Deciding Fragments of $(\alpha, \beta, \gamma, \delta)$-Privacy

Laouen Fernet

Kongens Lyngby 2021

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Communication protocols can be formalised as symbolic models, in order to study various security goals. The formulation of desired goals can be difficult for protocol designers and challenging with regards to automated verification. Most existing approaches considering privacy rely on an observational equivalence notion making it hard to express privacy goals and also verify them automatically.

The logical approach introduced as (α, β)-privacy allows to specify privacy goals in a declarative and intuitive manner. It is based on first-order logic with Herbrand universes, where a *payload formula* α defines information disclosed on purpose at an abstract level, and a *technical information formula* β includes cryptographic messages from the protocol specification. The idea is to verify if there is a consequence of β that does not follow from α alone. (α, β)-privacy enables verification of privacy goals based on a logical context, without relying on observational equivalence. However, procedures to perform automated verification are lacking for this approach.

This thesis deals with automated verification for the core problem (α, β)-privacy: the goal is to verify automatically if privacy holds in a given state of a transition system. The idea developed in this work is to compare a protocol specification with a concrete execution, in order to find relations between variables of the specification. The question is then whether the relations found between the variables break privacy or not. The method that we follow is to define algorithms that can be used to derive a decision procedure. The procedure is proved to be correct and implemented to provide basic computer-aided verification, constituting an initial step for software support.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science of the Technical University of Denmark in fulfilment of the requirements for acquiring a Master of Science in Engineering degree. The project corresponds to 30 ECTS and was carried out under the supervision of Associate Professor Sebastian Alexander Mödersheim.

The thesis deals with automated reasoning about privacy goals using the logical approach introduced as $(\alpha, \beta, \gamma, \delta)$-privacy.

The thesis consists of the report "Deciding Fragments of $(\alpha, \beta, \gamma, \delta)$-Privacy" and a file `Lib.hs` of Haskell source code.

Lyngby, 04-June-2021

Laouen Fernet

Acknowledgements

I would like to thank my supervisor Sebastian Mödersheim, who was of huge help through this project. His works developing the (α, β)-privacy approach were a fundamental basis for my thesis. I am very grateful for his guidance, which made it possible to provide meaningful formal proofs for the procedure designed.

I would also like to thank my friends for their support during this thesis, and especially the great people from Køkken 1000 which made my studies at DTU a very pleasant experience.

Contents

Chapter 1

Introduction

The problem of privacy is relevant in many fields, such as electronic voting, digital health information, mobile payments and distributed systems in general. Privacy is a security goal of its own, it cannot be described as regular secrecy. For example, in voting it is not the values of the votes that are secret, since there is a public tally, but rather the *relation* between a voter and a vote. It is best if privacy is taken into account during the design of communication protocols. But even then, it is difficult to get enough guarantees about privacy goals. Formal methods are a successful way of addressing the issue. By studying a protocol at an abstract level, they can be used to check digital applications against possible misuse.

The symbolic modelling of protocols allows to define various privacy goals. The standard approach uses the notion of observational equivalence [9, 8]: it is common to consider privacy as a bisimilarity between processes in the applied π-calculus. As an example, for electronic voting protocols, a privacy goal could be that two processes differing by a swap of votes are indistinguishable [4, 10, 17]. While tools exist to provide automated verification [3, 5], it can be hard to formalise a privacy goal as bisimilarity property, so automated verification is actually challenging. Moreover, it is not clear whether the goals defined in this way cover all properties implied by the privacy goals desired.

(α, β)-privacy [16] is an approach based on first-order logic, which allows us

to specify privacy goals without referring to observational equivalence and thus promises to give a new path for automated verification. It is also an intuitive and declarative way to specify privacy. Recent works have adapted this to transition systems and thus recast privacy as a reachability problem [12]; this context is referred to as $(\alpha, \beta, \gamma, \delta)$-privacy. This logical approach enables manual proving, nevertheless the lack of automation is a limitation for real-world applications of fully-fledged protocols. In the general case, (α, β)-privacy is undecidable because first-order logic is. However, there are fragments covering large classes of protocols that are actually decidable.

In this thesis, we investigate decidable fragments (α, β)-privacy. We develop algorithms and design a decision procedure allowing automated verification of privacy goals. We also tackle problems that arise from conditional branching. Even though this is not necessary in all protocols, it expands the scope of the procedure designed. Our work forms a basic framework to automatically verify privacy, where a modeller provides a protocol specification and a declaration of privacy goals.

1.1 Thesis structure

We give an overview of the general problem of privacy in protocols in Section 1.2, as well as an explanation of the method followed. Our idea is to consider an intruder that is trying to break privacy. In Chapter 2, we recall useful notions from previous works: we define Herbrand logic, and give a partial overview of (α, β)-privacy as introduced in [16] and its extension to transition systems, since this constitutes a necessary basis for our work.

In Chapter 3, we present our first contributions. We start by defining an *intruder theory*, which represents what our intruder is able to do, then we complement the notion of frames with what we call *frames with shorthands*. In Chapter 4, we define different algorithms, breaking down the problem in several steps. For these algorithms, we declare a number of properties that we prove in order to show correctness of the procedure. First, we limit our focus on protocols without branching. In this situation, the intruder uses their knowledge of the protocol specification and one concrete execution. They try to learn more information than what was intentionally disclosed, according to the rules of the intruder theory. Then, we lift in Chapter 5 our results to the case of protocols with branching, i.e. the intruder makes a number of hypotheses about which branch corresponds the actual protocol execution.

In Chapter 6, we discuss the limitations of the proposed procedure and outline

possible future work to tackle the remaining issues. Finally, we conclude in Chapter 7 by summing up the work achieved.

1.2 Background material

There are many examples of communication protocols that are not secure with regards to privacy. This is the case also for protocols which have been designed to provide some privacy goals. As an example, recent papers show privacy issues in voting protocols (Helios [4, 10]) as well as contact-tracing applications (GAEN API [6], SwissCovid [18, 14]). In such cases, it is hard to specify all desirable privacy goals using the notion of observational equivalence. Additionally, the standard approach cannot guarantee that the privacy goals verified cover all possibilities of misuse. These limits are the motivation for studying a new approach that is declarative and more intuitive.

This thesis builds on top of the notion of (α, β)-privacy. An essential concept of this logical approach is *Herbrand logic* introduced in [13], which is first-order logic with Herbrand universes. It allows to encode properties verified by some functions, e.g. the algebraic equations that must hold for cryptographic operators. The main definitions needed for this thesis will be given in Chapter 2. All of them are taken from [16], which specifies in details the logic and (α, β)-privacy. The basic idea is to define privacy goals as a *payload formula* α, which represents the publicly disclosed information. For instance, in an electronic protocol this could be the tally at the end of the vote. A *technical information formula* β includes α but also all information related to the protocol specification, e.g. actual cryptographic messages that have been exchanged. Then privacy is verified if the intruder cannot derive more information from β that they can from α, i.e. no additional sensitive information is leaked by the protocol execution. Contrary to the standard approach of observational equivalence, (α, β)-privacy can be used to declare privacy goals in a very intuitive manner. As it is reasoning about *any* logical consequence to verify privacy, it does provide guarantees that all corners have been covered.

The approach of (α, β)-privacy can be extended to transition systems with the concept of $(\alpha, \beta, \gamma, \delta)$-privacy [12], where a *truth formula* γ represents the real execution of the protocol, and a sequence of *conditional updates* δ represents the stateful execution. The question of whether privacy is verified by the entire transition system is to verify if it holds in every state. The core problem is therefore to study privacy in a single state, and it remains to lift this procedure to a transition system. This does not pose any theoretical difficulty, as it can be done by simply exploring all reachable states and applying the procedure in

each of them. In our case, we limit the scope of the procedure to specific classes of protocols, namely protocols with a bounded number of transactions without or with branching, and we will only verify privacy in a single state.

The formalisation that we use relies on the notion of *terms*, which is the symbolic representation of messages exchanged according to a protocol. Our definition of intruder theory uses this notion of terms, and the algorithms that we define rely on the set of algebraic equations specified. The approach taken in the modelling and the formal proving of the problem is inspired by chapters from [2], which deals with many topics around term rewriting. For instance, we want that the algebraic equations allow to generate a well-founded analysis rewriting system which is convergent. This means roughly that the order of the rules that the intruder can apply does not matter and that they can be applied a finite number of times.

Chapter 2

Preliminaries

The approach of (α, β)-privacy is based on logic. It uses more specifically Herbrand logic [13], which is first-order logic with Herbrand universes. The overall idea is that the specification of a protocol can be modelled in Herbrand logic. The verification of privacy that we perform considers this symbolic model of the protocol. Thus, we abstract away from actual implementation of protocols and cryptography, and apply formal methods to *prove* privacy goals.

In this chapter, we give the definitions from [16] that our work builds upon. First, we describe Herbrand logic in Section 2.1 and define the Herbrand universe that we will consider. To represent knowledge of messages, we use a definition of frames and explain how they are encoded in the logic in Section 2.2. We then specify the framework of (α, β)-privacy in Section 2.3.

2.1 Herbrand logic

The formalisation of (α, β)-privacy requires an appropriate logic. First-order logic might seem an intuitive choice at first, however it cannot be used properly in our case. Indeed, we would like to specify algebraic equations characterising the behaviour of cryptographic operators. The authors of [16] explain why it

is not possible to define axioms ensuring a correct interpretation of constants and cryptographic operators. However, the use of Herbrand universes solves the issues and the formalisation is therefore done in Herbrand logic.

Definition 2.1 (Syntax of Herbrand Logic [16]) Let $\Sigma = \Sigma_f \uplus \Sigma_i \uplus \Sigma_r$ be an alphabet that consists of

- a set Σ_f of *free function symbols*,
- a set Σ_i of *interpreted function symbols* and
- a set Σ_r of *relation symbols*,

all with their arities. We write f^n and r^n to denote a function symbol f and a relation symbol r of arity n, respectively.

We write $\mathsf{f}(t_1, \ldots, t_n)$ when $\mathsf{f} \in \Sigma_f$ and $f[t_1, \ldots, t_n]$ when $f \in \Sigma_i$, and we denote the set of considered *cryptographic operators* by the subset $\Sigma_{op} \subseteq \Sigma_f$. *Constants* are the special case of function symbols with arity 0; for an uninterpreted constant $\mathsf{c}^0 \in \Sigma_f$, we omit the parentheses and write simply c instead of $\mathsf{c}()$, whereas for interpreted constants $c^0 \in \Sigma_i$, we do not omit the square brackets for clarity and write $c[]$.

Let $\mathcal{V}$ be a countable set of *variable symbols*, disjoint from Σ. We denote with $\mathcal{T}_\Sigma(\mathcal{V})$ the set of all *terms* that can be built from the function symbols in Σ and the variables in $\mathcal{V}$. We simply write $\mathcal{T}_\Sigma$ when $\mathcal{V} = \emptyset$, and call its elements *ground terms* (over signature Σ). We define *substitutions* θ as is standard.

We define the set $\mathcal{L}_\Sigma(\mathcal{V})$ of *formulae* over the alphabet Σ and the variables $\mathcal{V}$ as usual: relations and equality of terms are *atomic formulae*, and *composed formulae* are built using conjunction $\wedge$, negation $\neg$, and existential quantification $\exists$.

The function *fv* returns the set of *free variables* of a formula as expected. □

We employ the standard syntactic sugar and write, for example, $\forall x.\phi$ for $\neg\exists x.\neg\phi$. We also write $x \in \{t_1, \ldots, t_n\}$ to abbreviate $x = t_1 \vee \cdots \vee x = t_n$.

Slightly abusing notation, we will also consider a substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ as a formula $x_1 = t_1 \wedge \cdots \wedge x_n = t_n$.

Definition 2.2 (Herbrand Universe and Algebra [16]) Formulae in Herbrand logic are always interpreted with respect to a given fixed set Σ_f of free symbols (since this set may contain symbols that do not occur in the formulae) and a congruence relation $\approx$ on $\mathcal{T}_{\Sigma_f}$. We may annotate all notions of the semantics with Σ_f and $\approx$ when it is not clear from the context.

We write $[\![t]\!]_\approx = \{t' \in \mathcal{T}_{\Sigma_f} \mid t \approx t'\}$ to denote the *equivalence class* of a term $t \in \mathcal{T}_{\Sigma_f}$ with respect to $\approx$. Further, let $U = \{[\![t]\!]_\approx \mid t \in \mathcal{T}_{\Sigma_f}\}$ be the set of all equivalence classes. We call U the *Herbrand universe* (since it is freely generated by the function symbols of Σ_f modulo $\approx$). Based on U, we define a Σ_f-algebra $\mathcal{A}$ that interprets every n-ary function symbol $\mathsf{f} \in \Sigma_f$ as a function $\mathsf{f}^{\mathcal{A}} : U^n \to U$ in the following standard way. $\mathsf{f}^{\mathcal{A}}([\![t_1]\!]_\approx, \ldots, [\![t_n]\!]_\approx) = [\![f(t_1, \ldots, t_n)]\!]_\approx$, where the choice of the representatives $t_1, \ldots, t_n$ of the equivalence classes is irrelevant because $\approx$ is congruent. $\mathcal{A}$ is sometimes also called the *quotient algebra* (in the literature sometimes denoted with $\mathcal{T}_{\Sigma_f}/\approx$). □

Definition 2.3 (Semantics of Herbrand Logic [16]) An *interpretation* $\mathcal{I}$ maps every interpreted function symbol $f \in \Sigma_i$ of arity n to a function $\mathcal{I}(f) : U^n \to U$ on the Herbrand universe, every relation symbol $r \in \Sigma_r$ of arity n to a relation $\mathcal{I}(r) \subseteq U^n$ on the Herbrand universe, and every variable $x \in \mathcal{V}$ to an element of U.

We extend $\mathcal{I}$ to a function on $\mathcal{T}_\Sigma(\mathcal{V})$ as expected:
$\mathcal{I}(\mathsf{f}(t_1, \ldots, t_n)) = \mathsf{f}^{\mathcal{A}}(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$ for $\mathsf{f} \in \Sigma_f$ and
$\mathcal{I}(f[t_1, \ldots, t_n]) = \mathcal{I}(f)(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$ for $f \in \Sigma_i$.

We define that $\mathcal{I}$ is a *model* of formula ϕ, in symbols $\mathcal{I} \models \phi$, as follows:

$\mathcal{I} \models s = t$	iff	$\mathcal{I}(s) = \mathcal{I}(t)$
$\mathcal{I} \models r(t_1, \ldots, t_n)$	iff	$(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) \in \mathcal{I}(r)$
$\mathcal{I} \models \phi \wedge \psi$	iff	$\mathcal{I} \models \phi$ and $\mathcal{I} \models \psi$
$\mathcal{I} \models \neg\phi$	iff	not $\mathcal{I} \models \phi$
$\mathcal{I} \models \exists x.\phi$	iff	there is a $c \in U$ such that $\mathcal{I}\{x \mapsto c\} \models \phi$

where $\mathcal{I}\{x \mapsto c\}$ denotes the interpretation that is identical to $\mathcal{I}$ except that x is mapped to c. *Entailment* $\phi \models \psi$ is defined as $\mathcal{I} \models \phi$ implies $\mathcal{I} \models \psi$ for all interpretations $\mathcal{I}$. We write $\phi \equiv \psi$ when both $\phi \models \psi$ and $\psi \models \phi$. We also use $\equiv$ in the definitions of formulae. Finally, we write $Sat(\phi)$ if ϕ has a model. □

2.2 Frames

Frames are a classical way of representing terms known by the intruder. The definition of frames taken here is slightly different than the standard definition and allows a convenient encoding in Herbrand logic through a couple of axioms. Frames and their static equivalence are used in this context to express (α, β)-privacy goals.

Definition 2.4 (Frame [16]) A frame is written as

$$F = \{\!| \mathsf{l}_1 \mapsto t_1, \ldots, \mathsf{l}_k \mapsto t_k |\!\},$$

where the l_i are distinguished constants and the t_i are ground terms *without any destructor or verifier* that do not contain any l_i. We call $\mathsf{l}_1, \ldots, \mathsf{l}_k$ and $t_1, \ldots, t_k$ the domain and the image of the frame, respectively. □

We may refer to the labels l_i of a frame as *memory locations*.

Definition 2.5 (Recipes and intruder-generable term [16]) The set of recipes is the least set that contains $\mathsf{l}_1, \ldots, \mathsf{l}_k$ and that is closed under all the cryptographic operators Σ_{op}. A frame F can be regarded as a substitution that replaces every l_i of its domain with the corresponding t_i. For a recipe r, we thus write $F\{\!| r |\!\}$ for the term obtained by applying this substitution to r. An intruder generable term (or just *generable term for short*) is any term t for which there is a recipe r with $t = F\{\!| r |\!\}$. □

We use frames to represent the intruder's knowledge of terms under the form of labelled messages. Thus, we can describe the actions performed by the intruder with recipes over the labels in the intruder knowledge. Note that by default, the intruder does not know the constants. They can however be explicitly included in the frame if needed.

As mentioned before, the frames and their static equivalence can be encoded in Herbrand logic. We define below two axioms allowing to do that [16]:

For every considered frame $F = \{\!| \mathsf{l}_1 \mapsto t_1, \ldots, \mathsf{l}_k \mapsto t_k |\!\}$, let kn_F be an interpreted unary function symbol and gen_F be a unary predicate. The idea is that kn_F encodes the knowledge of terms contained in the frame, and gen_F encodes the fact that the intruder is able to generate a term.

For a frame $\mathcal{F}$:

$$\begin{aligned}
\phi_{frame}(\mathcal{F}) \quad \equiv \quad & (\forall x.gen_{\mathcal{F}}(x) \iff \\
& \quad (x \in \{\mathsf{l}_1, \ldots, \mathsf{l}_k\} \vee \\
& \qquad \bigvee_{f^n \in \Sigma_{pub}} \exists x_1 \ldots x_n. \\
& \qquad\quad x = f(x_1, \ldots, x_n) \wedge gen_{\mathcal{F}}(x_1) \wedge \cdots \wedge gen(x_n))) \\
& \wedge \\
& (kn_{\mathcal{F}}[\mathsf{l}_1] = t_1 \wedge \cdots \wedge kn_{\mathcal{F}}[\mathsf{l}_k] = t_k) \\
& \wedge \\
& \quad \bigwedge_{f^n \in \Sigma_{pub}} (\forall x_1 \ldots x_n. \\
& \qquad gen_{\mathcal{F}}(x_1) \wedge \cdots \wedge gen(x_n) \implies \\
& \qquad\quad kn_{\mathcal{F}}[f(x_1, \ldots, x_n)] = f(kn_{\mathcal{F}}[x_1], \ldots, kn_{\mathcal{F}}[x_n]))
\end{aligned}$$

This axiom expresses that the intruder is able to generate every term in the frame, as well as all terms that correspond to applying a public function to generable terms. It also expresses that the intruder knows the terms stored in the memory locations, as well as the terms resulting from the application of a recipe.

For two frames $\mathcal{F}_1$ and $\mathcal{F}_2$:

$$\begin{aligned}
\phi_{\mathcal{F}_1 \sim \mathcal{F}_2} \quad \equiv \quad & (\forall x.gen_{\mathcal{F}_1}(x) \iff gen_{\mathcal{F}_2}(x)) \\
& \wedge \\
& (\forall x, y.gen_{\mathcal{F}_1}(x) \wedge gen_{\mathcal{F}_1}(y) \implies \\
& \quad (kn_{\mathcal{F}_1}[x] = kn_{\mathcal{F}_1}[y] \iff kn_{\mathcal{F}_2}[x] = kn_{\mathcal{F}_2}[y]))
\end{aligned}$$

This axiom expresses that the generable terms are the same in both frames, and also that given two generable terms, the intruder knows that they are equal (or not equal) in both frames at the same time.

