DTU Compute
Department of Applied Mathematics and Computer Science

A Logical Approach for Automated Reasoning about Privacy in Security Protocols

PhD Thesis

Laouen Fernet

Title:	A Logical Approach for Automated Reasoning about Privacy in Security Protocols
Authored by:	Laouen Fernet
Date:	November 2024
Copyright:	Reproduction of this publication in whole or in part must include the customary bibliographic citation, including author attribution, report title, etc.
Published by:	DTU, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Building 324, 2800 Kongens Lyngby Denmark www.compute.dtu.dk

Approval

This thesis was prepared at the Department of Applied Mathematics and Computer Science (DTU Compute) in fulfillment of the requirements for acquiring a PhD degree in Computer Science.

The research has been carried out under the supervision of Sebastian Mödersheim and Luca Viganò in the period from December 2021 to November 2024.

A substantial part of the work presented in this thesis is based on extensions to joint work with Sebastian Mödersheim and Luca Viganò, namely the following papers:

- "A Decision Procedure for Alpha-Beta Privacy for a Bounded Number of Transitions" [44] (published),
- "Private Authentication with Alpha-Beta-Privacy" [41] (published),
- "A decision procedure and typing result for alpha-beta privacy" [43] (submitted) and
- *A Compositionality Result for Alpha-Beta Privacy* [42] (manuscript).

Chapters 2 and 3 are based on [44, 43], Chapter 4 is based on [43], Chapter 5 is based on [42] and Chapter 6 is based on [41].

November 2024

[illegible]

Laouen Fernet

Abstract (English)

Security protocols are distributed systems that rely on cryptographic operations to provide security features. Formal verification of security protocols is important to guarantee that a protocol achieves its goals: the verification is done by reasoning about various properties such as secrecy, authentication and privacy. There are several approaches and techniques to obtain formal proofs of security. In many cases, these proofs are manually written, however this may lead to invalid results due to oversights by the prover or implicit assumptions. Automated verification has shown to be effective in preventing such errors, and there are nowadays a number of tools available that can check security properties of protocols with a very high degree of confidence. While secrecy and authentication goals are typically expressed as reachability properties, privacy is more subtle. A standard way to define privacy is to give a pair of systems that differ in some detail (e.g., two actions are performed by the same agent or by different agents) and check whether an attacker can distinguish the two systems (different notions of distinguishability can be used depending on the privacy property). An alternative approach is (α, β)-privacy, where the protocol specification declares the information that is allowed to be learned and a violation of privacy happens if the attacker manages to learn more than allowed, e.g., if they can deduce that in two instances of the protocol, some actions were performed by the same user (linkability attack). The formalism of (α, β)-privacy enables reasoning about privacy as a reachability property, which makes some proofs easier. Moreover, we do not have to think about possible attacks and prove equivalence between different systems, but we rather model the logical deductions that the attacker can make based on their observations of the real system. There are still several challenges when it comes to automation, which is the main topic of this thesis. We identify a decidable fragment of (α, β)-privacy and design a decision procedure that can check privacy goals, given a bound on the number of transitions in the protocol execution.

One kind of attacks on security protocols relies on the attacker reusing messages in a way that confuses the other participants. This issue arises when the protocol specification does not sufficiently distinguish the meaning of different messages. Considering a typed model where the protocol specifies for every message a type expressing its meaning, these attacks are called type-flaws and occur when a participant accepts a message of a different type than expected. For instance, someone may receive an encrypted message with an expected specific format, but the protocol does not check that the content under encryption is indeed of the right format. When working in such a typed model, it is very useful to establish typing results of the form: "if there exists an attack, then there exists a well-typed one." Effectively, this rules out all type-flaws. Moreover, the assumption that every message is well-typed significantly helps to establish other results on security protocols. Ideally, we want to verify whether protocols are resistant to type-flaws using purely syntactic requirements, since these can be checked statically without considering the semantics of the protocol execution.

One area that benefits from typing results is protocol composition. In the literature, the security goals of a protocol are typically studied with the protocol running in isolation.

However, oftentimes several protocols are running in parallel on the same device or are otherwise interacting with each other. For instance, one protocol could be used to establish a secure connection between two endpoints, while another protocol relies on this secure connection to exchange sensitive information. Another example is several applications sharing a common infrastructure such as an identity server. Protocol composition poses significant challenges to formal verification. Composed systems are much larger and complex than individual components and thus verification is more difficult. Moreover, components can be added or updated and we do not want to verify the entire composed system from scratch as soon as there is a slight change. It is thus desirable to obtain results for composing protocols securely in a modular way, i.e., we wish to derive the security of a composed protocol from the security of its components. While there are a number of results supporting protocol composition for goals like secrecy and authentication, it is harder to achieve compositionality for privacy properties.

