if } [X_f]^{\eta,\mathfrak{p}}_{\mu'} = \mathsf{false} \end{cases} \\
&= \begin{cases} \mu'\left[X_n \mapsto [\![\tau]\!]^{\$}_{\mathbb{M}}(\eta, \mathfrak{p}|^{\eta}_{(\mathsf{T_S},\tau)}[O_{\mathbb{M},\eta}(\mathsf{n}, \mu'[X_t])])\right] & \text{if } [X_f]^{\eta,\mathfrak{p}}_{\mu'} = \mathsf{true} \\ \mu' & \text{if } [X_f]^{\eta,\mathfrak{p}}_{\mu'} = \mathsf{false} \end{cases} \\
&= \begin{cases} \mu'\left[X_n \mapsto [\![\tau]\!]^{\$}_{\mathbb{M}}(\eta, \mathfrak{p}|^{\eta}_{(\mathsf{T_S},\tau)}[O_{\mathbb{M},\eta}(\mathsf{n}, [\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}})])\right] & \text{if } [\![f]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = \mathsf{true} \\ \mu' & \text{if } [\![f]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = \mathsf{false} \end{cases} \\
&= \begin{cases} \mu'\left[X_n \mapsto [\![n\ t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}\right] & \text{if } [\![f]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = \mathsf{true} \\ \mu' & \text{if } [\![f]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = \mathsf{false} \end{cases}
\end{aligned}
$$

which ends the proof.

Proof of (4) Immediately, $(\emptyset, n, \boldsymbol{t}, \mathsf{T_S}, \boldsymbol{f})$ captures p'' randomness with the respect to $\boldsymbol{\psi}, (t \mid f)$. We conclude using (Hcomp) (Hrnd) and Lemma 21.

Proof of (5) Immediately, $(\emptyset, n, \boldsymbol{t}, \mathsf{T_S}, \boldsymbol{f})$ is well-formed relatively to $(\boldsymbol{t} \mid \boldsymbol{f})$ and any assertion, so in particular $\boldsymbol{\psi}$. We conclude by composing well-formedness with Lemma 8 with (Hcomp) (Hflw) and (Hrnd). □

Rule ORACLE$_f$

Proof. Let p be a program that witnesses the validity of the bideduction judgement in the premisses of ORACLE$_f$, and let $(X_f, \vec{X}_t), (X_f, \vec{X}_o), (X_f, \vec{X}_s)$ be its return variables.

Let p' be the following program:

```
1 p;
2 if X_f
3  then
4   X_res ← G.f(X_t)[offset_k⃗(X⃗_o), offset_s⃗(X⃗_s)]
5  else
6   X_res ← zero;
7 return X_f, X_res
```

Let us show that p' witness the conclusion validity.

Let $\mathbb{M}$ be a model such that $\mathbb{M} \models \Theta \tilde{\wedge} \mathrm{Valid}(\boldsymbol{C}')$.

1. p' is a PTIME-adversary against $\mathcal{G}$;
2. p' flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\theta}$ w.r.t. $\boldsymbol{C}'$;
3. p' computes $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{w}}, (\boldsymbol{v} \mid \boldsymbol{F})$ with respect to $\boldsymbol{C}'$;
4. $\boldsymbol{C}'$ captures p' randomness; and

5. $\boldsymbol{C}'$ is well-formed relatively to $\vec{\boldsymbol{u}}, \boldsymbol{\varphi}$.

By constraint system inclusion of Lemma 4, we have that $\mathbb{M} \models \Theta \,\tilde{\wedge}\, \mathrm{Valid}(\boldsymbol{C})$. Hence :

(Hadv) p is an adversary against $\mathcal{G}$;

(Hflw) p flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\psi}$ w.r.t. $\boldsymbol{C}$;

(Hcomp) p computes $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{w}}, (\vec{\boldsymbol{t}} \mid \boldsymbol{F}), (\vec{\boldsymbol{o}} \mid \boldsymbol{F}), (\vec{\boldsymbol{s}} \mid \boldsymbol{F})$ with respect to $\boldsymbol{C}$;

(Hrnd) $\boldsymbol{C}$ captures p randomness; and

(Hwf) $\boldsymbol{C}$ is well-formed relatively to $\vec{\boldsymbol{u}}, \boldsymbol{\varphi}$.

Observes that p' is of the form $\mathsf{p}; \mathsf{p}''$. Immediately, we have that p'' is a PTIME adversary against $\mathcal{G}$, and

$$\boldsymbol{C}'' \text{ captures the randomness of } \mathsf{p}'' \text{ with respect to } \vec{\boldsymbol{w}}, (\vec{\boldsymbol{t}} \mid \boldsymbol{F}), (\vec{\boldsymbol{o}} \mid \boldsymbol{F}), (\vec{\boldsymbol{s}} \mid \boldsymbol{F})$$

with $\boldsymbol{C}'' = \prod_{v \in f.\mathrm{glob}_\$} (\emptyset, \mathsf{k}_v, o_{v\sharp}, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},v}, \boldsymbol{F}) \cdot \prod_{v \in f.\mathrm{loc}_\$} (\emptyset, \mathsf{r}_v, s_{v\sharp}, \mathsf{T}^{\mathsf{loc}}_{\mathsf{G}}, \boldsymbol{F})$. Also

$$\boldsymbol{C}'' \text{ is well-formed with respect to } \vec{\boldsymbol{w}}, (\vec{\boldsymbol{t}} \mid \boldsymbol{F}), (\vec{\boldsymbol{o}} \mid \boldsymbol{F}), (\vec{\boldsymbol{s}} \mid \boldsymbol{F}), \boldsymbol{\psi}.$$

By Hoare triples definition

$$\mathsf{p}'' \text{ flows from } \boldsymbol{\psi} \text{ to } \boldsymbol{\theta} \text{ w.r.t. } \boldsymbol{C}'';$$

and

$$\mathsf{p}'' \text{ computes } \vec{\boldsymbol{w}}, (\vec{\boldsymbol{t}} \mid \boldsymbol{F}), (\vec{\boldsymbol{o}} \mid \boldsymbol{F}), (\vec{\boldsymbol{s}} \mid \boldsymbol{F}) \triangleright (\boldsymbol{v} \mid \boldsymbol{F}) \text{ w.r.t. } \boldsymbol{C}''.$$

Proof of (1) We conclude by composition Lemma 17.

Proof of (2) We conclude by composition Lemma 12.

Proof of (3) We conclude by composition Lemma 14.

Proof of (4) We conclude by composition Lemma 21.

Proof of (5) We conclude by composition Lemma 8.

□

5.3.9 Computational rules

This section covers the soundness proofs for our computational rules. We won't detail every proof, but we still give the witness program. Crucially we made the proof of the Transitivity rule and Induction.

For the proof in this section we will need specific programs to handle lists and maps, that we define here. First, we assume the existence of [] the program constant for the empty list and of programs tail and head that processes lists such that

- $\mathsf{tail}\ l$ returns the tail of list l in constant time, and
- $\mathsf{head}\ l$ returns the head element of the list l in constant time.

Similarly, we assume the existence of programs to handle map, that will be used to store function's graphs. We let () be the constant program for the empty map and we write $map[t]$ and $map[t] \leftarrow y$ the programs where

- $map[t]$ returns the element labelled t in map map in time $O(|t|)$.
- $map[t] \leftarrow y$ assigns to the label t the element y in map map in time $O(|t|)$.

Rule LIBRARY

Proof. The term t is in $\mathcal{L}_{\mathsf{p}}$, hence there exists a PPTM $\mathcal{M}$ such that $\mathcal{M}(1^\eta, \rho_{\mathsf{a}}) = [\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$ for all η, for all model $\mathbb{M}$ and all logical tapes $\rho = (\rho_{\mathsf{a}}, \rho_{\mathsf{h}})$.

The program language being Turing-Complete, there exists then a program p_t with output variable res, such that

- p_t only accesses the tape $\mathfrak{p}[\mathtt{T}_{\mathtt{A}}, \mathsf{bool}]$.
- for all η, model $\mathbb{M}$, and logical tapes ρ and program tape $\mathfrak{p}$ s.t. ρ_{a} is a prefixe of $\mathfrak{p}[\mathtt{T}_{\mathtt{A}}, \mathsf{bool}]$; $(\!|p|\!)^{\eta,\mathfrak{p}}_{\mathbb{M}:\mathcal{E}}[\mathsf{res}] = [\![t]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$.

Then, p_t witnesses the bideduction in the conclusion. Remark p_t does not perform oracle calls, and samples only on tape $\mathtt{T}_{\mathtt{A}} \times \mathsf{bool}$, so p_t is a basic program, and

- p_t is a PTIME adversary against $\mathcal{G}$ by Lemma 15,
- p_t flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\varphi}$ by Lemma 10,
- $\emptyset$ captures p_t randomness by Lemma 19,

And, we immediately have that $\emptyset$ is well-formed relatively to $\vec{\boldsymbol{u}}$. □

Rule FA

Proof. Let p be a program that witnesses the validity of the bideduction judgement in the premises of FA, and let $X, (X_f, X_t^1) \dots (X_f, X_t^n)$ be its return variables.

The function g is in $\mathcal{L}_{\mathsf{p}}$, let $\tau_1 \to \cdots \tau_n \to \tau$ be its type. There exists a PPTM $\mathcal{M}$ such that for all inputs $m_1, \cdots, m_n$ in the serialized set of $[\![\tau_1]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} \times \cdots \times [\![\tau_n]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$,

$$\mathcal{M}(1^\eta, m_1, \cdots, m_n, \rho_{\mathsf{a}}) = [\![g]\!]^{\eta,\rho}_{\mathbb{M}[x_1 \mapsto \mathbb{1}^\eta_{m_1} \cdots x_1 \mapsto \mathbb{1}^\eta_{m_1}]:\mathcal{E}(x_1:\tau_1)\cdots(x_n:\tau_n)}$$

for all η, for all model $\mathbb{M}$ and all logical tapes $\rho = (\rho_{\mathsf{a}}, \rho_{\mathsf{h}})$.

By Turing-completeness, there exists a program p_g with output variable res and input variable $X_g^1 \dots X_g^n$ such that for ρ_a and all program tape $\mathfrak{p}$ s.t. ρ_{a} is a prefix of $\mathfrak{p}[\mathtt{T}_{\mathtt{A}}, \mathsf{bool}]$, for all inputs $m_1, \cdots, m_n$ in the serialized set of $[\![\tau_1]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} \times \cdots \times [\![\tau_n]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$

$$(\!|\mathsf{p}_g|\!)^{\eta,\mathfrak{p}}_{[X_g^1 \mapsto m_1, \cdots, X_g^2 \mapsto m_n]}[\mathsf{res}] = [\![g]\!]^{\eta,\rho}_{\mathbb{M}[x_1 \mapsto \mathbb{1}^\eta_{m_1} \cdots x_1 \mapsto \mathbb{1}^\eta_{m_1}]:\mathcal{E}(x_1:\tau_1)\cdots(x_n:\tau_n)}.$$

Let X_g be the return variable of p_g, and X_{res} a fresh variable, initialized to $\mathsf{witness}_\tau$ with τ type of X_{res}. Let p' be the following program:

```
1 p;
2 X_g^1 ← X_t^1;
3 ⋮
4 X_g^n ← X_t^n;
5 if X_f
6   then p_g; X_res ← X_g
7 return X, X_res
```

□

Rule IF-THEN-ELSE

Proof. Let $X_v, X_b, X_f, X_t, X_{\overline{f}}, X_{t'}$ be p's return variables.

```
1 p;
2 if X_b
3   then X_res ← X_t
4   else X_res ← X_t';
5 return X_v,X_{res}
```

□

Rule TRANSITIVITY

Proof. Let p_1 be a program that witnesses the validity of $\mathcal{E}, \Theta, \boldsymbol{C}^1, (\boldsymbol{\varphi}, \boldsymbol{\varphi'}) \vdash \vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{t}}$ and p_2 a program that witness the validity of $\mathcal{E}, \Theta, \boldsymbol{C}^2, (\boldsymbol{\varphi'}, \boldsymbol{\psi}) \vdash \vec{\boldsymbol{u}}, \vec{\boldsymbol{t}} \triangleright \vec{\boldsymbol{v}}$.

Let $\vec{X_u^1}$ be p_1's fresh input variables and $\vec{X_t^1}$ be p_1's fresh return variables. Let $\vec{X_u^2}, \vec{X_t^2}$ be p_2's fresh input variables. Let then $\mathsf{p'}$ be the following program.

```
1 X_u^2 ← X_u^1;
2 p_1;
3 X_t^2 ← X_t^1;
4 p_2
```

Let $\mathbb{M}$ be a model such that $\mathbb{M} \models \Theta \,\tilde{\wedge}\, \mathrm{Valid}(\boldsymbol{C}^1 \cdot \boldsymbol{C}^2)$. Let show that

1. $\mathsf{p'}$ is a PTIME adversary against $\mathcal{G}$;
2. $\mathsf{p'}$ flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\psi}$ w.r.t. $\boldsymbol{C}^1 \cdot \boldsymbol{C}^2$;
3. p computes $\vec{\boldsymbol{u}} \triangleright \vec{\boldsymbol{t}}\vec{\boldsymbol{v}}$;
4. $\boldsymbol{C}^1 \cdot \boldsymbol{C}^2$ captures $\mathsf{p'}$ randomness; and
5. $\boldsymbol{C}^1 \cdot \boldsymbol{C}^2$ is well-formed relatively to $\vec{\boldsymbol{u}}, \boldsymbol{\varphi}$.

By constraint system inclusion, we have that $\mathbb{M} \models \Theta \,\tilde{\wedge}\, \mathrm{Valid}(\boldsymbol{C}^1)$ and $\mathbb{M} \models \Theta \,\tilde{\wedge}\, \mathrm{Valid}(\boldsymbol{C}^2)$. Hence,

(Hadv1) p_1 is a PTIME adversary against $\mathcal{G}$;

(Hflw1) p_1 flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\varphi'}$ w.r.t. $\boldsymbol{C}^1$,$\vec{\boldsymbol{u}}$;

(Hcomp1) p_1 computes $\vec{u} \triangleright \vec{t}$ with respect to C^1;

(Hrnd1) C^1 captures p_1 randomness;

(Hwf1) C^1 is well-formed relatively to $\vec{u}, \varphi$;

and

(Hadv2) p_2 is a PTIME adversary against $\mathcal{G}$;

(Hflw2) p_2 flows from φ' to ψ w.r.t. C^2, $\vec{u}, \vec{t}$;

(Hcomp2) p_2 computes $\vec{u}\vec{t} \triangleright \vec{v}$ with respect to C^2;

(Hrnd2) C^2 captures p_2 randomness; and

(Hwf2) C^2 is well-formed relatively to $\vec{u}\vec{t}, \varphi'$;

Proof of (1) Immediate by composing lemma Lemma 17.

Proof of (2) Let p' be the following program, with input variables X_u^1 and return variables X_u^2, X_t.

```
1 X_u^2 ← X_u^1;
2 p_1;
3 X_t^2 ← X_t^1;
```

Immediately, p' has similar properties than p_1 (same computation, flows, etc.). Hence, by Lemma 12, we conclude that $\mathsf{p}'; \mathsf{p}_2$ flows from φ to ψ w.r.t. $C^1 \cdot C^2$.

Proof of (3) We conclude by same remarks on properties of program p' and Lemma 14.

Proof of (4) We conclude by same remarks properties of program p' and Lemma 21.

Proof of (5) Immediate by Lemma 8.

□

Rule Lambda-App

Proof. We only give the witness program.

```
1 p;
2 X_f, (X_g, X_t) ← res;
3 X_res ← X_g[X_t];
4 return X_f, X_res
```

□

Rules Lambda **and** Quantificator-$O \in \{\forall, \exists\}$

The witness program and proofs of rules of Lambda and Quantificator-$O \in \{\forall, \exists\}$ are very similar. In this part, we prove the 3 in one go. The element for which the proof differs will be annotated by the subscript $\{\lambda, \exists, \forall\}$. The retrieve the proof for one of rules, this element must be taken for the value of the corresponding rules.

Proof. The type τ is enumerable. There exists a machine $\mathcal{M}$, such that for all $\mathbb{M}$ and η, $\mathcal{M}(1^\eta)$ computes in PTIME a list $[a_0, a_1, \cdots, a_n]$ of all elements of $[\![\tau]\!]^\eta_{\mathbb{M}}$.

By Turing completeness, there exist a program $\mathsf{p}_{\tau_\mathsf{b}}$ that computes a list of elements $[a_0, a_1, \cdots, a_n]$ of type τ, covering all space $[\![\tau]\!]^\eta_{\mathbb{M}}$.

Let library elements

$$\mathsf{init}_w = \begin{cases} () \text{ if } w = \lambda \\ \mathsf{true} \text{ if } w = \exists \\ \mathsf{true} \text{ if } w = \forall \end{cases}$$

and

$$\mathsf{update}_w\ x\ t\ l = \begin{cases} (l[x] \leftarrow t) \text{ if } w = \lambda \\ \mathsf{or}\ t\ l \text{ if } w = \exists \\ \mathsf{and}\ t\ l \text{ if } w = \forall \end{cases}$$

Let X_u, X be p input variables. Then $\mathsf{p}'_{\{\lambda,\exists,\forall\}}$ is defined as follows:

```
1 p_τ;
2 l ← res;
3 X_res ← init_{λ,∃,∀};
4 while l ≠ [] do
5   X ← head l;
6   X_f, X_t ← p;
7   l ← tail l;
8   X_res ← update_{λ,∃,∀} X X_t X_res;
9 return X_f, if X_f then X_res
```

Let $\mathbb{M}$ be a model such that $\mathbb{M} \models \Theta \tilde{\wedge} \mathrm{Valid}(\prod_{(x:\tau)} \boldsymbol{C})$. Let show that

1. $\mathsf{p}'_{\{\lambda,\exists,\forall\}}$ is a PTIME adversary against $\mathcal{G}$;
2. $\mathsf{p}'_{\{\lambda,\exists,\forall\}}$ flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\varphi}$ w.r.t. $\prod_{(x:\tau)} \boldsymbol{C}$;
3. $\mathsf{p}'_{\{\lambda,\exists,\forall\}}$ computes $\vec{\boldsymbol{u}} \triangleright (v_{\{\lambda,\exists,\forall\}} \mid \boldsymbol{f})$;
4. $\prod_{(x:\tau)} \boldsymbol{C}$ captures $\mathsf{p}'_{\{\lambda,\exists,\forall\}}$ randomness; and
5. $\prod_{(x:\tau)} \boldsymbol{C}$ is well-formed relatively to $\vec{\boldsymbol{u}}, \boldsymbol{\varphi}$.

with

$$v_w = \begin{cases} \lambda(x:\tau).\boldsymbol{t} \text{ if } w = \lambda \\ \exists(x:\tau).\boldsymbol{t} \text{ if } w = \exists \\ \forall(x:\tau).\boldsymbol{t} \text{ if } w = \forall \end{cases}$$

Let $\mathbb{M}$ ba a model. Then, for all $a_i \in [\![\tau]\!]^\eta_{\mathbb{M}}$, we have $\mathbb{M}[x \mapsto \mathbb{1}^\eta_{a_1}] : \mathcal{E}(x:\tau) \models \mathrm{Valid}(\boldsymbol{C})$. Then for all $a_i \in [\![\tau]\!]^\eta_{\mathbb{M}}$,

(Hadv) p is an PTIME adversary against $\mathcal{G}$;

(Hflw) p flows from $\boldsymbol{\varphi}$ to $\boldsymbol{\varphi}'$ w.r.t. $\boldsymbol{C}$,$\vec{\boldsymbol{u}}, x$;

(Hcomp) p computes $\vec{\boldsymbol{u}}, x \triangleright \vec{\boldsymbol{t}}$ with respect to $\boldsymbol{C}$;

(Hrnd) $\boldsymbol{C}$ captures p randomness;

(Hwf) $\boldsymbol{C}$ is well-formed relatively to $\vec{\boldsymbol{u}}, x, \boldsymbol{\varphi}$;

all with respect to $\mathbb{M}[x \mapsto \mathbb{1}_{a_1}^{\eta}] : \mathcal{E}(x : \tau)$.

Proof of (1) By $\mathsf{enum}(\tau)$, we know that the size of l and the size of all its elements are bounded by polynomial in η, because p_τ is a PTIME programs with no inputs. By (Hadv) the cost of p is bounded by a polynomial $P(\eta + |X|)$ which itself is bounded by $P(\eta + m)$ with $m = \mathsf{max}(|a_i|)$. Then, the cost of all the operation under the while loop is bounded by a polynomial in $\eta + m$. Echoing remarks in Section 3.3.4, we have that p' is PTIME.

We conclude with Lemma 18 and (Hadv) that p' is a PTIME adversary.

Proof of (2) It is immediate with (Hflw) by invariant on the while loop on l.

Proof of (3) First, let $\mathsf{p}''_{\{\lambda,\exists,\forall\}}$ be the following program. Let $i \in \{0,1\}$, $\rho\ \mathcal{R}^{\eta}_{C_i,\mathbb{M}}\ \mathfrak{p}$ be tapes and let μ be a memory such that $\mathbb{M} : \mathcal{E}, \eta, \rho, \mu \models^{A} \varphi_i$.

```
1  X ← head l;
2  X_f , X_t ← p;
3  l ← tail l;
4  X_res ← update_{λ,∃,∀} X X_t X_res;
```

By (Hcomp) and assuming $\mathsf{update}_{\{\lambda,\exists,\forall\}}$ is interpreted as expected, we have that when $[\![f]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = \mathsf{true}$, $\mu(\mathsf{b}) = i$, and the head element of $\mu(l)$ is a :

$$(\!|\mathsf{p}''_w|\!)^{\eta,\mathfrak{p}}_{\mathbb{M}:\mathcal{E},\mu}[X_{res}] = \begin{cases} \mu(X_{res})[a \mapsto [\![t_i]\!]^{\eta,\rho}_{\mathbb{M}[x\mapsto a]:\mathcal{E}(x:\tau)}] & \text{if } w = \lambda \\ \mu(X_{res}) \vee [\![t_i]\!]^{\eta,\rho}_{\mathbb{M}[x\mapsto a]:\mathcal{E}(x:\tau)} & \text{if } w = \exists \\ \mu(X_{res}) \wedge [\![t_i]\!]^{\eta,\rho}_{\mathbb{M}[x\mapsto a]:\mathcal{E}(x:\tau)} & \text{if } w = \forall \end{cases}$$

Then, we have then

$$(\!|\mathsf{p}'_w|\!)^{\eta,\mathfrak{p}}_{\mathbb{M}:\mathcal{E},\mu}[X_{res}] = \begin{cases} [a_1 \mapsto [\![t_i]\!]^{\eta,\rho}_{\mathbb{M}[x\mapsto a_1]:\mathcal{E}(x:\tau)}, \cdots, a_n \mapsto [\![t_i]\!]^{\eta,\rho}_{\mathbb{M}[x\mapsto a_n]:\mathcal{E}(x:\tau)}] & \text{if } w = \lambda \\ \vee_{j\in[0,n]} [\![t_i]\!]^{\eta,\rho}_{\mathbb{M}[x\mapsto j_i]:\mathcal{E}(x:\tau)} & \text{if } w = \exists \\ \wedge_{j\in[0,n]} [\![t_i]\!]^{\eta,\rho}_{\mathbb{M}[x\mapsto j_i]:\mathcal{E}(x:\tau)} & \text{if } w = \forall \end{cases}$$

and ending the proof is immediate.

Proof of (4) Again, let p'' be the same program, and let $i \in \{0,1\}$, $\rho\ \mathcal{R}^{\eta}_{C_i,\mathbb{M}}\ \mathfrak{p}$ be tapes and let μ be a memory such that $\mathbb{M} : \mathcal{E}, \eta, \rho, \mu \models^{A} \varphi_i$.

Assume $\mu(\mathsf{b}) = i$, and the head element of $\mu(l)$ is some a_j, then by (Hrnd) we have that the execution of p'' relies on local samplings $L^j_{\$}$ and global samplings $G^j_{\$}$ such that

$$\begin{aligned} G^j_{\$} &\subseteq \{\ O_{\mathbb{M},\eta}(\mathsf{n}, a) \mid \langle \mathsf{n}, a, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},v} \rangle \in \mathcal{N}^{\eta,\rho}_{c,\mathbb{M}[x\mapsto a_j]},\ c \in C_i\ \} \\ L^j_{\$} &\subseteq \{\ O_{\mathbb{M},\eta}(\mathsf{n}, a) \mid \langle \mathsf{n}, a, \mathsf{T}^{\mathsf{loc}}_{\mathsf{G}} \rangle \in \mathcal{N}^{\eta,\rho}_{c,\mathbb{M}[x\mapsto a_j]},\ c \in C_i\ \} \end{aligned}$$

We end the proof with the fact that the program p relies on the local sampling $L_{\$}$ and global sampling $G_{\$}$ such that

$$G_{\$} = \cup_{j\in[1,n]} G^j_{\$}$$
$$L_{\$} = \cup_{j\in[1,n]} L^j_{\$}$$

Proof of (5) Immediate with Lemma 9. □

Rule Induction

Proof. Let p' be the program below.

```
1 X_t ← p_t;
2 l ← p_{τ_b} X_t;
3 X_res ← ();
4 while l ≠ [] do
5   X ← head l;
6   X_f, X_t ← p X X_res;
7   l ← tail l;
8   X_res[X] ← X_t;
9 return X_f, if X_f then X_res
```

Let $\mathbb{M}$ be a model such that $\mathbb{M} \models \Theta \tilde{\wedge} \text{Valid}(\prod_{(x:\tau)} \boldsymbol{C})$. Let show that

1. p' is a PTIME adversary against $\mathcal{G}$;
2. p' flows from $\boldsymbol{\varphi}^0$ to $\boldsymbol{\mathcal{I}}_{\leq}(\boldsymbol{t})$ w.r.t. $\boldsymbol{C}^0 \cdots \prod_{(x:\tau)} \boldsymbol{C}$;
3. p' computes $\vec{\boldsymbol{u}} \triangleright (v\ t \mid \boldsymbol{f})$;
4. $\prod_{(x:\tau)} \boldsymbol{C}$ captures p' randomness; and
5. $\prod_{(x:\tau)} \boldsymbol{C}$ is well-formed relatively to $\vec{\boldsymbol{u}}, \boldsymbol{\varphi}^0$.

Let $\mathbb{M}$ ba a model.
Then, for all $a_i \in [\![\tau_{\mathsf{b}}]\!]^{\eta}_{\mathbb{M}}$, we have $\mathbb{M}[x \mapsto \mathbb{1}^{\eta}_{a_1}] : \mathcal{E}(x:\tau) \models \text{Valid}(\boldsymbol{C})$, and for all $a_i \in [\![\tau]\!]^{\eta}_{\mathbb{M}}$,

(Hadv) p is an PTIME adversary against $\mathcal{G}$;

(Hflw) p flows from $\boldsymbol{\mathcal{I}}_{<}(x)$ to $\boldsymbol{\psi}$ w.r.t. $\boldsymbol{C}$,$\vec{\boldsymbol{u}}, x(\lambda(y:\tau_{\mathsf{b}}).(v\ y \mid y < x) \mid \boldsymbol{f})$;

(Hcomp) p computes $\vec{\boldsymbol{u}}, x, (\lambda(y:\tau_{\mathsf{b}}).(v\ y \mid y < x) \mid \boldsymbol{f}) \triangleright (v\ x \mid \boldsymbol{f} \wedge x \leq \boldsymbol{t})$ w.r.t. $\boldsymbol{C}$;

(Hrnd) $\boldsymbol{C}$ captures p randomness;

(Hwf) $\boldsymbol{C}$ is well-formed relatively to $\vec{\boldsymbol{u}}, x, (\lambda(y:\tau_{\mathsf{b}}).(v\ y \mid y < x) \mid \boldsymbol{f})$;

all with respect to $\mathbb{M}[x \mapsto \mathbb{1}^{\eta}_{a_1}] : \mathcal{E}(x:\tau)$.

Proof of (1) We know that the size of l and the size of all its elements are bounded by a constant. By (Hadv) the cost of p is PTIME and echoing remarks in Section 3.3.4, we have that p' is PTIME.

We conclude with Lemma 18 and (Hadv) that p' is a PTIME adversary.