The idea of the notion of static equivalence is whether the intruder is able to distinguish two frames. If they are able to generate a term with two recipes in one frame, they can check if the same holds in the second frame. Checking pairs of recipes can tell if the intruder is able to distinguish the frames or not. We would like to translate the intuitive property

$$\forall (r_1, r_2), \mathcal{F}_1\{\!|\, r_1 \,|\!\} \approx \mathcal{F}_1\{\!|\, r_2 \,|\!\} \iff \mathcal{F}_2\{\!|\, r_1 \,|\!\} \approx \mathcal{F}_2\{\!|\, r_2 \,|\!\}$$

into our logical framework.

This is formalised in Herbrand logic using the two axioms defined above:

Definition 2.6 (Static Equivalence of Frames [16]) Two frames $\mathcal{F}_1$ and $\mathcal{F}_2$ with the same set $\{\mathsf{l}_1, \ldots, \mathsf{l}_k\}$ of memory locations are statically equivalent (in symbols, $\mathcal{F}_1 \sim \mathcal{F}_2$) iff $Sat(\phi_{frame}(\mathcal{F}_1) \wedge \phi_{frame}(\mathcal{F}_2) \wedge \phi_{\mathcal{F}_1 \sim \mathcal{F}_2})$. □

Example 2.1 *Let $k, t_1, t_2 \in \mathcal{T}_\Sigma$ where $t_1 \neq t_2$. Consider the frames*

$$\mathcal{F}_1 = \{\!| \, \mathsf{l}_1 \mapsto \mathsf{scrypt}(k, t_1), \mathsf{l}_2 \mapsto k, \mathsf{l}_3 \mapsto t_1 \, |\!\}$$
$$\mathcal{F}_2 = \{\!| \, \mathsf{l}_1 \mapsto \mathsf{scrypt}(k, t_2), \mathsf{l}_2 \mapsto k, \mathsf{l}_3 \mapsto t_2 \, |\!\}$$

Then $\mathcal{F}_1 \sim \mathcal{F}_2$ because even though the frames are not equal ($t_1 \neq t_2$), there is no way to distinguish them.

Example 2.2 *Let $v_1, v_2 \in \mathcal{T}_\Sigma$. Consider the frames*

$$\mathcal{F}_1 = \{\!| \, \mathsf{l}_1 \mapsto \mathsf{h}(v_1), \mathsf{l}_2 \mapsto \mathsf{h}(v_2), \mathsf{l}_3 \mapsto \mathsf{h}(v_1) \, |\!\}$$
$$\mathcal{F}_2 = \{\!| \, \mathsf{l}_1 \mapsto \mathsf{h}(v_1), \mathsf{l}_2 \mapsto \mathsf{h}(v_2), \mathsf{l}_3 \mapsto \mathsf{h}(v_2) \, |\!\}$$

Then $\mathcal{F}_1 \not\sim \mathcal{F}_2$ because the pair of recipes $(\mathsf{l}_1, \mathsf{l}_3)$ distinguishes the frames.

2.3 (α, β)-privacy

The standard approach to verify privacy goals in protocols is based on observational equivalence, i.e. it consists in reasoning about the indistinguishability between two possible worlds. The problem can be encoded in frames and there exists decision procedures to verify static equivalence of frames. However, these methods are actually difficult to model and reason about. Instead of stating different possible worlds that should encode a privacy goal, we would like a more intuitive approach.

This is the problem that (α, β)-privacy addresses. It formalises privacy goals as logical formulae in a declarative manner. That makes it much easier for a modeller to specify the privacy properties that they desire. Moreover, it is not necessary to define a set of possible worlds to compare. We can reason about the protocol specification at the abstract level on one hand, and one real execution at the concrete level on the other hand. The core idea is to declare a *payload formula* α at the abstract level, defining intentionally released information, and a *technical information formula* β, including cryptographic messages and other knowledge at the technical level. That way, the modeller can use this framework to express privacy goals intuitively and provide a formal specification of a protocol.

This thesis is based on (α, β)-privacy as introduced in [16]. In the general case, (α, β)-privacy is not decidable because of the expressivity of Herbrand logic. However, there are identified fragments for which it is possible to design a decision procedure. While this can also be related to static equivalence of frames, (α, β)-privacy is more expressive. The decision procedure is based on deciding satisfaction of logical formulae, which corresponds to what a manual proof or automatic theorem prover would yield.

We have previously defined the syntax and semantics of Herbrand logic, as well as the encoding of intruder knowledge into frames. It remains to define how privacy goals are defined. In this chapter, we give a number of definitions related to privacy goals and theorems that are useful to design our procedure. We start by defining formally how to declare privacy goals and what an interesting consequence is. Then, we focus on the class of problem called *message-analysis problems*, which relates directly (α, β)-privacy and static equivalence of frames.

For the problem under consideration, we reason about interpretations and the different models of formulae. We describe the notion of *model-theoretical* (α, β)-privacy and some results. This is also put in relation to static equivalence of frames. Finally, we recall the concept of (α, β)-privacy in an entire transition system and explain why we only look at a single state.

2.3.1 Payload and technical information

As the authors explain in [16], the formalisation of (α, β)-privacy is inspired by zero-knowledge proofs for privacy. For these types of proofs, there is usually a prover that discloses some information intentionally; this would correspond to the payload information α. The participants of the protocol communicate by exchanging cryptographic messages (including hashing, encryption, signing etc.); this would correspond to the technical information β. Privacy is achieved if β does not leak any more useful information than α. To translate this notion in terms of logical formulae, we define an *interesting consequence* by such information following from β but not α. The idea relies on defining α over an alphabet $\Sigma_0 \subsetneq \Sigma$ (the abstract level only), while β is expressed over Σ.

Definition 2.7 (Interesting consequences [16]) Let $\Sigma_0 \subsetneq \Sigma$. Given a payload formula $\alpha \in \mathcal{L}_{\Sigma_0}(\mathcal{V})$ and a technical formula $\beta \in \mathcal{L}_{\Sigma}(\mathcal{V})$, where $\beta \models \alpha$ and $fv(\alpha) = fv(\beta)$ and both α and β are consistent, we say that a statement $\alpha' \in \mathcal{L}_{\Sigma_0}(fv(\alpha))$ is an *interesting consequence* of β (with respect to α) if $\beta \models \alpha'$ but $\alpha \not\models \alpha'$.

We say that β respects the privacy of α if it has no interesting consequences,

and that β violates the privacy of α otherwise. □

2.3.2 Privacy on messages

Many protocols can be specified by a number of cryptographic messages exchanged. In this situation, we assume that the intruder knows the protocol specification as well as one concrete execution. This can be encoded into a frame *struct* corresponding to the structural information coming from the specification (i.e. what the messages look like), and a frame *concr* corresponding to the actual execution. We consider the problem where *concr* is simply one instance of *struct*, i.e. the variables from *struct* have been instantiated. We can use the axioms defining static equivalence of frames for *struct* and *concr*. This gives the intruder access to the fact that *struct* and *concr* are statically equivalent, and they can then try to derive information by comparing pairs of recipes in the two frames. For simplicity, we will abuse notation and write also $struct[\cdot]$ and $concr[\cdot]$ in Herbrand logic instead of $kn_{struct}[\cdot]$ and $kn_{concr}[\cdot]$. Since *struct* and *concr* will then also have the same domain, it follows that gen_{struct} and gen_{concr} are equivalent and thus we will simply write *gen*. For such a problem, we specify a requirement on α so that it becomes possible to decide this fragment.

Definition 2.8 (Combinatoric α [16]) We call $\alpha \in \mathcal{L}_{\Sigma_0}(\mathcal{V})$ combinatoric if Σ_0 is finite and contains only uninterpreted constants. □

Thus, every model $\mathcal{I}$ of α maps the free variables of α to elements of the Herbrand universe induced by Σ_0. For each free variable x of α, we have $\mathcal{I}(x) = [\![c]\!]_{\approx}$ for some unique $c \in \Sigma_0$. For every $\mathcal{I}$ such that $\mathcal{I} \models \alpha$, we define the substitution $\theta_{\mathcal{I}}$ that has as domain the set of free variables of α, and such that $\theta_{\mathcal{I}}(x) = c$ iff $\mathcal{I}(x) = [\![c]\!]_{\approx}$ (note that $\theta_{\mathcal{I}}$ is unique modulo $\approx$). Recall that, by slight abuse of notation, we may treat a substitution $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ as the Herbrand formula $x_1 = t_1 \wedge \cdots \wedge x_n = t_n$. Thus α is equivalent to the disjunction of all such substitutions:

Lemma 2.9 *[16] For every combinatoric α, there is a finite set of substitutions Θ such that $\alpha \equiv \bigvee_{\theta \in \Theta} \theta$. We thus call Θ also the* models *of α.*

In this thesis, we will call the frame for the structural information *struct* and the frame for the concrete knowledge *concr*.

Definition 2.10 (Message-analysis problem [16]) Let α be combinatoric, θ a model of α, $struct = \{\!| \mathsf{l}_1 \mapsto t_1, \ldots, \mathsf{l}_k \mapsto t_k |\!\}$ for some $t_1, \ldots, t_k \in \mathcal{T}_\Sigma(fv(\alpha))$, and $concr = \theta(struct)$. Define

$$MsgAna(\alpha, struct, \theta) \equiv \alpha \wedge \phi_{frame}(concr) \wedge \phi_{frame}(struct) \wedge \phi_{concr \sim struct}$$

If $\beta \equiv MsgAna(\alpha, struct, \theta)$, then we say that β is a *message-analysis problem* (with respect to α, *struct*, and θ). □

In Chapter 4, we will study a message-analysis problem. Our procedure will make use of the fact that $concr = \theta(struct)$ for some substitution θ that is a model of α. It must be noted that, while our algorithms can use θ directly, the intruder does *not* have access to θ. Rather, for a term t_i at the structural level in *struct*, they also know the corresponding term $\theta(t_i)$ at the concrete level in *concr*. The intruder also knows that $struct \sim concr$, which will be used to find relations between variables that are consistent with θ.

2.3.3 Model-theoretical (α, β)-privacy

The concept of (α, β)-privacy has been defined with the notion of interesting consequences. We know consider a more semantic definition, using the models of the logical formulae. This section defines *model-theoretical* (α, β)-privacy and relates this approach to classical (α, β)-privacy. In the case of message-analysis problems, we present some results about the model-theoretical view. They will be used in order to prove correctness of our decision procedure in the next chapters.

Definition 2.11 (Model-theoretical (α, β)-privacy [16]) Consider Σ_0 and Σ as before, a formula α over Σ_0 and a formula β over Σ such that $fv(\alpha) = fv(\beta)$, both α and β are consistent and $\beta \models \alpha$. We say that (α, β)-privacy holds model-theoretically iff every Σ_0-model of α can be extended to a Σ-model of β, where a Σ-interpretation $\mathcal{I}'$ is an extension of a Σ_0-interpretation $\mathcal{I}$ if they agree on all variables and all the interpreted function and relation symbols of Σ_0. □

Theorem 2.12 *[16] If (α, β)-privacy holds model-theoretically, then it also holds in the classical sense. Conversely, if for every model $\mathcal{I}$ of α, there is a Σ_0-formula $\phi_\mathcal{I}$ that has only $\mathcal{I}$ as a model (with respect to Σ_0), then classical (α, β)-privacy implies model-theoretical (α, β)-privacy.*

Corollary 2.13 *[16] Consider an (α, β) pair, where α is combinatoric and Θ is the set of models of α. Then, β violates the privacy of α iff $\beta \models \neg\theta$ for any $\theta \in \Theta$.*

While classical (α, β)-privacy and model-theoretical (α, β)-privacy are not equivalent in the general case, when considering a combinatoric α (e.g. in message-analysis problems) the two notions coincide.

The theorem and its corollary allow us to design a decision procedure. Indeed, instead of having to reason about *any* possible interesting consequence, we can now focus more on the countably finite number of models of α. This is the foundation of the approach developed in this thesis: given a message-analysis problem, we will generate a logical formula ϕ (expressing the relations between the variables that the intruder is able to observe) that can be used to characterise the models of α.

2.3.4 Automation and the relation to static equivalence

We have shown in Section 2.2 how the problem of static equivalence of frames can be encoded in Herbrand logic. Moreover, we have defined a class of problems, namely message-analysis problems, using this notion of static equivalence between a structural frame *struct* and a concrete frame *concr*. We now relate this with the model-theoretical view of (α, β)-privacy.

Theorem 2.14 *[16] Let α be combinatoric, $\Theta = \{\theta_1, \ldots, \theta_n\}$ be the models of α, and $\beta \equiv MsgAna(\alpha, \Phi, \theta_1)$ for some $\theta_1 \in \Theta$. Then, we have that (α, β)-privacy holds iff $\theta_1(\Phi) \sim \ldots \sim \theta_n(\Phi)$.*

This theorem shows how privacy can be decided as a set of static equivalence problems. Our procedure is inspired by this view: our formula ϕ, generated using static equivalence properties, will be a way to characterise models. If ϕ really excludes some models of α, then we can illustrate how (α, β)-privacy is violated.

The interesting aspect is that we do not have to reason about two possible worlds to compare, as in standard observational equivalence for privacy. The formula α will encode a number of possible worlds (the different models), and then we can decide privacy by studying satisfiability of formulae.

2.3.5 (α, β)-privacy in Transition Systems

We have stated in Section 1.2 that we do not study in details (α, β)-privacy in an entire transition system. Nonetheless, we give below a definition of (α, β)-privacy for transition systems and then justify the choice of limiting to scope to a single state.

The general idea is to define a state with three formulae: α and β as before, alongside a *truth formula* γ which encode the instantiation of variables.

Definition 2.15 (Transition systems [16]) A state is a triple (α, β, γ), where α and β are as before and $\gamma \in \mathcal{L}_{\Sigma_0}(\mathcal{V})$ is such that $\gamma \models \alpha$ and γ is true in exactly one model of γ (with respect to Σ_0 and the free variables of α). We also call γ the *truth* and may also apply it to Σ_0-terms like a substitution.

Let $\mathcal{S}$ denote the set of all states. A transition system is a pair (I, R) where $I \in \mathcal{S}$ and $R \subseteq \mathcal{S} \times \mathcal{S}$. As is standard, the set of reachable states is the smallest set that contains I and that is closed under R, i.e.: if S is reachable and $(S, S') \in R$, then also S' is reachable.

We say that a transition system *satisfies privacy* iff (α, β)-privacy holds in every reachable state (α, β, γ). □

In this thesis, we will not study (α, β)-privacy at the level of a transition system. The reason is that the core problem is, given some α and β, whether β violates the privacy of α or not. The extension to transition system corresponds to verifying privacy in every reachable state. Therefore, we focus only on the problem of deciding (α, β)-privacy in a single state, and we do not to refer to an entire transition system. What is left out by our work in this thesis is only a specification of the method to explore every reachable state and apply our procedure. We do not see this as a real restriction, because there is no theoretical difficulty involved in this exploration of states.

Note also that we will study (α, β)-privacy with regards to a model θ encoding a concrete execution of the protocol. θ would then correspond to the truth γ when referring to a reachable state.

Chapter 3

Modelling the intruder

In Chapter 2, we have presented the definitions from previous works that constitute the basis of our contributions. We want to automatically reason about whether privacy is violated or not, given a certain knowledge. In order to do that, we specify in Section 3.1 the class of *intruder theories* that we support. The approach that we take to verify privacy is to consider an intruder trying to break privacy. That is, the intruder has some knowledge and can follow a number of rules (according to the intruder theory) in order to derive sensitive information. We complement this by the notion of *frames with shorthands* presented in Section 3.2. This will become useful when we describe the analysis of frames performed by the intruder in Chapter 4.

3.1 Intruder theory

Now that the syntax and semantics of the logic have been defined, we present a specification of what we call an intruder theory. A common approach is to specify a list of cryptographic properties with their properties. Therefore, our intruder theory includes this to determine a set of rules, that represent the behaviour of the intruder.

Definition 3.1 (Intruder theory) An *intruder theory* consists of

- a set of public functions $\Sigma_{pub} \subseteq \Sigma_f$,
- a set of variables with public range $\mathcal{V}_{pub} \subseteq \mathcal{V}$,
- a set of cryptographic operators $\Sigma_{op} \subseteq \Sigma_f$ and
- a set of algebraic equations.

The set Σ_{pub} identifies all functions that the intruder is allowed to use. The set $\mathcal{V}_{pub}$ identifies all variables for which the intruder knows the different possible values that they can take in a protocol execution. The cryptographic operators and their algebraic properties form a black-box model of the cryptography. We make the distinction between *constructors*, which are operators that can be used to compose terms, *destructors*, which are operators associated with a constructor to decompose terms, and *verifiers*, which are operators associated with a constructor to check the composition of terms. □

In general, we can allow an arbitrary set of cryptographic operators and algebraic equations. However, in this thesis we are concerned about verifying (α, β)-privacy for decidable fragments. Therefore, we give a definition of a subset of all intruder theories for which we will be able to design a decision procedure. Note that our procedure is parameterised over an intruder theory (respecting the following requirements).

Definition 3.2 (Convergent intruder theory) A *convergent intruder theory* is an intruder theory where:

The algebraic equations must be of one of the following forms:

- $\mathsf{destr}(k_1, \ldots, k_m, \mathsf{constr}(t_1, \ldots, t_n)) \approx t_i$, for a constructor of arity n and a corresponding destructor of arity $m + 1$.
- $\mathsf{verif}(k_1, \ldots, k_m, \mathsf{constr}(t_1, \ldots, t_n)) \approx \mathsf{yes}$, for a constructor of arity n and a corresponding verifier of arity $m + 1$.

The first m arguments of a destructor or verifier are called keys, and it is allowed to have 0 keys (i.e. $m = 0$). In case of a destructor, the term on the right-hand side must be one of the arguments of the constructor. For any destructor associated to a constructor, there must be one associated verifier. The same verifier can be associated to several destructors. Moreover, for any two of these algebraic equations where there is an occurrence of some $\mathsf{constr}(t_1, \ldots, t_m)$ and $\mathsf{constr}'(t'_1, \ldots, t'_n)$, it must be the case that either

- $\mathsf{constr} \neq \mathsf{constr}'$ or
- $m = n$ and $t_1 = t'_1, \ldots, t_m = t'_m$.

These constraints express that the algebraic equations must form a convergent rewriting system, i.e. the rewriting rules generated by reducing the application of a destructor on the left-hand side to the term on the right-hand side induce a confluent (diverging paths are always joining a common term at some point) and terminating reduction [2]. The congruence relation $\approx$ that the Herbrand universe uses is the least relation so that the algebraic equations hold.