In this thesis, we are concerned with the automation of verifying privacy properties of security protocols and with the proofs of typing and compositionality results. We use (α, β)-privacy, which is a symbolic approach that aims to provide a logical and intuitive way of specifying privacy goals. Our results support large classes of security protocols, with standard cryptographic operators, non-determinism, branching and statefulness. Our main contributions are as follows:

- Design of a decision procedure for a fragment of (α, β)-privacy.
- Implementation of the procedure in a prototype tool.
- Application of the procedure and the tool to case studies.
- A typing result for the class of type-flaw resistant protocols.
- A compositionality result for the class of composable protocols.

Abstract (dansk)

Sikkerhedsprotokoller er distribuerede systemer, der er afhængige af kryptografiske operationer for at levere sikkerhedsfunktioner. Formel verifikation af sikkerhedsprotokoller er vigtigt for at sikre, at en protokol opnår sine mål: verifikationen sker ved at ræsonnere om forskellige egenskaber som f.eks. hemmeligholdelse, autentificering og privathed. Der er flere tilgange og teknikker for at opnå formelle beviser af sikkerhed. I mange tilfælde er disse beviser håndskrevne, hvilket kan føre til ugyldige resultater grundet overseelser hos bevisføreren og implicitte antagelser. Automatiseret verifikation har vist sig at effektivt kunne forhindre sådanne fejl, og der er i dag en række værktøjer til rådighed, som kan tjekke sikkerhedsegenskaber af protokoller med en meget høj grad af tillid. Mens hemmeligholdelses- og autentificeringsmål typisk formuleres som opnåelighedsegenskaber, er privathedsmål mere subtile. En standard måde at definere privathedsegenskaber på, er ved at give to systemer, der adskiller sig på specifikke punkter (f.eks. hvor to handlinger udføres af en eller to agenter) og så derefter undersøge, om en angriber kan skelne mellem de to systemer (hvor de parametre, der skelnes på, afhænger af den pågældende privathedsegenskab). En alternativ tilgang er (α, β)-privathed, hvor en protokolspecifikation angiver de oplysninger, der må læres. Der sker en krænkelse af privathed, hvis det lykkes angriberen at lære mere end tilladt, f.eks. hvis denne kan udlede, at nogle handlinger blev udført af den samme bruger i to tilfælde af protokollen (angreb på linkbarhed). Formaliseringen af (α, β)-privathed gør det muligt at ræsonnere om privathed som en opnåelighedsegenskab, hvilket gør nogle beviser lettere. Desuden behøver vi ikke tænke på mulige angreb og bevise ækvivalens mellem forskellige systemer, i stedet modellerer vi de logiske udledninger, som angriberen kan foretage sig baseret på deres observationer af det virkelige system. Der er stadig flere udfordringer, når det gælder automatisering, hvilket er hovedemnet for denne afhandling. Vi identificerer et afgørligt fragment af (α, β)-privathed og designer en beslutningsprocedure, der kan verificere privathedsmål, givet en grænse for antallet af skridt i protokoludførelsen.

En slags angreb på sikkerhedsprotokoller går ud på, at angriberen genbruger beskeder på en måde, der forvirrer de andre deltagere. Dette problem opstår, når protokolspecifikationen ikke i tilstrækkelig grad skelner mellem betydningen af forskellige beskeder. I en typet model, hvor protokollen specificerer en type for hver besked, der indikerer beskedens betydning, kaldes disse angreb for typefejl, og opstår, når en deltager accepterer en besked af en anden type end forventet. For eksempel kan nogen modtage en krypteret besked med et forventet format, hvor protokollen ikke tjekker, at det krypterede indhold faktisk har det rigtige format. Når man arbejder i en sådan typet model, er det meget nyttigt at etablere typningsresultater af formen: "hvis der findes et angreb, så findes der et veltypet et." Dette udelukker effektivt alle typefejl. Desuden hjælper antagelsen om, at alle beskeder er veltypede, betydeligt når det kommer til at etablere andre resultater der omhandler sikkerhedsprotokoller. Ideelt set ønsker vi at verificere, hvorvidt protokoller er modstandsdygtige imod typefejl ved hjælp af udelukkende syntaktiske krav, da disse kan tjekkes statisk uden at tage hensyn til semantikken af protokoludførelsen.