The rest of the proof relies on the fact that since l has a value independent of η and ρ, the while loop can be inlined and we reuse results of the Transitivity rule. □

Chapter 6

Automation

Contents

6.1 Motivating example: an abstract mixnet **108**
6.2 Preliminary definitions **111**
6.2.1 Standard library 111
6.2.2 Assertion logic 113
6.3 Basic simulator synthesis **118**
6.3.1 Synthesis queries 118
6.3.2 Synthesis query rules 119
6.3.3 The basic simulator synthesis procedure 122
6.4 Inductive simulator synthesis **127**
6.4.1 Invariant synthesis 127
6.4.2 Example 129
6.4.3 Soundness 131

In the previous chapters, we established a theoretical framework for bideduction. However, applying our fine-grained proof system to cryptographic protocols by hand is impractical, especially when aiming to scale to actual protocols. To tackle this issue, one path is to automate the proofs.

In this chapter, we present our approach: a fully automated proof search procedure, designed with the idea to implement it within the Squirrel proof assistant. Our choice of full automation is driven by several considerations. First, our primary goal is to validate the approach experimentally; thus, developing a tactic-oriented proof search at this stage would be premature, due to a lot of added interface coding work. Second, by embedding our proof search in Squirrel, which is a tactic-based proof assistant, we retain the ability to manually simplify and structure proof goals before invoking the automated search which preserves part of the advantages of human guidance. Thirdly, our aim is also to replace legacy cryptographic tactics, which are automatic, so we aim at implementing an automatic tactic with our automation which is more natural for users of Squirrel to use.

This chapter is structured as follows: we give in Section 6.1 a motivating example which concretely shows the level of complexity we aim for our proof search: memoizing

```
game G_b = {            (* defines the games G_b, for b ∈ {0,1} *)
  sk ←$; ℓ ← [];        (* initializes sk and ℓ *)
  oracle pub() := { return (pub sk) }
  oracle left-right(m_0, m_1) := { r ←$;
                                   e ← enc m_b r (pub sk);
                                   ℓ ← e :: ℓ;
                                   return (if |m_0| = |m_1| then e else 0) }
  oracle decrypt(c) := { return (if c ∉ ℓ then dec c sk else 0) }}.
```

$x \leftarrow e$ assigns x to the value of e; and $x \xleftarrow{\$}$ assigns to x a randomly sampled value using a distribution based on the type of x.

Figure 6.1: The CCA2 cryptographic game.

simulators with time-sensitive memory invariants; we define our program library and assertion logic in Section 6.2; we present in Section 6.3 the basic proof-search procedure for bideduction, i.e. the component without induction; and we introduce in Section 6.4 our inductive proof-search technique.

6.1 Motivating example: an abstract mixnet

We present first a small but representative example drawn from our most recent and extensive case study. This example encapsulates what we aim for our automation to support.

> The automation strategy presented here is not our initial attempt. Our first attempt at automation relied on fixed-point computation [59]. It actually proves to work for simple examples but quickly appears insufficient for our needs. Indeed, this approach is blind to the temporal order, i.e. sequencing of protocol actions, and so it lacked the expressivity required to handle protocols such as NSL, as introduced in Chapter 1. In fact, these limitations seem to arise whenever CCA2 axiom is involved.

So, we illustrate the key difficulties encountered when synthesizing simulators for the *FOO protocol* [44]. FOO uses unspecified anonymous channel, that can instantiated by mixnets [63, 64], to achieve vote privacy. We use a high-level abstract modelling of decryption mixnets (in the style of [27]), which works in two phases. In the collection phase, voters send encrypted ballots to the mixnet, using its public key (pub k). The ballots are collected by a mixnet sub-process, which decrypts them and stores them into the ballot-box bb. Here, we consider a simple setting with a single honest voter V and dishonest voters are not explicitly modelled as we let the adversary play for them. Further, we let the adversary control V's vote, which V thus inputs from the network. After the collection phase, the mixnet sub-process P shuffles the ballot-box bb containing decrypted ballots, and publishes it.

```
system MixNet =
  new r; new k |                (* sample r and the secret key k *)
  Pk: out(c, pub k) |                              (* publish the public key pub k *)
  V: in(c,x); out(c, enc diff(x,dummy) r (pub k)) |    (* voter send its ballot *)
  M: !_i in(c,y);                                (* decrypt ballots and store them in bb *)
        bb(i) := if V < M i &&
                    y = enc diff(input@V,dummy) r (pub k)
                 then input@V else dec y k) |
  P: out(c, shuffle(bb)).                        (* publish stored decrypted ballots *)
```

Figure 6.2: An abstract mixnet protocol.

Modelling. We define in Figure 6.1 the CCA2 games for our encryption scheme, where: (pub k) is the public key associated to a private key k; (enc m r pk) is the encryption of plaintext m using public key pk and randomness r; (dec c k) is the decryption of ciphertext c using private key k. The log ℓ prevents the trivial attack in which the left-right challenge is sent to the decryption oracle.[1] The CCA2 assumption states that a probabilistic polynomial-time adversary with access to the game's oracles has a negligible probability of distinguishing whether it is interacting with $\mathcal{G}_0$ or $\mathcal{G}_1$.

We describe our abstract mixnet protocol using Squirrel's process algebra in Figure 6.2. We actually define *two* protocols through two process variants obtained by projecting the **diff** operators to their first or second component. The process obtained by projecting the **diff** operators to their first component corresponds to our abstract mixnet setting. We modify the output of the mixnet sub-process M using a conditional that will be instrumental when reasoning with the CCA2 game, but that does not change M's behavior: indeed, (input@V = dec y k) if (y = enc (input@V) r (pub k)). The process obtained by taking the second projection differs in that V will encrypt a dummy message instead of its input, and M will "magically" retrieve V's input when receiving that encryption. This variant is an *idealized* version of our protocol.

Cryptographic reduction. It should be quite obvious that an adversary will not be able to distinguish whether it is interacting with the real or idealized version of the protocol, assuming that it feeds V with an input that has the same length as dummy.

To formally show that our protocol is indistinguishable from its idealized version, it suffices to show there exists a CCA2 adversary that can simulate our protocols. More formally, we want to show the following *simplified* bideduction judgement:

$$(\mathcal{E}, t_0 : \mathsf{timestamp}), \boldsymbol{C}, (\boldsymbol{\varphi}, \boldsymbol{\psi}) : \emptyset \triangleright \mathbf{frame@}\boldsymbol{t_0}$$

for a valid constraint system $\boldsymbol{C}$ and assertion $\boldsymbol{\varphi}$ capturing the initial memory of CCA2.

For this example, we restrict to a non-adaptive setting in which the adversary must interact with the protocol along a fixed trace. We define the following timestamps variables, to capture time points of the protocols, such that a timestamp is an element of the data-type:

$$\mathsf{t} ::= \mathsf{init} \mid \mathsf{Pk} \mid \mathsf{V} \mid \mathsf{M(i)} \mid \mathsf{P}$$

For example, V denotes the timepoint at which the honest voter will send its encrypted ballot, and M(i) the timepoint at which the mixnet with session identifier i collects,

[1] This is actually the extension with the decryption oracle of IND-CPA game mentioned in Chapter 1.

```
1  frame, output, input ← [ ⊥ for t ∈ [ init; t0 ] ]  (* initialize arrays *)
2  bb ← [ ⊥ for i ∈ index ]                (* one more array initialization *)
3  ballotV ← None;                  (* to memoize V's encrypted ballot *)
4  input[init], output[init], frame[init] ← empty  (* initial timepoint*)
5  (* recursively compute input, output, frame and bb *)
6  for each t ∈ ]init; t0 ] {
7    input[t] ← att(frame[pred t]) (* the adversary computes the input *)
8    begin   (* simulate output[t] by case analysis on t *)
9      match t with
10     | Pk → output[t] ← G.pub()
11     | V →
12       ballotV ← Some (G.left-right(input[t], dummy)) (* memoize *)
13       output[t] ← Option.get(ballotV)
14     | M(i) →
15       (* in the condition below, we retrieve V's ballot from ballotV *)
16       if V < M(i) && input[t] = Option.get(ballotV)
17       then { bb[i] ← input[V] } (* bypass decryption oracle *)
18       else { bb[i] ← G.decrypt(input[t]) } (* safe decryption oracle call *)
19       output[t] ← empty (* no output there *)
20     | P → output[t] ← shuffle(bb)
21   end
22   frame[t] ← ⟨ frame[pred t], output[t] ⟩ (* add output to the frame *)
23 }
24 return (frame[t0])
```

We have Option.get (Some x) = x **and** Option.get(None) = ⊥**.**

Figure 6.3: Reduction to the CCA2 assumption.

decrypts and stores a ballot — we use the abstract type index to denote session identifiers. Timestamps are ordered by $<$, and for $\mathsf{t} \neq \mathsf{init}$, we let (pred t) denote the timestamp preceding t w.r.t. $<$.

Figure 6.3 shows a simulator $\mathcal{S}$ witnessing that the real and ideal mixnet protocols are indistinguishable up-to some timepoint t_0 by reduction to the CCA2 game (noted $\mathcal{G}$). On line 1, the simulator $\mathcal{S}$ initializes a number of timestamp-indexed arrays to store intermediate values: output[t] and input[t] store, resp, the output and input of the protocol at timepoint t; frame[t] is the sequence of all outputs from init to t included. The array cell bb[i] (line 2) will store the decrypted ballot processed by M(i). Finally, ballotV is initialized to None (line 3), and will be used to memoize V's ballot.

The simulator's main loop (lines 7–23) iterates over the timestamps in the trace]init; t_0]. For each such timestamp t, the input at time t is obtained (line 7) as the result of an attacker computation att($\cdot$) — modeled as an arbitrary unspecified procedure — taking all previous outputs as arguments: input[t] = att(frame[pred(t)]). Then, the next output is simulated (lines 8–21) depending on which action is considered. For t = V, we can call the **left-right** oracle — we assume that len (input[V]) = len dummy. For t = M(i), we need to distinguish whether V has occurred before or not.

If V has not occurred before M(i), then the log ℓ is empty and we can decrypt any message computed by the simulator, including input[M(i)]: it can thus simulate output[M(i)] easily.

Otherwise, V < M(i), and we cannot decrypt output[V] because it has been obtained from the **left-right** oracle and is thus in the game's log ℓ. Fortunately, output[M(i)] is written in such a way that this forbidden decryption is avoided. Note, however, that the simulator

needs to test whether input[M(i)] = enc **diff**(input[V],dummy) r (pub k) = output[V], thus it needs to use again the result of the call to **left-right** performed in V — calling the oracle again at this point would yield an encryption with a different random seed. Instead, our simulator exploits the fact that Option.get(ballotV) = output[V] when V < M(i), i.e. the voter's encrypted ballot has been memoized in ballotV.[2]

Finally, the sequence of all outputs up-to timepoint t is computed and stored in frame[t] (line 22).

The above analysis shows two key features (absent from [59]) needed for our simulator. First, we need simulators that **memoize the result of oracle calls** across recursive calls: the value of an oracle call performed in V must be reused later for M(i). Second, proving the correctness of such a simulator requires an invariant that tracks the value of the game's state (here, the log ℓ) after each step of the simulator's recursive process. Crucially, **time-sensitive invariants** are necessary, to express in our example that the log is empty before V but contains one element after it. Without such an invariant we would have to show that input[M(i)] ≠ enc **diff**(input[V],dummy) r k for the **else** branch, which is unnecessary.

6.2 Preliminary definitions

In this section, we first set up a standard library for our programs. We then explain how we use abstract interpretation to instantiate our assertion logic, and we introduce the abstract operations that go with it. The rest of the thesis — automation and implementation work — is based on this standard library and assertion logic.

6.2.1 Standard library

We present the assumptions we make on our standard library $\mathcal{L}$ to support pattern-matching and usual operation on messages. In particular, to support pattern-matching, we define algebraic data type, and destructor symbols. This solely serves to have a rigorous framework to match on timestamps, afterward.

Algebraic data-types. An *algebraic data-type declaration* (ADT) is of the form:

$$\begin{aligned} \tau ::= c_1 : \vec{\tau}_1 &\to \tau \\ \mid c_2 : \vec{\tau}_2 &\to \tau \\ &\dots \\ \mid c_n : \vec{\tau}_n &\to \tau \end{aligned}$$

$c_1, \dots, c_n$ are the *constructors* of the declaration, and each constructor c_i takes arguments of type $\vec{\tau}_i$.

We use an axiomatic approach, and see an ADT declaration as a convenient way of assuming the existence of a number of symbol declarations and axioms in the standard $\mathcal{L}$.

[2] In our simple example, we could get rid of ballotV and use output[V] instead. However, oracle calls are generally not directly used as outputs in protocols, which requires the use of memoization variables as done here.

First, we assume that τ is a base type, and that $\mathcal{L}$ contains the symbol declarations:

$$c_1 : \vec{\tau}_1 \to \tau \qquad \ldots \qquad c_n : \vec{\tau}_n \to \tau$$

Then, we require that the constructors $c_1, \ldots, c_n$ form of partition of τ, i.e. the following formulas must be axioms:

$$\forall x : \tau.\ \bigvee_{1 \le i \le n} \exists \vec{y} : \vec{\tau}_i.\ x = c_i\ \vec{y}$$
$$\bigwedge_{i \neq j} \forall \vec{y}_i : \vec{\tau}_i.\ \vec{y}_j : \vec{\tau}_j.\ c_i\ \vec{y}_i \neq c_j\ \vec{y}_j$$

We assume the existence of destructor symbols that allows to destruct x into one of its constituent constructor:

$$\mathsf{head} : \tau \to \mathsf{int} \qquad d_1 : \tau \to \vec{\tau}_1 \qquad \ldots \qquad d_n : \tau \to \vec{\tau}_n$$

We require that destructors and constructors behave as expected using the following axioms:

$$\bigwedge_i \forall \vec{y}.\ \mathsf{head}(c_i\ \vec{y}) = i \qquad \bigwedge_i \forall \vec{y}.\ d_i(c_i\ \vec{y}) = \vec{y}$$

Finally, we assume that the constructors and destructors are all poly-time:

$$\bigwedge_i \mathsf{adv}(c_i) \wedge \bigwedge_i \mathsf{adv}(d_i) \wedge \mathsf{adv}(\mathsf{head})$$

Pattern matching. For any ADT τ with constructors

$$c_1 : \vec{\tau}_1 \to \tau \qquad \ldots \qquad c_n : \vec{\tau}_n \to \tau$$

we assume the existence of a pattern-matching construct $(\mathsf{match}_\tau\ \cdot\ \mathsf{with}\ \cdot)$ symbol over τ with type:

$$\mathsf{match}\ \cdot\ \mathsf{with}\ \cdot : \tau \to (\vec{\tau}_i \to \tau_o)_{1 \le i \le n} \to \tau_o$$

We use the following usual notation for match terms:

$$\mathsf{match}\ t\ \mathsf{with}\ (c_i\ \vec{i} \mapsto u_i)_{1 \le j \le n} \stackrel{\mathsf{def}}{=} \mathsf{match}\ t\ \mathsf{with}\ (\lambda \vec{i}.\ u_i)_{1 \le j \le n}$$

Finally, we require that the interpretation of the match constructs is fixed, and satisfies the following axioms:

$$\left(\mathsf{match}\ c_i\ \vec{a}\ \mathsf{with}\ (u_j)_{1 \le j \le n}\right)_{1 \le i \le n} = u_i\ \vec{a}.$$

Sets of messages. We assume a (base) type message set and the following symbols:

$$\mathsf{make}_{\vec{\tau}} : (\vec{\tau} \to (\mathsf{message} * \mathsf{bool})) \to \mathsf{message\ set} \qquad \subseteq : \mathsf{message\ set} \to \mathsf{message\ set} \to \mathsf{bool}$$

$$\cup : \mathsf{message\ set} \to \mathsf{message\ set} \to \mathsf{message\ set}$$

$$\_\_::; : \mathsf{message} \to \mathsf{message\ set} \to \mathsf{message\ set} \qquad \mathsf{empty} : \mathsf{message\ set}$$

where we have one symbol $\mathsf{make}_{\vec{\tau}}$ for each sequence of type $\vec{\tau}$. As expected, $(\cdot :: \cdot)$ adds an element to a set, $\cup$ is set union, empty the empty set, etc. Further, $\mathsf{make}_{\vec{\tau}}$ is a set builder where $\mathsf{make}_{\vec{\tau}}\ (\lambda\vec{\alpha}.(t, f))$ represents the set of all terms t for any value of $\vec{\alpha}$ such that f holds. We use the following nicer notation for set builders:

$$\{t \mid \vec{\alpha} : f\} \stackrel{\mathsf{def}}{=} \mathsf{make}_{\vec{\tau}}\ (\lambda\vec{\alpha}.(t, f))$$

where $\vec{\alpha}$ are of type $\vec{\tau}$. All these symbols have the expected semantics. Notably, if $\{t \mid \vec{\alpha} : f\}$ is well-typed in $\mathcal{E}$ for any model $\mathbb{M}$ of $\mathcal{E}$, η and ρ, we have that:

$$[\![\{t \mid \vec{\alpha} : f\}]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}} = \bigcup_{\vec{a} \in [\![\vec{\tau}]\!]^{\eta}_{\mathbb{M}}} \begin{cases} [\![t]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto\vec{a}]} & \text{if } [\![f]\!]^{\eta,\rho}_{\mathbb{M}[\vec{\alpha}\mapsto\vec{a}]} = \mathsf{true}. \\ \emptyset & \text{otherwise} \end{cases}.$$

Terms VS expressions. In the rest of this section, we will mix terms and expressions

Note that expressions follow the same structure as terms. The differences are, first, that terms include lambda constructors, which expressions do not; and more importantly, they are interpreted in different environments. Expression variables are interpreted as program variables, using a memory, whereas term variables are interpreted within an environment.

Thus, another way to view expressions is as terms but that are in a reduced subset of terms. This is the view we will adopt for the remainder of the thesis.

In particular:

- any function on terms can indistinguishably act on terms and expressions.
- for any expression u, and substitution σ of variables toward terms, $u\sigma$ make sense.

6.2.2 Assertion logic

For our assertion logic, we use an abstract interpretation framework, capturing only *growing sets of terms*. Here, an assertion is an abstract memory that represents a set of concrete memories. We define in this section the abstract objects, the concretization and abstraction functions, as well as the abstract operations needed to manipulate these objects.

Symbolic sets

A symbolic set is a term $\{t \mid \vec{\alpha} : f\}$ of type message set. Also, a list of symbolic sets $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_n$ is a normalized representation of the term $\boldsymbol{s}_1 \cup \cdots \cup \boldsymbol{s}_n$ and the semantics of symbolic sets in naturally lifted to unions of symbolic sets.

Abstract memories

Abstract memories, usually written $\varphi, \psi, \ldots$, are finite maps from (program) variables to lists of symbolic sets of terms:

$$\varphi, \psi, \cdots \stackrel{\text{def}}{=} \epsilon \mid \varphi, (\ell \mapsto (s_1, \ldots, s_n))$$

where $\ell \in \mathcal{X}$ must be of type message set. An abstract memory φ may not contain two bindings for the same program variable ℓ, and we write $\varphi(\ell)$ the value of ℓ in φ. We write $\mathsf{dom}(\varphi)$ the domain of φ, i.e. $\mathsf{dom}(\epsilon) = \emptyset$ and $\mathsf{dom}(\varphi, (\ell \mapsto \ldots)) = \mathsf{dom}(\varphi) \cup \{\ell\}$.

Concretization. An abstract memory φ represents a set of concrete memories μ. Formally, this is captured by the satisfaction relation $\models_{\mathsf{A}}$, which is further parameterized by the model under consideration, the value of the security parameter, and a logical tape.

Definition 31. *Let φ be an abstract memory. Then, for any model $\mathbb{M}$, logical tape ρ and security parameter η, and program memory μ, we let*

$$\mathbb{M}, \eta, \rho, \mu \models_{A} \varphi$$

hold if and only if for any $x \in \mathsf{dom}(\varphi)$, $\mu(x) \subseteq [\![\varphi(x)]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$.
(Where we conflate the list of symbolic sets $\varphi(x) = s_1, \ldots, s_n$ and the term $s_1 \cup \cdots \cup s_n$.)

Abstract operations. We now equip abstract memories with a number of operations: inclusion, meet, join, etc.

First, given an environment $\mathcal{E}$, a model $\mathbb{M} : \mathcal{E}$ and two abstract memories φ, ψ well-typed in $\mathcal{E}$, we say that φ is included in ψ in $\mathbb{M}$, which we write $\mathcal{E}; \mathbb{M} \models_{\mathsf{A}} \varphi \sqsubseteq \psi$, iff.

$$\begin{aligned} &\mathsf{dom}(\varphi) = \mathsf{dom}(\psi) \\ \text{and } &\mathcal{E}; \mathbb{M} \models [\varphi(x) \subseteq \psi(x)]_{\mathrm{e}}. \qquad \text{(for all } x \in \mathsf{dom}(\varphi)) \end{aligned}$$

Then, we let $\mathcal{E}; \boldsymbol{\Theta} \models_{\mathsf{A}} \varphi \sqsubseteq \psi$ hold iff. $\mathcal{E}; \mathbb{M} \models_{\mathsf{A}} \varphi \sqsubseteq \psi$ for any model $\mathbb{M}$ such that $\mathcal{E}; \mathbb{M} \models_{\mathsf{A}} \boldsymbol{\Theta}$. Note that $\subseteq$ has the same definition than $\Rightarrow$ defined in Chapter 4. We rather use $\sqsubseteq$ than $\Rightarrow$ from now on since $\sqsubseteq$ is more natural when dealing with sets.

We write $\mathcal{E}; \boldsymbol{\Theta} \models_{\mathsf{A}} \varphi \dashv\models \psi$ if $\mathcal{E}; \boldsymbol{\Theta} \models_{\mathsf{A}} \varphi \sqsubseteq \psi$ and $\mathcal{E}; \boldsymbol{\Theta} \models_{\mathsf{A}} \psi \sqsubseteq \varphi$. Finally, we will omit $\mathcal{E}$ and $\boldsymbol{\Theta}$ when they are clear from context.

We consider the following (syntactic) meet and generalization operators on symbolic sets:

$$\begin{aligned} \{\boldsymbol{t} \mid \vec{\alpha} : \boldsymbol{f}\} \sqcap \boldsymbol{g} &\stackrel{\text{def}}{=} \{\boldsymbol{t} \mid \vec{\alpha} : \boldsymbol{f} \wedge \boldsymbol{g}\} \\ \forall x. \{\boldsymbol{t} \mid \vec{\alpha} : \boldsymbol{f}\} &\stackrel{\text{def}}{=} \{\boldsymbol{t} \mid x, \vec{\alpha} : \boldsymbol{f}\} \end{aligned}$$

We lift the operations $\cdot \sqcap \boldsymbol{g}$ and $\forall x. \cdot$ to lists of symbolic sets in the expected way, e.g.

$$(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_n) \sqcap \boldsymbol{g} \stackrel{\text{def}}{=} (\boldsymbol{s}_1 \sqcap \boldsymbol{g}, \ldots, \boldsymbol{s}_n \sqcap \boldsymbol{g}).$$

We also extend these operations to abstract memories, which we further equip with a join operator $\sqcup$:

$$\varphi \sqcap \boldsymbol{g} \stackrel{\text{def}}{=} \left(\ell \mapsto \varphi(\ell) \sqcap \boldsymbol{g}\right)_{\ell \in \mathsf{dom}(\varphi)}$$
$$\forall x.\varphi \stackrel{\text{def}}{=} \left(\ell \mapsto \forall x.\varphi(\ell)\right)_{\ell \in \mathsf{dom}(\varphi)}$$
$$\varphi \sqcup \psi \stackrel{\text{def}}{=} \left(\ell \mapsto \varphi(\ell), \psi(\ell)\right)_{\ell \in \mathsf{dom}(\varphi) \cap \mathsf{dom}(\psi)}$$

where in the $\sqcup$ rule we assume $\mathsf{dom}(\varphi) = \mathsf{dom}(\psi)$.

Properties. We summarize here a number of expected properties of our operations on abstract memories.

Proposition 5. *We have the structural properties:*

- $\sqcap$ *and* $\sqcup$ *are commutative and associative.*
- $\sqsubseteq$ *is transitive.*
- $\sqcap$ *distributes over* $\sqcup$*:*
$$(\varphi \sqcup \psi) \sqcap \boldsymbol{g} \dashv\models (\varphi \sqcap \boldsymbol{g}) \sqcup (\psi \sqcap \boldsymbol{g})$$
- $\forall$ *distributes over* $\sqcup$*:*
$$\forall x.\,(\varphi \sqcup \psi) \dashv\models (\forall x.\,\varphi) \sqcup (\forall x.\,\psi)$$
- $\sqcap$ *preserves* $\sqsubseteq$*:*
$$\varphi \sqsubseteq \psi \text{ implies } (\varphi \sqcap \boldsymbol{g}) \sqsubseteq (\psi \sqcap \boldsymbol{g})$$
- $\sqcup$ *preserves* $\sqsubseteq$*:*
$$\varphi_0 \sqsubseteq \psi_0 \text{ and } \varphi_1 \sqsubseteq \psi_1 \text{ implies } (\varphi_0 \sqcup \varphi_1) \sqsubseteq (\psi_0 \sqcup \psi_1)$$
- $\varphi \sqcap \top \dashv\models \varphi$
- *if* $\models [\boldsymbol{g_0} \Leftrightarrow \boldsymbol{g_1}]_{\mathrm{e}}$*, then* $\varphi \sqcap \boldsymbol{g_0} \dashv\models \varphi \sqcap \boldsymbol{g_1}$

We omit the details of the proof, which are straightforward.

We also have the following additional properties, one commuting $\forall$ and $\sqcap$, the other pushing a $\sqcup$ downward by turning it into a logical or $\vee$.

Proposition 6. *We have the commutation properties:*

- *(A)* $\forall x.\,(\varphi \sqcap \boldsymbol{g}) \dashv\models \varphi \sqcap (\exists x.\,\boldsymbol{g})$ *when* $x \notin \mathsf{fv}(\varphi)$*.*
- *(B)* $(\varphi \sqcap \boldsymbol{g}_0) \sqcup (\varphi \sqcap \boldsymbol{g}_1) \dashv\models \varphi \sqcap (\boldsymbol{g}_0 \vee \boldsymbol{g}_1)$

Again, we omit the details.

Finally, we lift abstract memories, the $\models^A$ definition and the above properties to bi-abstract memories, as explain in Chapter 4 and from now on drop the "bi" prefix. Also, in practice, we will always define bi-abstract memory through bi-symbolic sets, that we directly define by:

$$\boldsymbol{\varphi}, \boldsymbol{\psi}, \cdots \stackrel{\text{def}}{=} \epsilon \mid \boldsymbol{\varphi}, (\ell \mapsto (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_n))$$

(Note that, in this particular case, the left and right abstract memories have the same domain.)

Abstract set evaluation:

$$
\begin{aligned}
\mathsf{eval}_{\varphi}(x) &\stackrel{\mathsf{def}}{=} \begin{cases} \lightning & \text{if } x \notin \mathsf{dom}(\varphi) \\ s_1 \cup \cdots \cup s_n & \varphi(x) = s_1, \ldots, s_n \end{cases} \\
\mathsf{eval}_{\varphi}(s_0 \cup s_1) &\stackrel{\mathsf{def}}{=} \mathsf{eval}_{\varphi}(s_0) \cup \mathsf{eval}_{\varphi}(s_1) \\
\mathsf{eval}_{\varphi}(t :: s) &\stackrel{\mathsf{def}}{=} \{t \mid \emptyset, \top\} \\
&\quad \cup\ \mathsf{eval}_{\varphi}(s) \\
\mathsf{eval}_{\varphi}(\mathsf{empty}) &\stackrel{\mathsf{def}}{=} \epsilon \\
\mathsf{eval}_{\varphi}(s) &\stackrel{\mathsf{def}}{=} \lightning \qquad \text{(if no other rule applies)}
\end{aligned}
$$

Abstract boolean evaluation:

$$
\begin{aligned}
\mathsf{b\text{-}eval}_{\varphi}(t \notin s) &\stackrel{\mathsf{def}}{=} \bigwedge_{(\vec{\alpha}, t', f) \in \mathsf{eval}_{\varphi}(s)} \forall \vec{\alpha}.\, f \Rightarrow (t \neq t') \\
\mathsf{b\text{-}eval}_{\varphi}(f \wedge f') &\stackrel{\mathsf{def}}{=} \mathsf{b\text{-}eval}_{\varphi}(f) \wedge \mathsf{b\text{-}eval}_{\varphi}(f') \\
\mathsf{b\text{-}eval}_{\varphi}(f \vee f') &\stackrel{\mathsf{def}}{=} \mathsf{b\text{-}eval}_{\varphi}(f) \vee \mathsf{b\text{-}eval}_{\varphi}(f') \\
\mathsf{b\text{-}eval}_{\varphi}(\top) &\stackrel{\mathsf{def}}{=} \top \\
\mathsf{b\text{-}eval}_{\varphi}(\bot) &\stackrel{\mathsf{def}}{=} \bot \\
\mathsf{b\text{-}eval}_{\varphi}(f) &\stackrel{\mathsf{def}}{=} \lightning \qquad \text{(if no other rule applies)}
\end{aligned}
$$

Note that failure $\lightning$ propagates up-ward, e.g. $\mathsf{eval}_{\varphi}(s)$ fails if any sub-term of s fails to abstractly evaluate.