These equations lead to a function *ana* defined by

$$\begin{aligned} ana(\mathsf{constr}(t_1, \ldots, t_n)) = (\{k_1, \ldots, k_m\}, \\ \{(\mathsf{destr}, t_i) \mid \\ \mathsf{destr}(k_1, \ldots, k_m, \mathsf{constr}(t_1, \ldots, t_n)) \approx t_i\}) \end{aligned}$$

Intuitively, given a term that can be decomposed, *ana* returns the set of keys required for decomposition and all derivable terms according to the algebraic equations. With this definition, we require that different destructors associated with the same constructor use the same keys. □

Example 3.1 *Even though this thesis considers an arbitrary convergent intruder theory, we focus on a practical example that will be used in other illustrative examples. Let* $\Sigma = \Sigma_f \uplus \Sigma_i \uplus \Sigma_r$ *be an alphabet,* $\mathcal{V}$ *a set of variables, and* $\mathcal{V}_{pub} \subseteq \mathcal{V}$ *a set of variables with public range. We consider the cryptographic operators defined in Table 3.1 and* $\Sigma_{pub} = \Sigma_{op} \setminus \{\mathsf{priv}\}$*, i.e. all operators are public except for* priv*. As in [16],*

- $\mathsf{pub}(s)$ *and* $\mathsf{priv}(s)$ *represent an asymmetric key pair from a secret seed*[1] *s, where* pub *is a public function in* Σ_{op} *and* priv *is a private function in* $\Sigma \setminus \Sigma_{pub}$*;*
- $\mathsf{crypt}(p, r, t)$ *represents the asymmetric encryption of a message* t *with a public key* p *and randomness* r*;*
- $\mathsf{dcrypt}(p', t)$ *represents the decryption with private key* p' *of a message* t*, and the first property formalises that decryption with the correct private key yields the original message;*

[1] We make a slight change compared to [16], which associates one key pair to an agent, so that we can model arbitrary key infrastructures

- $\mathsf{vcrypt}(p', t)$ *and the second property formalise that we can check whether the message* t *can be correctly decrypted with private key* p'*;*
- $\mathsf{sign}(p', t), \mathsf{retrieve}(t)$ *and* $\mathsf{vsign}(p', t)$*, together with their properties, similarly formalise digital signatures (where we here model signatures that contain the plaintext so that it can be retrieved);*
- $\mathsf{scrypt}(k, t), \mathsf{dscrypt}(k, t)$ *and* $\mathsf{vscrypt}(k, t)$*, together with their properties, similarly formalise symmetric cryptography;*
- $\mathsf{pair}, \mathsf{proj}_i$ *and* vpair*, together with their properties, formalise that we assume to have a mechanism to concatenate plaintext so that it can later be decomposed in a unique way (sometimes called "serialisation");*
- h *is a cryptographic hash function (where the lack of destructors reflects that it is hard to find a pre-image).*

Constructors	**Destructors**	**Verifiers**	**Properties**
$\mathsf{pub}, \mathsf{priv}$			
crypt	dcrypt	vcrypt	$\mathsf{dcrypt}(\mathsf{priv}(s), \mathsf{crypt}(\mathsf{pub}(s), r, t)) \approx t$ $\mathsf{vcrypt}(\mathsf{priv}(s), \mathsf{crypt}(\mathsf{pub}(s), r, t)) \approx \mathsf{yes}$
sign	$\mathsf{retrieve}$	vsign	$\mathsf{retrieve}(\mathsf{sign}(\mathsf{priv}(s), t)) \approx t$ $\mathsf{vsign}(\mathsf{pub}(s), \mathsf{sign}(\mathsf{priv}(s), t)) \approx \mathsf{yes}$
scrypt	$\mathsf{dscrypt}$	$\mathsf{vscrypt}$	$\mathsf{dscrypt}(k, \mathsf{scrypt}(k, t)) \approx t$ $\mathsf{vscrypt}(k, \mathsf{scrypt}(k, t)) \approx \mathsf{yes}$
pair	proj_i	vpair	$\mathsf{proj}_i(\mathsf{pair}(t_1, t_2)) \approx t_i$ $\mathsf{vpair}(\mathsf{pair}(t_1, t_2)) \approx \mathsf{yes}$
h			

Table 3.1: Example set Σ_{op}

Additionally, we consider that natural integers are publicly known. This can be encoded in the logic with an interpreted constant $\mathsf{0}$ and an interpreted unary function s representing the increment by 1. We may use the notation $\mathsf{1}$ for $\mathsf{s}(\mathsf{0})$, $\mathsf{2}$ for $\mathsf{s}(\mathsf{s}(\mathsf{0}))$, and so on.

3.2 Frames with shorthands

When defining our analysis algorithm, we will make use of the algebraic equations of the intruder theory to derive new terms that can be added to the knowledge. These new terms are present in the analysed frame through *shorthands*, where the new labels are actually recipes for the original frame. A frame with

shorthands consists thus in a frame with additional labels. It can be seen as an extension of the original frame, where the new labels are not constants but behave in the same way when applying a recipe to the frame.

Definition 3.3 (Frame with shorthands) A frame with shorthands is written as

$$F' = \{\!| \mathsf{l}_1 \mapsto t_1, \ldots, \mathsf{l}_k \mapsto t_k, \mathsf{m}_1 \mapsto s_1, \ldots, \mathsf{m}_n \mapsto s_n |\!\},$$

where $F = \{\!| \mathsf{l}_1 \mapsto t_1, \ldots, \mathsf{l}_k \mapsto t_k |\!\}$ is a frame, the m_j are recipes over the l_i and the s_j are ground terms that do not contain any l_i. We call the mappings $\mathsf{m}_1 \mapsto s_1, \ldots, \mathsf{m}_n \mapsto s_n$ shorthands, and they must verify that $F\{\!| \mathsf{m}_j |\!\} \approx s_j$. □

The domain of a frame and a corresponding frame with shorthands can be considered the same, because a recipe containing a label m_j is equivalent to the same recipe where m_j is replaced by the recipe over $\mathsf{l}_1, \ldots, \mathsf{l}_k$ that defines it.

Example 3.2 *Let $k, t \in \mathcal{T}_\Sigma(\mathcal{V})$. Consider the frames*

$$F = \{\!| \mathsf{l}_1 \mapsto \mathsf{scrypt}(k,t), \mathsf{l}_2 \mapsto k |\!\}$$
$$F' = \{\!| \mathsf{l}_1 \mapsto \mathsf{scrypt}(k,t), \mathsf{l}_2 \mapsto k, \mathsf{dscrypt}(\mathsf{l}_2, \mathsf{l}_1) \mapsto t |\!\}$$

Then F' is the frame F with the shorthand $\mathsf{dscrypt}(\mathsf{l}_2, \mathsf{l}_1) \mapsto t$. Indeed, we have that $F\{\!| \mathsf{dscrypt}(\mathsf{l}_2, \mathsf{l}_1) |\!\} = \mathsf{dscrypt}(k, \mathsf{scrypt}(k,t)) \approx t = F'\{\!| \mathsf{dscrypt}(\mathsf{l}_2, \mathsf{l}_1) |\!\}$.

Chapter 4

Decision procedure

In order to verify privacy, we rely on an intruder theory. The situation that we study is that an intruder considers a protocol specification and a concrete execution of the protocol. Their objective is to break privacy if possible. For now, we restrict the problem to protocols without branching. In the formal specification of a protocol, we can distinguish a number of states, corresponding to the point reached during the protocol execution. The procedure that we design in this thesis concerns the verification of privacy in a single state of the protocol.

For this state, the privacy goals are expressed with the *payload formula* α, and the *technical information formula* β represents the cryptographic messages exchanged according to the protocol specification. The knowledge of the intruder is represented with frames. There is one frame encoding the protocol specification, i.e. how the messages look like. We call this the *structural frame* and designate it with the notation *struct*. There is also a second frame encoding one concrete execution of the protocol. We call this the *concrete frame* and designate it with the notation *concr*. The reasoning is that protocols are defined in public standards, so the intruder has access to this knowledge. Moreover, we assume that they were able to record one concrete execution[1].

[1]The knowledge of an intruder that would have recorded a sequence of protocol executions could be encoded in a single frame aggregating the labelled messages from several frames.

Since the execution follows the specification, the intruder knows that there is a correspondence between the structural frame *struct* (containing terms with variables) and the concrete frame *concr* (containing ground terms of one execution). This is expressed as the *static equivalence* between these two frames. Our idea is to define algorithms to decide static equivalence in this context. Besides simply determining whether privacy holds or not, we want to generate a *witness*, i.e. derive an *interesting consequence* showing what information the intruder learns that they were not supposed to.

More formally, we assume that the problem is a *message-analysis problem*: the privacy goals are expressed in the formula α, the concrete execution is encoded in the substitution θ which is a model of α, and $\beta \equiv MsgAna(\alpha, struct, \theta)$. In the rest of this chapter, we may write *concr* for $\theta(struct)$. Based on their knowledge, the intruder tries to find an interesting consequence of β with regards to α. The procedure designed in this chapter generates a formula ϕ characterising the relations between variables occurring in *struct*. We show that in this context ϕ can be used to decide privacy, i.e. it remains to decide if it follows from α alone or not. If ϕ really is an *interesting consequence*, then (α, β)-privacy does not hold in the state. When all variables have a finite range, the verification of whether ϕ constitutes a breach of privacy or not is a decidable problem. This last step of verification is not done directly by the procedure defined in the following, but it can already be done by existing checkers [1].

Example 4.1 *Consider a structural frame and corresponding concrete frame*

$$struct = \{\!|\, \mathsf{l}_1 \mapsto \mathsf{scrypt}(k, x), \mathsf{l}_2 \mapsto \mathsf{scrypt}(k, y), \mathsf{l}_3 \mapsto \mathsf{scrypt}(k, z) \,|\!\}$$
$$concr = \{\!|\, \mathsf{l}_1 \mapsto \mathsf{scrypt}(k, 0), \mathsf{l}_2 \mapsto \mathsf{scrypt}(k, 1), \mathsf{l}_3 \mapsto \mathsf{scrypt}(k, 0) \,|\!\}$$

where the variables x, y, z in struct represent some votes that have been encrypted by a trusted authority with a key k. Let the payload formula be $\alpha \equiv x, y, z \in \{0, 1\}$. In our example intruder theory, the intruder is not able to learn the values of the votes without the key. However, they can observe that $concr\{\!|\, \mathsf{l}_1 \,|\!\} = concr\{\!|\, \mathsf{l}_3 \,|\!\}$. Using static equivalence between struct and concr, they deduce that $struct\{\!|\, \mathsf{l}_1 \,|\!\} = struct\{\!|\, \mathsf{l}_3 \,|\!\}$. Therefore, they can learn that $x = z$. This constitutes a breach of privacy, as it does not follow from α.

Our procedure relies on the fact that the intruder knows $concr \sim struct$. The intruder knowledge is represented as a set of labelled terms. Analysis is performed to derive subterms, e.g. an encrypted term can be decomposed if the key can be composed. As mentioned previously, for a *message-analysis problem* the intruder knows that the structural and concrete frames are statically equivalent. The idea to find a candidate interesting consequence is to determine the relations between variables. During the analysis, if the decomposition of a term

is possible only at the structural level and not at the concrete level, then we can derive some information about terms being not equal and find inequalities between variables. After the analysis, it is possible to find equalities between variables using composition. Terms equal at the concrete level are also equal at the structural level. Therefore, if in *concr* there are several recipes to compose a term, then the same recipes must generate a unique corresponding term in *struct*. Moreover, it is also possible to derive more inequalities when terms are equal only at the structural level but not at the concrete level, as this goes against static equivalence. Note that the inequalities found during the analysis cannot be found later, because they correspond to decomposition failures (by opposition to composition checks that are performed later).

4.1 Method

We begin in Section 4.2 with describing helper functions, that are not specific to the general problem of (α, β)-privacy but simplify the main algorithms. We then go on with the problem of *composition* in Section 4.3. We make a clear distinction between composing terms at the concrete level, where the intruder can find recipes, and at the structural level, where the recipes require reasoning about the possible values of variables. Then in Section 4.4, we develop an algorithm that completely analyses a structural frame, where we use the decomposition rules coming from the algebraic equations of the intruder theory. Finally, we develop in Section 4.5 the main algorithm generating a formula ϕ characterising all relations between variables that the intruder is able to find.

The algorithms are presented in semi-formal definitions very close to *functional programming*. The rationale is that this paradigm uses a rather declarative approach, and makes the translation to Haskell code for implementation easier. Therefore, it is very well suited when working in the logical setting of (α, β)-privacy. In the definitions, we use bindings with the construct "**let** ... **in**", the construct "**and**" for conjunction of boolean expressions and conditional statements with the construct "**if** ... **then** ... **else**". We also use λ-expressions, and the construct "**all**" checking if every element in a list satisfy a predicate. A function returns when it reaches an expression that can be evaluated.

4.2 Helper functions

In this section, we define helper functions that do not constitute any new work. They solve small problems and will be used for the algorithms defined in the

rest of the thesis.

4.2.1 Depth and weight

The notion of depth and weight are not necessary for understanding the algorithms and procedure designed in this thesis. However, they will be used when writing termination proofs.

Let $t \in \mathcal{T}_\Sigma(\mathcal{V})$ be a term. It is either a variable or the application of a function f of arity $n \geq 0$. The depth of t represents the largest number of function applications inside the term. It is defined recursively as:

$$depth(t) = \begin{cases} 1 & \text{if } t \text{ is a variable} \\ 1 + \max_{1 \leq i \leq n} \{depth(t_i)\} & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Example 4.2 *Let f and g be functions of arity 2 and 1 respectively. Consider the term $f(0, g(0))$. Then*

$$depth(f(0, g(0))) = 1 + depth(g(0)) = 2 + depth(0) = 3$$

Let $t \in \mathcal{T}_\Sigma(\mathcal{V})$ be a term. It is either a variable or the application of a function f of arity $n \geq 0$. The weight of t represents the number of all subterms occurring in t (by opposition to just the top level). It is defined recursively as:

$$weight(t) = \begin{cases} 1 & \text{if } t \text{ is a variable} \\ 1 + \sum_{i=1}^{n} weight(t_i) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Example 4.3 *Let f and g be functions of arity 2 and 1 respectively. Consider the term $f(0, g(0))$. Then*

$$weight(f(0, g(0))) = 1 + weight(0) + weight(g(0)) = 3 + weight(0) = 4$$

4.2.2 Unification

The problem of unification concerns equality between terms. For two terms $s, t \in \mathcal{T}_\Sigma(\mathcal{V})$, a solution to unification is a substitution σ (called unifier) such that $\sigma(s) = \sigma(t)$. The unification problem is extended to sets of equalities

between terms, where a solution unifies all equalities. A unification algorithm is useful to solve the problem by finding a solution if it exists.

Unification is very relevant for our procedure deciding privacy goals: we want to know how the intruder is able to compose terms based on their knowledge. Composition at the structural level involves considering terms with variables; unification allows to express the possibilities considered by the intruder, which need to be consistent with what they know. In this thesis, we consider only *syntactic* unification. This means that terms are equal if they are written in the exact same way, by contrast with *equational* unification, which is unification modulo an equational theory. Note that it is fine to limit ourselves to syntactic unification, because in our procedure we follow rewriting rules from the intruder theory. This reduction removes layers of function applications, so that in the end syntactic equality is enough to check equality between terms.

Syntactic unification is a standard problem for which there exists algorithms computing solutions [15, 2]. We do not contribute anything to this, but a description of the implementation of one algorithm is presented in Appendix A. From now on, we assume that we can call a function *unify* solving this problem.

The function *unify* takes one argument:

- a set of pairs $\{(s_1, t_1), \ldots, (s_n, t_n)\}$.

It computes a *most general unifier* for the set of equalities. That is to say, a unifier σ such that for every $(s, t) \in \{(s_1, t_1), \ldots, (s_n, t_n)\}$, $\sigma(s) = \sigma(t)$ and any substitution σ' unifying the same equalities is an instance of σ. We write $\sigma \lesssim \sigma'$ to denote this quasi-ordering on substitutions [2]: $\sigma \lesssim \sigma'$ if there exists τ such that $\sigma' = \tau\sigma$.

The *unify* function returns *one* most general unifier if it exists, but in case there are several solutions it does not matter which one is computed because they are equal up to renaming [2].

In case of unifying a pair of terms (s, t), we abuse slightly the notation and write $\mathit{unify}(s = t)$. As a substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ can be expressed as a set of pairs $\{(x_1, t_1), \ldots, (x_n, t_n)\}$, we also allow to use the notation $\mathit{unify}(\sigma_1, \ldots, \sigma_n)$ to designate a most general unifier of all pairs from the σ_i.

Example 4.4 *Let f, g be functions of arity 2 and 1 respectively, and let $x, y, z \in \mathcal{V}$. Then*

$$\mathit{unify}(f(x, y) = f(g(z), 1)) = \{x \mapsto g(z), y \mapsto 1\}$$

4.2.3 Checks of equivalence classes

In our modelling, the knowledge of the intruder is encoded in frames. The axioms $\phi_{frame}(\mathcal{F})$ and $\phi_{\mathcal{F}_1 \sim \mathcal{F}_2}$ of (α, β)-privacy can be used to express static equivalence between two frames. This notion corresponds to comparing pairs of recipes applied to the two frames. During our procedure, we will make checks for such pairs of recipes. When comparing frames, we want to be able to generate a number of pairs allowing to decide static equivalence. Consider the case an equivalence class $[\![t]\!]_{\approx} = \{t_1, \ldots, t_n\}$. It suffices to pick one element, e.g. t_1, and compare it with all the other elements. Our function *pairsEcs* performs this operation over a set of equivalence classes, and returns a set of all the pairs generated by each equivalence class. This function could work for an arbitrary equivalence relation given as a parameter, but since we only use it for recipes found by the composition algorithms, we do not need to use an explicit parameter.

4.3 Composition

The problem of composition is the following: given a knowledge and a term, what recipes can the intruder use to generate the term? In our case, we have to consider separately the problem at the concrete level, i.e. how the intruder can generate ground terms, and the problem at the structural level, where the intruder also has to consider the possible value taken by variables.

The following functions rely on the intruder theory, in particular the set of public functions Σ_{pub} and the set of variables with public range $\mathcal{V}_{pub}$. Instead of making these sets explicit parameters, they are considered to have been fixed beforehand when defining the intruder theory to study. In the algorithms, the constructs "f is public" (for a function $f \in \Sigma$) and "x has a public range" (for a variable $x \in \mathcal{V}$) refer to membership to Σ_{pub} and $\mathcal{V}_{pub}$ respectively.

For a variable $x \in \mathcal{V}_{pub}$, the intruder knows all values that x can take. Therefore, they can compare all possibilities and see which one is correct, i.e. to what constant c the variable x is mapped. For instance, as we mentioned in the mocking voting protocol earlier, the intruder knows the values that a vote can take: the votes can be modelled as variables with public range. The way we handle it in our procedure is explained along with the algorithm for composition in a structural frame.

4.3.1 Composition in a ground frame

We focus first on the smaller problem of finding ways to compose a term based on the knowledge at the concrete level. The intruder has recorded a protocol execution, and this knowledge is represented by the frame *concr*. This frame contains only *ground* terms, that is to say terms without any variables. Given a term, the question is what recipes can be used to compose it, using terms in *concr*. The recipes are expressed in a similar structure as ground terms, they are either labels of the frame or the application of a function over subrecipes. We want to find all recipes for the term, using only constructors. The case of destructors will be taken care of in Section 4.4. The idea is that since the intruder theory includes a rewriting system, we can now restrict the problem to syntactic equality.

The recursive function *compose* takes two arguments:

- a ground frame *concr*;
- a ground term $t \in \mathcal{T}_\Sigma$.

It computes a set of recipes to compose the term t from the knowledge in *concr*. The term is assumed to be ground, because variables cannot be composed at the concrete level. t can be composed if there is a label mapping to it, or if the top level function is public and all subterms can be composed. In order to find all recipes (with constructors), we consider all possible combinations of the subrecipes.