Et område, der drager fordel af typningsresultater, er protokolsammensætning. I litteraturen undersøges sikkerhedsmålene for en protokol typisk med protokollen kørende isoleret. Men ofte kører flere protokoller parallelt på den samme enhed eller interagerer på anden måde med hinanden. For eksempel kan en protokol bruges til at etablere en sikker forbindelse mellem to slutpunkter, mens en anden protokol er afhængig af denne sikre forbindelse til at udveksle følsomme oplysninger. Et andet eksempel er flere applikationer, der deler en fælles infrastruktur som f.eks. en identitetsserver. Protokolsammensætning giver betydelige udfordringer for formel verifikation. Sammensatte systemer er meget større og mere komplekse end individuelle komponenter, og derfor er det sværere at verificere dem. Desuden kan komponenter tilføjes eller opdateres, og vi ønsker ikke at verificere hele det sammensatte system fra bunden, så snart der sker en lille ændring. Det er derfor fordelagtigt at opnå resultater for sikker sammensætningen af protokoller på en modulær måde, dvs. vi ønsker at udlede sikkerheden af en sammensat protokol ud fra sikkerheden af dens komponenter. Mens der er en række resultater, der understøtter protokolsammensætning for mål som hemmeligholdelse og autentificering, er det sværere at opnå sammensætning for privathedsegenskaber.

I denne afhandling beskæftiger vi os med automatisering af verificering af sikkerhedsprotokollers privathedsegenskaber og med beviser for typnings- og sammensætningsresultater. Vi bruger (α, β)-privathed, som er en symbolsk tilgang hvis formål er at give en logisk og intuitiv måde at specificere privathedsmål på. Vores resultater understøtter brede klasser af sikkerhedsprotokoller med sædvanlige kryptografiske funktioner, udeterministisk opførsel, forgrening og tilstandsfulde evner. Vores vigtigste bidrag er som følger:

- Design af en beslutningsprocedure for et fragment af (α, β)-privathed.
- Implementering af proceduren i form af et prototypeværktøj.
- Anvendelse af proceduren og værktøjet på casestudier.
- Et typningsresultat for klassen af protokoller, der er modstandsdygtige imod typefejl.
- Et sammensætningsresultat for klassen af sammensættelige protokoller.

Acknowledgements

I am most grateful to my supervisor Sebastian and co-supervisor Luca, for their guidance and support. Working with you has been a privilege and a pleasure. I remember the first time that I was introduced to the concept of formal verification for privacy in security protocols during one of Sebastian's lectures. This PhD project was a great opportunity to delve into the topic.

I am also very thankful to all my colleagues in the Software Systems Engineering section (and former colleagues in Formal Methods). I enjoyed spending time with you, both for discussing research in computer science and during the various social activities we organized outside work hours. I feel fortunate to have done my PhD in such good conditions. Special thanks are due to the people who have helped me with earlier versions of this thesis. Your comments have lead to significant improvements.

For my external research stay, I have to thank Luca and the other members of the Cybersecurity group in the Department of Informatics at King's College London. You have welcomed me and made my three-month stay a very enjoyable experience.

Finally, I would like to acknowledge my friends outside DTU and my family.

Contents

Chapter 1

Introduction

1.1 Motivation

Privacy is relevant for virtually any application handling data. For instance, a person getting ahold of medical records without consent may abuse personally identifiable information such as social security number, date of birth or address. Similar concerns arise for digital health in general, mobile payments, transport with smart cards, electronic voting etc. Therefore, studying privacy is critical, especially considering the increasing digitalization of applications. In order to protect sensitive information, it is crucial to have strong guarantees that distributed systems respect privacy. New digital applications need to be secured and protected against any misuse, such as surveillance, profiling, stalking, or coercion (e.g., a doctor should be able to make prescriptions without pressure from pharmaceutical companies).

1.2 Security protocols

In order to study how distributed systems and applications work, we consider the formal specifications of *security protocols*. They are protocols defining how messages are exchanged between several parties, often relying on cryptographic operations. A well-known example is the *Transport Layer Security* (TLS) protocol, that is commonly used to secure the connection between a client application like a browser and a webserver. TLS is a large protocol that starts by establishing a key between the client and the server; this function of *key exchange* is often realized by individual protocols. There are for instance several variants based on the so-called Diffie-Hellman key exchange.

Designing secure protocols is a hard problem: even for protocols that are designed with privacy in mind, there are often imperfections. For example, recent works show privacy issues in voting protocols (Helios [15, 35]) or contact-tracing applications (GAEN API [19], SwissCovid [73, 58]). Formal verification helps to discover issues in security protocols, and once the issues have been fixed it becomes possible to *prove* that the protocol is secure.