Figure 6.4: Abstract evaluation functions.

Abstract Evaluation

Given an abstract memory state φ and an expression s of type message set, the abstract evaluation $\mathsf{eval}_{\varphi}(s)$ of s in φ is either a term of type message set, or $\lightning$ if the evaluation failed. It is defined in Figure 6.4.

Given an abstract memory state φ and an expression f of type bool, the abstract evaluation $\mathsf{b\text{-}eval}_{\varphi}(f)$ of f in φ is either a logical term of type bool, or $\lightning$ if the evaluation failed. It is also defined in Figure 6.4.

The abstract evaluation functions are sound approximations of the evaluated expressions. This is captured by the following property, where: i) states that $\mathsf{eval}(\cdot)$ is a sound over-approximation of sets of bit-strings; and ii) states that $\mathsf{b\text{-}eval}(\cdot)$ is a sound boolean under-approximation.

Proposition 7. *Let s and f be expressions of type, resp., message set and bool such that*

$$\mathsf{eval}_{\varphi}(s) \neq \lightning \qquad \textit{and} \qquad \mathsf{b\text{-}eval}_{\varphi}(g) \neq \lightning .$$

Then, for any model $\mathbb{M}$, logical tape $\rho = (\rho_a, \rho_h)$ and security parameter η, side i, for any μ such that $\mathbb{M}, \eta, \rho, \mu \models_A \varphi$, we have:

- *i) the set inclusion* $[s]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu} \subseteq [\![\mathsf{eval}_{\varphi}(s)]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$
- *ii) if* $[\![\mathsf{b\text{-}eval}_{\varphi}(g)]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$ *then* $[g]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu}$

where $\mathfrak{p}$ *is the program tape filled with zeroes but for the sub-tape* $\mathtt{T_A}, \mathsf{bool}$*, which is equal to* ρ_a*; and where* $\vec{\tau}_{\vec{\alpha}}$ *are the type of* $\vec{\alpha}$*.*

Proof. The proof is by structural induction over the term s, and directly uses Definition 31 in the variable case. □

Remark 1. *In practice, we use a more complex boolean abstract evaluation function* $\mathsf{b\text{-}eval}(\cdot)$ *that also computes an over-approximation of a boolean expression* g*. Concretely, reusing the notations of Proposition 7, we have* $\mathsf{b\text{-}eval}_{\varphi}(g) = (g_\bot, g_\top)$ *where* $g_\bot, g_\top$ *are logical formulas such that:*

- *if* $[g]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu}$ *then* $[\![g_\top]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$*;*
- *if* $[\![g_\bot]\!]^{\eta,\rho}_{\mathbb{M}:\mathcal{E}}$ *then* $[g]^{\eta,\mathfrak{p}}_{\mathbb{M},i,\mu}$*.*

Computing an under-approximation allows to support more boolean program, notably negation $\neg$*, which simply swaps* $g_\top$ *and* $g_\bot$*.*

The abstract evaluations of an expression boast a stability property, in the sense that if they succeed (i.e. do not return $\lightning$) on some abstract memory $\boldsymbol{\varphi}$, then they will succeed on any abstract memory $\boldsymbol{\psi}$ which has the same domain as $\boldsymbol{\varphi}$. Formally:

Proposition 8. *For all abstract memories* $\boldsymbol{\varphi}, \boldsymbol{\psi}$ *such that* $\mathsf{dom}(\boldsymbol{\varphi}) = \mathsf{dom}(\boldsymbol{\psi})$*:*

- *for any expression* s*,* $\mathsf{eval}_{\varphi}(s) \neq \lightning$ *if and only if* $\mathsf{eval}_{\psi}(s) \neq \lightning$*;*
- *for any expression* f*,* $\mathsf{b\text{-}eval}_{\varphi}(f) \neq \lightning$ *if and only if* $\mathsf{b\text{-}eval}_{\psi}(f) \neq \lightning$*.*

Proof. This is an immediate induction over the definition of, resp., $\mathsf{eval}(\cdot)$ and $\mathsf{b\text{-}eval}(\cdot)$. □

The abstract evaluation function is monotonous w.r.t. the $\sqsubseteq$ ordering on abstract memories.

Proposition 9. *Let* $\mathcal{E}$ *be an environment* $\mathcal{E}$ *and* $\boldsymbol{\Theta}$ *some hypotheses. For all abstract memories* $\boldsymbol{\varphi}, \boldsymbol{\psi}$ *and expression* $\boldsymbol{t}$ *w.r.t.* $\mathcal{E}$*, if* $\mathsf{eval}_{\varphi}(\boldsymbol{t}) \neq \lightning$ *then:*

$$\text{if } \mathcal{E}; \boldsymbol{\Theta} \models \boldsymbol{\varphi} \sqsubseteq_A \boldsymbol{\psi} \quad \text{then} \quad \mathcal{E}; \boldsymbol{\Theta} \models \mathsf{eval}_{\varphi}(\boldsymbol{t}) \subseteq \mathsf{eval}_{\psi}(\boldsymbol{t})$$

Proof. This is an immediate induction over the definition of $\mathsf{eval}(\cdot)$. □

Proposition 10. *Let* $\mathcal{E}$ *be an environment. For all same-domain abstract memories* $\boldsymbol{\varphi}, \boldsymbol{\varphi}_0$ *w.r.t.* $\mathcal{E}$ *and expression* $\boldsymbol{t}$*, if* $\mathsf{eval}_{\varphi}(\boldsymbol{s}) \neq \lightning$ *then:*

$$\mathcal{E} \models \mathsf{eval}_{\varphi \sqcup \varphi_0}(\boldsymbol{s}) \subseteq \mathsf{eval}_{\varphi}(\boldsymbol{s}) \cup \bigcup_{x \in \mathsf{vars}(\boldsymbol{s})} \boldsymbol{\varphi}_0(x)$$

Proof. We show this by induction over the definition of $\mathsf{eval}(\cdot)$. In the variable case, we have:

$$\begin{aligned}\mathsf{eval}_{\varphi \sqcup \varphi_0}(x) &= (\varphi \sqcup \varphi_0)(x) \\ &= \varphi(x) \cup \varphi_0(x) \\ &= \mathsf{eval}_{\varphi}(x) \cup \varphi_0(x)\end{aligned}$$

The empty case is trivial. In the union case, we conclude easily using the induction hypothesis:

$$\begin{aligned}&\mathsf{eval}_{\varphi \sqcup \varphi_0}(s_0 \cup s_1) \\ =\ &\mathsf{eval}_{\varphi \sqcup \varphi_0}(s_0) \cup \mathsf{eval}_{\varphi \sqcup \varphi_0}(s_1) \\ =\ &\mathsf{eval}_{\varphi}(s_0) \cup \textstyle\bigcup_{x \in \mathsf{vars}(s_0)} \varphi_0(x) \cup \mathsf{eval}_{\varphi}(s_1) \cup \bigcup_{x \in \mathsf{vars}(s_1)} \varphi_0(x) \\ =\ &\mathsf{eval}_{\varphi}(s_0 \cup s_1) \cup \textstyle\bigcup_{x \in \mathsf{vars}(s_0) \cup \mathsf{vars}(s_1)} \varphi_0(x) \\ =\ &\mathsf{eval}_{\varphi}(s_0 \cup s_1) \cup \textstyle\bigcup_{x \in \mathsf{vars}(s_0 \cup s_1)} \varphi_0(x)\end{aligned}$$

(Above, we abuse notations and write $a \subseteq b$ instead of $\mathcal{E} \models a \subseteq b$.)
The $t :: s$ case similarly follows from the induction hypothesis. □

6.3 Basic simulator synthesis

We present a basic simulator synthesis procedure based on bideduction, that generates simulators *without loops or recursion* — it does not use the induction rule of bideduction.

We split the description of the basic synthesis procedure in several steps: first, we present *synthesis queries* (Section 6.3.1), which describe the inputs, outputs, and the specification of the synthesis main loop; then, we design low-level atomic steps of the simulator synthesis procedure through *synthesis query rules* (Section 6.3.2); finally, we present heuristics and orchestrate synthesis query rules to obtain our simulator synthesis procedure (Section 6.3.3).

6.3.1 Synthesis queries

The rules of Chapter 5 provide basic building blocks for deriving valid simulators but, in order to obtain a synthesis procedure, we need to determine when and how to use each rule. We first clarify what parts of a bideduction judgement are, respectively, *inputs* and *outputs* of the simulator synthesis procedure. To this effect, we use *synthesis queries* of the following form:

$$\mathcal{E}; \boldsymbol{\Theta}_i; \boldsymbol{C}_i; \varphi \vdash \boldsymbol{in} \triangleright_{\mathcal{G}} \boldsymbol{out} \rightsquigarrow (\boldsymbol{\Theta}_o; \boldsymbol{C}_o; \psi; \boldsymbol{w})$$

The components on the left of $\rightsquigarrow$ are *inputs* of the synthesis procedure, while components on the right of $\rightsquigarrow$ are *outputs*. For the sake of clarity, we coloured inputs in black and outputs in dark red in the query above[3]. This query is valid whenever the corresponding bideduction judgement is valid:

$$\mathcal{E}; \boldsymbol{\Theta}_i, \boldsymbol{\Theta}_o; \boldsymbol{C}_i \cdot \boldsymbol{C}_o; (\varphi, \psi) \vdash \boldsymbol{in} \triangleright_{\mathcal{G}} \boldsymbol{out}, \boldsymbol{w}$$

[3]To ensure accessibility, colours only encode redundant information.

Said otherwise, our synthesis procedure takes as inputs a set of hypotheses $\boldsymbol{\Theta}_i$ and initial constraints $\boldsymbol{C}_i$, a pre-condition $\boldsymbol{\varphi}$ on the state of game $\mathcal{G}$, the simulator's input $\boldsymbol{in}$ and its target output $\boldsymbol{out}$. It attempts to synthesize a simulator $\mathcal{S}$ that computes $\boldsymbol{out}$ when given $\boldsymbol{in}$ as inputs, and returns $\mathcal{S}$'s randomness constraints $\boldsymbol{C}_o$, the resulting post-condition $\boldsymbol{\psi}$, further hypotheses $\boldsymbol{\Theta}_o$ that must hold for $\mathcal{S}$ to be correct, and additional terms $\boldsymbol{w}$ that $\mathcal{S}$ computed while computing $\boldsymbol{out}$. The extra hypotheses $\boldsymbol{\Theta}_o$ are proof obligations that will be discharged to the user at the end of the simulator synthesis, together with a formula expressing the validity of the combined constraints $\boldsymbol{C}_i \cdot \boldsymbol{C}_o$.

The additional outputs $\boldsymbol{w}$ are called *memoization hints*, and will allow to re-use the result of oracle calls across recursive iterations of our final recursive simulators (see next Section 6.4). They will be added to inputs of further synthesis queries. To distinguish them from standard inputs, the inputs of synthesis queries are split into two sequences, noted $\boldsymbol{in}.\texttt{std}$ and $\boldsymbol{in}.\texttt{memo}$ for, resp., standard and memoization inputs. We may still use $\boldsymbol{in}$ as a single sequence when the distinction is irrelevant, e.g. $\boldsymbol{t} \in \boldsymbol{in}$ means that $\boldsymbol{t}$ belongs to either of the two sequences.

6.3.2 Synthesis query rules

We design rules for deriving synthesis queries, that provide a higher-level and more operational variant of the bideduction proof system. The validity of synthesis query rules can be established by combining several bideduction rules to derive the validity of the conclusion query from that of the premises.

We make use of a few standard automated reasoning utilities.

Normalization. We assume a weak head normalization function $\mathsf{whnf}^{\Theta}_{\mathcal{E}}(t)$. In practice, we only normalize modulo the definitions of $\mathcal{E}$ and some basic equations of Θ, but any normalization function ensuring $\mathcal{E}; \Theta \vdash [t = \mathsf{whnf}^{\Theta}_{\mathcal{E}}(t)]_{\mathrm{e}}$ is correct. Because our normalization only relies on a builtin part of Θ, we omit that component for brevity.

Unification. We also use unification: if u and v are terms well-typed in $(\mathcal{E}, \vec{x} : \vec{\tau})$, then $\mathsf{unify}^{\mathcal{E}}_{\vec{x}}(u = v)$ is a partial procedure that may return a substitution σ mapping a subset of $\vec{x}$ to well-typed terms in $\mathcal{E}$, such that $\mathcal{E}; \Theta \vdash [u\sigma = v\sigma]_{\mathrm{e}}$. We actually use unification on bi-terms, with the natural specification. This loose specification may be met by various unification procedures; in practice we use a simple one which only exploits definitions in $\mathcal{E}$ and basic equations from Θ.

The full set of rules used to present our basic simulator synthesis procedure is described in Figures 6.5, 6.6 and 6.7. We provide here a description of a selected but representative set of rules.

The Unreach rule. The Unreach rule is to be used when the term to be deduced is under an infeasible condition. The negation of the condition is discharged to the user, hence it is important to use this rule only as a last resort in an automatic synthesis context: otherwise, an invalid proof obligation might render the whole synthesis useless. **The Load rule.** An opposite strategy is used in Load: there, we attempt to instantiate an input $\lambda\vec{x}.(\boldsymbol{u} \mid \boldsymbol{g})$ to obtain the desired output $(\boldsymbol{o} \mid \boldsymbol{f})$; we determine a possible value for $\vec{x}$ by unification, and we verify *automatically* that under this instantiation, $\boldsymbol{g}$ is implied by $\boldsymbol{f}$ (this is denoted $\vdash_{\mathsf{auto}}$); it then only remains to verify that the values of $\vec{x}$ can themselves

Core rules. *In the* Trans *and* Case *rules,* $\overline{C}$ *denotes the sub-multiset of* C *where constraints with tag* $\mathsf{T}^{\mathsf{loc}}_{\mathsf{G}}$ *are removed; the remaining constraints may be duplicated without any impact on the constraints' validity.*

$$\textsc{Unreach}\ \frac{}{\mathcal{E};\Theta;C;\varphi \vdash in \triangleright (o \mid f) \rightsquigarrow ([\neg f]_{\mathrm{e}};\emptyset;\varphi;\epsilon)} \qquad \textsc{Conv}\ \frac{in' = \mathsf{whnf}^{\Theta}_{\mathcal{E}}(in) \qquad out' = \mathsf{whnf}^{\Theta}_{\mathcal{E}}(out) \qquad \mathcal{E};\Theta;C;\varphi \vdash in' \triangleright out' \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\varphi \vdash in \triangleright out \rightsquigarrow (\Theta';C';\psi;w)}$$

$$\textsc{Trans}\ \frac{\mathcal{E};\Theta;C;\varphi \vdash in \triangleright (o' \mid f) \rightsquigarrow (\Theta';C';\psi';w') \qquad \mathcal{E};\Theta;\overline{C}\cdot\overline{C'};\psi' \vdash in,(o' \mid f) \triangleright (o'' \mid f) \rightsquigarrow (\Theta'';C'';\psi'';w'')}{\mathcal{E};\Theta;C;\varphi \vdash in \triangleright (o',o'' \mid f) \rightsquigarrow (\Theta',\Theta'';C'\cdot C'';\psi'';w',w'')}$$

$$\textsc{Case}\ \frac{\begin{array}{c}\mathcal{E};\Theta;C;\varphi \vdash in \triangleright (c \mid f) \rightsquigarrow (\Theta_c;C_c;\psi_c;w_c) \\ \mathcal{E};\Theta;\overline{C}\cdot\overline{C_c};\psi_c \vdash in \triangleright (o \mid f \wedge c) \rightsquigarrow (\Theta_\top;C_\top;\psi_\top;w_\top) \\ \mathcal{E};\Theta;\overline{C}\cdot\overline{C_c}\cdot\overline{C_\top};\psi_c \vdash in \triangleright (o \mid f \wedge \neg c) \rightsquigarrow (\Theta_\bot;C_\bot;\psi_\bot;w_\bot) \\ \Theta' = \Theta_c,\Theta_\top,\Theta_\bot \qquad C' = C_c\cdot C_\top\cdot C_\bot \qquad w' = w_c,w_\top,w_\bot \\ \psi' = (\psi_\top \sqcap c) \sqcup (\psi_\bot \sqcap \neg c)\end{array}}{\mathcal{E};\Theta;C;\varphi \vdash in \triangleright (o \mid f) \rightsquigarrow (\Theta';C';\psi';w')}$$

$$\textsc{Oracle}\ \frac{\begin{array}{c}\sigma = \mathsf{unify}^{\mathcal{E}}_{\vec{x},\vec{y},\vec{z}}(o = o_f) \qquad \sigma(\vec{y}) = (\mathsf{k_v}\ p_{\mathsf{v}})_{\mathsf{v}\in\mathcal{G}.\mathsf{glob}_\$} \qquad \sigma(\vec{z}) = (\mathsf{r_v}\ s_{\mathsf{v}})_{\mathsf{v}\in f.\mathsf{loc}_\$} \\ \mathcal{E};\Theta;C;\varphi \vdash in \triangleright \left(\sigma(\vec{x}),(p_{\mathsf{v}})_{\mathsf{v}\in\mathcal{G}.\mathsf{glob}_\$},(s_{\mathsf{v}})_{\mathsf{v}\in f.\mathsf{loc}_\$} \mid g\right) \rightsquigarrow (\Theta';C';\varphi';w) \\ C'' = \left(\Pi_{\mathsf{v}\in\mathcal{G}.\mathsf{glob}_\$}(\emptyset,\mathsf{k_v},o_{\mathsf{v}},\mathsf{T}^{\mathsf{glob}}_{\mathsf{G},\mathsf{v}},g)\right)\cdot\left(\Pi_{\mathsf{v}\in f.\mathsf{loc}_\$}(\emptyset,\mathsf{r_v},s_{\mathsf{v}},\mathsf{T}^{\mathsf{loc}}_{\mathsf{G}},g)\right) \\ g_\mu = \text{b-eval}_{\varphi'}(c_f\,\sigma) \qquad \psi = \mathsf{post}^{\sigma}_{f}(\varphi')\end{array}}{\mathcal{E};\Theta;C;\varphi \vdash in \triangleright (o \mid g) \rightsquigarrow (\Theta',[g \Rightarrow g_\mu]_{\mathrm{e}};C'\cdot C'';\psi;w)}$$

Figure 6.5: Basic proof-search core rules.

be simulated. This rule requires that the implication has been verified, hence there is no risk of abusive applications as with Unreach.

The Oracle rule. The query synthesis rule for oracle calls, Oracle, is an effective version of $\textsc{Oracle}_f$ that relies on the following assumptions on oracle f:

- The body of f (in both sides of the game) is a sequence of random samplings of $f.\mathsf{loc}_\$$, followed by assignations and a final return statement of the form

  $$(\textbf{return}\ \mathsf{if}\ c_f\ \mathsf{then}\ o_f\ \mathsf{else}\ \mathsf{dummy}).$$

  The interesting result o_f is returned under a condition c_f; otherwise an irrelevant constant from $\mathcal{L}$ is returned.

- We further assume that the expression o_f does not contain memory locations: it may only refer to the oracle's inputs, and to local and global samplings.

We let $\vec{x}$ be the oracle's input variables. We also let $\vec{y} = \mathcal{G}.\mathsf{glob}_\$$ and $\vec{z} = f.\mathsf{loc}_\$$ for brevity. The assignation statements in f may refer to the logical variables $\vec{x},\vec{y},\vec{z}$ in addition to

Destruction rules.

$$\textsc{Name}\ \frac{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (o \mid f) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (\mathsf{n}\ o \mid f) \rightsquigarrow (\Theta';C' \cdot (\emptyset,\mathsf{n},o,\mathtt{T}_S,f);\psi;w)}$$

$$\textsc{FA.Quant}\ \frac{Q \in \{\forall,\exists,\lambda\} \qquad \mathtt{enum}_{\mathtt{poly}}(\tau) \qquad \mathcal{E},x:\tau;\varphi \vdash in \rhd (o \mid f) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (Q(x:\tau).\ o \mid f) \rightsquigarrow (\forall x.\Theta',\Pi_x.C';\forall x.\psi;\lambda x.w)}$$

$$\textsc{FA.ite}\ \frac{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (g \mid f),(o_0 \mid f \wedge g),(o_1 \mid f \wedge \neg g) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (\mathsf{if}\ g\ \mathsf{then}\ o_0\ \mathsf{else}\ o_1 \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

$$\textsc{FA.}\wedge\ \frac{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (g_0 \mid f),(g_1 \mid f \wedge g_0) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (g_0 \wedge g_1 \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

$$\textsc{FA.}\Rightarrow\ \frac{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (g_0 \mid f),(g_1 \mid f \wedge g_0) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (g_0 \Rightarrow g_1 \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

$$\textsc{FA.}\vee\ \frac{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (g_0 \mid f),(g_1 \mid f \wedge \neg g_0) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (g_0 \vee g_1 \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

$$\textsc{FA}\ \frac{s \in \mathcal{L} \qquad \mathcal{E};\Theta;C;\varphi \vdash in \rhd (o \mid f) \rightsquigarrow (\Theta';C';\psi;w)}{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (s\ o \mid f) \rightsquigarrow (\Theta';C';\psi;w)}$$

$$\textsc{FA.match}\ \frac{\begin{array}{c} t \in in.\mathtt{std} \\ \text{for any } i,\quad \mathcal{E},\vec{x}_i;\Theta;C;\varphi \vdash in,\vec{x}_i \rhd (u_i \mid f \wedge t = c_i\ \vec{x}_i) \rightsquigarrow (\Theta_i;C_i;\psi_i;w_i) \\ \Theta_{\mathsf{out}} = (\forall \vec{x}_i.\Theta_i)_i \qquad C_{\mathsf{out}} = \prod_i \forall \vec{x}_i.C_i \\ \psi_{\mathsf{out}} = \bigsqcup_i (\forall \vec{x}_i.\psi_i \sqcap (t = c_i\ \vec{x}_i)) \qquad w_{\mathsf{out}} = (\forall \vec{x}_i.w_i)_i \end{array}}{\mathcal{E};\Theta;C;\varphi \vdash in \rhd (\mathsf{match}\ t\ \mathsf{with}\ (c_i\ \vec{x}_i \mapsto u_i)_i \mid f) \rightsquigarrow (\Theta_{\mathsf{out}};C_{\mathsf{out}};\psi_{\mathsf{out}};w_{\mathsf{out}})}$$

Figure 6.6: Basic proof-search destructive rules.

program variables, i.e. memory locations. Note that, although conditionals are not allowed in the oracle's body, they still can be used inside the expressions in assignations. Although limited, this format is met by the CCA2 game and all cryptographic games that we have encountered so far.

Given an initial abstract memory φ and an assignation $\ell \leftarrow e$ of a message set memory location, we can abstractly evaluate e and the resulting memory, to obtain the updated abstract memory. This can be chained for all assignations in the body of f. We write $\mathsf{post}_f^{\sigma}(\varphi \mid g)$ the obtained abstract memory where σ is a substitution of domain $\vec{x},\vec{y},\vec{z}$. This defines a valid post-condition for a call to f with the values given by σ in a context where g holds and the memory satisfies φ.

Our Oracle rule proceeds as follows. First, it does not attempt to syntactically match the term to be computed with the oracle's return expression: instead, it matches it with o_f, and discharges a proof obligation which ensures that c_f holds when the oracle is called. We thus unify o_f and o to determine the values of $\vec{x},\vec{y},\vec{z}$. As before, the values of $\vec{y},\vec{z}$ must

Memory rule. *(For any type τ, we require that* $(\mathsf{witness}_\tau : \tau) \in \mathcal{L}$. *We write* witness *when the type τ is clear from the context.)*

LOAD

$$\frac{\begin{array}{c}\lambda\vec{x}.(\boldsymbol{u} \mid \boldsymbol{g}) \in \boldsymbol{in}.\mathtt{std} \\ \boldsymbol{\sigma} = \mathsf{unify}^{\mathcal{E}}_{\vec{x}}(\boldsymbol{u} = \boldsymbol{o}) \\ \boldsymbol{\sigma}_0 = \boldsymbol{\sigma}[\vec{x}\backslash\mathsf{dom}(\boldsymbol{\sigma}) \mapsto \mathsf{witness}] \\ \mathcal{E}; \boldsymbol{\Theta} \vdash_{\mathsf{auto}} [\boldsymbol{f} \Rightarrow \boldsymbol{g}\boldsymbol{\sigma}_0]_{\mathrm{e}} \\ \mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi} \vdash \boldsymbol{in} \triangleright (\vec{x}\boldsymbol{\sigma}_0 \mid \boldsymbol{f}) \rightsquigarrow (\boldsymbol{\Theta}'; \boldsymbol{C}'; \boldsymbol{\psi}; \boldsymbol{w})\end{array}}{\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi} \vdash \boldsymbol{in} \triangleright (\boldsymbol{o} \mid \boldsymbol{f}) \rightsquigarrow (\boldsymbol{\Theta}'; \boldsymbol{C}'; \boldsymbol{\psi}; \boldsymbol{w})}$$

Memoization rules.

MEMOIZE.STORE

$$\frac{\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi} \vdash \boldsymbol{in} \triangleright \boldsymbol{out} \rightsquigarrow (\boldsymbol{\Theta}'; \boldsymbol{C}'; \boldsymbol{\psi}; \boldsymbol{w})}{\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi} \vdash \boldsymbol{in} \triangleright \boldsymbol{out} \rightsquigarrow (\boldsymbol{\Theta}'; \boldsymbol{C}'; \boldsymbol{\psi}; \boldsymbol{w}, \boldsymbol{out})}$$

MEMOIZE.LOAD

$$\frac{\begin{array}{c}\lambda\vec{x}.(\boldsymbol{u} \mid \boldsymbol{g}) \in \boldsymbol{in}.\mathtt{memo} \\ \boldsymbol{\sigma} = \mathsf{unify}^{\mathcal{E}}_{\vec{x}}(\boldsymbol{u} = \boldsymbol{o}) \qquad \vec{y} = \vec{x}\backslash\mathsf{dom}(\boldsymbol{\sigma}) \\ \mathtt{enum}_{\mathtt{poly}}(\mathsf{type}_{\mathcal{E}}(\vec{y})) \\ \mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi} \vdash \boldsymbol{in} \triangleright \big(\boldsymbol{o} \mid \boldsymbol{f} \wedge (\forall\vec{y}.\ \neg\boldsymbol{g}\boldsymbol{\sigma})\big) \rightsquigarrow (\boldsymbol{\Theta}'; \boldsymbol{C}'; \boldsymbol{\psi}; \boldsymbol{w})\end{array}}{\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi} \vdash \boldsymbol{in} \triangleright (\boldsymbol{o} \mid \boldsymbol{f}) \rightsquigarrow (\boldsymbol{\Theta}'; \boldsymbol{C}'; \boldsymbol{\psi}; \boldsymbol{w})}$$

Figure 6.7: Basic proof-search memory rules.

be names, and the name indices as well as the arguments $\theta(\vec{x})$ must be bideducible.[4] The oracle is (implicitly) called with the abstract memory $\boldsymbol{\varphi}'$ obtained after that bideduction, and only when $\boldsymbol{g}$ holds — which implies $\boldsymbol{g}_\mu$ and then $\boldsymbol{c}_f$. The final abstract memory is $\mathsf{post}^{\sigma}_f(\boldsymbol{\varphi}' \mid \boldsymbol{g})$. Overall, our synthesis rule is justified using ORACLE$_f$, relying on the Hoare triple

$$\{\boldsymbol{\varphi}', \boldsymbol{g} \wedge \boldsymbol{g}_\mu\}\boldsymbol{out} \leftarrow O_f(\boldsymbol{\sigma}(\vec{x}))[\boldsymbol{\sigma}(\vec{y}); \boldsymbol{\sigma}(\vec{z})]\{\mathsf{post}^{\sigma}_f(\boldsymbol{\varphi}' \mid \boldsymbol{g})\}$$

which is is valid under the assumption $[\boldsymbol{g} \Rightarrow \boldsymbol{g}_\mu]_{\mathrm{e}}$.