Algorithm 1: Composition in a ground frame

$compose(concr, t) =$
  let $R = \{\mathsf{l} \mid \mathsf{l} \mapsto t \in concr\}$ **in**
  if $t = f(t_1, \ldots, t_n)$ *and* f *is public* **then**
    $R \cup \{f(r_1, \ldots, r_n) \mid r_1 \in compose(concr, t_1),$
    $\ldots,$
    $r_n \in compose(concr, t_n)\}$
  else
    R

Example 4.5 *Let* $a, b, c \in \mathcal{T}_\Sigma \setminus \mathcal{T}_{\Sigma_{pub}}$. *Consider the frame*

$$concr = \{\!|\, \mathsf{l}_1 \mapsto a, \mathsf{l}_2 \mapsto b, \mathsf{l}_3 \mapsto \mathsf{h}(a), \mathsf{l}_4 \mapsto \mathsf{scrypt}(a, c) \,|\!\}$$

Then

$$compose(concr, \mathsf{h}(a)) = \{\mathsf{h}(\mathsf{l}_1), \mathsf{l}_3\}$$

which shows that the intruder knows two ways to compose the term. However, here

$$compose(concr, c) = \{\}$$

because composing the term c requires using a destructor (decomposition of the encrypted term). The frame needs to be analysed first to access the term c.

Proposition 4.1 *Let concr be a ground frame and $t \in \mathcal{T}_\Sigma$. Then the call $compose(concr, t)$ terminates.*

Proof. Let *concr* be a ground frame and $t \in \mathcal{T}_\Sigma$. There are recursive calls to *compose* if t is the application of a public function over subterms, i.e. $t = f(t_1, \ldots, t_n)$ where f is a public function of arity $n \geq 1$. In this case, every subterm is of depth strictly inferior to that of t: by definition of *depth*, we have that $\forall i \in \{1, \ldots, n\}, depth(t_i) < depth(t)$. The same is true for every recursive call. We consider a sequence formed by the depths of the argument passed to *compose*. It is a strictly decreasing sequence of natural integers. Since $(\mathbb{N}, \leq)$ is a well-ordered set, such a sequence is finite so the call terminates. □

All the recipes found by this algorithm really are recipes to compose the term in the given frame.

Proposition 4.2 *Let concr be a ground frame and $t \in \mathcal{T}_\Sigma$. Then*

$$\forall r \in compose(concr, t), concr\{\!|\, r \,|\!\} = t$$

Proof. Let *concr* be a ground frame and $t \in \mathcal{T}_\Sigma$. We proceed by induction on the structure of t. Let $R = \{\mathsf{l} \mid \mathsf{l} \mapsto t \in concr\}$. Then $\forall r \in R, concr\{\!|\, r \,|\!\} = t$.

- If $t = f(t_1, \ldots, t_n)$ and f is private, then $compose(concr, t) = R$.
- If $t = f(t_1, \ldots, t_n)$ and f is public, then

$$compose(concr, t) = R \cup \{f(r_1, \ldots, r_n) \mid r_1 \in compose(concr, t_1), \ldots, r_n \in compose(concr, t_n)\}$$

  For every $r \in compose(concr, t) \setminus R$, there exists $r_1 \in compose(concr, t_1)$, $\ldots, r_n \in compose(concr, t_n)$ such that $r = f(r_1, \ldots, r_n)$. By induction,

the proposition holds for the t_i, so

$$\begin{aligned} concr\{\!|\, r \,|\!\} &= concr\{\!|\, f(r_1, \ldots, r_n) \,|\!\} \\ &= f(concr\{\!|\, r_1 \,|\!\}, \ldots, concr\{\!|\, r_n \,|\!\}) \\ &= f(t_1, \ldots, t_n) \\ &= t \end{aligned}$$

□

There is typically an infinite number of recipes for a given term: we could imagine that a destructor and a constructor (from one algebraic equation) are applied arbitrarily many times. The algorithm verifies a completeness property, in the sense that it finds all recipes that use only constructors. This is not considered a limitation, because the problem is for now only composing a term. A recipe with destructors corresponds to *decomposing* a term, which will be covered by the analysis of a frame in Section 4.4.

Proposition 4.3 *Let concr be a ground frame, $t \in \mathcal{T}_\Sigma$ and r be a recipe containing only constructors such that $concr\{\!|\, r \,|\!\} = t$. Then $r \in compose(concr, t)$.*

Proof. Let *concr* be a ground frame, $t \in \mathcal{T}_\Sigma$ and r be a recipe containing only constructors such that $concr\{\!|\, r \,|\!\} = t$. We proceed by induction on the structure of r. Let $R = \{\mathsf{l} \mid \mathsf{l} \mapsto t \in concr\}$.

- If r is a label, then $r \in R$.
- If $r = f(r_1, \ldots, r_n)$ (where the $r_1, \ldots, r_n$ contain only constructors because r does), then $t = f(t_1, \ldots, t_n)$ where $concr\{\!|\, r_1 \,|\!\} = t_1, \ldots, concr\{\!|\, t_n \,|\!\} = t_n$. By induction, the proposition holds for the r_i, so $r \in \{f(r'_1, \ldots, r'_n) \mid r'_1 \in compose(concr, t_1), \ldots, r'_n \in compose(concr, t_n)\}$.

□

4.3.2 Composition in a structural frame

Now that we have seen how to compose terms at the concrete level, we focus on the corresponding problem for the structural level. The knowledge of the

protocol specification is represented by a structural frame *struct*. This frame contains terms that are not necessarily ground, which means that the presence of variables must be accounted for. As for standard composition, the intruder is able to compose a term by using public constructors. However, the difference now is that the intruder considers the values that variables can take. They reason about the possibilities that are consistent with their knowledge. This involves solving unification problems between a term to compose and the terms in the frame *struct*. The question is what recipes can be used to compose a term, *and under which substitution*. Therefore, we want to find all recipes for the term, using only constructors, with a unifier for each recipe allowing the equality to hold.

Example 4.6 *Consider the frame* $struct = \{\!|\, \mathsf{l}_1 \mapsto \mathsf{h}(x), \mathsf{l}_2 \mapsto \mathsf{scrypt}(k, y) \,|\!\}$, *for some variables* x, y *and a key* k. *Can the intruder compose the term* $\mathsf{h}(y)$*? There is the possibility that* $x = y$, *in which case* l_1 *is actually a recipe to compose* $\mathsf{h}(y)$. *This holds under the unifier* $\{x \mapsto y\}$, *which is one possibility considered by the intruder. However, the intruder definitely knows that* l_2 *cannot be a recipe to compose* $\mathsf{h}(y)$, *since* y *also appears inside the encrypted term so* $\mathsf{h}(y) = \mathsf{scrypt}(k, y)$ *is not a possibility, no matter the substitution applied to the equality.*

In our algorithm, the substitution must map variables to constants, and it must substitute at least the variables with public range. We do not mention here all of these conditions in the algorithm definition, but the actual application in the overall procedure will use such a substitution since θ will be set as a model of the formula α. Indeed, the substitution θ is the substitution representing the instantiation of all variables occurring in *struct*.

The recursive function *composeUnder* takes three arguments:

- a substitution θ;
- a frame *struct*;
- a term $t \in \mathcal{T}_\Sigma(\mathcal{V})$.

It computes a set of pairs (recipe, substitution) to compose t under a unifier from the knowledge in *struct*. The term can be composed by unifying with a term that is an element of *struct*: if there exists a solution to the equality, then the label is a recipe.

For a variable x with public range, the value can be directly read from the substitution θ, and the constant it maps to is also a recipe. This is because

having a public range means that the intruder knows all values that the variable can take; all these values are public constants, so they do not need to be explicitly included in the frame with labels. The recipe is then the constant $\theta(x)$ itself, and the unifier is the substitution $\{x \mapsto \theta(x)\}$ derived from the substitution θ. As we mentioned in Section 2.3, this does not mean that the intruder has access to θ but simply that they know the specific public value $\theta(x)$.

For a public function at the top level, all combinations of composition for the subterms are also computed, as in *compose*. The additional computation for combining subrecipes is combining also the unifiers. This is done by solving again a unification problem, as the top level unifier must be a solution for equalities of all subterms at the same time.

Algorithm 2: Composition in a structural frame

$composeUnder(\theta, struct, t) =$
  let $RU = \{(\mathsf{l}, \sigma) \mid \mathsf{l} \mapsto t' \in struct, \sigma = unify(t = t')\}$ **in**
  if $t = x$ *and* x *has a public range* **then**
    $RU \cup \{(\theta(x), \{x \mapsto \theta(x)\})\}$
  else if $t = f(t_1, \ldots, t_n)$ *and* f *is public* **then**
    $RU \cup \{(f(r_1, \ldots, r_n), \sigma) \mid (r_1, \sigma_1) \in composeUnder(\theta, struct, t_1)$
    $\ldots,$
    $(r_n, \sigma_n) \in composeUnder(\theta, struct, t_n),$
    $\sigma = unify(\sigma_1, \ldots, \sigma_n)\}$
  else
    RU

Example 4.7 *Let $a, b \in \Sigma$ be constants and $x, y \in \mathcal{V}$. Consider the substitution $\theta = \{x \mapsto a, y \mapsto b\}$ and the frame $struct = \{\!| \mathsf{l}_1 \mapsto x, \mathsf{l}_2 \mapsto \mathsf{h}(y) |\!\}$. Then*

$$composeUnder(\theta, struct, \mathsf{h}(y)) = \{(\mathsf{l}_1, \{x \mapsto \mathsf{h}(y)\}), (\mathsf{l}_2, \varepsilon), (\mathsf{h}(\mathsf{l}_1), \{y \mapsto x\})\}$$

which shows that the intruder has, at the structural level, more than one way to compose the term. The substitutions are the constraints to make the recipe valid. It remains to be checked later if those recipes are equivalent at the concrete level. We use the symbol ε to denote the identity substitution.

Proposition 4.4 *Let θ be a substitution, $struct$ be a frame and $t \in \mathcal{T}_\Sigma(\mathcal{V})$. Then the call $composeUnder(\theta, struct, t)$ terminates.*

Proof. Let θ be a substitution, *struct* be a frame and $t \in \mathcal{T}_\Sigma(\mathcal{V})$. There are recursive calls to *composeUnder* if t is the application of a public function over

subterms, i.e. $t = f(t_1, \ldots, t_n)$ where f is a public function of arity $n \geq 1$. The proof is the same as for *compose*, i.e. the depths of the argument passed to *composeUnder* form a strictly decreasing sequence of natural integers. Such a sequence is finite so the call terminates. □

Similarly to the case of *compose*, the pairs (recipe, substitution) found by this algorithm really allow to compose the term in the given frame, under a unifier.

Proposition 4.5 *Let θ be a substitution, struct be a frame and $t \in \mathcal{T}_\Sigma(\mathcal{V})$. Then*

$$\forall (r, \sigma) \in composeUnder(\theta, struct, t), \sigma(struct\{\!|\, r \,|\!\}) = \sigma(t)$$

Proof. Let θ be a substitution, *struct* be a frame and $t \in \mathcal{T}_\Sigma$. We proceed by induction on the structure of t. Let $RU = \{(\mathsf{l}, \sigma) \mid \mathsf{l} \mapsto t' \in struct, \sigma = unify(t = t')\}$. Then $\forall (r, \sigma) \in RU, \sigma(struct\{\!|\, r \,|\!\}) = \sigma(t)$.

- If $t = f(t_1, \ldots, t_n)$ and f is private, then $composeUnder(\theta, struct, t) = RU$.
- If $t = f(t_1, \ldots, t_n)$ and f is public, then

$$\begin{aligned} composeUnder(\theta, struct, t) = RU \\ \cup \{(f(r_1, \ldots, r_n), \sigma) \mid \\ (r_1, \sigma_1) \in composeUnder(\theta, struct, t_1), \\ \ldots, \\ (r_n, \sigma_n) \in composeUnder(\theta, struct, t_n), \\ \sigma = unify(\sigma_1, \ldots, \sigma_n)\} \end{aligned}$$

For every $(r, \sigma) \in composeUnder(\theta, struct, t) \setminus RU$, there exists
$(r_1, \sigma_1) \in composeUnder(\theta, struct, t_1)$,
$\ldots$,
$(r_n, \sigma_n) \in composeUnder(\theta, struct, t_n)$
such that $r = f(r_1, \ldots, r_n)$ and $\sigma = unify(\sigma_1, \ldots, \sigma_n)$. By induction, the proposition holds for the t_i so

$$\begin{aligned} \sigma(struct\{\!|\, r \,|\!\}) &= \sigma(struct\{\!|\, f(r_1, \ldots, r_n) \,|\!\}) \\ &= f(\sigma(struct\{\!|\, r_1 \,|\!\}), \ldots, \sigma(struct\{\!|\, r_n \,|\!\})) \\ &= f(\sigma(t_1), \ldots, \sigma(t_n)) \\ &= \sigma(f(t_1, \ldots, t_n)) \\ &= \sigma(t) \end{aligned}$$

□

Again, the next proposition is close to a property of the *compose* algorithm. Here, the algorithm finds all recipes that use only constructors, and pairs them with a most general unifier.

Proposition 4.6 *Let θ be a substitution, struct be a frame, $t \in \mathcal{T}_\Sigma(\mathcal{V})$, r be a recipe and τ be a substitution such that $\tau(struct\{\!|\, r \,|\!\}) = \tau(t)$ and r contains only constructors. Then*

$$\exists \sigma, (r, \sigma) \in composeUnder(\theta, struct, t) \text{ and } \sigma \lesssim \tau$$

Proof. Let θ be a substitution, *struct* be a frame, $t \in \mathcal{T}_\Sigma$, r be a recipe and τ be a substitution such that $\tau(struct\{\!|\, r \,|\!\}) = \tau(t)$ and r contains only constructors. We proceed by induction on the structure of r. Let $RU = \{(\mathsf{l}, \sigma) \mid \mathsf{l} \mapsto t' \in struct, \sigma = unify(t = t')\}$.

- If r is a label, then $(r, \varepsilon) \in RU$.
- If $r = f(r_1, \ldots, r_n)$ (where the $r_1, \ldots, r_n$ contain only constructors because r does), then $t = f(t_1, \ldots, t_n)$ for some $t_1, \ldots, t_n$ and

$$\begin{aligned} &\tau(struct\{\!|\, r_1 \,|\!\}) = \tau(t_1) \\ &\ldots \\ &\tau(struct\{\!|\, t_n \,|\!\}) = \tau(t_n) \end{aligned}$$

  By induction, the proposition holds for the r_i, so

$$\begin{aligned} (r, \sigma) \in \{(f(r'_1, \ldots, r'_n), \sigma') \mid\ & (r'_1, \sigma'_1) \in composeUnder(\theta, struct, t_1), \\ & \ldots, \\ & (r'_n, \sigma'_n) \in composeUnder(\theta, struct, t_n), \\ & \sigma' = unify(\sigma'_1, \ldots, \sigma'_n)\} \end{aligned}$$

  Moreover, $\sigma \lesssim \tau$ since they unify the same equalities and σ is a most general unifier.

□

4.4 Analysis

The problem of analysing a frame is a major part of the work done in this thesis. The analysis of a frame corresponds to adding all possible derivable terms. In our case, it means that the intruder tries to apply the decomposition rules, i.e. derive terms from the use of destructors, according to the algebraic equations defined with the cryptographic operators in the intruder theory. It can be seen as a fixed-point computation, as we start with the original frame and apply analysis steps as long as possible. This problem has also been defined as *frame saturation* [7].

In the definition of intruder theory in Section 3.1, we have also specified a function *ana* performing one decomposition. The idea that we follow is to define a recursive function applying these rules until none can be applied anymore. When trying to analyse a term, we consider the set of keys required to decompose it. We perform the analysis at the structural level, i.e. in a frame *struct*, in accordance with static equivalence with a the concrete frame *concr*. The reason is that, at the structural level, the intruder reasons about the possibilities for variables as we mentioned for composition in Section 4.3. At the same time, they can check if the term can be analysed at the concrete level in *concr*. Thus, the intruder can see if their hypotheses for the variables (in form of the substitutions returned by *composeUnder*) are valid or not, with respect to the static equivalence between *struct* and *concr*. Moreover, it suffices to return the completely analysed frame based on *struct*, since *concr* in our case is defined to be $\theta(struct)$.

4.4.1 Decomposition rules

The decomposition rules that the intruder is allowed to apply come from the function *ana*, as defined in the intruder theory. It applies one analysis step to a term, that is to say one decomposition rule. Moreover, the intruder is not only interested in the actual terms they can learn from this decomposition but also the step that was taken. Thus, we record for each decomposition the destructors applied according the algebraic equations. What we add to the frame are directly subterms of the term to analyse. This corresponds to using the rewriting system generated by the set of algebraic equations, where we remove the applications of destructors.

The function *ana* takes one argument:

- a term $t \in \mathcal{T}_\Sigma(\mathcal{V})$.

It computes the set of terms required (keys) and the set of derivable terms resulting from analysing t. Each derivable term is paired with a function symbol, which is the destructor to be used as part of the recipe to compose the term. One analysis step will be allowed if and only if all of the keys can be composed. This corresponds to our assumption that the intruder is not able to break the cryptography; they can only follow the rules of the intruder theory.

Example 4.8 *Recall the example set of cryptographic operators Σ_{op} from Table 3.1. We provide below the definition of ana for this specific example.*

$$ana(t) = \begin{cases} (\{\mathsf{priv}(s)\}, \{(\mathsf{dcrypt}, t')\}) & \text{if } t = \mathsf{crypt}(\mathsf{pub}(s), r, t') \\ (\{k\}, \{(\mathsf{dscrypt}, t')\}) & \text{if } t = \mathsf{scrypt}(k, t') \\ (\{\}, \{(\mathsf{retrieve}, t')\}) & \text{if } t = \mathsf{sign}(p', t') \\ (\{\}, \{(\mathsf{proj}_1, t_1), (\mathsf{proj}_2, t_2)\}) & \text{if } t = \mathsf{pair}(t_1, t_2) \\ (\{\}, \{\}) & \text{otherwise} \end{cases}$$

This definition matches exactly the algebraic equations corresponding to the properties of the cryptographic operators. Let $k, t_1, t_2 \in \mathcal{T}_\Sigma(\mathcal{V})$. Consider the term $\mathsf{scrypt}(k, \mathsf{pair}(t_1, t_2))$. *Then*

$$ana(\mathsf{scrypt}(k, \mathsf{pair}(t_1, t_2)) = (\{k\}, \{(\mathsf{dscrypt}, \mathsf{pair}(t_1, t_2))\}$$

This means that if the intruder knows the key k, then they are able to decrypt the term and learn $\mathsf{pair}(t_1, t_2)$. *Moreover, they know that this comes from applying* dscrypt. *The destructor will be stored as part of a shorthand, and is not present in the derivable terms that we actually add to the intruder knowledge. The other possible decompositions have to be done in other applications of ana.*

4.4.2 Analysis of a structural frame

The function *ana* described above corresponds to a single analysis step. In order to completely analyse a frame, i.e. achieve saturation of the frame, we need to apply the decomposition rules until a fixed-point is reached. The intruder analyses a structural frame *struct* and derivable subterms are added with shorthands for decomposition recipes. This will make all derivable subterms available with only composition rules, after the frame has been completely analysed. This explains why using our algorithms *compose* and *composeUnder*, that cover recipes containing only constructors, is a reasonable choice.

We have stated previously that the analysis is performed in accordance with static equivalence between the frames *struct* and *concr*. It means that if a decomposition were successful (all keys can be composed) in *struct* but not

concr, then they would not be statically equivalent as the application of a verifier would distinguish the frames.