1.3 Privacy

Privacy properties are more complicated than standard goals of secrecy and authentication. For instance, for the privacy-type goal of unlinkability (an outsider cannot tell if two protocol sessions were executed by the same or different participants), the usual symbolic attacker model is not sufficient because it cannot directly model weak secrets like the name of a participant: the intruder may know the name of all participants, and the secret is rather which user has performed a particular action. It is standard to use equivalence notions here: we consider a pair of worlds and verify that the intruder cannot tell which world is the case. For instance, for unlinkability we have one world where some agent runs several protocol sessions and another world where different agents run one session each. However, reasoning about such equivalences is generally more complicated than reasoning about reachability properties (such as secrecy and authentication goals). Moreover, it is typical for verifying privacy in security protocols to define a list of properties (i.e., goals expressed with equivalences) to check. However, this is very technical and one cannot be sure that all relevant properties have been identified. This means that some attacks may still occur despite successful verification.

Instead, we want to reason about the knowledge an attacker has, and what implications they can derive from this knowledge. We use the formalism of (α, β)-privacy, which was introduced as an alternative way to define privacy-type properties in security protocols [63, 47]. (α, β)-privacy considers states that each represent one possible reality, and what the intruder knows about the reality in that state. This knowledge is not only in form of messages as in classic intruder models, but also in form of relations between messages, agents, etc. Together with a notion of what the intruder is allowed to know in a given state, we define a privacy violation if the intruder in any reachable state knows more than allowed. Privacy is then a question of reachability—a safety property—which is often easier to reason about and to specify than classical equivalence notions. First, one does not have to boil the privacy goal down to a distinction between two situations, which is often unnatural for more complicated properties. Second, one specifies goals positively by what the intruder is allowed to know rather than what they are not allowed to know (and thus unable to distinguish). This essentially means that in the worst case one is erring on the safe side, i.e., allowing less than the protocol actually reveals, and thus can be alerted by a counterexample. The expressive power of equivalence notions and of (α, β)-privacy is actually hard to relate in general, due to the different nature of the approaches. However, on concrete examples it seems one can always give reasonable adaptations from one approach to the other [63, 47].

1.4 Automated reasoning

Usually, formally verifying a protocol by hand is tedious and one easily overlooks an error in a corner case. Therefore, it is very useful to have tools that automate this formal verification. The main goal of this thesis is the development of new methods for automated verification of privacy. The idea is to give as input the specification of a security protocol, which includes the information intentionally disclosed. Then a program can check whether there are any attacks. With this tool, we can study existing systems to prove that either they respect privacy or they violate it (by showing an attack). Moreover, this can also be used during the development of applications with a privacy-by-design mindset.

In this thesis, we use a Dolev-Yao model for security protocols, i.e., messages of the protocol are represented with symbolic terms, and we assume that the intruder can control the network and act as a protocol participant but they cannot break cryptography. This symbolic model approach has proved to be a very effective basis for automated verification approaches, e.g., ProVerif [16], Tamarin [60], or CPSA [37].

1.5 Summary of the results

The work presented in this thesis is based on several papers, corresponding roughly to one chapter each.

- "A Decision Procedure for Alpha-Beta Privacy for a Bounded Number of Transitions" [44]: Our most important research question was: how to automate the verification of (α, β)-privacy in security protocols? In this publication, we define a decision procedure for privacy as reachability. In general, the problem is undecidable, and we thus have two restrictions to obtain a decidable problem: we bound the number of steps of honest agents, and we restrict ourselves to a particular class of algebraic theories. At the core of the procedure is a constraint-solving method, similar to previous works on protocol verification in the symbolic model, with adaptations to our formalism targeting (α, β)-privacy goals. The rest of the procedure uses the constraint-solving method as a building block. We define a state transition system as a symbolic execution of transactions specified by the protocol. We also define intruder experiments for the checks that the intruder can perform between messages in their knowledge. In order to support algebraic properties of cryptographic operators, we give an analysis strategy that saturates the intruder knowledge by decrypting messages as far as possible. Note that in our procedure, we never bound the number of steps that the intruder can make. Moreover, we have implemented the decision procedure in an open-source prototype tool and applied it to several protocols described in case studies.

- "A decision procedure and typing result for alpha-beta privacy" [43]: In this article, we extend the work from [44] and obtain a typing result. The new contributions are the introduction of a typed model where the protocol specifies an intended type for every message; the definition of *type-flaw resistant* protocols; and the proof of a typing result for (α, β)-privacy of the form "if there is an attack, then there is a well-typed one." For the proof arguments, we make several adaptations to the procedure verifying (α, β)-privacy, in particular with the introduction of pattern matching and a reformulation of the transitions for analyzing the intruder knowledge (we also prove that these transformations are correct). Our requirements for type-flaw resistance can be checked statically, and we prove that under these requirements, the procedure performs only well-typed instantiations of variables and well-typed intruder experiments. Our typing result is not limited to any bound on the number of transitions. Moreover, we revisit the case study protocols to check whether they satisfy our type-flaw resistance requirements.