6.3.3 The basic simulator synthesis procedure

Our basic synthesis procedure $\mathsf{synthesize}_\triangleright(\cdot)$ is a (recursive) function which takes the left part of a synthesis query (its inputs) and attempts to derive it by applying synthesis rules following a particular strategy. Upon success, it returns the right part of the synthesis query (its outputs). Our procedure is such that

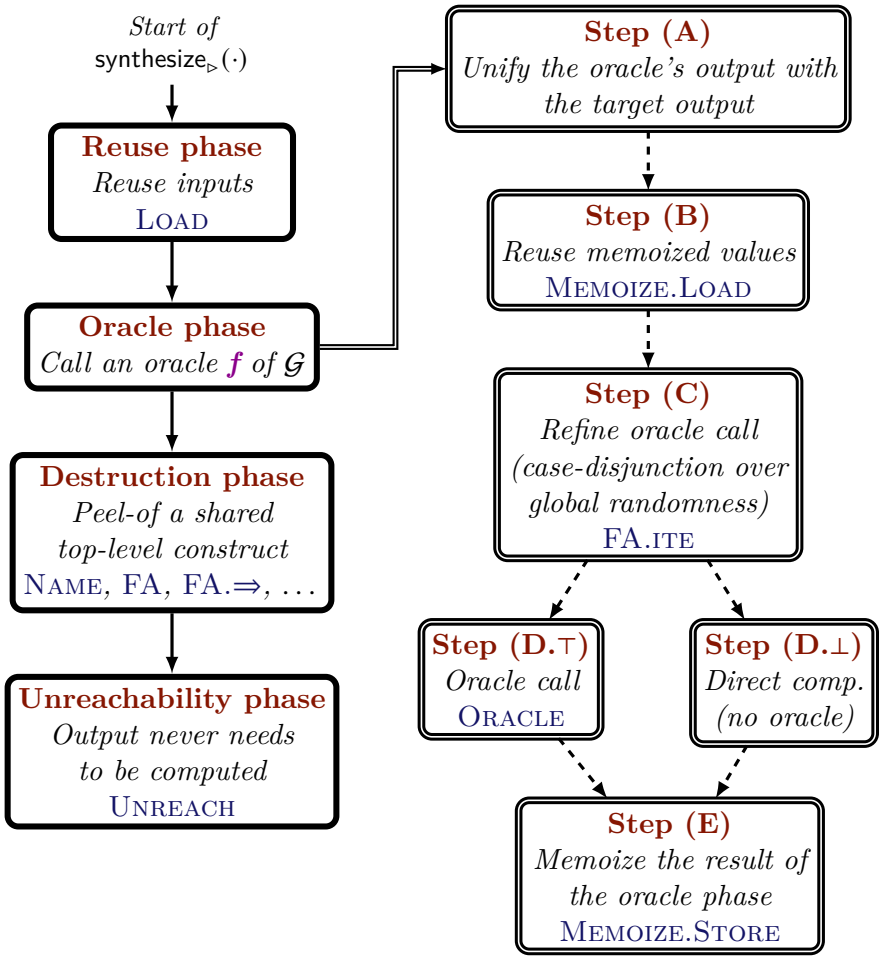
$$\begin{array}{ll}\text{if} & \mathsf{synthesize}_\triangleright(\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi} \vdash \boldsymbol{in} \triangleright \boldsymbol{out}) = (\boldsymbol{\Theta}'; \boldsymbol{C}'; \boldsymbol{\psi}; \boldsymbol{w}) \\ \text{then} & \mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi} \vdash \boldsymbol{in} \triangleright \boldsymbol{out} \rightsquigarrow (\boldsymbol{\Theta}'; \boldsymbol{C}'; \boldsymbol{\psi}; \boldsymbol{w}) \text{ is derivable.}\end{array}$$

The procedure has four different phases, as depicted in Figure 6.8 and described below. The phases are applied successively until one succeeds. A successful phase will usually generate bideduction premises, which are themselves resolved recursively by the procedure. Our procedure never backtracks: if a phase succeeds but generates invalid bideduction subgoals, the procedure will fail without trying the next phases. This makes the procedure less complete but more *efficient* and *predictable*. This latter property is crucial in a fully automated setting, as it helps the user construct an intuitive understanding of why the procedure failed and how this may be fixed.

The phases are, in the order in which they are attempted:

- The *reuse phase* checks if the target output $\boldsymbol{out}$ can be directly obtained from the inputs $\boldsymbol{in}.\mathtt{std}$.
- The *oracle phase* tries to obtain $\boldsymbol{out}$ by calling one of the oracles of the game.

[4] The samplings variables that were left free after the unification are ignored.



Phases are in boxes with a simple border. Steps of the oracle phase are in boxes with a double border. Phase/step names are in dark red. Each box includes the main rule they rely on, when applicable. Simple arrows indicates progression between phases (continue in case of failure). Dashed arrow indicates progression between steps of the oracle phase (continue in case of success). All phases and steps may generate new bideduction premises, which are resolved recursively by synthesize▷(·).

Figure 6.8: Control-flow of synthesize▷(·).

- The *destruction phase* checks if the left and right components of $\boldsymbol{out}$ start with the same top-level construct, which is then evaluated by the simulator being synthesized. The arguments of this construct are recursively synthesized by the procedure.

- Last, the *unreachable phase* let the simulator abandon the synthesis by asking the user to prove that output $\boldsymbol{out}$ never needs to be evaluated.

We detail each phase below.

Reuse phase

The **reuse phase** checks if the target $\boldsymbol{out}$ can be directly obtained from standard inputs. This phase only uses the Load rule, attempting to use it on all terms from $\boldsymbol{in}.\mathtt{std}$. Concretely, it scans the inputs and tries to apply Load on each term $\lambda\vec{x}.\,(\boldsymbol{u} \mid \boldsymbol{g})$ in the inputs $\boldsymbol{in}.\mathtt{std}$.[5]

Checking whether Load applies is fully automated and only requires a unification and a call to the automated reasoning solver $\vdash_{\mathsf{auto}}$. In case of success, it generates a bideduction premise that is resolved recursively by the synthesis procedure.

> Because this phase greedily uses the first relevant input, it could in principle make a wrong decision and cause the overall synthesis to fail. We never encountered this issue in practice.

Oracle phase

The oracles of a cryptographic game are stateful and probabilistic functions. Thus, when we call an oracle f, we are: i) modifying the internal state of the game; and ii) bideduction must add constraints indicating that we want to couple the randomness of the oracle with randomness of the logical target term. Both behaviours can lead to invalid simulators if we apply the oracle rule too greedily: in the first case, the modified internal state may prevent us from using another oracle later, e.g. because we wrongly added a value to a log; in the second case, the new oracle constraints may clash with other constraints existing or to come. In the oracle phase, we try to avoid such issues with two different techniques. First, we *reuse memoized values* as much as possible to avoid recalling an oracle that was already called (which would introduce contradictory constraints when the oracle has local randomness). Second, we automatically do a *case-analysis* to exclude cases that would add constraints that clash with the input constraints system.

Assume we want to answer the partial synthesis query $\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi} \vdash \boldsymbol{in} \triangleright_{\mathcal{G}} (\boldsymbol{o} \mid \boldsymbol{f})$ using

[5] We take the convention that when $\vec{x}$ is empty, $\lambda\vec{x}.\boldsymbol{t} \stackrel{\mathsf{def}}{=} \boldsymbol{t}$. With this convention, we can try to apply Load on any terms in the inputs $\boldsymbol{in}$.

the game:

```
game G = {
  k ←$; ℓ ← [];
  oracle o(x⃗) := {
    r ←$;
    ℓ_old ← ℓ;
    ℓ ← u :: ℓ;
    return (if c_f then o_f else 0) }
  ... (* other oracles *)
}
```

We describe how the oracle phase handles the oracle $\mathbf{o}$. For the sake of simplicity, the game we presented above considers a restricted oracle that only uses a single global sampling k, a single global state ℓ, etc. Generalizing this to an arbitrary oracle is straightforward.

We further assume that the program variable ℓ does not occur in the branching condition $\boldsymbol{c}_f$: this is w.l.o.g., since ℓ_{old} may occur, and $\ell = u :: \ell_{\mathsf{old}}$ at that program point. (Doing so allows evaluating $\boldsymbol{c}_f$ in the pre-condition $\boldsymbol{\varphi}$, which simplifies the presentation.)

Step (A). In step (A), the returned value (if $\boldsymbol{c}_f$ then $\boldsymbol{o}_f$ else 0) is split between the condition $\boldsymbol{c}_f$ and the value $\boldsymbol{o}_f$. Then, we unify the oracle output with the target term by computing $\boldsymbol{\sigma} = \mathsf{unify}^{\mathcal{E}}_{\vec{x},k,r}(\boldsymbol{o}_f = \boldsymbol{o})$. This substitution $\boldsymbol{\sigma}$ tells us what arguments $\vec{x}$ should be sent to $\mathbf{o}$, and how to map the game's randomness k and r to logical names. More precisely, we split $\boldsymbol{\sigma}$ into $\boldsymbol{\sigma}_{\mathsf{args}}, \{k \mapsto \mathsf{k}\ t_k\}, \{r \mapsto \mathsf{r}\ t_r\}$, and we require that k and r are logical names — otherwise, the oracle phase on oracle $\mathbf{o}$ fails. Further, we require that no program variables appear in $\boldsymbol{o}_f$, but for k, r and oracle inputs: this ensures that once we are done instantiating $\mathbf{o}$'s arguments $\vec{x}$ and the randomness offsets for k and r, the output $\boldsymbol{o}_f$ can be seen as a purely logical term that can be thus injected in the logical bideduction judgements.

Step (B). The memoization step (B) reuses memoized values by applying the Memoize.Load rule on all possible values. Concretely, this refines the cases in which $\mathbf{o}$ must be called by changing the target output $(\boldsymbol{o} \mid \boldsymbol{f})$ into $(\boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}})$ — for example, we can have $\boldsymbol{f}_{\mathsf{memo}} = \neg \boldsymbol{g}$ if $(\boldsymbol{o} \mid \boldsymbol{g}) \in \boldsymbol{in}.\mathtt{memo}$, i.e. if we memoized $(\boldsymbol{o} \mid \boldsymbol{g})$ from a past oracle call.

Step (C). Step (C) refines $\boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}}$ further by exploiting the input constraints $\boldsymbol{C}$. It looks in $\boldsymbol{C}$ for a constraint of the form $(\emptyset, \mathsf{k}, t'_k, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},k}, \dots)$, and does on case-disjunction on $\boldsymbol{g}_f \overset{\mathsf{def}}{=} (t'_k = t_k)$. Indeed, in case $t'_k \neq t_k$, applying the oracle rule would add a constraint $(\emptyset, \mathsf{k}, t_k, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},k}, \dots)$ which is incompatible with the existing constraint $(\emptyset, \mathsf{k}, t'_k, \mathsf{T}^{\mathsf{glob}}_{\mathsf{G},k}, \dots)$ — as adding this constraint would invalidate the whole proof. Concretely, the case-disjunction replaces $(\boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}})$ with

$$(\mathsf{if}\ \boldsymbol{g}_f\ \mathsf{then}\ \boldsymbol{o}\ \mathsf{else}\ \boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}})$$

which is then decomposed using the FA.ite rule. This yields two main bideduction subgoals,

$$(\boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}} \wedge \boldsymbol{g}_f) \quad \text{and} \quad (\boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}} \wedge \neg \boldsymbol{g}_f)$$

which are dealt with by, resp., the steps **(D.⊤)** and **(D.⊥)**.

Steps (D). **(D.⊥)** simply does a recursive call to the synthesis procedure bypassing the first oracle phase, to avoid infinite loops. **(D.⊤)** calls the ORACLE rule and must thus deal with the game's memory. First, it computes the post-condition $\boldsymbol{\psi}$ by applying the update $(\ell \leftarrow u :: \ell)$ to the pre-condition $\boldsymbol{\varphi}$:

$$\boldsymbol{\psi} \overset{\text{def}}{=} \boldsymbol{\varphi}\{\ell \mapsto \mathsf{eval}_{\boldsymbol{\varphi}}(u\sigma[\mathsf{b} \mapsto \#(0;1)] :: \ell)\}$$

Second, it discharges the condition $\boldsymbol{c}_f$ as a novel proof-obligation (using the output part $\boldsymbol{\Theta}_0$ of the synthesis query). This is done by abstractly evaluating $\boldsymbol{c}_f$ in ψ to obtain a purely logical formula, i.e. we add $\mathsf{b\text{-}eval}_{\boldsymbol{\varphi}}(\boldsymbol{c}_f)$ to the outputted subgoals $\boldsymbol{\Theta}_0$. Note that this step may fail if either $\mathsf{eval}_{\boldsymbol{\varphi}}(u\sigma[\mathsf{b} \mapsto \#(0;1)]) = \lightning$ or $\mathsf{b\text{-}eval}_{\boldsymbol{\varphi}}(\boldsymbol{c}_f) = \lightning$.

Step (E). Finally, step **(E)** memoizes the result using the MEMOIZE.STORE rule to avoid recomputing it later. Importantly, it does not memoize the term computed by the *oracle rule*, by rather the term computed by the whole *oracle phase*. That is, it memoizes $(\boldsymbol{o} \mid \boldsymbol{f})$ and not

$$(\boldsymbol{o} \mid \boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}} \wedge \boldsymbol{g}_f). \tag{6.1}$$

From a precision point-of-view, memoizing the latter would be sufficient. Indeed, when $\boldsymbol{f}_{\mathsf{memo}}$ or $\boldsymbol{g}_f$ do not hold, we can safely re-run the (sub-)simulators to re-compute the value of interest — note that we cannot re-run the simulator for (6.1), as it makes a stateful call to the oracle that cannot be safely replayed. But memoizing the more general term $(\boldsymbol{o} \mid \boldsymbol{f})$ is sound, more efficient (as proving that $\boldsymbol{f}$ holds using $\vdash_{\mathsf{auto}}$ is simpler than $\boldsymbol{f} \wedge \boldsymbol{f}_{\mathsf{memo}} \wedge \boldsymbol{g}_f$), and yields simpler proof-obligations in practice.

Destruction phase

The **destruction phase** checks if the left and right components of $\boldsymbol{out}$ start with the same top-level construct, which is then computed by the simulator being synthesized. The arguments of this construct are recursively synthesized by the procedure. This phase comes after the reuse and oracle phase, meaning that the target output could not be directly computed. In that case, the synthesis procedure checks if $\boldsymbol{out}$ is of the form $\#(\Delta\ \vec{\mathsf{out}}_0; \Delta\ \vec{\mathsf{out}}_1)$ where Δ is a term construct, i.e. either a symbol $s \in \mathcal{X}$ or a lambda λx. Crucially, Δ must be identical on both sides, e.g. it could be a symbol s but cannot be a pair of *distinct* symbols $\#(s_0; s_1)$. Then, we try to apply a destruction rule for Δ (see Figure 6.6). There is a single destruction rule for each possible Δ, except in the symbol case s where there are specialized rules that exploits particular properties of some builtin library functions.

Specialized rules such as FA.⇒ are always prioritized over the generic FA rule. Finally, we use the CONV rule beforehand to put the left and right components of $\boldsymbol{out}$ in *weak-head normal form*, which helps to apply the destruction rules more often. Applying this phase too early would often lead to invalid simulators: an obvious example of this would be to use FA on an encryption when it is necessary to obtain the encryption through the corresponding oracle of the CCA game.

INDUCTION

$$\frac{\begin{array}{c} \boldsymbol{t} \in \boldsymbol{in}.\mathtt{std} \qquad \boldsymbol{f} \in \boldsymbol{in}.\mathtt{std} \\ (\mathcal{E}, x : \tau); \boldsymbol{\Theta}; \overline{\boldsymbol{C}}; \boldsymbol{\mathcal{I}}_<(x) \vdash \boldsymbol{in}_{\mathsf{rec}} \rhd \big(\boldsymbol{u}\ x \mid \boldsymbol{f} \wedge x \leq \boldsymbol{t}\big) \rightsquigarrow (\boldsymbol{\Theta}_0; \boldsymbol{C}_0; \boldsymbol{\psi}; \boldsymbol{w}_0) \\ \mathcal{E}; \boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\varphi}_0 \sqsubseteq \boldsymbol{\mathcal{I}}_<(x_0) \qquad (\mathcal{E}, x : \tau); \boldsymbol{\Theta}, [x_0 < x]_{\mathsf{e}} \models_{\mathsf{A}} \boldsymbol{\mathcal{I}}_{\leq}(\mathsf{pred}\ x) \sqsubseteq \boldsymbol{\mathcal{I}}_<(x) \\ (\mathcal{E}, x : \tau); \boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\psi} \sqsubseteq \boldsymbol{\mathcal{I}}_{\leq}(x) \qquad < \in \mathcal{L}_{\mathsf{p}} \end{array}}{\mathcal{E}; \boldsymbol{\Theta}; \overline{\boldsymbol{C}}; \boldsymbol{\varphi}_0 \vdash \boldsymbol{in} \rhd (\boldsymbol{u}\ \boldsymbol{t} \mid \boldsymbol{f}) \rightsquigarrow (\forall x. \boldsymbol{\Theta}_0; \forall x. \boldsymbol{C}_0; \boldsymbol{\mathcal{I}}_{\leq}(\boldsymbol{t}); \lambda x. (\boldsymbol{w}_0 \mid x \leq \boldsymbol{t}))}$$

We require that $\boldsymbol{u} \equiv \lambda(x : \tau).\, \boldsymbol{u}_0$ where τ is a `fixed` and `finite` base-type. We assume a total order $<$ over τ. We let $\boldsymbol{in}_{\mathsf{rec}} \stackrel{\mathsf{def}}{=} (\boldsymbol{in}, x, \lambda y. (\boldsymbol{u}\ y \mid y < x))$. Let x_0 be the minimal element for $<$, and $\mathsf{pred}\ x$ the predecessor of x w.r.t. $<$ for any x (with the convention that $\mathsf{pred}\ x_0 = x_0$).

Figure 6.9: The time-sensitive induction rule.

Unreachable phase

Lastly, the **unreachable phase** lets the simulator abandon the synthesis by asking the user to prove that $\boldsymbol{out}$ never needs to be evaluated. It relies on UNREACH and is used only as a last resort, as it may produce invalid proof obligations. Moreover, it is always possible to postpone its application, so our last resort strategy cannot hurt.

More precisely, if $\boldsymbol{out}$ is $(\boldsymbol{u} \mid \boldsymbol{f})$, we ask to prove that $\boldsymbol{f}$ is never true by discharging $[\neg \boldsymbol{f}]_{\mathsf{e}}$ to the user.

6.4 Inductive simulator synthesis

In this section, we i) introduce the inductive synthesis, that embeds the basic synthesis within a larger procedure to infer inductive invariants, ii) illustrate it through an example and iii) establish the soundness of this procedure.

6.4.1 Invariant synthesis

Our procedure synthesizes a bideduction judgement whose output is a recursive term. Recall that, when performing induction, we distinguish between two types of invariants: memoizing invariants and memory invariants. Our procedure infers the invariants, and outputs subgoals and constraints system to synthesize a bideduction judgement, for which the first derivation rules is the INDUCTION rule of Chapter 5. We provide corresponding synthesis rules in Figure 6.9 of the former rule.

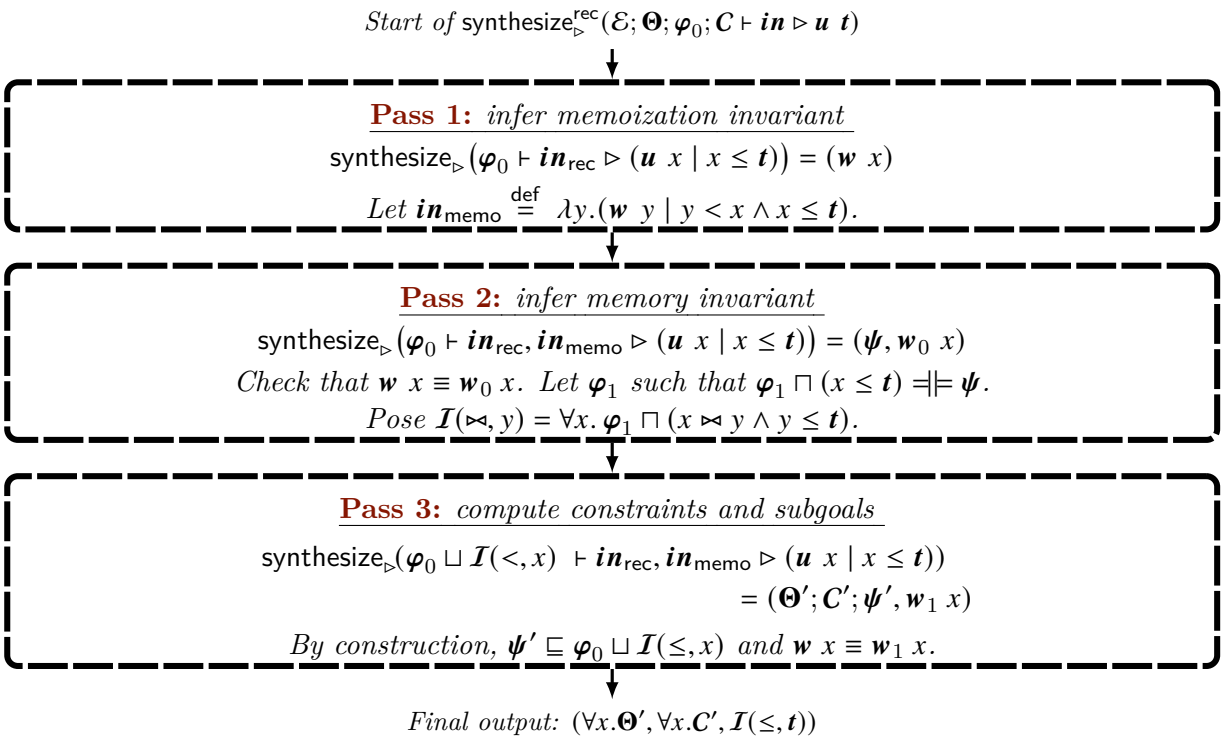
It operates in three phases: the first two infer memoizing invariants and memory invariants, respectively, while the third phase generates constraints and subgoals. Each phase builds upon our basic synthesis framework.

Memory and memoization invariants. Assume a function $\boldsymbol{u}$ defined by recurrence over some type τ using the well-founded order $<$. In a beautified syntax:

$$< : \tau \to \tau \to \mathsf{bool} \qquad \textbf{let rec}\ \boldsymbol{u}\ (x : \tau) = \boldsymbol{u}_0$$

where $\boldsymbol{u}_0$ is the body of $\boldsymbol{u}$'s definition.[6] We let $\leq$ be the reflexive closure of $<$.

[6] As indicated by the bold font, $\boldsymbol{u}_0$ is a bi-term. Formally, we have two symbols s_0 and s_1 defined by recurrence over x and whose bodies are, resp., the left and right component of $\boldsymbol{u}_0$. Then, we let



We assume $\boldsymbol{u}\equiv\lambda(x:\tau).\boldsymbol{u}_0$ where τ is is a `fixed` and `finite` base-type. We assume a total order $<$ over τ. We let $\boldsymbol{in}_{\mathsf{rec}}\overset{\mathsf{def}}{=}(\boldsymbol{in},\lambda y.(\boldsymbol{u}\ y\mid y<x))$. All calls to $\mathsf{synthesize}_{\triangleright}(\cdot)$ use the same environment $(\mathcal{E},x:\tau)$, initial hypotheses $\boldsymbol{\Theta}$, and initial constraints $\boldsymbol{C}$, which are thus omitted. We also omit components of $\mathsf{synthesize}_{\triangleright}(\cdot)$'s outputs that are discarded — e.g. in the first pass, only the outputted memoization hint $(\boldsymbol{w}\ x)$ is used.

Figure 6.10: Inductive simulator synthesis procedure $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\cdot)$.

Consider $\boldsymbol{t}$ of type τ, and assume we want to bideduce $(\boldsymbol{u}\ \boldsymbol{t})$ from some inputs $\boldsymbol{in}$. Excluding degenerated cases, doing so will require to recursively evaluate $\boldsymbol{u}$ on points $x<\boldsymbol{t}$. We can build such simulators using the following (simplified) induction rule:

$$\frac{\begin{array}{l}\mathcal{E},x:\tau;\ \boldsymbol{\mathcal{I}}_<(x)\ \vdash\\ \boldsymbol{in},x,\lambda y.(\boldsymbol{u}\ y,\boldsymbol{w}\ y\mid y<x\wedge x\le t)\triangleright(\boldsymbol{u}\ x,\boldsymbol{w}\ x\mid x\le \boldsymbol{t})\\ \qquad\qquad\rightsquigarrow(\boldsymbol{\mathcal{I}}_{\le}(x))\end{array}}{\mathcal{E};\boldsymbol{\varphi}_0\vdash \boldsymbol{in}\triangleright\boldsymbol{u}\ \boldsymbol{t}\rightsquigarrow(\boldsymbol{\mathcal{I}}_{\le}(\boldsymbol{t}))}$$

(For the sake of simplicity, we omit the constraints $\boldsymbol{C}$, hypotheses $\boldsymbol{\Theta}$, etc.)

There are two key ingredients (in colored boxes) which we describe next.

First, $\boldsymbol{\mathcal{I}}$ is a *memory invariant* describing the evolution of the game's state during the recursive evaluation of $\boldsymbol{u}$. It directly comes from the rule Induction. One remark, to initialize the induction (not shown here) we must ensure that $\boldsymbol{\varphi}_0$ entails $\boldsymbol{\mathcal{I}}_<(x_0)$ where x_0 is the smallest element w.r.t. $<$.

Second, $\boldsymbol{w}$ is a *memoization invariant*. Essentially, $(\boldsymbol{w}\ y)$ represents a set of intermediate values that have been computed during the bideduction of $(\boldsymbol{u}\ y)$ and that we decided

$\boldsymbol{u}=\#(s_0;s_1)$. For the sake of simplicity, we use a single symbol definition $\boldsymbol{u}$ using a bi-term as a body.

```
let rec output (t : timestamp) =
  match t with
  | Pk → pub k
  | V →
    let pk = pub k in
    enc diff(input V, dummy) r pk
  | M(i) → empty
  | P → shuffle(bb)
and frame (t : timestamp) =
  match t with
  | init → empty
  | _ → ⟨ frame (pred t), output t ⟩
and input (t : timestamp) =
  match t with
  | init → empty
  | _ → att(frame(pred t))
and bb (t : timestamp) =
  match t with
  | M i →
    let pk = pub k in
    let x =
      if V < M i &&
        (input t =
          enc diff(input V, dummy) r pk)
      then input V else dec (input t) k
    in
    bb[ i → x ] (* update cell i *)
  | _ → bb(pred t)
```

Figure 6.11: CCSA encoding of our abstract mixnet protocol.

to memoize. Thus, when bideducing $(\boldsymbol{u}\ x)$, we may reuse past memoized values $(\boldsymbol{w}\ y)$ for any $y < x$.

Invariant inference. Applying the induction rules requires to come-up with a memory invariant $\boldsymbol{\mathcal{I}}$ and a memoization invariant $(\boldsymbol{w}\ \cdot)$.

We define an inductive simulator synthesis procedure $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\cdot)$ that builds upon the basic synthesis of Section 6.3. This procedure is summarized in Figure 6.10, and relies on three passes of the basic synthesis procedure $\mathsf{synthesize}_{\triangleright}(\cdot)$, where the first pass infers a memoization invariant using the memoization hints returned by $\mathsf{synthesize}_{\triangleright}(\cdot)$, the second pass computes a memory invariant, and the third and last pass computes the final proof-obligations $\boldsymbol{\Theta}'$ to be discharged to the user and the constraints $\boldsymbol{C}'$ guaranteeing the existence of a probabilistic coupling justifying our reduction.

Crucially, the memory invariant of the second pass is an inductive invariant by construction (see Theorem 4 below). Thus, we do not need to check that this invariant is inductive.

6.4.2 Example

To understand this technical procedure, let us illustrate it on our abstract mixnet of Section 6.1. We give in Figure 6.11 a CCSA encoding of this protocol using recursive functions. Let us reduce the equivalence of the left and right versions of frame t_0 to CCA2, where t_0 is some constant timestamp. We unroll the execution of $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\boldsymbol{\varphi}_0 \vdash \emptyset \triangleright \mathsf{frame}\ t_0)$ on the initial memory $\boldsymbol{\varphi}_0 = (\ell \mapsto \epsilon)$ of the CCA2 game. Let t be the inductive step variable.

Obviously, detailing all passes, phases and steps of our synthesis procedure is not realistic, and would be too verbose to be of any use. Instead, we let the reader get an intuitive understanding of how the recursive functions in Figure 6.11 are simulated. Further, the simulator of Figure 6.3, which we manually wrote, should help intuit what is going on.