Example 4.9 *Let $k_1, k_2, m \in \Sigma \setminus \Sigma_{pub}$ be constants and $x, y, z \in \mathcal{V}$, where $k_1 \neq k_2$. Consider the substitution $\theta = \{x \mapsto k_1, y \mapsto m, z \mapsto k_1\}$ and the frame $struct = \{\!| \mathsf{l}_1 \mapsto \mathsf{scrypt}(x, y), \mathsf{l}_2 \mapsto z |\!\}$. Then the completely analysed frame is*

$$struct_{ana} = \{\!| \mathsf{l}_1 \mapsto \mathsf{scrypt}(x, y), \mathsf{l}_2 \mapsto z, \mathsf{dscrypt}(\mathsf{l}_2, \mathsf{l}_1) \mapsto y |\!\}$$

because the decryption is successful at the concrete level: in $concr = \theta(struct)$, the intruder is able to compose the key $\theta(x) = k_1$ with the recipe l_2, so they can use the same recipe in struct for the corresponding key x.

However, for the substitution $\theta' = \{x \mapsto k_1, y \mapsto m, z \mapsto k_2\}$, then the analysed frame would be

$$struct_{ana} = \{\!| \mathsf{l}_1 \mapsto \mathsf{scrypt}(x, y), \mathsf{l}_2 \mapsto z |\!\}$$

since here the intruder cannot compose the required key. When trying to compose the key x, the algorithm composeUnder would have returned $(\mathsf{l}_2, \{x \mapsto z\})$ as a possibility, but this does not work in $concr' = \theta'(struct)$ so no terms can be added.

In order to compute the fixed-point of the completely analysed frame, we define a recursive function *analyseRec* that will take terms to analyse, apply one analysis step from calling *ana*, add terms if the decomposition was successful, and call itself to perform the other analysis steps.

To tackle the problem, we first consider that the intruder knowledge has been split into three frames. That way, we can make the distinction between the terms that have to be analysed in the future, the terms that might be decomposed later, and the terms that have already been completely analysed. Note that we do need to consider the terms "on hold", i.e. that might be decomposed later, because the intruder might learn at a later point the required keys.

Example 4.10 *Let $k_1, k_2, m \in \Sigma \setminus \Sigma_{pub}$ be constants and $x, y, z \in \mathcal{V}$. Consider the substitution $\theta = \{x \mapsto k_1, y \mapsto m, z \mapsto k_2\}$ and the frame $struct = \{\!| \mathsf{l}_1 \mapsto \mathsf{dscrypt}(x, y), \mathsf{l}_2 \mapsto \mathsf{pair}(x, z) |\!\}$.*

Suppose the intruder analyses first $\mathsf{dscrypt}(x, y)$. They are not able to compose the key x (with recipes containing only constructors), so the decomposition fails.

However, when they analyse $\mathsf{pair}(x, z)$, they can learn the terms x and z since no keys are required to decompose pairs. Therefore, a shorthand $\mathsf{m}_1 \mapsto x$ is added to the frame, where $\mathsf{m}_1 = \mathsf{proj}_1(\mathsf{l}_2)$. Now, the intruder is able to compose

the key x with only constructors, with the recipe m_1. Thus, they can actually decrypt the term $\mathsf{dscrypt}(x, y)$. This shows why it is important to put terms "on hold" during analysis in case they the intruder becomes able to analyse them at a later point.

Before defining the recursive function *analyseRec* computing a fixed-point, we present first the wrapper-function *analyse*, which simply calls *analyseRec* with the arguments properly initialised.

The function *analyse* takes two arguments:

- a substitution θ;
- a structural frame *struct*.

It analyses a structural frame *struct*, where the substitution θ maps all variables occurring in *struct* to constants. It computes the analysed frame $struct_{ana}$ and a set of substitutions E inconsistent with static equivalence between *struct* and $concr = \theta(struct)$. The substitutions in E correspond to unifiers such that a decomposition step fails at the concrete level but succeeds at the structural level: the unifier allowing the intruder to compose all keys would also allow to distinguish the frames. The intruder records those substitutions that are not consistent with their knowledge, so that they can rule out possibilities.

Algorithm 3: Analysis of a structural frame (wrapper)

$analyse(\theta, struct) =$
$\quad analyseRec(\theta, struct, \{\!|\ |\!\}, \{\!|\ |\!\}, \{\})$

When performing the analysis of a structural frame *struct*, all terms are initially considered "new" in the sense that they have to be analysed. There are, at the start, no elements "on hold" or "done". Moreover, we also indicate an empty set as the initial value of the set E of substitutions inconsistent with static equivalence of the frames. We denote the result of the analysis with $(struct_{ana}, E) = analyse(\theta, struct)$.

It remains now to define the function *analyseRec* that actually performs the analysis of the frame.

The recursive function *analyseRec* takes five arguments:

- a substitution θ;

- a frame with terms to analyse N (new);
- a frame with terms that could be analysed again later H (on hold);
- a frame with terms that are completely analysed D (done);
- a set of substitutions E inconsistent with static equivalence between the structural and concrete frames.

It analyses mappings "recipe to term" at the structural level, assuming static equivalence to a corresponding concrete frame. The fixed-point is reached when no mapping can be further analysed, i.e. the argument for "new" terms is an empty frame ($N = \{\!|\ |\!\}$). When considering a term to analyse, the function *ana* is called to apply, if possible, a decomposition rule.

The result of applying *ana* gives a set of keys K required to decompose the term, and a set FT of pairs (destructor, term) of derivable terms (with a destructor to define a shorthand). The first step is to check if there are indeed any new terms to learn or if the intruder already knows them. We filter the pairs for new terms in a set FT_{new}. If it is empty, there is nothing more to do, the intruder can analyse the next term. Otherwise, we try to decompose the term.

If the analysis fails in *concr*, i.e. at least one key cannot be composed at the concrete level, then it also fails in *struct* and no new terms can be added. However, since composition of all keys at the structural level might succeed even in this case, the unifiers allowing to compose all keys in *struct* are not consistent with the static equivalence. We add such substitutions to the set E.

If the analysis is successful in *concr*, i.e. all keys can be composed at the concrete level, then it is also successful in *struct* and recipes for the new terms use the recipes of keys in the concrete frame. The shorthands added at this point use the destructors paired with the new terms, and some recipes found for composing the keys. We not only add the terms coming from FT_{new}, but also the keys. We put the new mappings in the frame LT_{new} and add this to the new terms to analyse. All terms that were on hold also need to be analysed again, as the intruder might be able to successfully decompose them with their new knowledge.

The choice of these recipes is irrelevant: since the keys are also added to the frame, the other ways to compose them can be found with *compose*. The notation "*pick*" in the definition below refers to this choice, it means "take any one element from the set".

The function makes analysis steps as long as there are terms that can be added,

i.e. until a fixed-point is reached (frame saturation). This happens when the argument representing terms to analyse is an empty frame.

Algorithm 4: Analysis of a structural frame (recursive)

$$
\begin{array}{l}
\mathit{analyseRec}(\theta, N, H, D, E) = \\
\quad \textbf{if } N = \{\!|\ |\!\} \textbf{ then} \\
\quad\quad (H \cup D, E) \\
\quad \textbf{else} \\
\quad\quad \textbf{let } \{\!|\, \mathsf{l} \mapsto t \,|\!\} \cup LT = N \\
\quad\quad\quad (K, FT) = \mathit{ana}(t) \\
\quad\quad\quad \{k_1, \ldots, k_n\} = K \\
\quad\quad\quad FT_{new} = \{(f, t') \in FT \mid \forall r, r \mapsto t' \notin D\} \textbf{ in} \\
\quad\quad \textbf{if } FT_{new} = \{\} \textbf{ then} \\
\quad\quad\quad \mathit{analyseRec}(\theta, FT, H, \{\!|\, \mathsf{l} \mapsto t \,|\!\} \cup D, E) \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textbf{let } \mathit{struct} = N \cup H \cup D \\
\quad\quad\quad\quad \mathit{concr} = \theta(\mathit{struct}) \textbf{ in} \\
\quad\quad\quad \textbf{if } \{\} \in \{\mathit{compose}(\mathit{concr}, \theta(k)) \mid k \in K\} \textbf{ then} \\
\quad\quad\quad\quad \textbf{let } E_{new} = \{\sigma \mid (r_1, \sigma_1) \in \mathit{composeUnder}(\theta, \mathit{struct}, k_1), \\
\quad\quad\quad\quad\quad\quad\quad\quad \ldots, \\
\quad\quad\quad\quad\quad\quad\quad\quad (r_n, \sigma_n) \in \mathit{composeUnder}(\theta, \mathit{struct}, k_n), \\
\quad\quad\quad\quad\quad\quad\quad\quad \sigma = \mathit{unify}(\sigma_1, \ldots, \sigma_n)\} \textbf{ in} \\
\quad\quad\quad\quad \mathit{analyseRec}(\theta, LT, \{\!|\, \mathsf{l} \mapsto t \,|\!\} \cup H, D, E \cup E_{new}) \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \textbf{let } LT_{new} = \{\!|\, f(r_1, \ldots, r_n, \mathsf{l}) \mapsto t' \mid \\
\quad\quad\quad\quad\quad\quad\quad\quad (f, t') \in FT_{new}, \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{pick}\ r_1 \in \mathit{compose}(\mathit{concr}, \theta(k_1)), \\
\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{pick}\ r_n \in \mathit{compose}(\mathit{concr}, \theta(k_n))\,|\!\} \\
\quad\quad\quad\quad\quad\quad\quad\quad \cup \{\!|\, r \mapsto k \mid k \in K, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{pick}\ r \in \mathit{compose}(\mathit{concr}, \theta(k)), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \forall t', r \mapsto t' \notin \mathit{struct}\,|\!\} \textbf{ in} \\
\quad\quad\quad\quad \mathit{analyseRec}(\theta, LT_{new} \cup LT \cup H, \{\!|\ |\!\}, \{\!|\, \mathsf{l} \mapsto t \,|\!\} \cup D, E)
\end{array}
$$

Example 4.11 *Let $s, r, t \in \mathcal{T}_\Sigma$ and $x, y, z, u \in \mathcal{V}$. We consider the substitution $\theta = \{x \mapsto s, y \mapsto r, z \mapsto t, u \mapsto s\}$ and the frame*

$$\mathit{struct} = \{\!|\, \mathsf{l}_1 \mapsto \mathsf{crypt}(\mathsf{pub}(x), y, z), \mathsf{l}_2 \mapsto \mathsf{pair}(\mathsf{priv}(u), \mathsf{pub}(u)) \,|\!\}$$

Then

$$\begin{aligned} \mathit{analyse}(\theta, \mathit{struct}) = (\{\!| \, \mathsf{l}_1 \mapsto \mathsf{crypt}(\mathsf{pub}(x), y, z), \\ \mathsf{l}_2 \mapsto \mathsf{pair}(\mathsf{priv}(u), \mathsf{pub}(u)), \\ \mathsf{proj}_1(\mathsf{l}_2) \mapsto \mathsf{priv}(u), \\ \mathsf{proj}_2(\mathsf{l}_2) \mapsto \mathsf{pub}(u), \\ \mathsf{dcrypt}(\mathsf{proj}_1(\mathsf{l}_2), \mathsf{l}_1) \mapsto z \, |\!\}, \{\}) \end{aligned}$$

Proposition 4.7 *Let θ be a substitution and struct be a frame. Then the call $\mathit{analyse}(\theta, \mathit{struct})$ terminates.*

Proof. Let θ be a substitution and *struct* be a frame. Since by definition $\mathit{analyse}(\theta, \mathit{struct}) = \mathit{analyseRec}(\theta, \mathit{struct}, \{\!| \, |\!\}, \{\!| \, |\!\}, \{\})$, what we really want to show is that the call to *analyseRec* terminates. We now consider that the frame *struct* has been split into three frames N, H, D and denote with E the set of substitutions passed as argument to *analyseRec*. We abuse the notation and write $\mathit{weight}(N \cup H)$ to mean $\sum_{\mathsf{l} \mapsto t \in N \cup H} \mathit{weight}(t)$. We consider the tuple $(\mathit{weight}(N \cup H), \#N)$. When analysing the mapping $\mathsf{l} \mapsto t \in N$:

- If there are no new terms to be added from the analysis of t, $\mathsf{l} \mapsto t$ is removed from N and put in D for the recursive call. Then $\mathit{weight}(N \cup H)$ has decreased by $\mathit{weight}(t)$.
- If the analysis of t fails, $\mathsf{l} \mapsto t$ is removed from N and put in H for the recursive call. Then $\mathit{weight}(N \cup H)$ stays the same but $\#N$ has decreased by 1.
- If the analysis of t succeeds, $\mathsf{l} \mapsto t$ is removed from N and put in D. The new terms from the analysis and the terms that were on hold are put in N. Then $\mathit{weight}(N \cup H)$ has decreased by at least 1 (t is not present anymore but some of its subterms might be).

The lexicographic order on $(\mathbb{N}, \leq) \times (\mathbb{N}, \leq)$ forms a well-order and the sequence of tuples for the recursive calls is a strictly decreasing sequence bounded by $(0, 0)$, so such a sequence is finite and the call terminates. □

The analysis only adds shorthands to the original frame. It makes the derivable terms accessible by composition, but it does not modify the information encoded by the frame. To be absolutely rigorous, we have $\mathit{struct}_{ana}\{\!| \, r \, |\!\} \approx \mathit{struct}\{\!| \, r \, |\!\}$ only if the unifiers allowing to compose the keys at the structural level (e.g. $\{x \mapsto u\}$ in Example 4.11) have been applied to the frame. That is to say, when

the intruder finds equalities between variables that *must* hold during analysis, then the substitution is applied to the term so that we only keep the necessary variables (we do not need to distinguish variables that are equal). This step should technically be verified against privacy, i.e. perform a check to see if these equalities already violate the privacy of the payload α. In the rest of the thesis, we assume implicitly that these steps have been performed, so that the analysed frame really contains proper shorthands.

Proposition 4.8 *Let θ be a substitution and struct be a frame. Then*

$$\forall r, \mathit{struct}_{ana}\{\!|\, r \,|\!\} \approx \mathit{struct}\{\!|\, r \,|\!\},$$

where $(\mathit{struct}_{ana}, E) = \mathit{analyse}(\theta, \mathit{struct})$.

Proof. Let θ be a substitution, *struct* be a frame and r be a recipe. Let $(\mathit{struct}_{ana}, E) = \mathit{analyse}(\theta, \mathit{struct})$. All mappings $\mathsf{l} \mapsto t$ in *struct* are also in struct_{ana}. New terms are added according to the analysis rules of the intruder theory, so the analysed term must be a constructor applied to some terms. When analysing $\mathsf{l} \mapsto \mathsf{constr}(t_1, \ldots, t_n)$ the frame is augmented with mappings of the form $\mathsf{destr}(r_1, \ldots, r_m, \mathsf{l}) \mapsto t_i$, where there is an algebraic equation $\mathsf{destr}(k_1, \ldots, k_m, \mathsf{constr}(t_1, \ldots, t_n)) \approx t_i$ and the $r_1, \ldots, r_m$ are recipes for the $k_1, \ldots, k_m$. Therefore, the labels that the analysis adds are themselves recipes over the labels from the original frame. struct_{ana} is the frame *struct* with shorthands. □

The analysis finds all derivable terms. Shorthands have been determined for any successful decomposition of terms.

Proposition 4.9 *Let θ be a substitution and struct be a frame. Then for every recipe r, there exists a recipe r' containing only constructors such that $\mathit{struct}_{ana}\{\!|\, r' \,|\!\} \approx \mathit{struct}\{\!|\, r \,|\!\}$, where $(\mathit{struct}_{ana}, E) = \mathit{analyse}(\theta, \mathit{struct})$.*

Proof. Let θ be a substitution, *struct* be a frame and r be a recipe. Let $(\mathit{struct}_{ana}, E) = \mathit{analyse}(\theta, \mathit{struct})$. We proceed by induction on the structure of r. We consider the occurrence of a destructor f such that no subrecipe for the arguments of f contains destructors.

- If the destructor is applied to a label and the decomposition is successful, then during analysis a shorthand $\mathsf{m} = f(\ldots)$ has been added, i.e. $\mathit{struct}_{ana}\{\!|\, \mathsf{m} \,|\!\} \approx \mathit{struct}\{\!|\, f(\ldots) \,|\!\}$.

- If the destructor is applied to a constructor and the decomposition is successful, then the intruder has performed a useless step and it can be replaced by a term without the destructor (from the algebraic equations).
- If the decomposition is not successful, then we can replace the application of f by a corresponding constructor g_f that does not occur in any algebraic equations. This g_f represents failed decomposition, and can be considered a constructor because it can only be used to compose terms, there is no way to reduce the terms with any rewriting rule.

We have covered all cases since the subrecipes do not contain destructors. By induction, we are able to replace all occurrences of destructors in the recipe. A similar argument can be made for verifiers, which are basically a special case of destructors. We can define a recipe r' which is the same as r but all occurrences of destructors and verifiers have been replaced by the methods listed above. □

The definition of the function *analyse* uses an assumption of static equivalence between a structural frame and its concrete instantiation characterised by θ. The analysis preserves this property of static equivalence. In the rest of this thesis, we will refer to the set of models of α as Θ. Our approach is to reason about the possible interpretations, so we quantify our results over models in Θ.

Proposition 4.10 *Let θ be a substitution and struct be a frame. Then*

$$\forall \mathcal{I} \in \Theta, \mathcal{I} \models \mathit{struct} \sim \theta(\mathit{struct}) \iff \mathcal{I} \models \mathit{struct}_{ana} \sim \theta(\mathit{struct}_{ana}),$$

where $(\mathit{struct}_{ana}, E) = \mathit{analyse}(\theta, \mathit{struct})$.

Proof. Let θ be a substitution, *struct* be a frame and $\mathcal{I} \in \Theta$.
Let $(\mathit{struct}_{ana}, E) = \mathit{analyse}(\theta, \mathit{struct})$.

$$\begin{aligned}
&\mathcal{I} \models \mathit{struct} \sim \theta(\mathit{struct}) \\
&\quad\iff \\
&(\forall (r_1, r_2), \mathcal{I}(\mathit{struct})\{\!|\, r_1 \,|\!\} \approx \mathcal{I}(\mathit{struct})\{\!|\, r_2 \,|\!\} \iff \\
&\qquad\qquad \theta(\mathit{struct})\{\!|\, r_1 \,|\!\} \approx \theta(\mathit{struct})\{\!|\, r_2 \,|\!\}) \\
&\quad\iff \\
&(\forall (r_1, r_2), \mathcal{I}(\mathit{struct}_{ana})\{\!|\, r_1 \,|\!\} \approx \mathcal{I}(\mathit{struct}_{ana})\{\!|\, r_2 \,|\!\} \iff \\
&\qquad\qquad \theta(\mathit{struct}_{ana})\{\!|\, r_1 \,|\!\} \approx \theta(\mathit{struct}_{ana})\{\!|\, r_2 \,|\!\}) \\
&\quad\iff \\
&\mathcal{I} \models \mathit{struct}_{ana} \sim \theta(\mathit{struct}_{ana})
\end{aligned}$$

□

The algorithm presented here does not simply return the analysed frame, but also a set of substitutions. All substitutions returned are inconsistent with the assumption of static equivalence. This notion of inconsistency is formally expressed in the following proposition:

Proposition 4.11 *Let θ be a substitution and struct be a frame. Then*

$$\forall \mathcal{I} \in \Theta, \mathcal{I} \models \mathit{struct} \sim \theta(\mathit{struct}) \implies \forall \sigma \in E, \mathcal{I} \models \neg\sigma,$$

where $(\mathit{struct}_{ana}, E) = \mathit{analyse}(\theta, \mathit{struct})$.