- *A Compositionality Result for Alpha-Beta Privacy* [42]: In this paper, we extend the (α, β)-privacy specification language with constructs useful for expressing protocol composition; for instance, one protocol may call another as a subprotocol. Our definition of protocol composition is general and allows for protocols sharing messages. We introduce a notion of roles, which allows for sequencing transactions with particular interleavings. We also add assertions that are checked during the symbolic execution of the protocol: these assertions can be used to express protocol goals other than privacy. Composed protocols may share secrets, and these can be declassified during the protocol execution: the non-leakage of secrets is another protocol goal to verify, besides assertions and regular (α, β)-privacy goals. We define *composable* protocols and prove a compositionality result for (α, β)-privacy of the form "if every component protocol is secure, then the composition is also secure." This result enables modular verification of (α, β)-privacy goals: our approach relies on the specification, for each component in a composed system, of an abstract interface. We then consider each component with only the interface of the others, instead of the entire composed system.

- "Private Authentication with Alpha-Beta-Privacy" [41]: In this publication, we focus on two protocols: BAC [57], typically used with RFID tags like in epassports, and Private Authentication [1], used for communication between agents without revealing to an outsider who is talking to whom. We discuss several variants of the protocols and explain how to use the `noname` tool implementing the decision procedure from [44]. The case studies go into details about various modeling choices and show how we can find attack traces that can easily be explained. The protocol modeling illustrates the benefits of the declarative approach of (α, β)-privacy.

As an additional contribution, we have reworked and refined the material from the papers forming the basis of this thesis. In particular, we have made an effort to present the results in successive chapters that generally build on top of each other, with consistent notation.

1.6 Outline of the thesis

In Chapter 2, we present the notion of (α, β)-privacy in transition systems and define the problem of privacy as reachability. In Chapter 3, we present our decision procedure, which uses symbolic representations to compute the reachable states of a protocol and then verifies, in each state, whether (α, β)-privacy holds. In Chapter 4, we introduce a type system for security protocols and define the class of *type-flaw resistant* protocols, for which we obtain a typing result of the form "if there is an attack, then there is a well-typed one." In Chapter 5, we extend the grammar of protocols with constructs enabling *composition*, such as the ability for one protocol to call a procedure as a subprotocol. We define the class of *composable* protocols, for which we obtain a compositionality result. This allows for modular verification of composed systems. In Chapter 6, we explain how to use our tool `noname`, which implements the decision procedure from Chapter 3, and we describe in details the results of applying the tool to two case study protocols: BAC and Private Authentication. In Chapter 7, we conclude by summarizing our results and discussing future work. The appendix contains all the proofs of correctness and additional details for models of protocols.

Chapter 2

Preliminaries

This chapter is based on [44, 43].

Mödersheim and Viganò [63] introduce (α, β)-privacy as a reachability problem in a state transition system, where each state contains two formulas α and β. Intuitively, α represents what the intruder may know (e.g., the result of an election) and β what the intruder has observed (e.g., the encrypted votes). Then, a state (α, β) violates privacy iff some model of α can be excluded by the intruder knowing β, i.e., the intruder in that state can rule out more than allowed, e.g., in a voting protocol the intruder finds out that two of the voters have voted the same, even though this does not follow from the election result. The transition system violates (α, β)-privacy iff some reachable state does.

2.1 (α, β)-privacy for a state

[63] focuses on how to define (α, β) pairs for a fixed state, and describes a state-transition relation only briefly by an example. Let us also start with a fixed state. The formulas α and β are in *Herbrand logic* [55], a variant of First-Order Logic (FOL), with the difference that the universe is the quotient algebra of the *Herbrand universe* (the set of all terms that can be built with the function symbols) modulo a congruence relation $\approx$. This congruence specifies algebraic properties of cryptographic operators. For concreteness, we use the congruence defined in Fig. 2.1; the class of properties supported by our result is in Definition 3.4.1. The quotient algebra consists of the $\approx$-equivalence classes of terms.