◇ *Pass 1.* After the first pass, we have the memoization hints:

$$\begin{aligned} \boldsymbol{w}\ t \stackrel{\mathsf{def}}{=}\ & (\mathsf{encV} \mid t = V \wedge t \leq t_0), && \text{(line 6)} \\ & \lambda i.\, (\mathsf{encV} \mid \mathsf{V} < \mathsf{M\ i} \wedge t = \mathsf{M\ i} \wedge t \leq t_0) && \text{(line 24)} \\ & \lambda i.\, (\mathsf{decM} \mid \ldots \wedge t = \mathsf{M\ i} \wedge t \leq t_0) && \text{(line 25)} \end{aligned}$$

where encV = enc **diff**(input V, dummy) (pub k) and decM = dec (input t) k. (We omit some details of the decM hint, as it will not be used.) We annotated each memoization hint with the corresponding line in Figure 6.11. Remark that the call to the **left-right** oracle line 24 is guarded by the test V < M i (at line 22), which thus appears in the corresponding memoization hint. After some simplification, this yields the following memoized values at time t:

$$\begin{aligned} \boldsymbol{in}_{\mathsf{memo}} \stackrel{\mathsf{def}}{=}\ & (\mathsf{encV} \mid V < t \wedge t \leq t_0), \\ & \lambda i.\, (\mathsf{encV} \mid \mathsf{V} < \mathsf{M\ i} \wedge \mathsf{M\ i} < t \wedge t \leq t_0),\ \ldots \end{aligned}$$

Note that the second memoized value is subsumed by the first. Since it could be difficult to parse, the above invariant can be read as follows: when computing $\boldsymbol{u}\ t$ for $t \leq t_0$, we have already computed encV if

- V happens before t, and/or
- M i happens before t for some i.

◇ *Pass 2.* The second pass computes the memory invariant of our simulator. Crucially, the memoized values in $\boldsymbol{in}_{\mathsf{memo}}$ allow to reduce the number of oracle calls through re-use, which is critical to simulate our mixnet protocol. Concretely, looking at Figure 6.11, we see that we may need to call the **left-right** oracle to simulate the computation at lines 6 and 24. For line 6, this adds encV to ℓ whenever $t = \mathsf{V}$. For line 24, we know that $t = \mathsf{M\ i}$ and we are operating under the condition that V < M i (line 22): thus, we can re-use the memoized value in $\boldsymbol{in}_{\mathsf{memo}}$, and never need to call the oracle **left-right** oracle there. Thus, after some simplifications which we omit, we obtain the post-condition

$$\boldsymbol{\psi} \stackrel{\mathsf{def}}{=}\ (\ell \mapsto \{\mathsf{encV} \mid t = \mathsf{V} \wedge t \leq t_0\})\,,$$

which yields the memory invariant

$$\boldsymbol{\mathcal{I}}_{\bowtie}(t) \stackrel{\mathsf{def}}{=}\ (\ell \mapsto \{\mathsf{encV} \mid t_1 : t_1 = \mathsf{V} \wedge t_1 \bowtie t \wedge t \leq t_0\})\,,$$

or equivalently $(\ell \mapsto \{\mathsf{encV} \mid \mathsf{V} \bowtie t \wedge t \leq t_0\})$. By Theorem 4 (presented later), this is an inductive invariant of our simulator.

◇ *Pass 3.* The last pass computes the proof-obligations $\boldsymbol{\Theta'}$ and constraints $\boldsymbol{C'}$, using the memory and memoization invariants, resp, $\boldsymbol{\mathcal{I}}$ and $\boldsymbol{in}_{\mathsf{memo}}$. Then, in our implementation, all generated goals are automatically discharged by Squirrel automated reasoning tactic **auto**. We refer the curious reader to [65] (file `motivating.sp`) for details — invariants in that file are however less readable than the ones presented here, which have been manually simplified.

6.4.3 Soundness

We now state the theorem ensuring the soundness of $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\cdot)$. First, we must describe the class of cryptographic hardness games supported by our result. We already saw that we only support game whose global mutable state are logs (or sets) of message values. Further, we must require that there are no complex flows between the different logs of the game. E.g., a game with two logs ℓ and ℓ', and with an update of the form $\ell \leftarrow \ell \cup \ell'$ in one of its oracles, is not supported by our procedure. (To our knowledge, this assumption is at no loss, as we do not know of any cryptographic game featuring such patterns.) Formally:

Assumption 1. *For any* update $\ell \leftarrow \boldsymbol{s}$ *in an oracle of* $\mathcal{G}$*, the only global* mutable *variable that* $\boldsymbol{s}$ *may depends upon is* $\{\ell\}$.

Finally, we assume that all assertions' domains cover the game global variables. Then, our rules never add elements to an assertion's domain and thus unions of assertions are always defined.

We can now state our main soundness theorem.

Theorem 4. *Let* $\mathcal{G}$ *satisfy Assumption 1. Let* $\boldsymbol{u} \equiv \lambda(x : \tau).\boldsymbol{u}_0$ *where* τ *is a* `fixed` *and* `finite` *base-type. Let* $<$ *be a total order over* τ *such that* $<\in \mathcal{L}_p$*. If*

$$\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{\varphi}_0; \boldsymbol{C} \vdash \boldsymbol{in} \triangleright \boldsymbol{u}\ \boldsymbol{t}) = (\boldsymbol{\Theta}', \boldsymbol{C}', \boldsymbol{\psi})$$

and $\boldsymbol{t} \in \boldsymbol{in}$*, then we have:*

$$\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi}_0 \vdash \boldsymbol{in} \triangleright \boldsymbol{u}\ \boldsymbol{t} \rightsquigarrow (\boldsymbol{\Theta}'; \boldsymbol{C}'; \boldsymbol{\psi}; \boldsymbol{w}).$$

To make the proof, we will first need a lemma to characterize the stability of the basic synthesis procedure, that is crucial to our invariant inference approach. The lemma is presented and proven below.

Lemma 22. *If basic synthesis succeeds starting from* $\boldsymbol{\varphi}$*:*

$$\mathsf{synthesize}_{\triangleright}(\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi} \vdash \boldsymbol{in} \triangleright \boldsymbol{out}) = (\boldsymbol{\Theta}_0; \boldsymbol{C}_0, \boldsymbol{\psi}, \boldsymbol{w})$$

then for any abstract memory $\boldsymbol{\varphi}_0$ *with the same domain as* $\boldsymbol{\varphi}$*,*

$$\begin{array}{l}
\textit{for any } \boldsymbol{\varphi}' \textit{ such that } \mathcal{E} \models_A \boldsymbol{\varphi}' \sqsubseteq \boldsymbol{\varphi} \sqcup \boldsymbol{\varphi}_0 \\
\textit{there exists } \boldsymbol{\Theta}'_0, \boldsymbol{C}'_0, \boldsymbol{\psi}' \textit{ such that} \\
\mathsf{synthesize}_{\triangleright}(\mathcal{E}; \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi}' \vdash \boldsymbol{in} \triangleright \boldsymbol{out}) = (\boldsymbol{\Theta}'_0; \boldsymbol{C}'_0, \boldsymbol{\psi}', \boldsymbol{w}) \\
\textit{where } \mathcal{E} \models_A \boldsymbol{\psi}' \sqsubseteq \boldsymbol{\psi} \sqcup \boldsymbol{\varphi}_0
\end{array}$$

Said otherwise, if the pre-condition $\boldsymbol{\varphi}'$ *is at-most the original pre-condition* $\boldsymbol{\varphi}$ *extended with* $\boldsymbol{\varphi}_0$*, then the synthesis procedure starting from* $\boldsymbol{\varphi}'$ *yields a post-condition* $\boldsymbol{\psi}'$ *that is at-most* $\boldsymbol{\psi}$ *extended with* $\boldsymbol{\varphi}_0$*. Further, the memoization hints* $\boldsymbol{w}$ *are left unchanged. Note, however, that the proof-obligations* $\boldsymbol{\Theta}'_0$ *and outputted constraints* $\boldsymbol{C}'_0$ *may be different.*

Proof. We prove this by induction over the execution of the basic synthesis procedure, walking synchronously through the pair of executions of $\mathsf{synthesize}_{\triangleright}(\cdot)$ starting from, resp., $\boldsymbol{\varphi}$ and $\boldsymbol{\varphi}'$.

Then, we analyze each phase of the procedure as described in Figure 6.8 and check that if the phase succeeds on $\boldsymbol{\varphi}$, then it also succeeds on $\boldsymbol{\varphi}'$ and produces identical outputs, but for the post-condition $\boldsymbol{\varphi}' \sqsubseteq \boldsymbol{\varphi} \sqcup \boldsymbol{\varphi}_0$, the proof-obligations $\boldsymbol{\Theta}'_0$, and outputted constraints $\boldsymbol{C}'_0$. During this execution, we must prove that the arguments of all recursive calls to $\mathsf{synthesize}_{\triangleright}(\cdot)$ are identical except for the pre-condition, that must be of the form, resp., $\boldsymbol{\varphi}_{\mathsf{rec}}$ and $\boldsymbol{\varphi}'_{\mathsf{rec}} \sqsubseteq \boldsymbol{\varphi}_{\mathsf{rec}} \sqcup \boldsymbol{\varphi}_0$;

⋄ *The reuse phase.* This phase scans the inputs in the terms $\boldsymbol{in}.\mathtt{std}$, looking for a valid application of the Load rule. The success of this rule only depends on the inputs $\boldsymbol{in}.\mathtt{std}$, the assumptions $\boldsymbol{\Theta}$, and the environment $\mathcal{E}$. Thus, it satisfies the wanted property.

⋄ *The oracle phase.* Consider the application of an oracle $\mathbf{o}$.

First, step **(A)** decomposes the oracle's outputs into a condition $\boldsymbol{c_f}$ and an output $\boldsymbol{o_f}$. This decomposition is syntactic and thus independent of the pre-condition. Then, it unifies the oracle outputs with $\boldsymbol{out}$, which only depends on $\boldsymbol{out}$, the game $\mathcal{G}$, and the environment $\mathcal{E}$ (since unification is modulo convertibility in $\mathcal{E}$).

Then, step **(B)** reuses memoized values using the rule Memoize.Load, which only depends on $\boldsymbol{in}$ and its tagging (more precisely, on $\boldsymbol{in}.\mathtt{memo}$), and on $\mathcal{E}$ (due to the unification modulo convertibility). Thus, this step refines $\boldsymbol{out}$ into $(\boldsymbol{out} \mid \boldsymbol{h})$ for some boolean terms $\boldsymbol{h}$, for both executions of $\mathsf{synthesize}_{\triangleright}(\cdot)$.

Step **(C)** does a case-analysis to reduce when the output term $\boldsymbol{out}$ is to be computed using the oracle $\mathbf{o}$. This case-analysis only depends on the substitution $\boldsymbol{\sigma}$ computed in step **(A)**, and uses the deconstruction rule FA.ite which is independent from the pre-condition.

(D.⊥) only does a recursive call to $\mathsf{synthesize}_{\triangleright}(\cdot)$, which is fine using the induction hypothesis. Step **(D.⊤)** is the most complex step. First, it outputs the updated pre-condition. Consider the case where there is a single update $(\ell \leftarrow \boldsymbol{s})$ (the general case is similar). The post-conditions $\boldsymbol{\psi}$ and $\boldsymbol{\psi}'$ are then:

$$\boldsymbol{\psi} \stackrel{\mathsf{def}}{=} \boldsymbol{\varphi}\{\ell \mapsto \mathsf{eval}_{\boldsymbol{\varphi}}(\boldsymbol{s}\sigma[\mathsf{b} \mapsto \#(0;1)])\}$$
$$\boldsymbol{\psi}' \stackrel{\mathsf{def}}{=} \boldsymbol{\varphi}'\{\ell \mapsto \mathsf{eval}_{\boldsymbol{\varphi}'}(\boldsymbol{s}\sigma[\mathsf{b} \mapsto \#(0;1)])\}$$

We start by observing that as both abstract memories $\boldsymbol{\varphi}$ and $\boldsymbol{\varphi}'$ have the same domain, we know by Proposition 8 that $\mathsf{eval}_{\boldsymbol{\varphi}}(\boldsymbol{s}')$ succeeds iff. $\mathsf{eval}_{\boldsymbol{\varphi}'}(\boldsymbol{s}')$ does, with $\boldsymbol{s}' = \sigma[\mathsf{b} \mapsto \#(0;1)]$. Thus, either both oracle phases fail or both oracle phases succeed, ensuring that they remain synchronized. Now, we must show that $\boldsymbol{\psi}' \sqsubseteq \boldsymbol{\psi} \cup \boldsymbol{\varphi}_0$, which amounts to proving that for any $y \in \mathsf{dom}(\boldsymbol{\varphi})$:

$$\boldsymbol{\psi}'(y) \subseteq (\boldsymbol{\psi} \cup \boldsymbol{\varphi}_0)(y) \tag{6.2}$$

For any $y \neq \ell$, $\boldsymbol{\psi}(y) = \boldsymbol{\varphi}(y)$ and $\boldsymbol{\psi}'(y) = \boldsymbol{\varphi}'(y)$. Thus, Eq. (6.2) is immediate from the fact that:

$$\boldsymbol{\varphi}' \sqsubseteq \boldsymbol{\varphi} \sqcup \boldsymbol{\varphi}_0$$

It remains to check the case $y = \ell$:

$$
\begin{aligned}
& \boldsymbol{\psi}'(\ell) \\
&= \mathsf{eval}_{\boldsymbol{\varphi}'}(\boldsymbol{s}') \\
&\subseteq \mathsf{eval}_{\boldsymbol{\varphi} \sqcup \boldsymbol{\varphi}_0}(\boldsymbol{s}') && \text{(By Proposition 9)} \\
&\subseteq \mathsf{eval}_{\boldsymbol{\varphi}}(\boldsymbol{s}') \cup \textstyle\bigcup_{x \in \mathsf{vars}(\boldsymbol{s}')} \boldsymbol{\varphi}_0(x) && \text{(By Proposition 10)} \\
&= \mathsf{eval}_{\boldsymbol{\varphi}}(\boldsymbol{s}') \cup \boldsymbol{\varphi}_0(\ell) && \text{(By Assumption 1)} \\
&= \boldsymbol{\psi}(\ell) \cup \boldsymbol{\varphi}_0(\ell) \\
&= (\boldsymbol{\psi} \sqcup \boldsymbol{\varphi}_0)(\ell)
\end{aligned}
$$

Which is what we needed. Then, the procedure analyzes the output condition $\boldsymbol{c}_f$ in, resp., $\boldsymbol{\psi}$ and $\boldsymbol{\psi}'$. As before, we now that both abstract evaluation $\mathsf{b\text{-}eval}_{\boldsymbol{\psi}}(\boldsymbol{c}_f)$ and $\mathsf{b\text{-}eval}_{\boldsymbol{\psi}'}(\boldsymbol{c}_f)$ are synchronized using Proposition 8, which concludes the analysis of the **(D.⊤)**.

Finally, step **(E)** applies MEMOIZE.STORE to add a memoization hints, which only depends on $\boldsymbol{out}$.

⋄ *The destruction phase.* The destruction phase peels-off a top-level construct of $\boldsymbol{out}$ and applies a deconstruction rule (see Figure 6.6). To peel-off a top-level construct, we put the term in weak-head normal form using the CONV rule on $\boldsymbol{out}$. This only depends on $\mathcal{E}$ and $\boldsymbol{out}$.[7] We conclude the analysis of this phase by observing that none of the deconstruction rules use the pre-condition or inputs, nor modifies the post-condition or output memoization hints. All rules are straightforward, but for the FA.MATCH rule, which obtains the final post-condition by re-composing the post-conditions of all match cases.

More precisely, consider an FA.MATCH instance:

$$
\boldsymbol{\varphi} \vdash \boldsymbol{in} \triangleright \mathsf{match}\ \boldsymbol{t}\ \mathsf{with}\ (c_i\ \vec{x}_i \mapsto \boldsymbol{u_i})_i \rightsquigarrow \boldsymbol{\psi}_{\mathsf{out}} \tag{6.3}
$$

We simplified the instance by removing the guard $(\cdots \mid \boldsymbol{f})$ around the match. Further, we omitted most components of the query judgement to focus on the the pre- and post-conditions, which are the interesting part. In premise, we know that for any i, the procedure $\mathsf{synthesize}_{\triangleright}(\cdot)$ concluded and produced the completed query judgement:

$$
\vec{x}_i, \boldsymbol{\varphi} \vdash \boldsymbol{in}, \vec{x}_i \triangleright (\boldsymbol{u_i} \mid \boldsymbol{t} = c_i\ \vec{x}_i) \rightsquigarrow \boldsymbol{\psi}_i \tag{6.4}
$$

and we know that:

$$
\boldsymbol{\psi}_{\mathsf{out}} = \textstyle\bigsqcup_i \big(\forall \vec{x}_i.\, \boldsymbol{\psi}_i \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i)\big) \tag{6.5}
$$

Consider $\boldsymbol{\varphi}' \sqsubseteq \boldsymbol{\varphi} \sqcup \boldsymbol{\varphi}_0$. We must show that running $\mathsf{synthesize}_{\triangleright}(\cdot)$ starting from the left-hand side of Eq. (6.3) where we replaced $\boldsymbol{\varphi}$ by $\boldsymbol{\varphi}'$ verify the three following points:

- (i) It succeeds, and produces a completed query judgement of the form:

$$
\boldsymbol{\varphi}' \vdash \boldsymbol{in} \triangleright \mathsf{match}\ \boldsymbol{t}\ \mathsf{with}\ (c_i\ \vec{x}_i \mapsto \boldsymbol{u_i})_i \rightsquigarrow \boldsymbol{\psi}'_{\mathsf{out}}. \tag{6.6}
$$

[7] Our implementation uses more complex reduction rules that may exploits the assumptions $\boldsymbol{\Theta}$. This does not pose any issues here, since we may depend on $\boldsymbol{\Theta}$.

- (ii) The novel post-condition $\boldsymbol{\psi}'_{\mathsf{out}}$ satisfies:

$$\boldsymbol{\psi}'_{\mathsf{out}} \sqsubseteq \boldsymbol{\psi}_{\mathsf{out}} \sqcup \boldsymbol{\varphi}_0.$$

- (iii) The memoization hints of the query judgement in Eq. (6.3) and Eq. (6.6) are identical.

First, by applying the induction hypothesis to Eq. (6.4) for every $\boldsymbol{i}$, we get that:

$$\vec{x}_i; \boldsymbol{\varphi}' \vdash \boldsymbol{in}, \vec{x}_i \rhd (\boldsymbol{u_i} \mid \wedge \boldsymbol{t} = c_i\ \vec{x}_i) \rightsquigarrow \boldsymbol{\psi}'_i \tag{6.7}$$

$$\text{where} \quad \boldsymbol{\psi}'_i \sqsubseteq \boldsymbol{\psi}_i \sqcup \boldsymbol{\varphi}_0 \tag{6.8}$$

Thus, synthesis of the match term succeeds (that is, point (i) holds), and produces a post-condition:

$$\boldsymbol{\psi}'_{\mathsf{out}} = \bigsqcup_i \big(\forall \vec{x}_i.\, \boldsymbol{\psi}'_i \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i)\big) \tag{6.9}$$

By Eq. (6.8), we know that $\boldsymbol{\psi}'_i \sqsubseteq \boldsymbol{\psi}_i \sqcup \boldsymbol{\varphi}_0$ and thus (using the properties of Proposition 5):

$$\forall \vec{x}_i.\, \boldsymbol{\psi}'_i \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i) \sqsubseteq \forall \vec{x}_i.\, ((\boldsymbol{\psi}_i \sqcup \boldsymbol{\varphi}_0) \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i))$$

Further:

$$\begin{aligned}
&\forall \vec{x}_i.\, ((\boldsymbol{\psi}_i \sqcup \boldsymbol{\varphi}_0) \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i)) \\
\sqsubseteq\ & (\boldsymbol{\psi}_i \sqcup \boldsymbol{\varphi}_0) \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i) && \text{(Proposition 6.(A))} \\
\sqsubseteq\ & \big(\boldsymbol{\psi}_i \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big) \\
& \sqcup \big(\boldsymbol{\varphi}_0 \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big) \\
\sqsubseteq\ & \forall \vec{x}_i.\, \big(\boldsymbol{\psi}_i \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i)\big) && \text{(Proposition 6.(A))} \\
& \sqcup \big(\boldsymbol{\varphi}_0 \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big)
\end{aligned}$$

Consequently, using the definition of $\boldsymbol{\psi}'_{\mathsf{out}}$ in Eq. (6.9) and taking the join of the previous symbolic inclusion over all $\boldsymbol{i}$, we have that:

$$\begin{aligned}
& \boldsymbol{\psi}'_{\mathsf{out}} \\
\sqsubseteq\ & \bigsqcup_i \begin{pmatrix} \forall \vec{x}_i.\, \big(\boldsymbol{\psi}_i \sqcap (\boldsymbol{t} = c_i\ \vec{x}_i)\big) \\ \sqcup \big(\boldsymbol{\varphi}_0 \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big) \end{pmatrix} \\
\sqsubseteq\ & \boldsymbol{\psi}_{\mathsf{out}} \sqcup \bigsqcup_i \big(\boldsymbol{\varphi}_0 \sqcap (\exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big) && \text{(Using Eq. (6.5))} \\
\sqsubseteq\ & \boldsymbol{\psi}_{\mathsf{out}} \sqcup \big(\boldsymbol{\varphi}_0 \sqcap (\bigvee_i \exists \vec{x}_i.\, \boldsymbol{t} = c_i\ \vec{x}_i)\big) && \text{(Prop. 6.(B))} \\
\sqsubseteq\ & \boldsymbol{\psi}_{\mathsf{out}} \sqcup \boldsymbol{\varphi}_0 && \text{(Since } (c_i)_i \text{ are constructors)}
\end{aligned}$$

Which concludes the proof of item (ii).

For point (iii), we observe that the induction hypothesis guarantees that the memoization hints in Eq. (6.4) and Eq. (6.7) are identical. Since the final memoization hints of Eq. (6.3) and Eq. (6.6) are obtained by recomposing the memoization hints all all premises, we know that they are left unchanged.

This concludes the proof of (iii), and the proof of the FA.MATCH case.

⋄ *The unreachability phase.* The unreachability phase uses the UNREACH rule which leaves the pre-condition unchanged and directly outputs it. This satisfies our invariant. □

We now prove Theorem 4:

Proof of Theorem 4. Since the execution of $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\cdot)$ succeeded, we know, looking at the second and third pass as described in Figure 6.10, that:

$$\begin{aligned}\mathsf{synthesize}_{\triangleright}\big(\boldsymbol{\varphi}_0 \vdash \boldsymbol{in}_{\mathsf{rec}}, \boldsymbol{in}_{\mathsf{memo}} \triangleright (\boldsymbol{u}\ x \mid x \leq \boldsymbol{t})\big) \\ = (\boldsymbol{\varphi}_1 \sqcap (x \leq \boldsymbol{t}), \boldsymbol{w}\ x)\end{aligned} \tag{6.10}$$

$$\mathsf{synthesize}_{\triangleright}\left(\begin{array}{l}\boldsymbol{\varphi}_0 \sqcup \boldsymbol{\mathcal{I}}_{<}(x) \\ \quad \vdash \boldsymbol{in}_{\mathsf{rec}}, \boldsymbol{in}_{\mathsf{memo}} \\ \quad \triangleright (\boldsymbol{u}\ x \mid x \leq \boldsymbol{t})\end{array}\right) = (\boldsymbol{\Theta}_0; \boldsymbol{C}_0; \boldsymbol{\psi}'; \boldsymbol{w}\ x) \tag{6.11}$$

where both calls to $\mathsf{synthesize}_{\triangleright}(\cdot)$ use the same environment $(\mathcal{E}, x : \tau)$, initial hypotheses $\boldsymbol{\Theta}$, and initial constraints $\boldsymbol{C}$. Further, we recall that:

$$\boldsymbol{in}_{\mathsf{rec}} \stackrel{\mathsf{def}}{=} \big(\boldsymbol{in}, \lambda y.(\boldsymbol{u}\ y \mid y < x)\big) \tag{6.12}$$

$$\boldsymbol{in}_{\mathsf{memo}} \stackrel{\mathsf{def}}{=} \lambda y.(\boldsymbol{w}\ y \mid y < x) \tag{6.13}$$

$$\boldsymbol{\mathcal{I}}_{\bowtie}(y) = \forall x.\, \boldsymbol{\varphi}_1 \sqcap (x \bowtie y \wedge y \leq \boldsymbol{t}). \tag{6.14}$$

Using Eq. (6.11) and by the soundness of the $\mathsf{synthesize}_{\triangleright}(\cdot)$ procedure, we know that:

$$\begin{aligned}&(\mathcal{E}, x : \tau); \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi}_0 \sqcup \boldsymbol{\mathcal{I}}_{<}(x) \vdash \\ &\quad \boldsymbol{in}_{\mathsf{rec}}, \boldsymbol{in}_{\mathsf{memo}} \triangleright (\boldsymbol{u}\ x \mid x \leq \boldsymbol{t}) \rightsquigarrow (\boldsymbol{\Theta}_0; \boldsymbol{C}_0; \boldsymbol{\psi}'; \boldsymbol{w}\ x)\end{aligned}$$

or equivalently, moving $(\boldsymbol{w}\ x)$ from the right of $\rightsquigarrow$ to its left, and weakening $(\boldsymbol{w}\ x)$ into $(\boldsymbol{w}\ x \mid x \leq \boldsymbol{t})$ (which we can do, since $t \in \boldsymbol{in}.\mathtt{std}$):

$$\begin{aligned}&(\mathcal{E}, x : \tau); \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi}_0 \sqcup \boldsymbol{\mathcal{I}}_{<}(x) \vdash \\ &\quad \boldsymbol{in}_{\mathsf{rec}}, \boldsymbol{in}_{\mathsf{memo}} \triangleright (\boldsymbol{u}\ x, \boldsymbol{w}\ x \mid x \leq \boldsymbol{t}) \rightsquigarrow (\boldsymbol{\Theta}_0; \boldsymbol{C}_0; \boldsymbol{\psi}'; \epsilon)\end{aligned}$$

Then, using the definitions of $\boldsymbol{in}_{\mathsf{rec}}$ and $\boldsymbol{in}_{\mathsf{memo}}$ in Eq. (6.12) and Eq. (6.13), and re-arranging the values on the left of $\triangleright$, we have:

$$\begin{aligned}&(\mathcal{E}, x : \tau); \boldsymbol{\Theta}; \boldsymbol{C}; \boldsymbol{\varphi}_0 \sqcup \boldsymbol{\mathcal{I}}_{<}(x) \vdash \\ &\boldsymbol{in}, \lambda y.(\boldsymbol{u}\ y, \boldsymbol{w}\ y \mid y < x) \triangleright (\boldsymbol{u}\ x, \boldsymbol{w}\ x \mid x \leq \boldsymbol{t}) \\ &\qquad\qquad \rightsquigarrow (\boldsymbol{\Theta}_0; \boldsymbol{C}_0; \boldsymbol{\psi}'; \epsilon)\end{aligned} \tag{6.15}$$

We are now ready to apply the Induction rule of Figure 6.9, where:

- we are inductively computing $\lambda(x : \tau).\, (\boldsymbol{u}\ x, \boldsymbol{w}\ x)$;
- forall x the memory invariant *before* time-point x is $\boldsymbol{\varphi}_0 \sqcup \boldsymbol{\mathcal{I}}_{<}(x)$;
- forall x the memory invariant *after* time-point x is $\boldsymbol{\mathcal{I}}_{\leq}(x)$ (and *not* $\boldsymbol{\varphi}_0 \sqcup \boldsymbol{\mathcal{I}}_{\leq}(x)$).

It remains to check all premises of the Induction rule. Most premises clearly hold by hypothesis, but for the fact that:

A) $\boldsymbol{\varphi}_0$ entails the initial memory invariant, i.e.:

$$\mathcal{E}; \boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\varphi}_0 \sqsubseteq \boldsymbol{\varphi}_0 \sqcup \boldsymbol{\mathcal{I}}_{<}(x_0)$$

where x_0 is the minimal element w.r.t. $<$.

B) The post-condition of Eq. (6.15) is a post-fixpoint:

$$(\mathcal{E}, x : \tau); \boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\psi}' \sqsubseteq \boldsymbol{\mathcal{I}}_{\leq}(x);$$

C) The memory invariant after $\mathsf{pred}\ x$ entails the memory invariant before x:

$$(\mathcal{E}, x : \tau); \boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\mathcal{I}}_{\leq}(\mathsf{pred}\ x) \sqsubseteq \boldsymbol{\varphi}_0 \sqcup \boldsymbol{\mathcal{I}}_{<}(x)$$

Condition A) trivially follows from the monotonicity property of $\sqcup$ w.r.t. $\sqsubseteq$ of Proposition 5.