Proof. Let θ be a substitution, *struct* be a frame and $\mathcal{I} \in \Theta$ such that $\mathcal{I} \models \mathit{struct} \sim \theta(\mathit{struct})$. Let $(\mathit{struct}_{ana}, E) = \mathit{analyse}(\theta, \mathit{struct})$ and $\sigma \in E$. The substitution σ has been found during analysis of some mapping $\mathsf{l} \mapsto t$ where all keys can be composed in the current *struct* under some unifier but the corresponding ground keys cannot be composed in $\theta(\mathit{struct})$. Let f be the verifier allowing to check for decomposition of the term t with the keys $k_1, \ldots, k_m$. Let
$(r_1, \sigma_1) \in \mathit{composeUnder}(\theta, \mathit{struct}, k_1)$,
$\ldots$,
$(r_m, \sigma_m) \in \mathit{composeUnder}(\theta, \mathit{struct}, k_m)$.
We define the recipe $r = f(r_1, \ldots, r_m, \mathsf{l})$ corresponding to the decomposition, and the substitution $\sigma = \mathit{unify}(\sigma_1, \ldots, \sigma_m)$ corresponding to the unifier allowing to compose all keys. Then $\sigma(\mathit{struct}\{\!|\, r \,|\!\}) \approx \sigma(\mathit{struct}\{\!|\, \mathsf{yes} \,|\!\})$ because the decomposition is successful in *struct*. However, the same decomposition is not possible in $\theta(\mathit{struct})$, so $\theta(\mathit{struct}\{\!|\, r \,|\!\}) \not\approx \theta(\mathit{struct}\{\!|\, \mathsf{yes} \,|\!\})$. Since $\mathcal{I}(\mathit{struct}) \sim \theta(\mathit{struct})$, we also have that $\mathcal{I}(\mathit{struct})\{\!|\, r \,|\!\} \not\approx \mathcal{I}(\mathit{struct})\{\!|\, \mathsf{yes} \,|\!\})$. Therefore, $\mathcal{I}$ does not model σ, because if it did there would be a pair of recipes, namely (r, yes), to distinguish the frames $\mathcal{I}(\mathit{struct})$ and $\theta(\mathit{struct})$. □

4.5 Relations between variables

In the two previous sections, we have solved the problem of composition of a term, both at the concrete and structural level, and of analysis of a structural frame. These steps are part of a larger procedure, which we describe now. Recall that our goal is to determine the *relations* between the variables of the protocol specification. By relations, we mean in this context whether variables are equal or not. At the structural level, the intruder knows how messages exchanged

during protocol executions look like: the terms in the structural knowledge include variables, which are instantiated to constants at the concrete level. The procedure designed generates a formula ϕ, which contains all equalities and inequalities between variables that the intruder is able to derive from their knowledge. We relate this formula ϕ to the problem of (α, β)-privacy. We present an algorithm that, for a given state of some protocol specification, finds relations between variables. We show that this procedure allows automated verification of privacy goals.

For our algorithm generating the formula ϕ encoding relations between variables, we want that the frames have been analysed so that we can use composition to compare all derivable terms. We consider the general case of one structural frame *struct* and one concrete frame *concr*. The goal is to find which variables are actually equal to each other, and which are not. Indeed, at the structural level the terms include variables. A variable is simply a placeholder that, in any protocol execution, is substituted to a constant. Privacy goals, encoded by the formula α, are expressed on the structural level of the protocol specification. For instance, it could be $\alpha \equiv x, y, z \in \{0, 1\}$ in a simple voting protocol, where the variables x, y, z correspond to some votes. The intruder can try to see if they can deduce $x = y$ or $y \neq z$ or other relations like this, based on their knowledge from β. If so, then privacy is violated.

Therefore, it makes sense to find how these equalities and inequalities between concrete terms translate to variables before their instantiation. This is used so that the formula ϕ becomes a consequence of the intruder knowledge, and thus we can check if it is an *interesting consequence* breaking privacy.

We specify a function *findRelations* that starts by performing the analysis of a frame before trying to find more relations. In this situation, the intruder has the knowledge of a structural frame *struct* and one protocol execution recorded as the concrete frame $concr = \theta(struct)$. We use the substitution θ, that maps all variables to constants, to denote this instantiation.

The result of the analysis of *struct* includes the analysed frame $struct_{ana}$ as well as a set of substitutions E, that we have shown contains substitutions inconsistent with static equivalence between *struct* and $\theta(struct)$. Thus, these substitutions encode some inequalities between variables. This has to be included in the formula ϕ, since it already constitutes some relations between variables that the intruder was able to deduce.

Example 4.12 *Let $k_1, k_2, m \in \Sigma$ be constants and $x, y, z \in \mathcal{V}$. Consider the substitution $\theta = \{x \mapsto k_1, y \mapsto m, z \mapsto k_1\}$ and the frame*

$$struct = \{\!| \, \mathsf{l}_1 \mapsto \mathsf{scrypt}(x, y), \mathsf{l}_2 \mapsto z \, |\!\}$$

The result of the analysis is

$$analyse(\theta, struct) = (\{\!|\, \mathsf{l}_1 \mapsto \mathsf{scrypt}(x, y), \mathsf{l}_2 \mapsto x, \mathsf{dscrypt}(\mathsf{l}_2, \mathsf{l}_1) \mapsto y \,|\!\}, \{\})$$

Then we would like our algorithm to find the possible relations between the variables. For instance, intuitively we can see that $x = z \wedge y \neq z$, because the decryption was successful (so $x = z$) and the intruder is able to compare the pair of recipes $(\mathsf{l}_2, \mathsf{dscrypt}(\mathsf{l}_2, \mathsf{l}_1))$ (so $y \neq z$) with composition in the analysed frame. There are more relations that the intruder could deduce, e.g. $z \neq \mathsf{dscrypt}(x, y)$ (which in this case also follows from $x = z$).

The function *findRelations* takes two arguments:

- a substitution θ;
- a structural frame *struct*.

It finds relations between the variables occurring in a structural frame *struct*, based on static equivalence with the concrete frame $concr = \theta(struct)$. It computes a set of equalities and a set of inequalities between variables occurring in a structural frame *struct*, based on static equivalence with a concrete frame *concr*. The equalities and inequalities are encoded in a logical formula ϕ.

First, the intruder tries to compose the terms inside *concr* in different ways. This is done by calling *compose* on the different labels and shorthands in the analysed concrete frame. If the intruder has several ways to compose a term, i.e. the composition algorithm returned several recipes, then pairs of recipes from these possibilities must also generate a unique term in *struct*. This provides a number of equalities. When we have all equalities, we compute a most general unifier so that we can write the results in a simpler way: an equality in our final formula ϕ will be of the form $x = t$ for some variable x and some term t.

Then, the intruder tries to compose the terms inside *struct* in different ways, under some unifiers. If they are able to compose a term in several ways, then we check if the pairs of recipes generate a unique term in *concr*. If it is the case, then there is nothing to deduce, it is just from static equivalence. However, if a pair of recipes distinguish the frames, i.e. we have found (l, r) such that $concr\{\!|\, \mathsf{l} \,|\!\} \neq concr\{\!|\, r \,|\!\}$, then the intruder knows that the unifier attached to r is not consistent with the static equivalence. They can deduce the negation of the unifier, i.e. a disjunction of inequalities.

Algorithm 5: Relations between variables

$$
\begin{array}{l}
\mathit{findRelations}(\theta, \mathit{struct}) = \\
\quad \textbf{let } (\mathit{struct}_{ana}, E) = \mathit{analyse}(\theta, \mathit{struct}) \\
\qquad \mathit{concr}_{ana} = \theta(\mathit{struct}_{ana}) \\
\qquad \mathit{pairs} = \mathit{pairsEcs}(\{\mathit{compose}(\mathit{concr}_{ana}, t) \mid \exists \mathsf{I}, \mathsf{I} \mapsto t \in \mathit{concr}_{ana}\}) \\
\qquad \mathit{eqs} = \{(\mathit{struct}_{ana}\{\!|\, r_1 \,|\!\}, \mathit{struct}_{ana}\{\!|\, r_2 \,|\!\}) \mid (r_1, r_2) \in \mathit{pairs}\} \\
\qquad \mathit{ineqs} = E \cup \{\sigma' \mid \mathsf{I} \mapsto t \in \mathit{struct}_{ana}, \\
\qquad\qquad\qquad\qquad (r, \sigma') \in \mathit{composeUnder}(\theta, \mathit{struct}_{ana}, t), \\
\qquad\qquad\qquad\qquad \mathsf{I} \neq r, \\
\qquad\qquad\qquad\qquad \mathit{concr}_{ana}\{\!|\, \mathsf{I} \,|\!\} \neq \mathit{concr}_{ana}\{\!|\, r \,|\!\}\} \\
\qquad \sigma = \mathit{unify}(\mathit{eqs}) \textbf{ in} \\
\quad \sigma \wedge \bigwedge_{\tau \in \mathit{ineqs}} \neg\tau
\end{array}
$$

In this definition, the result is the formula $\phi \equiv \sigma \wedge \bigwedge_{\tau \in ineqs} \neg\tau$. We do not follow strictly pseudo-code notation here for simplicity.

In general, there are more inequalities found than equalities. This depends on the length of the frame, because when composing a term the intruder tries to unify with the terms from their knowledge. The formula returned could also be simplified somewhat, for instance as defined here it could return both $y \neq z$ and $z \neq y$ (this would come from two unifiers computed separately), but then $y \neq z$ is enough. There are not efforts made to take into account symmetry and transitivity at this point. Still, the formula is not complex (it is just equalities and inequalities between a variable and a term) and at the same time enough to check for privacy.

Example 4.13 *We model a toy protocol corresponding to a doctor issuing a sick note to one of their patients. The doctor has a public and private key pair, they can log in to a trusted authority to receive a certificate, and they can use their key to sign a sick note. The patient will be able to show their sick note along with the doctor's certificate to their employer.*

We write ta[] for the name of the trusted authority, and we represent the key pair of the trusted authority with $\mathsf{pub}(s)$ *(known by everyone) and* $\mathsf{priv}(s)$ *(only known by the authority), where s was some seed used to generate the pair. We denote with* $\mathcal{D}$ *the names of all doctors registered to the trusted authority. We consider the situation where the intruder has recorded the connections of two doctors to the authority, as well as two different sick notes. This is encoded in*

the frame

$$\begin{aligned} \mathit{struct} = \{\!|\, & \mathsf{l}_1 \mapsto \mathsf{scrypt}(k_1, \mathsf{pair}(\mathsf{pair}(x_1, \mathsf{pw}(x_1, ta[])), \mathsf{pub}(x_1))), \\ & \mathsf{l}_2 \mapsto \mathsf{scrypt}(k_1, \mathsf{sign}(\mathsf{priv}(s), \mathsf{pair}(t_1, \mathsf{pub}(x_1)))), \\ & \mathsf{l}_3 \mapsto \mathsf{sign}(\mathsf{priv}(x_1), \mathsf{pair}(n_1, a)), \\ & \mathsf{l}_4 \mapsto \mathsf{sign}(\mathsf{priv}(s), \mathsf{pair}(t_1, \mathsf{pub}(x_1))), \\ & \mathsf{l}_5 \mapsto \mathsf{scrypt}(k_2, \mathsf{pair}(\mathsf{pair}(x_2, \mathsf{pw}(x_2, ta[])), \mathsf{pub}(x_2))), \\ & \mathsf{l}_6 \mapsto \mathsf{scrypt}(k_2, \mathsf{sign}(\mathsf{priv}(s), \mathsf{pair}(t_2, \mathsf{pub}(x_2)))), \\ & \mathsf{l}_7 \mapsto \mathsf{sign}(\mathsf{priv}(x_2), \mathsf{pair}(n_2, b)), \\ & \mathsf{l}_8 \mapsto \mathsf{sign}(\mathsf{priv}(s), \mathsf{pair}(t_2, \mathsf{pub}(x_2))) \,|\!\} \end{aligned}$$

where the messages mean respectively

1. *a doctor x_1 logs in with their password $\mathsf{pw}(x_1, s[])$ and sends their public key $\mathsf{pub}(x_1)$ (encrypted session);*
2. *the trusted authority sends back a certificate for the doctor's public key with a timestamp (encrypted session);*
3. *the doctor issues a sick note for a patient a, that they sign with their private key (n_1 may contain information related to formats, health data etc.);*
4. *the doctor also shares the certificate they received earlier from $ta[]$ so that other can verify the signature of the sick note;*
5. *a doctor x_2 logs in and sends their public key;*
6. *$ta[]$ sends a certificate for the doctor's public key;*
7. *the doctor issues a sick note for a patient b;*
8. *the doctor also shares their certificate.*

We do not add it explicitly to the frame, but the intruder has also access to $\mathsf{pub}(s)$ so they can verify signatures. By writing the name of a doctor as the seed, we mean that the same doctor always uses the same key pair.

The payload is $\alpha \equiv x_1, x_2 \in \mathcal{D}$, i.e. x_1 and x_2 are just the names of some doctors. We consider the concrete instantiation

$$\theta = \{x_1 \mapsto d, a \mapsto A, x_2 \mapsto d, b \mapsto B, \dots\}$$

which corresponds to the fact that the same doctor issued both sick notes. The values of the rest of the variables are not really relevant for the privacy goal.

Without going into the details as it becomes quite cumbersome to write (a good argument for an automated procedure), we mention what the algorithm would find in such a situation.

When performing the analysis of the frame struct, the intruder is able to open the signatures since they know $\mathsf{pub}(s)$. *Since at the concrete level the doctors' key pairs are the same, they are equal in struct as well. So our function findRelations will compute a unifier for, among other things, the equality* $\mathsf{pub}(x_1) = \mathsf{pub}(x_2)$ *and therefore find* $x_1 = x_2$ *as part of the formula* ϕ. *Without reporting all of the relations found, we can already see that* ϕ *does not follow from* α *so* (α, β)*-privacy does not hold.*

We formalise the correctness of the decision procedure that has been described. We argue that the algorithm *findRelations* is sound and complete, i.e. the formula ϕ generated encoding relations between variables really corresponds to static equivalence between *struct* and $concr = \theta(struct)$.

Proposition 4.12 *Let* θ *be a substitution and struct be a frame. Then*

$$\forall \mathcal{I} \in \Theta, \mathcal{I} \models struct \sim \theta(struct) \iff \mathcal{I} \models \phi,$$

where $\phi \equiv findRelations(\theta, struct)$.

Proof. Let θ be a substitution, *struct* be a frame and $\mathcal{I} \in \Theta$. Let $\phi \equiv findRelations(\theta, struct)$, $(struct_{ana}, E) = analyse(\theta, struct)$ and $concr_{ana} = \theta(struct_{ana})$.

- If $\mathcal{I} \not\models struct \sim \theta(struct)$: then $\mathcal{I}(struct_{ana}) \not\sim concr_{ana}$, so there exists a pair of recipes (r_1, r_2) that distinguishes the frames.
  - If $\mathcal{I}(struct_{ana})\{\!|\, r_1 \,|\!\} \not\approx \mathcal{I}(struct_{ana})\{\!|\, r_2 \,|\!\}$ and for the concrete frame $concr_{ana}\{\!|\, r_1 \,|\!\} \approx concr_{ana}\{\!|\, r_2 \,|\!\}$: then the function *findRelations* has found an equivalent pair of recipes (with constructors only) when trying to find equalities between terms. So there exists a substitution σ such that $\phi \models \sigma$ and $\sigma(struct_{ana}\{\!|\, r_1 \,|\!\}) = \sigma(struct_{ana}\{\!|\, r_2 \,|\!\})$. This unifier has been found by the algorithm from the set of equalities. Assume by contradiction that $\mathcal{I} \models \phi$, then also $\mathcal{I} \models \sigma$ and thus $\mathcal{I}(struct_{ana}\{\!|\, r_1 \,|\!\}) \approx \mathcal{I}(struct_{ana}\{\!|\, r_2 \,|\!\})$ which is false. So $\mathcal{I} \not\models \phi$.
  - If $\mathcal{I}(struct_{ana})\{\!|\, r_1 \,|\!\} \approx \mathcal{I}(struct_{ana})\{\!|\, r_2 \,|\!\}$ and for the concrete frame $concr_{ana}\{\!|\, r_1 \,|\!\} \not\approx concr_{ana}\{\!|\, r_2 \,|\!\}$: then the function *findRelations* has made an equivalent check when trying to find inequalities between terms. So there exists a substitution τ such that $\phi \models \neg\tau$ and

$\tau(struct_{ana}\{\!|\, r_1 \,|\!\}) \approx \tau(struct_{ana}\{\!|\, r_2 \,|\!\})$. This unifier has been found by the algorithm from the set of inequalities. Moreover, τ is a most general unifier of the equality $struct_{ana}\{\!|\, r_1 \,|\!\} = struct_{ana}\{\!|\, r_2 \,|\!\}$ so $\mathcal{I} \models \tau$. Therefore $\mathcal{I} \not\models \phi$.

- If $\mathcal{I} \models struct \sim \theta(struct)$: then also $\mathcal{I} \models struct_{ana} \sim \theta(struct_{ana})$. For every mapping $\mathsf{l} \mapsto t \in concr_{ana}$, the recipes in $compose(concr_{ana}, t)$ form an equivalence class. For every $(r_1, r_2) \in pairsEcs(compose(concr_{ana}, t))$, we have by definition that $concr_{ana}\{\!|\, r_1 \,|\!\} = concr_{ana}\{\!|\, r_2 \,|\!\}$.

  Since $\mathcal{I}(struct_{ana}) \sim concr_{ana}$, then also $\mathcal{I}(struct_{ana})\{\!|\, r_1 \,|\!\} \approx \mathcal{I}(struct_{ana})\{\!|\, r_2 \,|\!\}$. Therefore, $\mathcal{I} \models \sigma$ where

$$\begin{aligned}\sigma = unify(\{(&struct_{ana}\{\!|\, r_1 \,|\!\}, struct_{ana}\{\!|\, r_2 \,|\!\}) \mid \\ &\mathsf{l} \mapsto t \in concr_{ana}, \\ &(r_1, r_2) \in pairsEcs(compose(concr_{ana}, t))\})\end{aligned}$$

  Besides the exceptions found during analysis, a substitution τ such that $\mathsf{l} \mapsto t \in struct_{ana}$, $(r, \tau) \in composeUnder(\theta, struct_{ana}, t)$, $\mathsf{l} \neq r$ and $concr_{ana}\{\!|\, \mathsf{l} \,|\!\} \not\approx concr_{ana}\{\!|\, r \,|\!\}$ is found by the function *findRelations* and is inconsistent with the static equivalence between $struct_{ana}$ and $concr_{ana}$. Let *ineqs* be the set of substitutions E found during analysis union with the substitutions found by the *findRelations* algorithm. Then $\forall \tau \in ineqs, \mathcal{I} \models \neg\tau$. Therefore, $\mathcal{I} \models \sigma \wedge \bigwedge_{\tau \in ineqs} \neg\tau$ which is exactly $\mathcal{I} \models \phi$.

□

We now have everything to relate directly our results to (α, β)-privacy. We have designed a procedure that generates a formula ϕ of relations between variables, and this ϕ can be used to verify privacy for a message-analysis problem.