Given an alphabet Σ, a Σ-*interpretation* $\mathcal{I}$ interprets variable and relation symbols as usual in the Herbrand universe induced by Σ and $\approx$; the interpretation of the function symbols is determined by the Herbrand universe. We have a model relation $\models_\Sigma$ as expected. By construction, $\mathcal{I} \models_\Sigma s \doteq t$ iff $\mathcal{I}(s) \approx \mathcal{I}(t)$. We say that ϕ *entails* ψ, and write $\phi \models_\Sigma \psi$, when every Σ-interpretation that is a model of ϕ is also a model of ψ. We write $\phi \equiv_\Sigma \psi$ when $\phi \models_\Sigma \psi$ and $\psi \models_\Sigma \phi$.

We now fix the alphabet Σ that contains all symbols we use, namely cryptographic functions, a countable set of constants representing agents, nonces and so on, and some relation symbols. We also have the set of variable symbols $\mathcal{V}$. Each protocol specification will fix a sub-alphabet $\Sigma_0 \subset \Sigma$ that we call the *payload symbols*; we call $\Sigma \setminus \Sigma_0$ the *technical symbols*. All α formulas use only symbols in Σ_0 and variables. In the rest of the thesis, we often omit the alphabet and just write $\models$ to mean $\models_{\Sigma_0}$, and $\equiv$ to mean $\equiv_{\Sigma_0}$.

The main idea of (α, β)-privacy is that we distinguish between the actual privacy goal (e.g., an unlinkability goal talking only about agents) and the means to achieve it (e.g., the cryptographic messages exchanged).

$$\begin{array}{ll}
\mathsf{dcrypt}(s_1, s_2) \approx t & \text{if } s_1 \approx \mathsf{inv}(k) \text{ and } s_2 \approx \mathsf{crypt}(k, t, r) \\
\mathsf{dscrypt}(k, s) \approx t & \text{if } s \approx \mathsf{scrypt}(k, t, r) \\
\mathsf{open}(k, s) \approx t & \text{if } s \approx \mathsf{sign}(\mathsf{inv}(k), t) \\
\mathsf{pubk}(s) \approx k & \text{if } s \approx \mathsf{inv}(k) \\
\mathsf{proj}_1(s) \approx t_1 & \text{if } s \approx \mathsf{pair}(t_1, t_2) \\
\mathsf{proj}_2(s) \approx t_2 & \text{if } s \approx \mathsf{pair}(t_1, t_2) \\
\text{and } \ldots \approx \mathbf{ff} & \text{otherwise}
\end{array}$$

Figure 2.1: The congruence used for examples in this thesis: crypt and dcrypt are asymmetric encryption and decryption, scrypt and dscrypt are symmetric encryption and decryption, sign and open are signing and verification/opening, pair is a transparent function and the proj_i are the projections, inv gives the private key corresponding to a public key, and pubk gives the public key from a private key. Here k, t, r and the t_i are variables standing for arbitrary messages. When the conditions are not met, the functions give $\mathbf{ff}$, which is a constant indicating failure of decryption or parsing. If crypt and scrypt are used as binary functions, we consider their deterministic variants where the random factor r has been fixed and is omitted for simplicity.

Definition 2.1.1 (Adapted from [63]). *Given two formulas α over Σ_0 and β over Σ with $fv(\alpha) \subseteq fv(\beta)$, where fv denotes the free variables, we say that* (α, β)-privacy holds *iff for every Σ_0-interpretation $\mathcal{I} \models_{\Sigma_0} \alpha$ there exists a Σ-interpretation $\mathcal{I}' \models_{\Sigma} \beta$ such that $\mathcal{I}$ and $\mathcal{I}'$ agree on the variables in $fv(\alpha)$ and on the relation symbols in Σ_0.*

We call the formula α the *payload*, defining the privacy goal. For example, for unlinkability in an RFID-tag protocol, we may have a fixed set $\{t_1, t_2, t_3\}$ of tags and in a concrete state, the intruder has observed that two tags have run a session. Then α in that state may be $x_1, x_2 \in \{t_1, t_2, t_3\}$, meaning that the intruder is only allowed to know that both x_1 and x_2 are indeed tags, but not, for instance, whether $x_1 \doteq x_2$. In our approach, the formulas α that can occur fall into a fragment where we can always compute a finite representation of all models, in particular the variables like the x_i in the example will always be from a fixed finite domain.

For the formula β, we employ the concept of frames: a *frame* has the form $F = l_1 \mapsto t_1. \cdots .l_n \mapsto t_n$, where the l_i are distinguished variables called *labels* and the t_i are *messages* (terms that do not contain labels). This represents that the intruder has observed (or initially knows) messages $t_1, \ldots, t_n$ and we give each message a unique label. We call the set $\{l_1, \ldots, l_n\}$ the *domain* of F. A frame can be used as a substitution, mapping labels to messages.