For condition B), we apply Lemma 22 on the basic synthesis execution of Eq. (6.10), using $\boldsymbol{\mathcal{I}}_{<}(x)$ as inductive memory invariant. This tells us that the post-condition $\boldsymbol{\psi}'$ of the basic synthesis execution of Eq. (6.11) is such that

$$(\mathcal{E}, x : \tau); \boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\psi}' \sqsubseteq (\boldsymbol{\varphi}_1 \sqcap (x \leq \boldsymbol{t})) \sqcup \boldsymbol{\mathcal{I}}_{<}(x). \tag{6.16}$$

Let us show that:

$$(\mathcal{E}, x : \tau); \boldsymbol{\Theta} \models_{\mathsf{A}} (\boldsymbol{\varphi}_1 \sqcap (x \leq \boldsymbol{t})) \sqsubseteq \boldsymbol{\mathcal{I}}_{\leq}(x) \tag{6.17}$$

$$(\mathcal{E}, x : \tau); \boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\mathcal{I}}_{<}(x) \sqsubseteq \boldsymbol{\mathcal{I}}_{\leq}(x) \tag{6.18}$$

Recall that the definition of $\boldsymbol{\mathcal{I}}$ is in Eq. (6.14). Property Eq. (6.17) can be established by instantiating the quantification in $\boldsymbol{\mathcal{I}}$ with x, and noting that

$$((\boldsymbol{\varphi}_1 \sqcap (x \leq \boldsymbol{t})) \sqsubseteq \boldsymbol{\varphi}_1 \sqcap (x \leq x \wedge x \leq \boldsymbol{t})).$$

Property Eq. (6.18) follows by unfolding $\sqsubseteq$ into its semantics and then doing some straightforward reasoning on the ordering $\leq$.

Continuing from Eq. (6.17) and Eq. (6.18), and using Proposition 5, we get that:

$$(\mathcal{E}, x : \tau); \boldsymbol{\Theta} \models_{\mathsf{A}} (\boldsymbol{\varphi}_1 \sqcap (x \leq \boldsymbol{t})) \sqcup \boldsymbol{\mathcal{I}}_{<}(x) \sqsubseteq \boldsymbol{\mathcal{I}}_{\leq}(x)$$

which, together with Eq. (6.16) and the transitivity of $\sqsubseteq$, proves condition B).

For condition C), we must show that:

$$(\mathcal{E}, x : \tau); \boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\mathcal{I}}_{\leq}(\mathsf{pred}\ x) \sqsubseteq \boldsymbol{\varphi}_0 \sqcup \boldsymbol{\mathcal{I}}_{<}(x)$$

which we obtain by showing the stronger property:

$$(\mathcal{E}, x : \tau); \boldsymbol{\Theta} \models_{\mathsf{A}} \boldsymbol{\mathcal{I}}_{\leq}(\mathsf{pred}\ x) \sqsubseteq \boldsymbol{\mathcal{I}}_{<}(x).$$

which is straightforward from the definition of $\boldsymbol{\mathcal{I}}_{\bowtie}(x)$ in Eq. (6.14), instantiating the quantifier in $\boldsymbol{\mathcal{I}}$ with x on both side. □

Chapter 7

Implementation and Case Studies

Contents

7.1 Preliminary on Squirrel syntax **137**
7.2 Implementation **139**
7.2.1 Inputs of the **crypto** tactic 140
7.2.2 Outputs of the **crypto** tactic 141
7.3 Case studies **143**
7.4 The FOO protocol **144**
7.4.1 Cryptographic primitives and assumptions 145
7.4.2 The FOO protocol security 153
7.4.3 Proof 155

To conclude this thesis, we implemented our synthesis into Squirrel, making it available as a tactic, **crypto**, in Squirrel proofs. We start this chapter by giving a quick overview of Squirrel syntax and proofs in Section 7.1, and we present how the synthesis is made available to users in Section 7.2. We present these case studies in Section 7.3, highlighting what was achieved and why they validate our approach. Finally, we show that our method can handle more complex protocols, by tackling the FOO protocol — the largest case study in Squirrel to date. This case study relies on new hardness assumption and integrate **crypto** in a large proof. The proofs' high-level description is presented in Section 7.4.

7.1 Preliminary on Squirrel syntax

In this section, we present elements of the Squirrel syntax and usage prior of the thesis. These elements are necessary to understand the rest of the chapter.

Types and operators. In Squirrel, we can declare new types and abstract library functions, called operators. We can declare a new type ty with labels labels with the following syntax: **type** ty[labels]. Among the Squirrel labels, we will make uses of the followings ones:

- serializable, which says that the type is encodable in bitstrings,
- large, which ensures that the probability of collision of two sampling on this type is negligible.

An operator f of type τ is declared with the syntax **op** f :τ. For example, the types and operators needed for the CCA2 game can be declared as follows:

```
type seed[ serializable,large ].      (* seed *)
type sk_enc[ serializable,large ].    (* secret key *)
type pk_enc[ serializable ].          (* public key *)
type ctxt[ serializable ].            (* cyphertext *)

(* Encryption's scheme primitives: public key, encryption and decryption. *)
op pk_enc : sk_enc → pk_enc.
op encr : message → pk_enc → seed → ctxt.
op decr : ctxt → sk_enc → message.
```

Systems. In Squirrel, the mutually recursive functions frame, input and output have built-in definitions. Their mutual definition is implemented directly into Squirrel, when it is not protocol-depend, and leaves to the users the definition of timestamp symbols and output. This is done through *systems*, a system representing a protocol. Bi-systems — or bi-protocols — are made available through bi-terms, with the specific constructor **diff**, the exact counterpart of our theoretical #(_; _).

To continue on illustrating the implementation, we provide below the mixnet system declaration done in Squirrel, as presented in Chapter 6 along with name declaration.

```
(* Encryption key and encryption seed names.*)
name k:sk_enc.
name r:seed.

(* The mixnet system. *)
system MixNet =
  PUB : out(c, (pub k)) |

 (* the voter `V` sends its encrypted ballot *)
 V: (in(c,x);
     out(c, enc diff(x,dummy) r (pub k))) |

 (* collect and decrypts ballot into `bb` *)
 M: (!i in(c,y);
        bb(i) :=
          if V < M i &&
             y = enc diff(input@V,dummy) r (pub k)
          then input@V
          else dec y k) |

 (* publish `bb` once collection (i.e. sub-processes `M`) are done *)
 P: (let BB = λ i ⇒ bb i in out(c, shuffle BB)).
```

Proofs. Finally, we can declare axioms in SQUIRREL with the following syntax:

```
axiom Name @system:S vars : f
```

where

- Name is the name of the axiom,
- @system:S indicates the system (or systems) in which the axiom holds, S can be a system name or any when the axioms hold in any system,
- vars is a list of variable, and
- f is a formula.

Lemmas and global lemmas (lemmas on, respectively, local or global formulae) are also declared using the same syntax with the commands lemma and **global** lemma.

For example, we present below the usual axiom on encryption and the protocol indistinguishability global lemma for the MixNet system.

```
(* Axiom stating that the decryption of the encryption of m with the correct key retrieves m *)
axiom decr_encr @system:any m r sk : decr (encr m (pk_enc sk) r) sk = m.

(* The following lemma proves that for any timestamp `t`,
if the voter has sent its message `V ≤ t`, the two protocols
decribed by `MixNet` system are indistinguishable.

The additional hypothesis `[ len (input@V) = len dummy ]` ensures the bi-system
differs on encryptions of same-length messages,
to respect the length condition of the CCA2 hardness assumption.
*)
global lemma _ @system:MixNet t : adv(t) ⇛ [ V ≤ t ] ⇛ [ len (input@V) = len dummy ] ⇛
equiv(frame@t).
```

To start the proof of lemmas and global lemma, we use the command **Proof**.. Then follows a sequence of tactics, and the proof ends with either **Qed**. when the proof is done, or **Abort**. to abort the proof.

7.2 Implementation

We have implemented our simulator synthesis procedure in SQUIRREL [66]. This implementation is mainly located in one file, src/core/crypto.ml, that implements the proof search procedure. It significantly relies on the existing matching procedure and automatic reasoning capabilities in SQUIRREL. This extension allows users to specify arbitrary games, and exploit the synthesis procedure through a tactic called **crypto**, which takes as input the game to be used and some (optional) initial constraints. Upon success, the tactic reduces the equivalence to be proved to proof obligations corresponding to the subgoals generated by the synthesis procedure and the formulae establishing the validity of the generated constraint system.

Scope. Our goal was for **crypto** to reach the expressivity level of SQUIRREL's legacy cryptographic tactics, while being able to tackle new cryptographic games. Crucially, legacy cryptographic tactics, as well as **crypto**, are not expected to apply in all scenarios: a typical SQUIRREL proof consists in modifying the proof-goal using its indistinguishability logic [24] to pave the way for the application of a cryptographic game. Furthermore, since SQUIRREL is an interactive proof assistant, we aim for **crypto** to have a low running time (i.e. a few seconds). This explains the design choices of our procedure: the up-front processing of induction, the ad-hoc processing for invariant inference and the absence of backtracking.

In this section, we go back to the motivating example of Section 6.1, to presents its formalization in SQUIRREL and illustrate how our framework is embedded in SQUIRREL. The corresponding file can be found in [65] in `motivating.sp`.

7.2.1 Inputs of the crypto tactic

The **crypto** tactic is expected to be used in a proof environment — between a command **Proof** and **Qed** or **Abort** commands. It takes several inputs: the game of the hardness assumption, and optional initial constraint system and flags. We detail them in this section.

Games. The **crypto** tactic comes with a new syntax to declare arbitrary games. This syntax follows the one of Chapter 3. A game declaration is of the form

```
game Name = { ... }
```

The user can declare global samplings, variables and oracles. This is respectively done with the following declarations:

$$\begin{aligned}&\textbf{rnd}[name] : [type]\\&\textbf{var}[var] = [term]and\\&\textbf{oracle}[oracle] = \{\ldots\}.\end{aligned}$$

Below, we give a SQUIRREL syntax for the CCA2 game.

```
game CCA2 = {
  rnd key : sk_enc;
  var log = empty_set;

  oracle pk = { return pk_enc key}

  oracle encrypt (m0,m1 : message) = {
    rnd seed: seed;
    var c0 = encr m0 (pk_enc key) seed;
    var c1 = encr m1 (pk_enc key) seed;
    log := add diff(c0,c1) log ;
    return if (len m0) = (len m1) then encr diff(m0,m1) (pk_enc key) seed
  }

  oracle decrypt (c : ctxt) = { return if not (mem c log) then decr c key }
}
```

Initial constraint system. An initial constraint system can be given to the **crypto** tactic for the simulator' synthesis procedure. This is done by giving elements of the form (rnd : n) where rnd is a global random value in the game, and n a name. When the name is indexed, the implementation checks that they are bideducible *without oracle calls nor random samplings*.

Flags. It is possible to give flags to the tactic, which are

- time_sensitive: this enables the first two passes of our simulator synthesis to synthesize memoizing simulators and generate time-sensitive invariants. Without this flag, the **crypto** tactic synthesize a simulator without memoization and with time-insensitive invariants.
- no_subgoals_on_failure: this disables the unreachable phase in the basic procedure of simulator synthesis.

Finally, the proof of the MixNet example is:

```
global lemma secure_mixnet @system:MixNet t : adv(t) ⇛̃ [ V ≤ t ] ⇛̃ [ len (input@V) = len dummy ]
⇛̃ equiv(frame@t).
Proof.
  intro HA Hlen.
  crypto
    ~time_sensitive        (* use the improved synthesis procedure *)
    CCA2                   (* use the `CCA2` game *)
    (key:k);               (* map name `k` to `key` in `CCA2` *)
  auto.
Qed.
```

7.2.2 Outputs of the crypto tactic

The SQUIRREL proof above proves the equivalence of the left and right components of the process MixNet directly by reduction to CCA2. It generates 8 subgoals that are automatically proved by **auto**. It also prints the final constraint system, the final abstract memory and all the generated subgoals.

First, let us give a sketch of what the synthesis procedure outputs on the proof on our mixnet example. This can be retrieved by a verbose mode of the tactic **crypto**. There is one encryption enc **diff**(input@V, dummy) r (pub k) that appears in:

- in output@V, guarded by condition V ≤ t.
- in output@M(i) for any index i guarded by the condition M(i) ≤ t && V < M(i).

That means the synthesis does the following oracle calls:

- one for output@V if the mixnet has not run and used the oracle, that is ∀ i, not M(i) ≤ pred V || not (M(i) ≤ t && V < M(i)).
- one for output@M(i0) if i0 is the first index such that: the mixnet does the call and the voter hasn't run before. That is not (V ≤ pred (M(i0))) || not (V ≤ t) and ∀ i, not (M(i) ≤ pred (M(i0))) || not (M(i) ≤ t && V < M(i))

This is reflected in the final memory and then final constraint systems.

Final constraint system. The final constraint system contains the initial constraint given as input and the two associating name r to a local sampling of the game.

```
(* Initial constraint given to crypto*)
{ k , Gkey }

(* Contraint generated for name `r` when synthesizing output@M(i0)*)
{ r , L | ∀ i0 : ((∀ (i:index), not (M(i) ≤ pred (M(i0))) || not (V < M(i)) || not (M(i) ≤ t))
                && (not (V ≤ pred (M(i0))) || not (V ≤ t)))
                && V < M(i0)
                && M(i0) ≤ t }

(* Contraint generated for name `r` when synthesizing output@V *)
{ r , L | (∀ (i:index), not (M(i) ≤ pred V) || not (V ⟨ M(i)) || not (M(i) ≤ t)) && V ≤ t }
```

Final memory. Similarly, the log is updated during these two oracles calls, which leads to the following abstract memory.

```
log → [  (* Logged during the synthesis of output@V *)
      { enc diff(input@V, dummy) r (pub k) | ∀ τ0 :
                    (∀ (i:index), not (M(i) ≤ pred V) || not (V < M(i)) || not (M(i) ≤ t))
                && V < τ0 && τ0 ≤ t },
      (* Logged during the synthesis of output@M(i0) *)
      { enc diff(input@V, dummy) r (pub k) | ∀ i0,τ0 :
          ((∀ (i:index), not (M(i) ≤ pred (M(i0))) || not (V ⟨ M(i)) || not (M(i) ≤ t))
          && (not (V ≤ pred (M(i0))) || not (V ≤ t)))
          && V < M(i0) && M(i0) < τ0 && τ0 ≤ t } ]
```

Validity subgoals. Two kinds of subgoals are generated by our tactic. First, we have formulae justifying the validity of the final constraints. We give here an example of the freshness local formula that originates from the two constraints that associate r to a local sampling.

```
(* The condition of the constraint for output@M(i)...*)
∀ (i:index), ((∀ (i0:index), not (M(i0) ≤ pred (M(i))) || not (V < M(i0)) || not (M(i0) ≤ t))
                && (not (V ≤ pred (M(i))) || not (V ≤ t)))
                && V < M(i)
                && M(i) ≤ t
 (*...and ...*)
 &&
 (*..the condition of the constraint for output@V...*)
 (∀ (i:index), not (M(i) ≤ pred V) || not (V < M(i)) || not (M(i) ≤ t)) && V ≤ t
 (* ...do not both holds at the same time. *)
 ⇒ false
```

Oracles subgoals. Finally, note that a call to the oracle encrypt with messages m_0 and m_1 requires that m_0 and m_1 are of same length. This is left to the user. We present here the formula returned to the user that originates for the oracle call when synthesizing output@V.

Protocol	Hypotheses	Synthesis	Property
Hash Lock	PRF	Insensitive	Strong secrecy
Basic Hash	EUF-MAC and PRF	Insensitive	Unlinkability
Global CPA	CPA	Insensitive	Basic indistinguishability
Private Authentication	$\text{CPA}^*_{\$}$	Insensitive	Anonymity
Partial NSL	CCA2*	Insensitive	CCA rewriting
NSL	CCA2*	Sensitive	Bob'key secrecy - CCA step

Figure 7.1: The case studies. For each protocol, we give the cryptographic games (hypotheses) used in the proof and the kind of security property proved. Cryptographic assumptions that were not available in the tool before this thesis are marked by an asterisk $*$. The Synthesis column precises whether the synthesis needed for the proof could rely on time-insensitive *invariants* or needed the *memoizing and time sensitive* invariants presented in Chapter 6.

```
(∀ (i:index), not (M(i) ≤ pred V) || not (V < M(i)) || not (M(i) ≤ t)) &&  V ≤ t
⇒ len (input@V) = len dummy
```

7.3 Case studies

We validate our approach first on several case studies, which we briefly describe below. These case studies are available on the main Squirrel repository [66] (in sub-directory `examples/crypto/`), except for the NSL case study, which can be found in [65] (in sub-directory `nsl/Bob-secrecy/`).

In these case studies, we were able to reprove Squirrel examples without the legacy tactics, as well as integrate new hardness assumption, which provided promising results.

> These case studies, except for NSL, were also provided as artifact material in [59] in a *non-maintained* and self-contained public repository [67]. It provided the source code of Squirrel at this date, the case studies (in sub-directory `case-studies-ccs/`), as well as HTML files that allow to replay the runs of Squirrel on each example without installing the tool.

In Figure 7.1, we provide a table summarizing the cases studies. We give some additional details below.

Hash Lock. The file `hash-lock.sp` presents the Squirrel proof of our example in Chapters 3 to 5, i.e. strong secrecy for the Hash Lock protocol, derived from the PRF game.

Basic Hash. We then illustrate how our **crypto** tactic can eventually replace existing tactics, on the example of the Basic Hash protocol, which was already proved unlinkable using the EUF and PRF legacy tactics. We adapt the same arguments using **crypto** with both EUF and PRF games in the file `basic-hash.sp`. This showed that our bi-

deduction verification was already powerful enough for real examples without memoizing and time-sensitive invariants.

Global CPA. Legacy cryptographic tactics in Squirrel can only handle #(_;_) in indistinguishabilities, and not in the protocols. Interestingly, **crypto** does not have this limitation: in `global-cpa.sp`, we prove the equivalence between two protocols outputting different values (of the same length) using **crypto** on the CPA game; such equivalences are often useful when reasoning about protocols.

Private authentication. We show, obviously, that our approach is not limited to cryptographic assumptions already supported by Squirrel. We prove anonymity of the Private Authentication protocol [68] in the file `private-authentication.sp` using a previously unsupported cryptographic assumption, $\text{CPA}_{\$}$, which roughly states the indistinguishability between an encrypted message and a fresh random sampling.

NSL. In the file `nsl.sp`, we prove a key step of a partial version of the NSL protocol [3], which relies on the CCA_2 game that was previously unsupported in Squirrel. This version has no additional actions to capture key secrecy: it proves the equivalence of the NSL protocol with its ideal version where encryptions are replaced by dummy ones, as presented in Chapter 1. This proof does not use the time-sensitive aspect of the simulator synthesis: we manage the induction outside the call to crypto, to make some deduction step by hand. This technic would not work with the protocol expressing the key secrecy. Hence, this last case study provides the proof of the CCA2 idealization game-hop for the protocol modelling Bob's key secrecy property (This is the protocol shown in Chapter 1). It relies on the memoizing simulators and time-sensitive invariants aspects to conclude.

7.4 The FOO protocol

We verified the security of the FOO e-voting protocol [44]. We chose FOO because it relies on cryptographic assumptions and primitives (CCA2, blind signatures, commitments) that were never considered in Squirrel before this work; and because a CCSA pen-and-paper proof of security for FOO already existed [27], giving us a head start.

As in [27], we focus on proving that FOO provides vote privacy, following Benaloh's vote swapping definition [69]. While the high-level structure of our proof follows that of [27], our security analysis is significantly more complicated, for two reasons. First and foremost, we *mechanized* our proof in Squirrel, which yields a development of approximately 10 kLoC (in contrast, [27] only provides a two-page proof sketch). Second, we consider an arbitrary number of dishonest voters, where [27] only has one — thus, their protocol only has a fixed number of agents. Having an unbounded number of agents significantly complicates the security analysis, as it forces us to rely on inductive reasoning.

We describe next FOO's cryptographic primitives, the high-level structure of the protocol, the modelling of vote privacy, and finally our Squirrel proof.

The Squirrel proof can be found in [65].

7.4.1 Cryptographic primitives and assumptions

To achieve its security goals, FOO uses mixnets, encryptions, blind signatures, and commitments. In this section, we present each primitive and the associated cryptographic game. We give special attention to the blind signature scheme, as the game we used in the proof is not the usual game associated with it. Our game is a specifically designed version, and using it requires pen and paper proof reductions to the usual one. The SQUIRREL games can all be found in the file `foo/Games.sp`.

Some primitives will share seeds type, whose declaration is recalled below.

```
type seed[ serializable,large ].
```

Asymmetric encryptions

Asymmetric encryption and its hardness assumption were presented and used several times in this thesis. We point to Section 7.2 for their SQUIRREL formalization.

Mixnets

A *mixnet* [63] is a (sub-)protocol which allows a collection of agents to send messages while hiding the relations between messages and senders. Typically, the agent's messages are encrypted with the mixnet public key. Once all messages have been received, the mixnet shuffles them at random, and publishes their decryptions.

> In practice, to protect the users' privacy against the mixnet itself, mixnets are composed of several independent servers, where each server does one round of shuffling and decryptions, typically augmented with zero-knowledge proofs of correct shuffling. Here we assume our mixnets to be honest agents.

We use the abstract modelling of shuffle presented in Chapter 6. For each mixnet, we assume the existence of a single (fictitious) public key whose corresponding secret key is held by an honest agent representing the mixnet as a whole. Once the mixnet is done receiving messages, it decrypts them and outputs their shuffling. Following [27], we leave the shuffling function unspecified, and only require that shuffling is invariant by permutation of its inputs through the following axiom:

```
axiom shuffle @system:any f p : bijective p → shuffle f = shuffle(λ i. f (p i))
```

Commitment

Cryptographic commitments [70] allow a user to commit to a value v while hiding v until it publishes the key k_c needed to open the commit. A commitment must be *binding* (a commit of v cannot be opened into a value $v' \neq v$) and *hiding* (as long as k_c remains secret, no adversary can learn anything about v from its commit). To prove FOO's privacy, we only need the commitment hiding.

Also, during the proof we exploited a second assumption on commitments: that they hide not only committed values but also keys. It trivially follows from the commitment

hiding property, as an adversary that can compute the commitment key can open commit and win against the commitment hiding.

For these hardness assumptions, we extended our games to *parameterized games*. A parameter is a value provided by the adversary at the beginning of the game to be set in the game's memory. It is identified by the flag #init in the syntax. Note that this is equivalent to define an initialization oracle and restricting the game to ensure this oracle is called first. In SQUIRREL it translates into an adversarial term that has to be provided when calling **crypto**. Here is the formalization in SQUIRREL:

```
type k_comm[ serializable,large ].  (* Commitement key *)

(* Commitement scheme primitives:  commit, and open.*)
op comm  : message → k_comm → message.
op copen : message → k_comm → message.

(* Axiom stating that a commit opened with its key yields the commited value.*)
axiom copen_comm @system:any x k : copen (comm x k) k = x.

(* Commitment hiding game.*)
game CommitmentHiding = {
  oracle challenge (m0,m1 : message) = {
    rnd key : k_comm;
    return comm diff(m0,m1) key;
  }
}.

(* Commitment key hiding game, consequence of the above game.*)
game CommitmentKeyHiding = {
  rnd key : k_comm;

  (* game parameter declaration *)
  let commited_message = #init;

  oracle commit = {
    return comm commited_message key;
  }
  oracle challenge (guess : k_comm) = {
    return diff(key = guess,false);
  }
}.
```

Notice that in the commitment hiding game, the key is a local sampling. We use this formalization as it simplified the games (there is no need for logs). This reduces to the classical game which has a global key. On the contrary, note that the key in the key hiding game is global. However, the oracle of the game can morally be called only once on a single input: the value with which the game is initialized.

Blind signature

Blind signatures [71] allow a user to ask for the signature of a message m to a signer $\mathcal{V}$ *without revealing* m *to* $\mathcal{V}$. In FOO, blind signatures are used to authenticate the voters' ballots without revealing its content to the authority who signs the ballots. This allows to consider a dishonest authority when analysing vote privacy.

```
type pk_sign[ serializable ].              (* public verification key *)
type sk_sign[ serializable,large ].        (* secret signing key *)
type token_bsign[ serializable,large ].    (* blinding token, sampled by the voters *)
type blinded[ serializable ].              (* blinded message, to be signed *)
type bsigned[ serializable ].              (* blinded signature *)
type signed[ serializable ].               (* unblinded signature *)

(* pk ← bskg(sk) generates the public key `pk` associated to the secret key `sk` *)
op bskg : sk_sign → pk_sign.

(* b ← blind(m,pk,t) computes the blinding `b` of a message `m` *)
op blind    : message → pk_sign → token_bsign → blinded.

(* bs ← bskg(b,sk,r) computes the blinded signature of the blinded message `b` *)
op bsign    : blinded → sk_sign → seed → bsigned.

(*  acc ← baccept(m,pk,t,bs) checks if `bs` is a blinded signature for message `m` *)
op baccept : message → pk_sign → token_bsign → bsigned → bool.

(* ub ← unblind(m,pk,t,bs) unblinds the blinded signature `bs` of message `m` using the token `t` *)
op unblind  : message → pk_sign → token_bsign → bsigned → signed.

(* ver ← bverifi(m,ub,pk) checks if `ub` is an unblinded signature of `m` *)
op bverif   : message → signed → pk_sign → bool.
```

Figure 7.2: Abstract types and operators modeling blind signatures.

In our formalization, we use a two-round blind signature, modelled by the abstract types and operators in Figure 7.2. Concretely, let sk be a secret key and $\mathsf{pk} = \mathsf{bskg}(\mathsf{sk})$ the associated public key. To obtain the signature of m, the user will generate a fresh blinding token t, and use it to compute the blinding $b \leftarrow \mathsf{blind}(m, \mathsf{pk}, \mathsf{t})$ of m and, crucially, b should not reveal anything about the message m. Then, the signer can blindly sign b by computing $\mathsf{bs} \leftarrow \mathsf{bsign}(b, \mathsf{sk}, \mathsf{r})$ (where r is the signature randomness). Then, the user can check that bs is a valid blind signature of m using $\mathsf{baccept}(m, \mathsf{pk}, \mathsf{t}, \mathsf{bs})$, and unblind bs to retrieve the unblinded signature ub through $\mathsf{unblind}(m, \mathsf{pk}, \mathsf{t}, \mathsf{bs})$ — note that the random blinding token t is needed here, which the signer does not know. Finally, any third party can check that ub is a valid signature for m using $\mathsf{bverif}(m, \mathsf{ub}, \mathsf{pk})$.

The proof relies on the Selective Failure Blindness [72], hardness assumption, which state that the blinding hides the blinded.

Furthermore, we design a game adapted for our proof of FOO: the Adaptative Selective Failure Blindness; and prove it reduces to the Selective Failure Blindness assumption. Below, we present the Selective Failure Blindness and Adaptative Selective Failure Blindness notions, prove the reduction from the latter to the former, and give the Squirrel formalization of the Adaptative Selective Failure Blindness.

> The usual hardness assumption for blind signature blinding is the Blindness [71] property. However, it was not applicable in our situation.
> The Selective Failure Blindness [72], is a stronger notion than Blindness, and, any blind signature scheme can be efficiently modified into a Selective Failure Blind

signature scheme [73]. Thus, relying on this stronger assumption is at little cost, and we assume our blind scheme respects the Selective Failure Blindness assumption.

Selective Failure Blindness. Formally, the Selective Failure Blindness [72] game is captured by the $\mathsf{SFBlind}$ experiment in Figure 7.3. This is a three phase experiment. In the first phase (*P1*), the adversary chooses a pair of messages (m_0, m_1) to be signed, as well as the public signing key (which may thus be dishonestly generated). In the second phase (*P2*), the adversary is provided with the blinding of m_0 and m_1, in an order that depends on b: if $b = 0$, it gets the blinding of (m_0, m_1); and of (m_1, m_0) if $b = 1$. More concisely, it gets (m_A, m_B) where $A = b$ and $B = 1 - b$. Then, it produces two candidates blind signatures bs_A and bs_B of, resp., m_A and m_B. Finally, in the last phase (*P3*), the experiment computes the unblinding ub_A and ub_B of the blind signatures bs_A and bs_B. Crucially, the experiment masks *both* signatures if *any one of them* was not correctly signed by the adversary. Then, it forwards them to the adversary in an order that is independent of b, i.e. it sends ub_0 and ub_1 — if the adversary got $\mathsf{ub}_A, \mathsf{ub}_B$ instead, it could trivially break the game by checking, e.g., whether ub_A is a valid signature for m_0.[1] Furthermore, the adversary is provided with the two bits $\mathsf{acc}_A, \mathsf{acc}_B$ informing it of which blinding were rejected by the user. If $\mathsf{acc}_A, \mathsf{acc}_B$ are both true, this information is already available to the adversary from the fact that $(\mathsf{ub}_A, \mathsf{ub}_B) \neq (\bot, \bot)$. But if that is not the case, the adversary may know which blinded signature was rejected (see [72, 73] for details).