Proposition 4.13 *Let α be combinatoric, $\theta \in \Theta$ be a model of α and $struct = \{\!|\, \mathsf{l}_1 \mapsto t_1, \ldots, \mathsf{l}_k \mapsto t_k \,|\!\}$ for some $t_1, \ldots, t_k \in \mathcal{T}_\Sigma(fv(\alpha))$.*
Let $\beta \equiv MsgAna(\alpha, struct, \theta)$ and $\phi \equiv findRelations(\theta, struct)$. Then

$$(\alpha, \beta)\text{-privacy holds iff } \forall \mathcal{I} \in \Theta, \mathcal{I} \models \phi$$

Proof. Let α be combinatoric, $\theta \in \Theta$ be a model of α and $struct = \{\!|\, \mathsf{l}_1 \mapsto t_1, \ldots, \mathsf{l}_k \mapsto t_k \,|\!\}$ for some $t_1, \ldots, t_k \in \mathcal{T}_\Sigma(fv(\alpha))$. Let $\beta \equiv MsgAna(\alpha, struct, \theta)$

and $\phi \equiv \mathit{findRelations}(\theta, \mathit{struct})$. Then

$$
\begin{aligned}
&(\alpha, \beta)\text{-privacy holds} \\
&\iff \\
&\forall \mathcal{I} \in \Theta, \mathcal{I}(\mathit{struct}) \sim \theta(\mathit{struct}) \\
&\iff \\
&\forall \mathcal{I} \in \Theta, \mathcal{I} \models \mathit{struct} \sim \theta(\mathit{struct}) \\
&\iff \\
&\forall \mathcal{I} \in \Theta, \mathcal{I} \models \phi
\end{aligned}
$$

□

In this chapter, we have seen what an intruder is able to deduce from the structural frame *struct*, a concrete execution *concr*, and the static equivalence between these two frames. We have specified a number of algorithms that solves composition of terms, both at the concrete and structural level, analysis of a structural frame, and verification of (α, β)-privacy. Our procedure can be used to automate the verification. It is terminating, sound and complete for the decidable fragment formed by message-analysis problems.

Chapter 5

Decision procedure for protocols with branching

The procedure designed in the previous chapter is restricted to the situation where the intruder has the knowledge of one structural frame and one concrete frame. However, for many protocols the specification and the representation as a transition system involves conditional branching. For example, it might be the case that an encrypted message contains a different value depending on the result of some test. In such cases, when considering one state in the transition system, the intruder needs to make hypotheses about which branch of the protocol specification has actually been executed.

This means that the intruder knowledge can now be represented as a number of different structural frames, and we continue to assume that they have been able to record a single execution of the protocol. In this context, we lift the procedure working on one *struct* to a procedure working on several structural frames $struct_1, \ldots, struct_n, n \geq 2$. The motivation for studying this problem is that our decision procedure will support a larger class of protocols than the previous restricted problem without branching.

As before, the intruder knowledge is represented with frames. The difference with the case without branching is that the analysis is now performed for all different structural frames considered possible. Then, the relations between

variables are found independently for each of these hypotheses. This is done by reusing the functions described in Chapter 4. In the rest of this chapter, we distinguish the structural frames with an index $i \in \{1, \ldots, n\}$. One and only one of the possibilities really happened during the protocol execution. This particular structural frame is marked with the index 1. This is of course not something that the intruder knows, but it will be used to denote the concrete frame of the protocol execution with $concr = \theta(struct_1)$. The substitution θ characterises the instantiation of variables for the actual execution. What we mean by the intruder making one hypothesis is that they follow the procedure assuming static equivalence between some $struct_i$ ($i \in \{1, \ldots, n\}$) and $concr$.

The problem can still be expressed in terms of *message-analysis problems*. To express the situation in terms of (α, β)-privacy, we consider now that the intruder studies a set of message-analysis problems. Each of them corresponds to one hypothesis made by the intruder. We follow a method very close to the case of a single structural frame. The procedure designed in this chapter generates a number of formulae $\phi_1, \ldots, \phi_n$ characterising the relations between variables, for each hypothesis made by the intruder. We argue and prove that these formulae allow automated verification of privacy goals.

5.1 Static equivalence

In the case we study in this chapter, the knowledge of the intruder can be encoded into several structural frames and one concrete frame. Moreover, it includes the static equivalence between these frames, as the protocol execution must correspond to the protocol specification. We have seen in Chapters 2 and 4 how the notion of static equivalence is closely related to verifying privacy. Now that the intruder makes a number of hypotheses, they will try to exclude some possibilities if they can.

This involves reasoning about static equivalence in a way that we did not need before. Indeed, in Chapter 4, it was always the case that $concr = \theta(struct)$, so it was not necessary to actually verify static equivalence between them: we have designed our procedure to work in such a situation by assuming that static equivalence held. However, the intruder knows only knows that there is some $struct_i$ such that $concr = \theta(struct_i)$, but they do not know which one. In order to rule out hypotheses, it is useful to be able to decide static equivalence. This is not fundamentally a requirement, but makes it easier to talk about the hypotheses that the intruder is able to rule out.

5.1.1 Decision algorithm for ground frames

Static equivalence of frames is not specific to the problem we tackle in this thesis. For now, we forget about the (α, β)-privacy setting and consider the general case of two frames. The question is how to verify if they are statically equivalence, that is to say: is there any way to distinguish them? The approach taken is to generate a number of checks, based on the recipes found with the *compose* function. We require that the frames are ground and saturated, i.e. no term in the frames contains variables and they are completely analysed. The reason for these conditions is that the composition algorithm we apply is correct for a ground frame, and that the frame must have been analysed beforehand to handle the case of decompositions.

The function *areStatEq* takes two arguments:

- a ground and analysed frame $\mathcal{F}_1$;
- a ground and analysed frame $\mathcal{F}_2$.

It performs a number of checks to determine whether the frames are statically equivalent or not. Static equivalence holds if and only if all of the checks succeed, i.e. the frames agree on all pairs of recipes checked. To find all relevant checks, we apply *compose* successively on all terms inside the frames. Since we assume that they have been completely analysed, this gives sets of recipes to compose all derivable terms. We then convert the sets of recipes into pairs to checks, with our function *pairsEcs*, since recipes for the same term form an equivalence class. All pairs of recipes from the frame $\mathcal{F}_1$ must generate a unique term in $\mathcal{F}_2$, and vice-versa.

Algorithm 6: Static equivalence between ground frames

$areStatEq(\mathcal{F}_1, \mathcal{F}_2) =$
  let $C_1 = pairsEcs(\{compose(\mathcal{F}_1, t) \mid \exists \mathsf{l}, \mathsf{l} \mapsto t \in \mathcal{F}_1\})$
    $C_2 = pairsEcs(\{compose(\mathcal{F}_2, t) \mid \exists \mathsf{l}, \mathsf{l} \mapsto t \in \mathcal{F}_2\})$ **in**
  (**all** $(\lambda\ (r_1, r_2).\mathcal{F}_2\{\!|\, r_1 \,|\!\} = \mathcal{F}_2\{\!|\, r_2 \,|\!\})\ C_1$
    and all $(\lambda\ (r_1, r_2).\mathcal{F}_1\{\!|\, r_1 \,|\!\} = \mathcal{F}_1\{\!|\, r_2 \,|\!\})\ C_2$)

Example 5.1 *Let $k, v_1, v_2 \in \Sigma$ be constants. Consider the frames*

$$\mathcal{F}_1 = \{\!|\, \mathsf{l}_1 \mapsto \mathsf{scrypt}(k, v_1), \mathsf{l}_2 \mapsto \mathsf{scrypt}(k, v_2) \,|\!\}$$
$$\mathcal{F}_2 = \{\!|\, \mathsf{l}_1 \mapsto \mathsf{scrypt}(k, v_1), \mathsf{l}_2 \mapsto \mathsf{scrypt}(k, v_1) \,|\!\}$$

Then

$$areStatEq(\mathcal{F}_1, \mathcal{F}_2) = false$$

because the pair of recipes $(\mathsf{l}_1, \mathsf{l}_2)$ *distinguishes the frames.*

The fact that the frames are completely analysed justifies the restriction to composition for finding appropriate checks. Indeed, *compose* computes a finite set of recipes. In the definition of static equivalence, the frames must agree on *any* pair of recipes, and there is an infinite number of recipes. We argue that the algorithm we defined is still correct.

Proposition 5.1 *Let* $\mathcal{F}_1, \mathcal{F}_2$ *be ground and analysed frames. Then*

$$areStatEq(\mathcal{F}_1, \mathcal{F}_1) = true \iff \mathcal{F}_1 \sim \mathcal{F}_2$$

.

Proof. Let $\mathcal{F}_1, \mathcal{F}_2$ be ground and analysed frames.

- If $areStatEq(\mathcal{F}_1, \mathcal{F}_2) = false$: the algorithm has found a pair of recipes that distinguishes the frames so $\mathcal{F}_1 \not\sim \mathcal{F}_2$.
- If $areStatEq(\mathcal{F}_1, \mathcal{F}_2) = true$: the frames are analysed, so for any pair (r_1, r_2) of recipes, there is a pair (r'_1, r'_2) of recipes containing only constructors such that $\mathcal{F}_i\{\!|\, r'_1 \,|\!\} \approx \mathcal{F}_i\{\!|\, r_1 \,|\!\}$ and $\mathcal{F}_i\{\!|\, r'_2 \,|\!\} \approx \mathcal{F}_i\{\!|\, r_2 \,|\!\}$ (for $i \in \{1, 2\}$).
  - If r'_1 or r'_2 is a label, then the check is found by the algorithm when calling *compose*.
  - Otherwise, there is a function f such that $r'_1 = f(u_1, \ldots, u_n)$ and $r'_2 = f(v_1, \ldots, v_n)$. Then for checking the pair (r'_1, r'_2), it suffices to check all pairs (u_i, v_i).

  Thus, the pairs generated from calling *compose* on the terms in the frames are enough to verify static equivalence.

□

We are now able to decide if two frames are statically equivalent, assuming that they are ground and have been analysed before. Going back to the (α, β)-privacy problem at hand, we will make use of this method to rule out hypotheses: if the intruder is able to observe that their candidate frame for the concrete execution is not statically equivalent to the actual execution *concr* that they have recorded, then they can exclude this possibility.

5.2 Analysis

5.2.1 Analysis of several structural frames

The intruder knows several structural frames $struct_1, \ldots, struct_n$. In order to reuse our method to find relations, we again need to analyse the structural frames. The different $struct_i$ considered by the intruder are analysed separately. This analysis procedure is exactly the same as before, which means that we call the function *analyse* for every possibility. Note that this means, for a possibility $struct_i$, that the analysis is done relying on static equivalence between $struct_i$ and $\theta(struct_i)$. The substitution θ is the same for all possibilities, because it simply encodes the single concrete execution recorded by the intruder. There is only one "true" structural frame corresponding to *concr*, and it is $struct_1$. The intruder will try to see if the hypotheses they make are consistent with this single concrete execution, or if they can rule out some possibilities.

The function *analyseMulti* takes two arguments:

- a substitution θ;
- a set of possible structural frames *structs*.

It analyses a set of structural frames *structs*. Every possibility is analysed separately, resulting in a number of pairs (analysed frames, substitutions), where the substitutions returned are inconsistent with the static equivalence of the structural and concrete frames.

Algorithm 7: Analysis of several structural frames

$$analyseMulti(\theta, structs) = \bigsqcup \{analyse(\theta, struct) \mid struct \in structs\}$$

Note that even though we extensively use set notations in this thesis, the actual algorithms preserve the orders of elements (which are in lists rather than sets). Therefore, we consider that we can still identity the different values returned by *analyseMulti* with the same indices $1, \ldots, n$ as before the analysis.

5.3 Relations between variables for several structural frames

In a similar way to analysis, the method to find relations between variables is basically lifting the results of Chapter 4 to several structural frames. When we defined the function *findRelations*, we set the concrete frame to be $\theta(struct)$ since it was the only possibility. Now, the intruder still considers a single protocol execution. But they have to compare it with the different structural frames of their hypotheses. Thus, we can apply the procedure for finding relations to the result of the analysis and this particular concrete frame, which is the same for every possibility.

The function *findRelationsMulti* takes two arguments:

- a substitution θ;
- a set of possible structural frames $\{struct_1, \ldots, struct_n\}$.

It finds relations between variables for a set of possible structural frames. When the intruder considers several possibilities, each frame is analysed and relations are found for each possibility. The result is a set of analysed frames, each with a formula encoding equalities and inequalities between variables of the frame.

The difference with our previous procedure, is that now we do want to verify whether the $struct_i$ and $concr$ are statically equivalent. If they are, the intruder will try to deduce relations for the current hypothesis. If they are not, the intruder will rule out the current hypothesis and thus break privacy right away. We follow the same approach as before for finding relations, but we will specify two algorithms: *findRelationsMulti* will apply the analysis and method to find relations to every hypothesis, and *relations* will adapt the algorithm for finding relations to the case of one hypothesis.

Algorithm 8: Relations between variables for several structural frames

$findRelationsMulti(\theta, \{struct_1, \ldots, struct_n\}) =$
  let $S = \{struct_1, \ldots, struct_n\}$
    $SE = analyseMulti(\theta, S)$
  $concr = \theta(struct_{1\ ana})$ **in**
  $\{(struct_{ana}, relations(\theta, struct_{ana}, concr, E) \mid (struct_{ana}, E) \in SE\}$

In this definition, we denote with $struct_{1\ ana}$ the analysed frame after analysis of $struct_1$.

The function *relations* takes four arguments:

- a substitution θ;
- a possible structural frame $struct_i$;
- a concrete frame $concr$;
- a set of substitutions E inconsistent with $struct_i \sim \theta(struct_i)$ (coming from the analysis).

There is actually little difference to the algorithm from Chapter 4. Our new algorithm takes one $concr$, that might not be $\theta(struct_i)$, and a set of substitutions E of substitution inconsistent with the static equivalence between $struct_i$ and $\theta(struct_i)$ (these have been found during analysis of the frame). We start by making a check of static equivalence between $\theta(struct_i)$ and $concr$, corresponding to verify if the intruder can rule out the possibility. If they can rule out their current hypothesis, then we set $\phi_i \equiv false$; the intruder has broken privacy. Otherwise, we apply the same method as before to generate ϕ_i.

Algorithm 9: Relations between variables for a specific hypothesis

$$
\begin{array}{l}
relations(\theta, struct, concr, E) = \\
\quad \textbf{if } areStatEq(\theta(struct), concr) \textbf{ then} \\
\quad\quad \textbf{let } (struct_{ana}, E) = analyse(\theta, struct) \\
\quad\quad\quad pairs = pairsEcs(\{compose(concr, t) \mid \exists \mathsf{I}, \mathsf{I} \mapsto t \in concr\}) \\
\quad\quad\quad eqs = \{(struct_{ana}\{\!|\, r_1 \,|\!\}, struct_{ana}\{\!|\, r_2 \,|\!\}) \mid (r_1, r_2) \in pairs\} \\
\quad\quad\quad ineqs = E \cup \{\sigma' \mid \mathsf{I} \mapsto t \in struct_{ana}, \\
\quad\quad\quad\quad\quad\quad\quad\quad (r, \sigma') \in composeUnder(\theta, struct_{ana}, t), \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{I} \neq r, \\
\quad\quad\quad\quad\quad\quad\quad\quad concr\{\!|\, \mathsf{I} \,|\!\} \neq concr\{\!|\, r \,|\!\}\} \\
\quad\quad\quad \sigma = unify(eqs) \textbf{ in} \\
\quad\quad \sigma \wedge \bigwedge_{\tau \in ineqs} \neg\tau \\
\quad \textbf{else} \\
\quad\quad false
\end{array}
$$

We note with $\phi_i, i \in \{1, \ldots, n\}$ the formulae encoding the relations between variables that were returned by the procedure, for several structural frames $struct_1, \ldots, struct_n$. As for the case with a single structural frame, we formalise the correctness of the procedure. The formulae $\phi_1, \ldots, \phi_n$ that have been generated correspond each to the assumption of static equivalence between structural and concrete frames. Note that here, the concrete frame $concr = \theta(struct_1)$ is the same for all possibilities considered by the intruder.

Proposition 5.2 *Let θ be a substitution and $struct_1, \ldots, struct_n$ be frames. Let $concr = \theta(struct_1)$. Then*

$$\forall \mathcal{I} \in \Theta, \forall i \in \{1, \ldots, n\}, \mathcal{I} \models struct_i \sim concr \iff \mathcal{I} \models \phi_i$$

PROOF. Let θ be a substitution, $struct_1, \ldots, struct_n$ be frames, $\mathcal{I} \in \Theta$ and $i \in \{1, \ldots, n\}$. Let $concr = \theta(struct_1)$.

- If $\mathcal{I} \not\models struct_i \sim \theta(struct_1)$: then $\mathcal{I}(struct_i) \not\sim \theta(struct_1)$, so $\phi_i \equiv false$ (as we make a check by calling *areStatEq*) and therefore $\mathcal{I} \not\models \phi_i$.
- If $\mathcal{I} \models struct_i \sim \theta(struct_1)$: then also $\mathcal{I} \models struct_{i\ ana} \sim \theta(struct_{1\ ana})$. The argument is the same as for the case of a single *struct*. We can use it because of the transitivity of static equivalence: the substitutions inconsistent with $struct_i \sim \theta(struct_i)$ (from the analysis) are also inconsistent with $struct_i \sim \theta(struct_1)$.

□

Again, we relate directly our result with the notion of (α, β)-privacy. This time, a model must verify the conjunction of all the relations found. The reason is that, if there is any possibility that the intruder is able to rule out, then the intruder has broken privacy.

Proposition 5.3 *Let α be combinatoric, $\theta \in \Theta$ be a model of α and $structs = \{struct_1, \ldots, struct_n\}$ be a set of frames with terms over $fv(\alpha)$.*

Let $\beta \equiv MsgAna(\alpha, struct_1, \theta)$. Then

$$(\alpha, \beta)\text{-privacy holds iff } \forall \mathcal{I} \in \Theta, \forall i \in \{1, \ldots, n\}, \mathcal{I} \models \phi_i$$

where $\{(struct_{i\ ana}, \phi_i) \mid i \in \{1, \ldots, n\}\} = findRelationsMulti(\theta, structs)$.

PROOF. Let α be combinatoric, $\theta \in \Theta$ be a model of α and let $structs = \{struct_1, \ldots, struct_n\}$ be a set of frames with terms over $fv(\alpha)$.

Let $\beta \equiv MsgAna(\alpha, struct_1, \theta)$, $concr = \theta(struct_1)$ and $\{(struct_{i\ ana}, \phi_i) \mid i \in$

$\{1, \ldots, n\}\} = \mathit{findRelationsMulti}(\theta, \mathit{structs})$. Then

$$
\begin{aligned}
&(\alpha, \beta)\text{-privacy holds} \\
&\iff \\
&\forall \mathcal{I} \in \Theta, \mathcal{I}(\mathit{struct}_1) \sim \cdots \sim \mathcal{I}(\mathit{struct}_n) \sim \mathit{concr} \\
&\iff \\
&\forall \mathcal{I} \in \Theta, \forall i \in \{1, \ldots, n\}, \mathcal{I} \models \mathit{struct}_i \sim \mathit{concr} \\
&\iff \\
&\forall \mathcal{I} \in \Theta, \forall i \in \{1, \ldots, n\}, \mathcal{I} \models \phi_i
\end{aligned}
$$

□

We have thus successfully extended our decision procedure to support protocols with branching, in which the intruder reasons about different possibilities. We have leveraged the notion of static equivalence of frames in order to prove the correctness of our procedure. We have shown how our work is related to typical (α, β)-privacy. Therefore, we have managed to provide a method for automated verification of privacy for decidable fragments of (α, β)-privacy.

Chapter 6

Discussion

This thesis introduces a number of algorithms and a decision procedure to automatically verify privacy goals of communication protocols, without and with branching. We have argued the correctness of the method presented in this work, under a number of assumptions. These assumptions restrict the scope of the procedure to a specific class of intruder theories. This means that some protocols are not yet supported by the situations considered until now. We see our decision procedure as a first step towards the development of machine-checking of (α, β)-privacy goals.