To describe intruder deductions, we define a subset Σ_{pub} of the function symbols to be *public*: they represent operations the intruder can perform on known messages. For instance, all symbols used in Fig. 2.1 are public except for inv, since getting the private key is not an operation that everyone can do themselves.[1] A *recipe* (in the context of a frame F) is any term that consists of only labels (in the domain of F) and public function symbols, so it represents a computation that the intruder can perform on F. We write $F(r)$ for the message generated by the recipe r with the frame F: all labels in the recipe are replaced with the respective messages from the frame.

[1]The use of inv is just one possible model, and one could choose to model private keys differently, e.g., with public functions for key pair generation and secret seeds. In this thesis we use inv as it makes our examples simpler.

Two frames F_1 and F_2 with the same domain are *statically equivalent*, written $F_1 \sim F_2$, iff for every pair (r_1, r_2) of recipes, we have $F_1(r_1) \approx F_1(r_2) \Leftrightarrow F_2(r_1) \approx F_2(r_2)$. This means that the intruder cannot distinguish F_1 and F_2, since any experiment they can make (i.e., comparing the outcome of two computations r_1, r_2) either gives in both frames the same result or in both frames not.

Static equivalence can be encoded in Herbrand logic [63] and so for instance we can have as part of β formulas like $\mathit{concr} \sim \mathit{struct}$, where struct is a frame with variables and $\mathit{concr} = \mathcal{I}(\mathit{struct})$ for some model $\mathcal{I} \models \alpha$ is the concrete frame of intruder observations. As an example, let $\alpha = x_1, x_2 \in \{0, 1\}$, $\mathit{struct} = l_1 \mapsto \mathsf{h}(k, x_1).l_2 \mapsto \mathsf{h}(k, x_2)$ and $\mathit{concr} = l_1 \mapsto \mathsf{h}(k, 0).l_2 \mapsto \mathsf{h}(k, 1)$. Observe that α has four models, but in the two models where $\mathcal{I}(x_1) = \mathcal{I}(x_2)$ we have $\mathcal{I} \not\models_\Sigma \mathit{concr} \sim \mathit{struct}$, because for these models l_1 and l_2 give the same message in struct but different messages in concr. Since in this example $\mathit{concr} \sim \mathit{struct}$ is part of β, β allows the intruder to rule out two models of α, thus (α, β)-privacy does not hold.

2.2 (α, β)-privacy for a transition system

So far we have been talking about only a single (α, β) pair, i.e., a single state of a larger transition system. Gondron, Mödersheim, and Viganò [47] define a language for specifying transition systems where the reachable states and their (α, β) pairs are defined by executing atomic transactions. We present their formalization with some minor adaptations to ease our further development.

2.2.1 Specification

We distinguish two sorts of variables: the *privacy variables* $\mathcal{V}_{\mathit{privacy}}$, which are denoted with lower-case letters like x and are all introduced in the form $x \in D$ for a finite domain D of public constants from Σ_0, and the *intruder variables* $\mathcal{V}_{\mathit{intruder}}$, which are denoted with upper-case letters like X for messages received and cell reads in a transaction.

We also distinguish *destructor* and *constructor* function symbols. In Fig. 2.1 we have that dcrypt, $\mathsf{dscrypt}$, open, pubk, proj_1 and proj_2 are destructors whereas the rest are constructors. Moreover, we call pair and inv transparent functions, because one can get all their arguments without any key (but recall that inv is not a public function).

Much of the processes defined below follows standard process calculus constructs. The special constructs of (α, β)-privacy are the non-deterministic choice of privacy variables and release. Choice comes in two flavors: $\star$ if the choice is privacy relevant (as in Example 2.2.1 just after the definition), and $\diamond$ if not. The release is used to declare that a certain fact ϕ may now be known to the intruder; we discuss this construct and what formulas can be released a bit later.

Definition 2.2.1 (Protocol specification). *A* protocol specification *consists of*

- *a number of* transaction processes P_i*, where the* P_i *are left processes according to the syntax below, describing the atomic transactions that participants in the protocol can execute;*
- *a number of* memory cells*, e.g.,* $\mathsf{cell}[\cdot]$*, together with a ground context* $C[\cdot]$ *for each memory cell defining the initial value of the memory, so that initially* $\mathsf{cell}[t] = C[t]$*; and*
- *a* Σ_0*-formula* γ_0 *that fixes the interpretation of the relation symbols.*