Finally, the adversary wins if its guess b' is correct, i.e. $b = b'$. We say that a blind signature scheme satisfies the Selective Failure Blindness assumption if:

$$\mathsf{Adv}^{\mathsf{SFBlind}}(\eta) \stackrel{\mathsf{def}}{=} \left| \Pr(\mathsf{SFBlind}^{\eta}_{\mathcal{A}}) - \frac{1}{2} \right|$$

is negligible in η for all polynomial-time adversary $\mathcal{A}$.

Adaptive Selective Failure Blindness. The Selective Failure Blindness game provides the bits acc_A and acc_B non-adaptively to the adversary. This may be limiting in practice. Indeed, consider the following scenario[2]:

- The adversary first interacts with some user A to obtain the blinded message $\mathsf{blind}(m_A, \mathsf{pk}, t_A)$. At that point, it must send back the candidate blinded signature bs_A to A.
- Then, the adversary does the same for B: it obtains $\mathsf{blind}(m_B, \mathsf{pk}, t_B)$, and sends back the candidate blinded signature bs_B.
- Then the adversary eventually obtains the (guarded) unblinding of bs_0 and bs_1.

Further, assume that both A and B abort their execution if they obtain invalid blind signature — which is a reasonable behaviour for them. Then, we cannot directly apply

[1] Similarly, if both signatures were not masked when any of them was incorrect, the adversary could trivially win by correctly signing the blinding of m_A and incorrectly signing the blinding of m_B. Then, if ub_0 is valid, it means that $b = 0$. Otherwise, $b = 1$.

[2] This is the pattern we encountered with FOO's proof.

experiment $\mathsf{SFBlind}^{\eta}_{\mathcal{A}} = \{$
$b \xleftarrow{\$} \mathsf{bool};$

(P1: the adversary chooses the public key and messages to sign *)*
$(\mathsf{pk}, m_0, m_1) \leftarrow \mathcal{A}(1^{\eta});$

(P2: send blinded messages to be signed by the adversary *)*
$t_0 \xleftarrow{\$} \mathsf{token}; \quad t_1 \xleftarrow{\$} \mathsf{token};$
$A = b; \quad B = 1 - b;$
$(\mathsf{bs}_A, \mathsf{bs}_B) \leftarrow \mathcal{A}(\mathsf{blind}(m_A, \mathsf{pk}, t_A), \mathsf{blind}(m_B, \mathsf{pk}, t_B));$

(P3: adversary guesses the side *)*
$\mathsf{acc}_i \leftarrow \mathsf{baccept}(m_i, \mathsf{pk}, t_i, \mathsf{bs}_i); \qquad (\forall i \in \{0,1\})$
$\mathsf{ub}_i \leftarrow \mathsf{unblind}(m_i, \mathsf{pk}, t_i, \mathsf{bs}_i); \qquad (\forall i \in \{0,1\})$
(mask both unblinded signatures if any of them failed *)*
if $\neg(\mathsf{acc}_0 \wedge \mathsf{acc}_1)$ **then** $\{\mathsf{ub}_0 \leftarrow \bot; \mathsf{ub}_1 \leftarrow \bot;\}$ **else skip**
(unblinded signatures provided in an order independent of b *)*
(acceptance bits provided in the order (A, B) *)*
$b' \leftarrow \mathcal{A}(\mathsf{ub}_0, \mathsf{ub}_1, \mathsf{acc}_A, \mathsf{acc}_B);$

return$(b = b')$
$\}$

The adversary $\mathcal{A}$ is a stateful polynomial-time probabilistic program. All procedures and random samplings are implicitly parametrized by the security parameter η. The start of each phase is indicated by *P1*,*P2*, or *P3* in comment.

Figure 7.3: The Selective Failure Blindness cryptographic games.

the Selective Failure Blindness because we cannot simulate A and B's behaviour without knowing acc_A and acc_B *during phase* (*P2*) instead of at the start of phase *P3*.

To solve this issue, we propose the Adaptive Selective Failure Blindness game in Figure 7.4. In this game, the adversary may obtain the acceptance bits acc_A and acc_B as soon as they are available during phase *P2*, instead of at the beginning of phase *P3*. Concretely, in phase *P2*, the adversary can set the blind signatures bs_A and bs_B in the order of its choosing, using the **setBlindSig** oracle. Then, once bs_X has been set, the adversary may obtain acc_X by calling **getAcc**(X).

We say that a blind signature scheme satisfies the Adaptive Selective Failure Blindness assumption if:

$$\mathsf{Adv}_{\mathsf{ASFBlind}}(\eta) \stackrel{\mathsf{def}}{=} \left|\Pr(\mathsf{ASFBlind}^{\eta}_{\mathcal{A}}) - \frac{1}{2}\right|$$

is negligible in η, for all polynomial-time adversary $\mathcal{A}$.

Proposition 11. *Any Selective Failure blind signature is an Adaptive Selective Failure scheme. More precisely:*

$$\mathsf{Adv}_{\mathsf{ASFBlind}}(\eta) \leq 4 \times \mathsf{Adv}_{\mathsf{SFBlind}}(\eta)$$

experiment $\mathsf{ASFBlind}^{\eta}_{\mathcal{A}} = \{$
  $b \xleftarrow{\$} \mathsf{bool};$

  (P1: the adversary chooses the public key and messages to sign *)*
  $(\mathsf{pk}, m_0, m_1) \leftarrow \mathcal{A}(1^{\eta});$

  (P2: send blinded messages to be signed by the adversary *)*
  $t_0 \xleftarrow{\$} \mathsf{token};\quad t_1 \xleftarrow{\$} \mathsf{token};$
  $A = b;\quad B = 1 - b;$
  $\mathsf{bs}_0 \leftarrow \bot; \mathsf{bs}_1 \leftarrow \bot;$
  $() \leftarrow \mathcal{A}^{\mathbf{setBlindSig},\mathbf{getAcc}}(\mathsf{blind}(m_A, \mathsf{pk}, t_A), \mathsf{blind}(m_B, pk, t_B));$

  (P3: adversary guesses the side *)*
  $\mathsf{acc}_i \leftarrow \mathsf{baccept}(m_i, \mathsf{pk}, t_i, \mathsf{bs}_i);\qquad (\forall i \in \{0,1\})$
  $\mathsf{ub}_i \leftarrow \mathsf{unblind}(m_i, \mathsf{pk}, t_i, \mathsf{bs}_i);\qquad (\forall i \in \{0,1\})$
  (mask both unblinded signatures if any of them failed *)*
  if $\neg(\mathsf{acc}_0 \wedge \mathsf{acc}_1)$ **then** $\{\mathsf{ub}_0 \leftarrow \bot; \mathsf{ub}_1 \leftarrow \bot;\}$ **else skip**
  (unblinded signatures provided in an order independent of* b **)*
  $b' \leftarrow \mathcal{A}(\mathsf{ub}_0, \mathsf{ub}_1);$

  return$(b = b')$
}.

where
  oracle **setBlindSig**$(X, y) := \{$ **if** $(\mathsf{bs}_X = \bot)$ **then** $\mathsf{bs}_X \leftarrow y$ **else skip** $\}$
  oracle **getAcc**$(X) := \{$
    if $(\mathsf{bs}_X \neq \bot)$ **then return** $\mathsf{baccept}(m_X, \mathsf{pk}, t_X, \mathsf{bs}_X)$ **else skip**
  $\}$

The adversary $\mathcal{A}$ is a stateful polynomial-time probabilistic program. All procedures and random samplings are implicitly parametrized by the security parameter η. The start of each phase is indicated by *P1*,*P2*, or *P3* in comment.

Figure 7.4: The Adaptative Selective Failure Blindness cryptographic games.

Proof. This is a standard guessing step. Take $\mathcal{A}$ an adversary against the ASFBlind experiment. W.l.o.g., we assume that $\mathcal{A}$ calls each oracles **setBlindSig** and **getAcc** exactly once for $X = A$ and for $X = B$. We build $\mathcal{B}$ against SFBlind:

- Phase *P1*: $\mathcal{B}$ simulates $\mathcal{A}$ without any modification.
- Phase *P2*: $\mathcal{B}$ simulates $\mathcal{A}$. When $\mathcal{A}$ calls **setBlindSig**(X, bs_X), $\mathcal{B}$ logs the pair (X, bs_X). At the end of this phase, $\mathcal{B}$ returns the two logged values bs_A and bs_B. When $\mathcal{A}$ calls **getAcc**, the adversary $\mathcal{B}$ answers with a bit sampled uniformly at random. Further, it stores this bit for the next phase.
- Phase *P3*: $\mathcal{B}$ checks whether it correctly guessed the values of acc_A and acc_B. If that is not the case, it aborts its execution and return a bit b'' sampled uniformly at random. Otherwise, it simulates $\mathcal{A}$ without any modification and returns $\mathcal{A}$'s result.

Let Guess be the event indicating that $\mathcal{B}$ correctly guessed the values of acc_A and acc_B. Then,

$$\begin{aligned}
&\Pr(\mathsf{ASFBlind}^{\eta}_{\mathcal{B}}) \\
&= \Pr(\mathsf{ASFBlind}^{\eta}_{\mathcal{B}} \wedge \mathsf{Guess}) + \Pr(\mathsf{ASFBlind}^{\eta}_{\mathcal{B}} \wedge \neg\mathsf{Guess}) \\
&= \Pr(\mathsf{ASFBlind}^{\eta}_{\mathcal{B}} \wedge \mathsf{Guess}) + \Pr(b'' = b \wedge \neg\mathsf{Guess}) \\
&= \Pr(\mathsf{ASFBlind}^{\eta}_{\mathcal{B}} \wedge \mathsf{Guess}) + \frac{1}{2} \cdot \frac{3}{4} \\
&= \Pr(\mathsf{SFBlind}^{\eta}_{\mathcal{A}} \wedge \mathsf{Guess}) + \frac{3}{8} \\
&= \frac{1}{4} \cdot \Pr(\mathsf{SFBlind}^{\eta}_{\mathcal{A}}) + \frac{3}{8}
\end{aligned}$$

Thus,

$$\Pr(\mathsf{ASFBlind}^{\eta}_{\mathcal{B}}) - \frac{1}{2} = \frac{1}{4} \cdot \left(\Pr(\mathsf{SFBlind}^{\eta}_{\mathcal{A}}) - \frac{1}{2}\right)$$

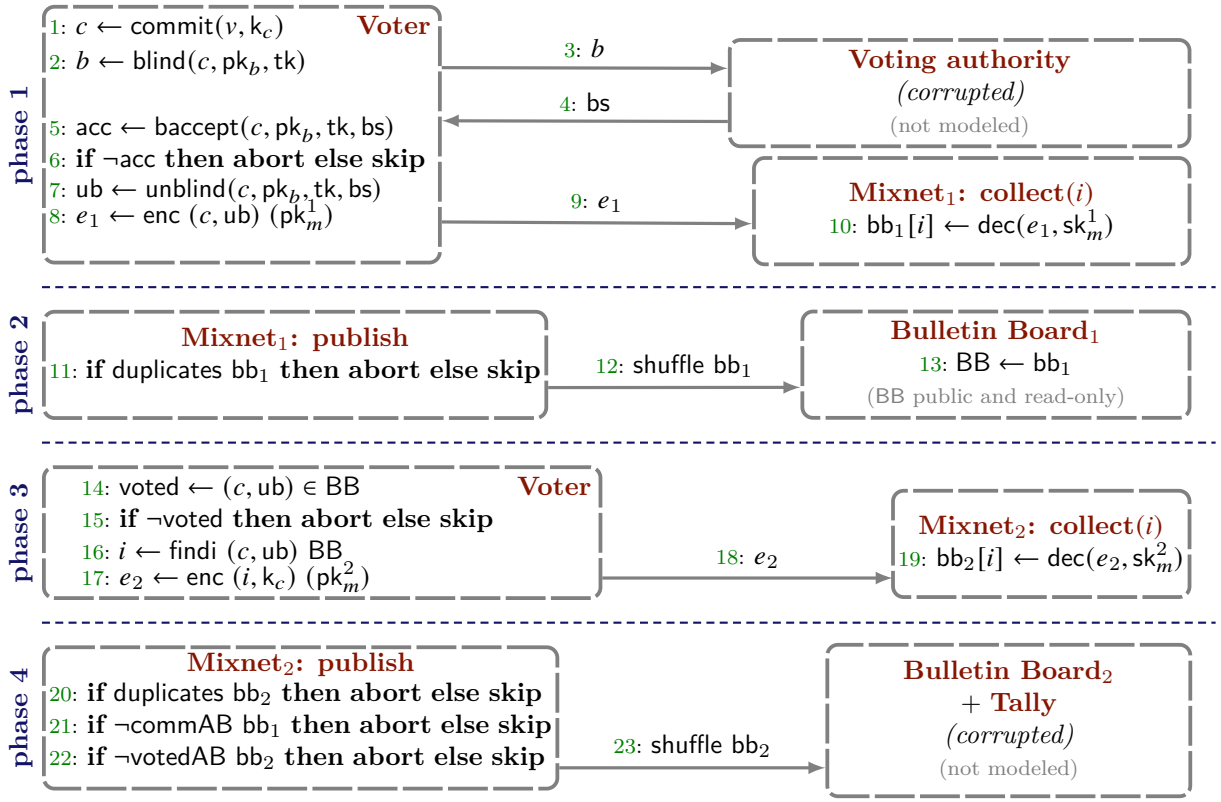
which concludes this proof. □

Encoding in Squirrel. In Figures 7.3 and 7.4, we gave the blindness games using a standard style for a cryptographer. But, in this style, these games do not fall into the class of cryptographic assumptions supported by the procedure we designed in Chapter 6. Fortunately, the games can be rewritten to fall into this class, using an alternative non-trivial encoding.

The experiment we define for Adaptative Selective Failure Blindness is phased. Phases are not natively supported in our games, but we use logs to encode them. Its Squirrel formalization can be found in Figure 7.5. The game defines two logs: logA and logB, which are supposed to register the signatures for A's blinding and B's blinding. Concretely, a non-empty log means that the corresponding acc oracle has been called on the value the log contains. With that idea we can prevent accA and accB oracles from being called several times, and ensure the unblind oracle is called on the same signatures, after accA and accB oracles.

```
game AdaptativeSelectiveFailureBlindness = {
  rnd tokenA : token_bsign;
  rnd tokenB : token_bsign;

  (* initialization parameters *)
  let m0 = #init;
  let m1 = #init;
  let pk = #init;

  var logA = empty_set;
  var logB = empty_set;

  (* oracle blinding and acc for A *)
  oracle blindingA = { return blind diff(m0,m1) pk tokenA;}

  oracle accA sA = {
    var logA' = logA;
    logA := add sAm logA;
    return if subseteq logA' (singleton sA) then baccept diff(m0,m1) pk tokenA sA
  }

  (* oracle blinding and acc for A *)
  oracle blindingB = { return blind diff(m1,m0) pk tokenB;}

  oracle accB sB = {
    var logB' = logB;
    var sBm = format sB;
    return if subseteq logB' (singleton sB) then  baccept diff(m1,m0) pk tokenA sB
  }

  (* unblind oracle *)
  oracle unblind (sA,sB : bsigned) = {
    var logA' = logA;
    var logB' = logB;

    var accA = baccept diff(m0,m1) pk tokenA sA;
    var accB = baccept diff(m1,m0) pk tokenB sB;
    var ub0 = unblind m0 pk diff(token0,token1) diff(sA,sB);
    var ub1 = unblind m1 pk diff(token1,token0) diff(sB,sA);

    logA := add sA logA;
    logB := add sB logB;

    return
      if subseteq logA' (singleton sA) && subseteq logB' (singleton sB) then
        if accA && accB then (ub0, ub1)
    }
}.
```

Figure 7.5: Adaptative Selective Failure Blindness SQUIRREL formalization.



Lines are labeled by integers (1, 2, ...) for quick referencing. Integer labels only roughly indicate the ideal execution order of protocol operations: e.g., the adversary may trigger the input 9 before the voter sent its blinded ballot to be signed (output 3).

Figure 7.6: The FOO e-voting protocol.

7.4.2 The FOO protocol security

FOO involves three parties: voters, who cast their ballots; the administrator, who issues credentials to voters (without learning their votes); and the collector, who gathers, mixes, and publishes ballots and results. In this section we present the modelling of FOO, how we expressed the privacy property and a high-level description of the SQUIRREL proof.

Modelling FOO

The FOO e-voting protocol, described in Figure 7.6, has four phases. In the first two phases, the voters publish the commit to their vote on a public bulletin board. In the last two phases, the voters publish their commit token to the public bulletin board, which allows to open the commits to the votes and to tally the election.

Phase 1. In **phase 1**, the voter computes the commitment c to its vote (1) using the commit token k_c. Then, blind signatures are used to let the voting authority authenticate the ballot anonymously: the voter blinds its commit (2), sends it to the voting authority (3) which sends back the blinded signature (4) that the voter verifies (5,6) and unblinds (7) unto the signature ub. Then, the voter publishes the ballot (c, ub), comprising the commit to its vote and the signature authenticating it, to the public bulletin board. Here, the

voter's anonymity is preserved by interposing a mixnet between the voter and the bulletin board. Concretely, ballots are encrypted and sent to the mixnet (8,9), which decrypts and collects them (10).

Phase 2. Once the first voting phase is over, we move to **phase 2**, where the mixnet shuffles all ballots and sends them in bulk to the bulletin board (12) which publishes them (13) — we assume that BB is a public read-only variable, which models the fact that everybody shares the same view on the immutable bulletin board. Before sending the ballots to the bulletin board, the mixnet checks that there are no duplicates ballots (11), which is necessary for privacy (see [74]).

Phase 3. In **phase 3**, the voter aborts if its ballot is not on the public bulletin board BB (14,15). Then, it retrieves the index i of its ballot on BB, and sends to the second mixnet (17,18) the encryption of the pair (i, k_c) of its ballot index and its commit token, which is decrypted and collected by the mixnet (19).

Phase 4. Finally, in **phase 4**, the second mixnet publishes the shuffled commit tokens (23). As in phase 2, it first checks that there are no duplicates (20). The additional checks (21,22) are necessary to model the privacy property, and are discussed below.

Privacy modelling

Model. We let the adversary control the bulletin board signing authority, and the tally. We consider two honest voters $\mathbf{Voter_A}$ and $\mathbf{Voter_B}$, and an arbitrary number of dishonest voters — dishonest voters are not modelled explicitly, as they are adversary-controlled, but implicitly, by letting the mixnets collect an arbitrary number of inputs. Following Benaloh's definition of privacy [69], we must show that

$$\begin{aligned} & \mathbf{P} \mid \mathbf{Voter_A}(v_0) \mid \mathbf{Voter_B}(v_1) \\ \sim\ & \mathbf{P} \mid \mathbf{Voter_A}(v_1) \mid \mathbf{Voter_B}(v_0) \end{aligned} \tag{7.1}$$

where v_0 and v_1 are arbitrary votes chosen by the adversary and $\mathbf{P}$ is the rest of the protocol, i.e. the mixnets and the bulletin board. As usual, we must rule out trivial privacy attacks against the indistinguishability in Eq. (7.1). Indeed, an adversary can trivially break security by letting the first voter cast its ballot, but blocking the ballot of the second voter. Further, the adversary does not cast any dishonest ballots itself. Then, inspecting the final bulletin board breaks the indistinguishability, as it only contains the first voter's vote.

Our model rules out this trivial attack by having the final mixnet publish the ballot box only if it contains the ballots of the two honest agents, for both phases (see checks 21,22).

Squirrel encoding. We encode the protocol in Squirrel following closely the description in Figure 7.6. The reader can refer to the file `foo/processes.sp` for details. To follow the protocol structure, we add axioms in Squirrel to capture the phases our protocol. These phases are delimited by two key timestamps: MVP and MOP that respectively

Group of files	LoC
Definitions and utilities	1652
Reduction to Privacy_CCA	3274
Deduction steps and shuffle opening	3242
Cryptographic arguments	2054

Table 7.1: Overview of the Squirrel development for FOO.

correspond to the time points after the first and the second mixnet outputs. This implies that

- frame@pred(MVP) corresponds to the frame at the end of **phase 1**;
- frame@MVP corresponds to the frame at the end of **phase 2**;
- frame@pred(MOP) corresponds to the frame at the end of **phase 3**; and
- frame@MOP corresponds to the frame at the end of **phase 4**.

7.4.3 Proof

We define in Squirrel a pair of systems, called Privacy_real, corresponding to Eq. (7.1). Contrary to the informal protocol description, our system never aborts: instead, if one of the aborting conditions of Figure 7.6 fails, the corresponding agent will output a dummy message. Thus it is always possible to keep executing the protocol until its very last action MOP (corresponding to 23), where the second mixnet publishes commit tokens. We show that for any trace ending with MOP, the two frames (with and without the votes swapped) are indistinguishable:

```
global theorem vote_privacy @system:Privacy_real : equiv(frame@MOP).
```

We provide in Table 7.1 an overview of how the 10 kLOC of the Squirrel development are split across the different steps of our proof, explained it this section.

CCA2 rewriting. The first step in the proof is to reduce this theorem to the same one but for a modified pair of systems, called Privacy_CCA, where all encryptions are replaced by same-length encryptions of zeroes. This step is itself justified by two reductions to the CCA2 game, for sk_m^1 and sk_m^2. In the resulting systems, the messages sent to the mixnets are completely hidden from the attacker, which is necessary for several reasoning steps in the rest of the proof.

Case analysis. Let us note φ the condition expressing that the two honest votes have successfully gone through the whole protocol. That is, the protocol never aborts. The conditions φ is then a conjunction of four conditions, i.e. $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$, defined as follows:

- In **phase 1**, both **Voter$_A$** and **Voter$_B$** did not abort if they accept the blind signature. We let φ_1 be this conditions, i.e. $\varphi_1 = \mathsf{acc}_A \wedge \mathsf{acc}_B$.

- In **phase 2**, the mixnet does not abort on condition $\varphi_2 \stackrel{\text{def}}{=}$ not duplicates bb_1.
- In **phase 3**, both $\mathbf{Voter_A}$ and $\mathbf{Voter_B}$ do not abort if they find their commit in the Bulletin Board. We let φ_3 be this conditions, i.e. $\varphi_3 = \mathsf{voted}_A \wedge \mathsf{voted}_B$.
- In **phase 4**, the mixnet does not abort on conditions φ_1, φ_2, φ_3 and $\varphi_4 \stackrel{\text{def}}{=}$ not duplicates bb_2.

Next, we proceed by case analysis [30] over the condition φ. This yield two subgoals for our proofs:

---

equiv(φ,**if** φ **then** frame@MOP).

---

and

---

equiv(φ,**if** not φ **then** frame@MOP).

Crucially, note that the condition φ also has to be deduced.

Deduction. Our goal now is to reduce the above indistinguishabilities to simpler indistinguishabilities. The idea is that we want to simplify the equivalence as much as possible for the application of cryptographic assumptions. To that end, we use *deduction*. Deduction is a predicate in SQUIRREL such that $\mathsf{u} \triangleright \mathsf{v}$ when there exists an *adversarial function* f such that $\mathsf{f}\ \mathsf{u} = \mathsf{v}$. In particular, we have that $\mathsf{u} \triangleright \mathsf{v}$ and **equiv**(u) implies **equiv**(v). We are going to find vectors of terms $\vec{r}$ and $\vec{r^{not}}$ such that

$$\vec{r} \triangleright \varphi, \textbf{if } \varphi \textbf{ then } \mathsf{frame@MOP} \text{ and } \vec{r^{not}} \triangleright \varphi, \textbf{if not } \varphi \textbf{ then } \mathsf{frame@MOP}.$$

Then, we reduce both subgoals' equivalence to proving the equivalences **equiv**($\vec{r}$) and **equiv**($\vec{r^{not}}$).

The deduction predicate is transitive. We exploited this fact to reduce our proof goals step by step. In particular, we show below the main deduction step used in both proofs. In both proofs, we are going at some point to prove that

$$\vec{t} \triangleright \textbf{if not dupplicate } bb_i \wedge \psi \textbf{ then shuffle } bb_i$$

for some vector of term $\vec{t}$ and boolean term ψ. This is the general form of mixnet outputs.

A crucial observation is that to deduce a shuffle $\mathsf{shuffle}(\lambda i.\ f\ i)$ of a function f it is sufficient to know the list of its inputs $(f\ i)$. Because of the key property of shuffles, we can modify the shuffle into $\mathsf{shuffle}(\lambda i.\ f\ (p\ i))$ for any permutation p of our choice in order to obtain the list of inputs $(f\ i)$ in a convenient order.

In practice, the only elements in shuffles we are interested in are $\mathbf{Voter_A}$ and $\mathbf{Voter_B}$'s data, if present. Based on the earlier observation, we design the following lemma in a simplified version in Figure 7.7 on shuffle, which basically extracts two points $(f\ i)$ and $(f\ j)$ from a shuffle of f to be the first and second elements of the list. Applying this lemma will be referred to as the shuffle.

Depending on φ, we are going to open the mixnets' shuffles differently, in order to apply specific cryptographic arguments (see later).

Finally, in both cases, the condition not duplicate bb_1 or not duplicate bb_2 implies that once $\mathbf{Voter_A}$ and $\mathbf{Voter_B}$'s data have been extracted the rest on the shuffle can only be

global lemma **@system**:Privacy_CCA open_shuffle (i,j:**index**) (φ:**bool**) (f: **index** → **message**) :
(φ,**if** φ **then** {i,j}, (i = j), f i, f j, $f_{\setminus\{i,j\}}$) ▷ (**if** φ **then** shuffle f)

Figure 7.7: Deduction lemma for shuffles. There, the element $\{i, j\}$ is the set of two elements i and j and $f_{\setminus\{i,j\}}$ is the function f where the image of both i and j are set to a dummy value. The elements {i,j} and (i = j) under condition φ are needed to extract the two specific points from f under condition φ during the proof.

an attacker computation with previous messages. That is how we deduce the component $f_{\setminus\{i,j\}}$. This argument hides auxiliary cryptographic arguments that are performed as part of the above decomposition. For example, the commit of an honest voter cannot be confused with that of another voter (honest or not), hence the failure of the duplication check at (11) can only be caused by two dishonest ballots, which is not a concern for privacy. Or we show that commits and commit keys remain secret in the relevant early phases of the protocol, using the blinding and commitment hiding and key-hiding assumptions on a truncated system where the later mixnets are ineffective.

Case φ. When φ holds, the attacker has access to the final publication of the commit keys. Although **A** and **B**'s commits can thus be opened to the vote they contain, their voting material from the first phase is still private thanks to blind signatures.

That is why it is crucial in this case that we open both shuffles to organize the voting material by *vote* rather than by *identity*. We let k_c^i c_i, acc_i and ub_i be the commitment key, commit, acceptance condition and unblinding (as in Figure 7.6) of the voter who voted v_i. The two shuffles (of keys and commits) will be opened by extracting in first position data related to 0 and in second position data related to 1.

Furthermore, in this case, the deduction is phased. The difficulty here is that we need the conditions in φ to rewrite terms it is guarding, but that φ also contains elements we want to erase though deduction.

In this case we find r1,r2,r3 and r4 such that

(1 *)* r4, $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$, **if** $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ **then** frame@pred(MOP) ▷ φ,**if** φ **then** frame@MOP
(2 *)* r3, $\varphi_1 \wedge \varphi_2$, **if** $\varphi_1 \wedge \varphi_2$ **then** frame@MVP ▷ $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$,**if** $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ **then** frame@pred(MOP)
(3 *)* r2, φ_1, **if** φ_1 **then** frame@pred(MVP) ▷ $\varphi_1 \wedge \varphi_2$,**if** $\varphi_1 \wedge \varphi_2$ **then** frame@MVP
(4 *)* r1, frame@init ▷ φ_1,**if** φ_1 **then** frame@pred(MVP)

In particular, the subgoals (1) and (3) correspond to shuffle openings, applying what we have seen earlier, the deduction (1) is done by having k_c^0, k_c^1 in r4, and the deduction (3) with c_0, c_1, if $\mathsf{acc}_0 \wedge \mathsf{acc}_1$ then $(\mathsf{ub}_0, \mathsf{ub}_1)$ in r2. Removing replicate, we have

r1,r2,r3,r4 ▷ m,b_A, acc_A, b_B, acc_B, c_0, c_1 **if** $\mathsf{acc}_0 \wedge \mathsf{acc}_1$ **then** $(\mathsf{ub}_0, \mathsf{ub}_1)$)

where m contains irrelevant cryptographic material (encryption keys, etc.), b_A the blinding received by $\mathbf{Voter_A}$, and b_B the blinding received by $\mathbf{Voter_B}$.