6.1 Limitations

It must be noted that the procedure designed works in one state of a transition system. We have mentioned why lifting our work, considering a single state, to an entire transition system is not a real limitation as it poses no theoretical difficulty. However, the idea of studying privacy in transition systems assumes that the modeller has defined a protocol specification that can be translated as such a transition system. This requires a formal description of the protocol in a certain manner, which does not necessarily exist yet. We consider though that the development of automated verification procedures, supporting more

and more protocols, is a good incentive for protocols designers and researchers to come up with these formal descriptions.

One assumption restricting the problem considered concerns the notion of equality between terms. We have clearly stated that the algorithms, as defined in this thesis, support algebraic equations inducing a convergent rewriting system. Many security protocols use require more equations to hold. For example, the first version of the Diffie-Hellman key exchange uses commutativity of exponentiation [11]: for three numbers g, a, b, we have that $(g^a)^b = (g^b)^a$. On the syntactic level, the corresponding terms are not equal. Protocols making use of such properties cannot be verified by the procedure designed with the current definitions, because the properties like commutativity of exponentiation do not fit with our definition of convergent intruder theory.

This thesis considers problems that can be expressed as message-analysis problems in the (α, β)-privacy framework. There are protocols that fall outside of this scope, in this case the procedure designed cannot be applied to decide privacy.

6.2 Future work

(α, β)-privacy is a novel approach to verifying privacy goals. It allows to manually find proofs when studying privacy. However, automation is paramount to develop the approach further and enable its application to complex real-life protocols. We feel that the procedure designed in this thesis forms a good starting point towards the development of automated verification and tool support.

While we have made a number of contributions, enabling automated verification of privacy for a decidable fragment, it would be very fruitful to study other fragments of (α, β)-privacy. In every problem that we studied, we have for example considered that $fv(\alpha) = fv(\beta)$, i.e. the variables of β are part of the payload formula. It would be interesting to consider the case of $fv(\alpha) \subsetneq fv(\beta)$, i.e. β contains some variables which are not expressed in the payload formula. Our procedure is based on the idea of relations between variables, and we generate a formula ϕ that can be used to verify privacy. However, the same results cannot apply trivially if our ϕ includes variables that are not from α, because then we cannot use it as a witness of an interesting consequence.

We can point out also the study of other fragments of (α, β)-privacy, that do not have the same restrictions on the α and β formulae. Together with the extension to $(\alpha, \beta, \gamma, \delta)$-privacy [12], there is the also the case of the quantitative approach

to privacy. In this context, we would need to revisite our idea of relations between variables to not reason simply about equality or inequality, but rather include a notion of probability.

Another potentially fruitful path is to relate more specifically (α, β)-privacy with other existing approaches, e.g. observational and trace equivalence. This has already been described in [16], and we think that there is still room for more research in this area. We have mostly worked on static equivalence of frames with the particular application to (α, β)-privacy. We can still investigate classical privacy goals and translate them into the logical setting of (α, β)-privacy.

A promising path for improvement is the refinement of our algorithms to support arbitrary equational theories. We have already mentioned that the class of intruder theories supported by our procedure is not enough for some classical properties of protocols. Therefore, keeping our core idea but reasoning about equational theories in general would be a good way to extend the scope of our tool.

On more practical note, one possibility to improve this tool support is to investigate an input language. While this thesis does not focus on the details of implementation (because it follows closely the algorithms specifications), it can already be used in a Haskell program. For now, the modeller has to define directly the protocol specification at the data structure level. It would be more convenient to provide this through a text file in an appropriate language. The solution could be either choosing an existing language allowing to express a protocol specification in an easy way, or to design a dedicated input language.

Chapter 7

Conclusion

This thesis is about automated verification of privacy goals, based on the logical approach of (α, β)-privacy developed in [16]. We have defined a number of algorithms and designed a procedure that enables this automated reasoning. Our idea was to consider a protocol specification that can be translated as a transition system, and to provide a decision procedure in one state of the protocol execution. The approach was to find relations between variables of the protocol, which can be used to check whether there is a breach of privacy or not.

In Chapter 2, we recalled definitions from [16] about the logic used, Herbrand logic, the notions of messages, operators and algebraic properties as well as the encoding of knowledge into frames. In particular, Section 2.3 reuses definitions from previous works to present the concept of (α, β)-privacy. We explained what constitutes a breach of privacy and related to the problem to the question of static equivalence of frames.

In Chapter 3, we have made our first contributions by defining the notion of intruder theory, representing the behaviour of the intruder through a number of rules. We have identified more specifically what we call convergent intruder theories, on which we have been able to base our decision procedure. We have also complemented the notion of frames with the concept of frames with shorthands.

In Chapter 4, we have presented our main contributions. We defined several algorithms breaking down the problem into the notions of composition of terms, analysis of frames and relations between variables. We have considered what the intruder is able to deduce by making the distinction between concrete information from an actual protocol execution, and structural information about the protocol specification. We have argued for the correctness of the procedure that we designed and have thus shown that it really can be used to decide privacy.

Chapter 5 is concerned with extending the results to protocols with branching, where several possibilities of the protocol execution form different hypotheses. We have described the choices of modelling and how the procedure can support this class of protocols.

Finally, in Chapter 6 we have stated clearly the assumptions made in this thesis, and the current limitations of our procedure. We have also identified promising leads for future work.

Parts of our implementation of the decision procedure designed are discussed in Appendix A, where we describe some interesting points.

Appendix A

Implementation

We give some explanations related to implementation details. The whole implementation is provided in a source code file `Lib.hs`, and we do not present all of it here. We make use of types and functions from standard libraries. In Chapters 4 and 5, we often use set notations. In the code, we rather work on lists, that is to say sequences instead of sets. A sequence is an ordered set, where elements can appear multiple times. In Haskell, the `Maybe` monad is typically used to represent a computation that might fail. In our case, we use it when solving a unification problem. If there exists a solution, then the algorithm returns a most general unifier. Otherwise, it fails. This result is wrapped as a monadic computation, so that it can be used conveniently in the other functions.

```
import qualified Data.List
import qualified Data.Map
import          Data.Maybe
```

```
as L
as M
( catMaybes
, fromMaybe
, isJust
, mapMaybe
, maybeToList
)
```

A.1 Types

The first step of the implementation was to define the data structures needed by the different functions. We thus define types to represent function symbols from our alphabet, variables, terms, recipes, substitution and frames.

```
-- | A function symbol is represented as a string.
type Function = String
-- | A variable name is represented as a string.
type Variable = String

-- | A @Term@ is either a @Variable@ or a @Function@ applied to a list of
-- subterms. A constant is a @Function@ applied to the empty list
-- (no subterms).
data Term = Var Variable
          | Fun Function [Term]
          deriving (Eq, Ord)

-- | A recipe is a @Function@ applied to subrecipes. It should not be a
-- @Variable@, since this is not enforced by this definition it will be
-- checked when relevant in functions using recipes.
type Recipe = Term
-- | A @Substitution@ maps @Variable@ to @Term@.
type Substitution = M.Map Variable Term
-- | A @Frame@ maps @Recipe@ (called labels) to @Term@. It is used to
-- represent the knowledge of the intruder.
type Frame = M.Map Recipe Term
```

A.2 Intruder theory

We have stated that our procedure is parameterised over a convergent intruder theory. In our implementation, we have made tests by using the intruder theory defined in Example 3.1.

A.2.1 Public functions and variables with public range

One component of the intruder theory is the set of public functions and the set of variables with public range. We have defined this in the code by specifying a list of public function symbols and a list of variable names. Then, the property of being public is simply membership to those lists. Recall that in our example

intruder theory, all cryptographic operators are public except for priv (private keys are supposed to be kept secret, they cannot be composed by the intruder if they do not know the key directly). For the variables with public range, we have only made a few tests with some specific variables names, it is an arbitrary example of $\mathcal{V}_{pub}$.

```
-- | Names of public functions.
publistFun =
  [ "pub"
  , "crypt"
  , "dcrypt"
  , "vcrypt"
  , "sign"
  , "retrieve"
  , "vsig"
  , "scrypt"
  , "dscrypt"
  , "vscrypt"
  , "pair"
  , "proj1"
  , "proj2"
  , "vpair"
  , "h"
  ]

-- | A @Function@ is public if it is in @publistFun@.
public :: String → Bool
public f = f `elem` publistFun

-- | Names of variables with public range.
publistVar = [v1, v2, v3]

-- | A @Variable@ has a public range if it is in @publistVar@.
pubvar :: String → Bool
pubvar x = x `elem` publistVar
```

A.2.2 Analysis rewriting system

The implementation of *ana* follows the specification given in the example. We use pattern-matching to determine which decomposition rule we can apply.

```
-- | Compute the list of @Term@ required (keys) and the list of derivable
-- @Term@ resulting from analysing a @Term@.
-- Each derivable @Term@ is paired with a @Function@, the destructor to
-- be used as part of the @Recipe@ to compose the @Term@.
```

```
ana :: Term → ([Term], [(Function, Term)])
ana (Fun "scrypt" [k , t ])                = ([k],  [( "dscrypt", t)])
ana (Fun "crypt" [Fun "pub" [s], r, t ])   = ([Fun "priv" [s]],  [( "dcrypt", t)])
ana (Fun "sign"  [k , t ])                 = ([],   [( "retrieve", t)])
ana (Fun "pair"  [t1, t2])                 = ([],   [( "proj1", t1), ("proj2", t2)])
ana _                                      = ([],   [])
```

A.3 Unification

We have mentioned that several algorithms exist to solve unification. In our implementation, we have followed a standard algorithm based on a set of rules, to apply until no longer possible [2]. While this is not a very efficient algorithm, it has the benefit of being not too difficult to understand. If efficiency becomes a problem, then it will be possible to implement other algorithms [15, 2]. The rules to apply are:

- **Delete:** an equality is deleted if the two sides are syntactically equal;
- **Decompose:** an equality between two function applications is transformed into all equalities between the arguments;
- **Clash:** if the two functions are not the same then there is no solution (for syntactic unification);
- **Orient:** a variable is put to the left-hand side of an equality (to go towards a solved form);
- **Occurs check:** a variable cannot be unified with a term that contains it;
- **Eliminate:** when found, a mapping from a variable to a term is applied to all other equalities.

```
-- | Compute a most general unifier of the equalities given. The result
-- is a @Maybe Substitution@, which is @Nothing@ if there is no unifier.
unify :: [(Term, Term)] → Substitution → Maybe Substitution
unify []           sigma = Just sigma
unify ((s, t) : ts) sigma = if s == t
  then unify ts sigma -- Delete
  else case (s, t) of
    (Fun f us, Fun g vs) → if f == g
      then unify (zip us vs ++ ts) sigma -- Decompose
      else Nothing -- Clash
    (Fun f us, Var x) → unify ((t, s) : ts) sigma -- Orient
```

```
    (Var x   , _    ) → if x `elem` vars t
      then Nothing -- Occurs check
      else
        let sigma' = M.insert x t sigma
        in  unify (substituteList sigma' ts) sigma' -- Eliminate

-- | Compute a most general unifier of the equality between two @Term@.
unifyEq :: Term → Term → Maybe Substitution
unifyEq s t = unify [(s, t)] M.empty
```

This implementation calls a function `vars` that returns the list of variables occurring in a term and a function `substituteList` that applies a substitution to both sides of a list of equalities.

A.4 Combinations as a cartesian product

Many of our semi-formal definitions use set-builder notation. For the problem of composition of a term, we have stated that our algorithm combines all recipes for subterms. In the code, these combinations are computed as a cartesian product.

The recursive function *cartProd* takes one argument:

- a list of list.

It computes the cartesian product of a list of lists. It is useful in that it generates all combinations possible of taking one element in each list.

The base case is the cartesian product of an empty list. In that case, there are no lists in which elements can be taken. So the only combination is itself an empty list. The general case is the cartesian product of at least one list X. In that case, we consider the cartesian product of the rest of the lists XS. Note that with these notations, X is a list while XS is a list of lists. A combination is then taking one element $x \in X$ and one list Y from the recursive call.

The implementation is only ever used on recipes, but it is valid for any type.

```
-- | Compute the cartesian product of a list of lists.
cartProd :: [[ a]]  → [[a]]
cartProd []          = [[]]
cartProd (xs : xss) = [ x :  ys |  let yss = cartProd xss, x ← xs, ys ← yss ]
```

Example A.1 *Let* `rss = [[r1, r2, r3], [r4, r5]]`. *Then*

```
cartProd rss = [[r1, r4], [r1, r5], [r2, r4], [r2, r5], [r3, r5]]
```

A.5 Relations between variables

Our implementation differs slightly from the algorithms present in Chapters 4 and 5 in that *findRelations* actually calls *relations*. Indeed, the case of a single structural frame *struct* is a special case of our procedure for several $struct_i$, where we have $concr = \theta(struct)$.

```
-- | Compute a list of equalities and a list of inequalities between
-- @Variable@ occurring in a structural @Frame@, based on static
-- equivalence with a concrete @Frame@.
-- The frames are assumed to have been already analysed. Equalities
-- between @Variable@ are found when there are several recipes to compose
-- a term in the concrete frame. Inequalities are found when there are
-- pairs of recipes distinguishing the two frames. The unifiers such that
-- terms are equal in the structural frame but not in the concrete frame
-- are not consistent with the static equivalence, they are exceptions.
relations
  :: Substitution                    -- ^ The instantiation of variables
  → Frame                            -- ^ The structural frame
  → Frame                            -- ^ The concrete frame
  → [Substitution]                   -- ^ The list of exceptions found
  → ([(Term, Term)], [[( Term, Term)]]) -- ^ The equalities and inequalities
relations theta struct concr exceptions =
  let pairs = pairsEcs . L.map (compose concr) $ M.elems concr
      eqs   = L.map (\(r1, r2) → (cook struct r1, cook struct r2)) pairs
      ineqs = L.map subToEqs $ exceptions ++ L.nub
        [ sigma
        | (l, t    ) ← M.assocs struct
        , (r, sigma) ← composeUnder theta struct t
        , l /= r
        , sigma `notElem` exceptions
        , cook concr l /= cook concr r
        ]
      unifier = unify eqs M.empty
  in  case unifier of
        Nothing    → error $ "There␣is␣no␣unifier!␣" ++ showEqs eqs
        Just sigma → (subToEqs sigma, ineqs)
```

In this definition, we call a function `cook` that applies a recipe to a frame and a function `subToEqs` that converts a substitution to a set of equalities (pairs).

Instead of defining a data structure for logical formulae, since we only have a formula ϕ expressing relations between variables our function *findRelations* actually returns a set of equalities and a set of inequalities, where inequalities are the negation of a unifier.

```
-- | Find relations between the variables occurring in a structural
-- @Frame@, based on static equivalence with the concrete @Frame@ found
-- with the @Substitution@.
-- The @Frame@ is first analysed. Then @relations@ is called on the
-- result of the analysis.
findRelations
  :: Substitution                    -- ^ The instantiation of variables
  → Frame                            -- ^ The structural frame
  → ([(Term, Term)], [[( Term, Term)]]) -- ^ The equalities and inequalities
findRelations theta struct =
  let (structAna, exceptions) = analyse theta struct
      concrAna                = getConcr theta structAna
  in  relations theta structAna concrAna exceptions
```

The checks for static equivalence with the function *areStatEq* are actually used as a filter: the intruder is only interested in the formulae ϕ_i for the hypotheses that were not ruled out, since we have specified in Chapter 5 that if the hypothesis can be ruled out the corresponding formula is simply $\phi_i \equiv \mathit{false}$.

```
-- | Find relations between variables for a list of possible structural
-- @Frame@, using the same @Substitution@ for each of the possibility.
-- When the intruder considers several possibilities, each frame is
-- analysed and relations are found for each possibility. The result
-- is a list of analysed @Frame@, each with a list of equalities and a
-- list of inequalities.
findRelationsMulti
  :: Substitution                                -- ^ The theta
  → [Frame]                                      -- ^ The possible structs
  → [(Frame, ([(Term, Term)], [[( Term, Term)]]))] -- ^ The relations
findRelationsMulti theta [] =
  error $ "The␣list␣of␣possibilities␣is␣empty,␣at␣least␣one␣frame␣is"
        ++ "␣required."
findRelationsMulti theta structs@(struct1 : structis) =
  let structAnaExs = analyseMulti theta structs
      struct1ana   = fst $ head structAnaExs
      concr1ana    = getConcr theta struct1ana
  in  [ (structAna, relations theta structAna concr1ana exceptions)
      | (structAna, exceptions) ← structAnaExs
      , let concrAna = getConcr theta structAna
      , areStatEq concr1ana concrAna
      ]
```

Bibliography

[1] Max-Planck-Institut für Informatik, SPASS theorem prover. `https://www.spass-prover.org`. Accessed: 2021-01.

[2] Baader, F., and Nipkow, T. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] Basin, D., Dreier, J., and Sasse, R. Automated symbolic proofs of observational equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1144–1155.

[4] Bernhard, D., Pereira, O., Smyth, B., and Warinschi, B. Adapting Helios for provable ballot privacy. In *ESORICS'11: 16th European Symposium on Research in Computer Security, volume 6879 of LNCS* (2011), Springer, pp. 335–354.

[5] Blanchet, B. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security 1*, 1–2 (oct 2016), 1–135.

[6] Boutet, A., Castelluccia, C., Cunche, M., Dmitrienko, A., Iovino, V., Miettinen, M., Nguyen, T. D., Roca, V., Sadeghi, A.-R., Vaudenay, S., Visconti, I., and Vuagnoux, M. Contact Tracing by Giant Data Collectors: Opening Pandora's Box of Threats to Privacy, Sovereignty and National Security. University works, EPFL, Switzerland ; Inria, France ; JMU Würzburg, Germany ; University of Salerno, Italy ; base23, Geneva, Switzerland ; Technical University of Darmstadt, Germany, dec 2020.

[7] Comon, H. Static equivalence, 2020. Chapter 6 of Lecture notes in Formal and Computation Proofs, module of MPRI.

[8] Comon-Lundh, H., and Cortier, V. Computational soundness of observational equivalence, 2014.

[9] Cortier, V., and Delaune, S. A method for proving observational equivalence, 2010.

[10] Cortier, V., Drăgan, C. C., Dupressoir, F., Schmidt, B., Strub, P. Y., and Warinschi, B. Machine-checked proofs of privacy for electronic voting protocols. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), pp. 993–1008.

[11] Diffie, W., and Hellman, M. New directions in cryptography. *IEEE Transactions on Information Theory 22*, 6 (1976), 644–654.

[12] Gondron, S., Mödersheim, S., and Viganò, L. Privacy as reachability, 2021. Submitted for publication.

[13] Hinrichs, T., and Genesereth, M. Herbrand logic. Tech. Rep. LG-2006-02, Stanford University, Stanford, CA, 2006.

[14] Iovino, V., Vaudenay, S., and Vuagnoux, M. On the effectiveness of time travel to inject COVID-19 alerts. Cryptology ePrint Archive, Report 2020/1393, 2020.

[15] Martelli, A., and Montanari, U. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4*, 2 (1982), 258–282.

[16] Mödersheim, S., and Viganò, L. Alpha-beta privacy. *ACM Trans. Priv. Secur. 22*, 1 (2019), 7:1–7:35.

[17] Moran, M., and Wallach, D. S. Verification of STAR-vote and evaluation of FDR and ProVerif. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 10510* (2017), 422–436.

[18] Vaudenay, S., and Vuagnoux, M. Analysis of SwissCovid. `https://lasec.epfl.ch/people/vaudenay/swisscovid/swisscovid-ana.pdf`, 2020. Accessed: 2021-04.