Hence, by transitivity we reduce to a simpler indistinguishability

equiv(m,b_A, acc_A, b_B, acc_B, c_0, c_1 **if** $\mathsf{acc}_0 \wedge \mathsf{acc}_1$ **then** $(\mathsf{ub}_0, \mathsf{ub}_1)$)

Finally, note that

$$b_A = \mathbf{diff}(b_0, b_1)$$
$$b_B = \mathbf{diff}(b_1, b_0)$$
$$\mathsf{acc_A} = \mathbf{diff}(\mathsf{acc_0}, \mathsf{acc_1}) \text{ and}$$
$$\mathsf{acc_B} = \mathbf{diff}(\mathsf{acc_1}, \mathsf{acc_0}).$$

We thus have an equivalence that has the form of a sequence of message of an adversary against the Adaptative Selective Failure Blindness game and we end the proof using **crypto** by reduction to this game.

Case not φ. When φ does not hold, the commitment keys are not revealed by the second mixnet. Hence, privacy follows from the commitment hiding property.

Before reducing to the hiding property, we must deal with the fact that the attacker may have altered $\mathbf{Voter_A}$ or $\mathbf{Voter_B}$ messages (e.g. blocking only $\mathbf{Voter_A}$'s messages), so we must organize shuffles' inputs by identity. We open the shuffles by always putting $\mathbf{Voter_A}$'s data first and $\mathbf{Voter_B}$'s data second, when they are present in the shuffles.

As before we reduce to a simple indistinguishability using deduction, showed below. This step is simpler than the case where φ holds, and it is done at once; we do not deduced phase by phase.

---

equiv(m',c_A,c_B)

---

where m' is a set of irrelevant terms, $c_\mathbf{A}$ the commit of $\mathbf{Voter_a}$ and $c_\mathbf{B}$ the commit of $\mathbf{Voter_B}$. Note that

$$c_\mathbf{A} = \mathsf{commint}\ \mathbf{diff}(v_0, v_1)\ k_{cA} \text{ and} \qquad c_\mathbf{B} = \mathsf{commint}\ \mathbf{diff}(v_1, v_0)\ k_{cB},$$

with k_{cA} and k_{cB} the commitment keys of respectively $\mathbf{Voter_a}$ and $\mathbf{Voter_b}$. Then, **crypto** reduces to the commitment's hiding game to end the proof.

Conclusion

Summary

We added support for cryptographic reductions to arbitrary cryptographic games in the CCSA-HO logic. To that end, we designed a novel variant of bideduction, improving upon the existing version by allowing to synthesize simulators that can perform oracle calls and random samplings. Our key contributions are: our novel bideduction judgement, which captures the existence of a simulator witnessing a cryptographic reduction, a bideduction rule that lifts this judgement to an indistinguishability of the logic by cryptographic reduction, and a proof system to derive bideduction judgements. Two key ingredients made this possible. First, we introduced the notion of constraint systems, which register randomness usage and ownership. Constraint system are crucial to ensure that the synthesized simulator handles randomness as an adversary must, and to couple the simulator's sample space with the logic's sample space. Second, we equipped the bideduction predicate with a Hoare-sytle assertion logic to capture the evolution of a game's memory through the proof system. The soundness of the proof system was proved. Once we defined the bideduction logic and proof system, we implemented an automated procedure that searches for bideduction proofs without backtracking. This proof search procedure can synthesize memoizing simulators and infer precise time-sensitive memory invariants. These two aspects are essential to lift a restriction we identified with the reductions to the CCA2 cryptographic game. Finally, we made our procedure available as a Squirrel tactic called **crypto** and used it to validate our approach on several case studies. Our evaluation showed that the tactic could replace legacy cryptographic tactics in practical cases as well as add support for new cryptographic assumptions not supported by Squirrel before. Nowadays, the **crypto** tactic is commonly used by other Squirrel users. Finally, we stress-tested our approach on a large case study: the FOO e-voting protocol. It is the most complex Squirrel proof to date. It also notably provides a novel proof method in Squirrel, relying on deduction.

Limitations and future work

The natural continuation for this work lies in the usage of **crypto** by Squirrel users. Indeed, we have already identified minor coding improvement that can be made in the embedding behind our proof search within Squirrel (e.g. ad-hoc handling of reachability games, etc.). Also, the usage of **crypto** will also test our framework on new case studies, and likely uncover other limitations. However, this work also lead to more fundamental open questions.

Exact semantics. We want to stress that our proof system relies on exact semantics: the semantics of the bideduction judgment relies on a notion of exact computation. As a result the rewriting rules of our proof system rewrite a term u into v only when u is exactly equal to v ($[u = v]_e$). Less obvious, however, is that the assertion logic is also used in an exact way. Indeed, looking back at the validity of our oracle triples, we see that it holds for any memory satisfying the pre-condition, for any tape. While this has not limited us in the use cases we have encountered so far, it remains theoretically unsatisfying: the CCSA-HO framework supports notions of overwhelming truth in predicates, which conveniently abstracts away probabilistic considerations. Yet, our framework introduces a gap here. Relaxing the exact semantics limitation also raises intriguing questions. For example, if we allow for "approximately correct simulators" — simulators that interact correctly with a game only with an overwhelming probability to deduce a term — how do we formally define them? And more critically, how do we justify the soundness of our rules? In this work, soundness follows from cryptographic assumptions, but with such approximations, this no longer holds strictly: the simulator no longer corresponds to a direct interaction with a game.

Assertion logic. The assertion logic we implemented is intentionally focused only on monotonous logs. While this suffices for our use cases, there is significant room for improvement. For instance, even "simple" cases like booleans are not directly supported—though workarounds exist using logs, native boolean support would be more intuitive, especially for phased games. Another key extension would be finite maps, a natural next step beyond logs. Finite maps are ubiquitous in cryptographic games (e.g., for modelling random oracles) and would greatly expand the framework's expressiveness.

Full automation. We opted for full automation because it aligns with the legacy cryptographic tactics usage in Squirrel, which was already fully automated, and offers a more user-friendly embedding compared to a tactics-based approach. However, this choice comes with trade-offs.

First, we might want to improve our proof search (e.g. to use rewriting lemmas, etc.) but efficiency would likely be a major obstacle for improvement. Since our tactic operates within a proof assistant, even a one-minute runtime can be prohibitive, restricting the scalability of our proof search.

Second, full automation reduces our control over the choices made during proof synthesis. This lack of flexibility demands significant effort in tuning the system, and understanding the proof search to ensure the synthesis works as intended. We can explore alternative mechanization strategies. The approach based on tactics is often more flexible as it does

not reduce user control. However, our synthesis procedure's three-phase structure and generation of long, complex invariants make a purely tactics-based approach impractical. These challenges suggest that a hybrid approach might strike a better balance between automation and control.

System guessing. There is currently a marge of improvement from cryptographic proofs to our approach. Often in practice, a game-hopping proof is thought as a system rewriting step: given a pair of games $(\mathcal{G}_0, \mathcal{G}_1)$ and a single system S_0, we define S_1 such that the indistinguishability of S_0 and S_1 reduces to the indistinguishability of $(\mathcal{G}_0, \mathcal{G}_1)$. Frameworks like CryptoVerif [38] follows that practice and implements the guessing part of system. Furthermore, S_0 and S_1 are (implicit) arguments of **crypto**. and, in Squirrel, users need to manually write down S_0 and S_1 for each proof step, this has been a time-consuming task for our proof of the FOO protocol. Mechanizing this process — i.e. infering S_1 from S_0— and automating it in Squirrel would significantly improve usability.

Theory follow up. Our new framework for simulator synthesis was designed with CCSA-H0 and Squirrel in mind. From a theoretical perspective, the key question is how applicable the framework can be to the broader domain of cryptographic verification. This leads to two main open questions. First, what can we say about the expressiveness of our logic? We aim to better characterize its capabilities: Is it complete, or can we identify the specific classes of reductions it effectively captures? Second, we can explore the framework's generality to understand how tightly it is tied to Squirrel. An intriguing direction is to generalize and adapt our approach to other systems, such as EasyCrypt, which could reveal its broader applicability. In particular, our framework roughly captures reductions of Squirrel-like protocols to games, which is less general than cryptographic reductions, we could explore how to adapt our framework for reductions of *games* to games.

List of Figures

1.1 The Niedam-Schroder protocol. . . . 3
1.2 Mallory attack. . . . 4
1.3 Needham-Schroder-Lowe protocol . . . 4

3.1 The Hash-Lock protocol . . . 40
3.2 Games for the PRF cryptographic assumption. . . . 42
3.3 Reduction to the PRF assumption. . . . 43
3.4 Syntax of programs. . . . 45
3.5 Syntax of games defined over oracle names O. . . . 45
3.6 Initial memory of a game $\mathcal{G}$ w.r.t. $\mathbb{M}$ and side bit i. . . . 47
3.7 Semantics of expressions w.r.t. a model $\mathbb{M} : \mathcal{E}$. . . . 48
3.8 Program semantics w.r.t. a model $\mathbb{M} : \mathcal{E}$, a side $i \in \{0, 1\}$ and a game $\mathcal{G}$. . . . 49

4.1 Reduction to the PRF assumption (copy of Figure 3.3). . . . 55
4.2 Constraint validity conditions . . . 59

5.1 Structural bideduction rules . . . 80
5.2 Computational bideduction rules . . . 81
5.3 Adversarial bideduction rules . . . 82

6.1 The CCA2 cryptographic game. . . . 108
6.2 An abstract mixnet protocol. . . . 109
6.3 Reduction to the CCA2 assumption. . . . 110
6.4 Abstract evaluation functions. . . . 116
6.5 Basic proof-search core rules. . . . 120
6.6 Basic proof-search destructive rules. . . . 121
6.7 Basic proof-search memory rules. . . . 122
6.8 Control-flow of $\mathsf{synthesize}_{\triangleright}(\cdot)$. . . . 123
6.9 The time-sensitive induction rule. . . . 127
6.10 Inductive simulator synthesis procedure $\mathsf{synthesize}^{\mathsf{rec}}_{\triangleright}(\cdot)$. . . . 128
6.11 CCSA encoding of our abstract mixnet protocol. . . . 129

7.1 Case studies . . . 143
7.2 Abstract types and operators modeling blind signatures. . . . 147
7.3 The Selective Failure Blindness cryptographic games. . . . 149
7.4 The Adaptative Selective Failure Blindness cryptographic games. . . . 150
7.5 Adaptative Selective Failure Blindness Squirrel formalization. . . . 152
7.6 The FOO e-voting protocol. . . . 153
7.7 Deduction lemma for shuffles. . . . 157

Bibliography

[1] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Commun. ACM*, vol. 21, p. 993–999, Dec. 1978.

[2] A. Kerckhoffs, "La cryptographie militaire," *J. Sci. Militaires*, vol. 9, no. 4, pp. 5–38, 1883.

[3] G. Lowe, "Breaking and fixing the needham-schroeder public-key protocol using fdr," in *Tools and Algorithms for the Construction and Analysis of Systems* (T. Margaria and B. Steffen, eds.), (Berlin, Heidelberg), pp. 147–166, Springer Berlin Heidelberg, 1996.

[4] "Documentation." Signal. https://signal.org/docs/.

[5] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3." RFC 8446, Aug. 2018.

[6] V. Cortier, C. C. Dragan, F. Dupressoir, and B. Warinschi, "Machine-checked proofs for electronic voting: Privacy and verifiability for belenios," in *CSF*, pp. 298–312, IEEE Computer Society, 2018.

[7] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, and A. Sorniotti, "An authentication flaw in browser-based single sign-on protocols: Impact and remediations," *Computers & Security*, vol. 33, pp. 41–58, 2013.

[8] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *2017 IEEE Symposium on Security and Privacy*, pp. 483–502, IEEE, 2017.

[9] "CVE-2014-0160 aka. the Heartbleed bug.." Available from MITRE, 2013.

[10] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, IEEE, 2019.

[11] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow,

S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[12] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of Computer and System Sciences*, vol. 28, no. 2, pp. 270–299, 1984.

[13] D. Dolev and A. C. Yao, "On the security of public key protocols (extended abstract)," in *22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, USA, 28-30 October 1981*, pp. 350–357, IEEE Computer Society, 1981.

[14] C. Cremers, A. Dax, and A. Naska, "Formal analysis of SPDM: Security protocol and data model version 1.2," in *32nd USENIX Security Symposium (USENIX Security 23)*, (Anaheim, CA), pp. 6611–6628, USENIX Association, Aug. 2023.

[15] D. A. Basin, R. Sasse, and J. Toro-Pozo, "The EMV standard: Break, fix, verify," *CoRR*, vol. abs/2006.08249, 2020.

[16] T. Wallez, J. Protzenko, and K. Bhargavan, "Treekem: A modular machine-checked symbolic security analysis of group key agreement in messaging layer security," in *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025* (M. Blanton, W. Enck, and C. Nita-Rotaru, eds.), pp. 4375–4390, IEEE, 2025.

[17] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 435–450, 2017.

[18] J. B. Almeida, M. Barbosa, G. Barthe, M. Campagna, E. Cohen, B. Gregoire, V. Pereira, B. Portela, P.-Y. Strub, and S. Tasiran, "A machine-checked proof of security for aws key management service," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, (New York, NY, USA), p. 63–78, Association for Computing Machinery, 2019.

[19] D. Unruh, "The impossibility of computationally sound xor," *Cryptology ePrint Archive*, 2010.

[20] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," *IACR Cryptol. ePrint Arch.*, p. 332, 2004.

[21] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Advances in Cryptology - EUROCRYPT 2006* (S. Vaudenay, ed.), (Berlin, Heidelberg), pp. 409–426, Springer Berlin Heidelberg, 2006.

[22] M. Fischlin and A. Mittelbach, "An overview of the hybrid argument." Cryptology ePrint Archive, Paper 2021/088, 2021.

[23] G. Bana and H. Comon-Lundh, "A computationally complete symbolic attacker for equivalence properties," in *Proceedings of the 2014 ACM SIGSAC Conference on*

Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014 (G. Ahn, M. Yung, and N. Li, eds.), pp. 609–620, ACM, 2014.

[24] D. Baelde, A. Koutsos, and J. Lallemand, "A higher-order indistinguishability logic for cryptographic reasoning," in *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2023, Boston, MA, USA, June 26-29, 2023*, pp. 1–13, IEEE, 2023.

[25] G. Scerri and R. Stanley-Oakes, "Analysis of key wrapping apis: Generic policies, computational security," in *CSF*, pp. 281–295, IEEE Computer Society, 2016.

[26] H. Comon and A. Koutsos, "Formal computational unlinkability proofs of RFID protocols," in *CSF*, pp. 100–114, IEEE Computer Society, 2017.

[27] G. Bana, R. Chadha, and A. K. Eeralla, "Formal analysis of vote privacy using computationally complete symbolic attacker," in *ESORICS (2)*, vol. 11099 of *Lecture Notes in Computer Science*, pp. 350–372, Springer, 2018.

[28] A. Koutsos, "The 5G-AKA authentication protocol privacy," in *EuroS&P*, pp. 464–479, IEEE, 2019.

[29] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and S. Moreau, "An interactive prover for protocol verification in the computational model," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pp. 537–554, IEEE, 2021.

[30] D. Baelde, S. Delaune, A. Koutsos, and S. Moreau, "Cracking the stateful nut: Computational proofs of stateful security protocols using the squirrel proof assistant," in *CSF*, pp. 289–304, IEEE, 2022.

[31] C. Cremers, C. Fontaine, and C. Jacomme, "A logic and an interactive prover for the computational post-quantum security of protocols," in *SP*, pp. 125–141, IEEE, 2022.

[32] S. Jeanteur, L. Kovács, M. Maffei, and M. Rawson, "Cryptovampire: Automated reasoning for the complete symbolic attacker cryptographic model," in *2024 IEEE Symposium on Security and Privacy (SP)*, (Los Alamitos, CA, USA), pp. 259–259, IEEE Computer Society, may 2024.

[33] L. Kovács and A. Voronkov, "First-order theorem proving and vampire," in *CAV*, vol. 8044 of *Lecture Notes in Computer Science*, pp. 1–35, Springer, 2013.

[34] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin, "Computer-aided security proofs for the working cryptographer," in *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings* (P. Rogaway, ed.), vol. 6841 of *Lecture Notes in Computer Science*, pp. 71–90, Springer, 2011.

[35] D. A. Basin, A. Lochbihler, and S. R. Sefidgar, "Crypthol: Game-based proofs in higher-order logic," *J. Cryptol.*, vol. 33, no. 2, pp. 494–566, 2020.

[36] P. G. Haselwarter, E. Rivas, A. Van Muylder, T. Winterhalter, C. Abate, N. Sidorenco, C. Hritcu, K. Maillard, and B. Spitters, "Ssprove: A foundational framework for modular cryptographic proofs in coq," *ACM Trans. Program. Lang. Syst.*, vol. 45, no. 3, pp. 15:1–15:61, 2023.

[37] D. Baelde, C. Fontaine, A. Koutsos, G. Scerri, and T. Vignon, "A probabilistic logic for concrete security," in *37th IEEE Computer Security Foundations Symposium, CSF 2024, Enschede, Netherlands, July 8-12, 2024*, pp. 324–339, IEEE, 2024.

[38] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Trans. Dependable Secur. Comput.*, vol. 5, no. 4, pp. 193–207, 2008.

[39] G. Barthe, J. M. Crespo, B. Grégoire, C. Kunz, Y. Lakhnech, B. Schmidt, and S. Z. Béguelin, "Fully automated analysis of padding-based encryption in the computational model," in *CCS*, pp. 1247–1260, ACM, 2013.

[40] A. J. Malozemoff, J. Katz, and M. D. Green, "Automated analysis and synthesis of block-cipher modes of operation," in *CSF*, pp. 140–152, IEEE Computer Society, 2014.

[41] V. T. Hoang, J. Katz, and A. J. Malozemoff, "Automated analysis and synthesis of authenticated encryption schemes," in *CCS*, pp. 84–95, ACM, 2015.

[42] G. Barthe, E. Fagerholm, D. Fiore, A. Scedrov, B. Schmidt, and M. Tibouchi, "Strongly-optimal structure preserving signatures from type II pairings: synthesis and lower bounds," *IET Inf. Secur.*, vol. 10, no. 6, pp. 358–371, 2016.

[43] J. Gancher, S. Gibson, P. Singh, S. Dharanikota, and B. Parno, "Owl: Compositional verification of security protocols via an information-flow type system," in *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, (Los Alamitos, CA, USA), pp. 1130–1147, IEEE Computer Society, may 2023.

[44] A. Fujioka, T. Okamoto, and K. Ohta, "A practical secret voting scheme for large scale elections," in *Advances in Cryptology — AUSCRYPT '92* (J. Seberry and Y. Zheng, eds.), (Berlin, Heidelberg), pp. 244–251, Springer Berlin Heidelberg, 1993.

[45] M. Rusinowitch, R. Küsters, M. Turuani, and Y. Chevalier, "An np decision procedure for protocol insecurity with xor," in *Logic in Computer Science, Symposium on*, (Los Alamitos, CA, USA), p. 261, IEEE Computer Society, jun 2003.

[46] H. Comon-Lundh and V. Shmatikov, "Intruder deductions, constraint solving and insecurity decision in presence of exclusive or," in *LICS*, p. 271, IEEE Computer Society, 2003.

[47] S. Bursuc, H. Comon-Lundh, and S. Delaune, "Deducibility constraints and blind signatures," *Inf. Comput.*, vol. 238, pp. 106–127, 2014.

[48] H. Comon-Lundh, V. Cortier, and G. Scerri, "Tractable inference systems: An extension with a deducibility predicate," in *CADE*, vol. 7898 of *Lecture Notes in Computer Science*, pp. 91–108, Springer, 2013.

[49] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE (1)*, pp. 215–224, ACM, 2010.

[50] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, pp. 62–73, ACM, 2011.

[51] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," in *POPL*, pp. 599–612, ACM, 2017.

[52] D. Perelman, S. Gulwani, D. Grossman, and P. Provost, "Test-driven synthesis," in *PLDI*, pp. 408–418, ACM, 2014.

[53] S. Kremer and M. Ryan, "Analysis of an electronic voting protocol in the applied pi calculus," in *ESOP*, vol. 3444 of *Lecture Notes in Computer Science*, pp. 186–200, Springer, 2005.

[54] S. Delaune, M. Ryan, and B. Smyth, "Automatic verification of privacy properties in the applied pi calculus," in *IFIPTM*, vol. 263 of *IFIP Advances in Information and Communication Technology*, pp. 263–278, Springer, 2008.

[55] R. Chadha, V. Cheval, Ş. Ciobâcă, and S. Kremer, "Automated verification of equivalence properties of cryptographic protocols," *ACM Trans. Comput. Log.*, vol. 17, no. 4, p. 23, 2016.

[56] V. Cortier, C. C. Dragan, F. Dupressoir, B. Schmidt, P. Strub, and B. Warinschi, "Machine-checked proofs of privacy for electronic voting protocols," in *IEEE Symposium on Security and Privacy*, pp. 993–1008, IEEE Computer Society, 2017.

[57] C. C. Dragan, F. Dupressoir, E. Estaji, K. Gjøsteen, T. Haines, P. Y. A. Ryan, P. B. Rønne, and M. R. Solberg, "Machine-checked proofs of privacy against malicious boards for selene & co," in *CSF*, pp. 335–347, IEEE, 2022.

[58] T. E. development team, "The EasyCrypt Prover repository," accessed august 2025. https://github.com/EasyCrypt/easycrypt/.

[59] D. Baelde, A. Koutsos, and J. Sauvage, "Foundations for cryptographic reductions in CCSA logics," in *CCS*, pp. 2814–2828, ACM, 2024.

[60] A. Juels and S. A. Weis, "Defining strong privacy for rfid," *ACM Trans. Inf. Syst. Secur.*, vol. 13, Nov. 2009.

[61] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *J. ACM*, vol. 33, p. 792–807, Aug. 1986.

[62] G. Barthe, T. Espitau, B. Grégoire, J. Hsu, L. Stefanesco, and P. Strub, "Relational reasoning via probabilistic coupling," in *LPAR*, vol. 9450 of *Lecture Notes in Computer Science*, pp. 387–401, Springer, 2015.

[63] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Commun. ACM*, vol. 24, no. 2, pp. 84–88, 1981.

[64] D. Wikström, "A universally composable mix-net," in *TCC*, vol. 2951 of *Lecture Notes in Computer Science*, pp. 317–335, Springer, 2004.

[65] D. Baelde, A. Koutsos, and J. Sauvage, "Foo proof repository." https://gitlab.inria.fr/jusauvag/crypto-foo.

[66] T. S. development team, "The Squirrel Prover repository," August 2024. https://github.com/squirrel-prover/squirrel-prover/.

[67] D. Baelde, A. Koutsos, and J. Sauvage, "Artifact for "foundations for cryptographic reductions in ccsa logics." ⟨hal-04650670⟩, 2024.

[68] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication," *ACM Trans. Comput. Syst.*, vol. 8, p. 18–36, feb 1990.

[69] J. D. C. Benaloh, *Verifiable secret-ballot elections*. Yale University, 1987.

[70] M. Naor, "Bit commitment using pseudorandomness," *J. Cryptol.*, vol. 4, no. 2, pp. 151–158, 1991.

[71] D. Chaum, "Blind signatures for untraceable payments," in *CRYPTO*, pp. 199–203, Plenum Press, New York, 1982.

[72] J. Camenisch, G. Neven, and A. Shelat, "Simulatable adaptive oblivious transfer," in *EUROCRYPT*, vol. 4515 of *Lecture Notes in Computer Science*, pp. 573–590, Springer, 2007.

[73] M. Fischlin and D. Schröder, "Security of blind signatures under aborts," in *Public Key Cryptography*, vol. 5443 of *Lecture Notes in Computer Science*, pp. 297–316, Springer, 2009.

[74] V. Cortier and B. Smyth, "Attacking and fixing Helios: An analysis of ballot secrecy," *J. Comput. Secur.*, vol. 21, no. 1, pp. 89–148, 2013.

RÉSUMÉ

Cette thèse étudie la vérification des protocoles cryptographiques dans le cadre CCSA, une approche de vérification formelle basée sur une logique probabiliste pour prouver les propriétés de sécurité dans le modèle computationnel. Cette approche est implémentée dans l'assistant de preuve SQUIRREL. Cette thèse s'intéresse à la mécanisation des réductions cryptographiques, une technique de preuve centrale en cryptographie où la sécurité d'un protocole est réduite à une hypothèse calculatoire cryptographique par la construction d'un simulateur.
Avant cette thèse, le cadre CCSA fournissait des axiomes logiques dont la validité était établie manuellement par des réductions. Ces réductions sont une source possible d'erreurs et les axiomes logiques n'avait été conçus seulement pour un nombre restreint d'hypothèses calculatoires (par exemple, CCA, PRF, EUF-MAC). Chaque axiome nécessitait également un effort d'implémentation, lui aussi source d'erreurs. Malheureusement, ces tâches (conception, preuve et implémentation des axiomes) étaient inaccessibles aux utilisateurs typiques, limitant ainsi la capacité de l'approche CCSA à passer à l'échelle.
La contribution majeure de cette thèse est une approche permettant de capturer des réductions vers des jeux cryptographiques arbitraires dans le cadre de la logique CCSA. Nous introduisons une logique dont le prédicat central, le prédicat de bidéduction, formalise l'existence d'un simulateur justifiant une réduction cryptographique. Nous proposons ensuite un système de preuve pour dériver ces prédicats, qui infère implicitement les simulateurs. Nous avons en outre implémenté dans SQUIRREL une procédure de recherche de preuve qui synthétise des simulateurs qui mémoïsent et génèrent des invariants sensibles au temps pour justifier la correction des simulateurs inférés. Notre implémentation élargit significativement la portée des preuves dans SQUIRREL, en étendant l'ensemble des hypothèses calculatoires cryptographiques supportées. Pour valider notre approche, nous l'avons appliquée à des études de cas, en reproduisant des preuves existantes dans SQUIRREL et en traitant de nouveaux cas qui n'étaient pas prouvables auparavant. Ce travail culmine avec la première preuve mécanisée de la confidentialité des votes pour le protocole de vote électronique FOO — la plus grande preuve réalisée à ce jour dans SQUIRREL.

MOTS CLÉS

Jeux cryptographiques, Système de vérification, Logique CCSA, Preuves de sécurité

ABSTRACT

This thesis investigates cryptographic protocol verification in the CCSA framework, a formal verification approach based on a probabilistic logic for proving security properties in the computational model. This framework is implemented in the Squirrel proof assistant. The main focus of the thesis is the mechanization of cryptographic reductions — a core proof technique in cryptography in which the security of a protocol is reduced to a cryptographic hardness assumption via the construction of a simulator.
Prior to this thesis, the CCSA framework provided logical axioms whose soundness was established through manual, error-prone reductions to a fixed set of cryptographic hardness assumption (e.g., CCA, PRF, EUF-MAC). Each axiom also necessitated implementation efforts, which were prone to errors. Unfortunately, these tasks (designing, proving, and implementing the axioms) were inaccessible to typical users, thus limiting the scalability of the CCSA approach.
The main contribution of this thesis is a framework that captures reductions to arbitrary cryptographic games for the CCSA framework. We introduce a logic with a core predicate, the *bideduction predicate*, which express the existence of a simulator justifying a cryptographic reduction. We then provide a proof system to derive such predicates, implicitly inferring simulators. We further implement in SQUIRREL a proof search procedure that synthesize memoizing simulators and generates time-sensitive invariants to justify the inferred simulator's correctness. Our implementation significantly extends SQUIRREL's scope as it extends the set of supported cryptographic hardness assumptions. To validate our approach, we apply it to case studies, reproving existing SQUIRREL case studies and analysing new ones which were not provable in SQUIRREL before. This work culminates with the first mechanized proof of ballot privacy for the FOO e-voting protocol the largest proof conducted in SQUIRREL to date.

KEYWORDS

Cryptographic games, Verification system, CCSA logic, Security proofs