We define left, center, *and* right processes *as follows:*

P_l			*Left process*
	$::=$	$\mathsf{mode}\ x \in D.P_l$	*Non-deterministic choice*
	$\mid$	$\mathsf{rcv}(X).P_l$	*Receive*
	$\mid$	P_c	*Center process*
P_c			*Center process*
	$::=$	$\mathsf{try}\ X := d(t,t)\ \mathsf{in}\ P_c\ \mathsf{catch}\ P_c$	*Destructor application*
	$\mid$	$X := \mathsf{cell}[t].P_c$	*Cell read*
	$\mid$	$\mathsf{if}\ \phi\ \mathsf{then}\ P_c\ \mathsf{else}\ P_c$	*Conditional statement*
	$\mid$	$\nu X_1, \dots, X_k.P_r$	*Fresh constants*
P_r			*Right Process*
	$::=$	$\mathsf{snd}(t).P_r$	*Send*
	$\mid$	$\mathsf{cell}[t] := t.P_r$	*Cell write*
	$\mid$	$\star\ \phi.P_r$	*Release*
	$\mid$	0	*Terminate (nil process)*

where mode *is either* $\star$ *or* $\diamond$*,* ϕ *is a quantifier-free Herbrand formula, and* d *is a destructor. Destructors are not allowed to occur elsewhere in terms. For simplicity, we have denoted destructors as binary functions, but we may similarly use unary destructors (like* proj_i *and* pubk *in the example).*

We require that a transaction P *is a* closed *left process, i.e.,* $fv(P) = \emptyset$*—we define the* free variables $fv(P)$ *of a process* P *as expected, where the non-deterministic choices, receives, cell reads and fresh constants are binding. Moreover, for destructor applications:*

$$fv(\mathsf{try}\ X := d(k,t)\ \mathsf{in}\ P_1\ \mathsf{catch}\ P_2) = fv(d(k,t)) \cup (fv(P_1) \setminus \{X\}) \cup fv(P_2)\ .$$

Finally, a bound variable cannot be instantiated a second time.

Example 2.2.1 (Running example). Let us consider the following transaction where a server non-deterministically chooses an agent x and a yes/no-decision y, receives a message, tries to decrypt it with their own private key and then sends the decision encrypted with the public key of x:

```
⋆ $x \in \mathsf{Agent}$. ⋆ $y \in \{\mathsf{yes}, \mathsf{no}\}$.
rcv($M$).
try $N := \mathsf{dcrypt}(\mathsf{inv}(\mathsf{pk}(\mathsf{s})), M)$ in
  if $y \doteq \mathsf{yes}$ then
      $\nu R.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, N), R)).0$
  else $\nu R.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{no}, R)).0$
catch 0
```

Here the $\star$ means that the choice of x and y is privacy relevant and the intruder may (at least for now) only learn that $x \in \mathsf{Agent}$ and $y \in \{\mathsf{yes}, \mathsf{no}\}$. The outgoing message has a different form depending on y: in the positive case the server also includes the content N of the encrypted message M they received (and if the message is not of the right format, then the transaction simply terminates); in either case the encryption is randomized with a fresh R. We may omit R if we want to model non-randomized encryption. pk is a public function (modeling a fixed public-key infrastructure known to everybody). $\lhd$

Observe that privacy variables are introduced only by non-deterministic choices $\mathsf{mode}\ x \in D$. If the mode is $\star$, the transaction augments α by $x \in D$, thus specifying that the intruder may not know more about x unless we also explicitly release some information about x. If the mode is $\diamond$, the transaction augments β by $x \in D$. In this case it is *not* in itself a violation of privacy if the intruder learns more about x, but it may lead to a privacy violation if this allows for finding out more about the variables in α. This is useful if one wants to keep the privacy specification independent of some rather technical secret. In our model, the intruder knows which transaction is executed, but in general does not know which branch is taken. Using, for example, $\diamond\ z \in \{1, 2\}.\mathsf{if}\ z \doteq 1\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$, one can avoid to make it (a priori) visible to the intruder whether P_1 or P_2 is executed, but if the intruder finds out which one, it is not a privacy violation in itself.

2.2.2 Semantics

The semantics follows [47], with small adaptations. It is defined as a state transition system where each transition corresponds to the execution of one transaction. Thus, transactions are atomic: they cannot run concurrently with another transaction. In particular, when reading from and writing to memory cells, no race conditions can occur and we thus do not need locking mechanisms. A transaction thus consists in receiving input, checking this input (possibly reading from memory), then making a decision (possibly updating the memory), and finally sending an output and releasing information.

The atomicity of transactions has an advantage: we can easily formalize how the intruder can reason about what is happening. In particular, we assume that the intruder at each point knows which transaction is executed and what process a transaction contains. What the intruder does not know in general are the concrete values of the variables and the truth values of conditions, and thus in which branch of an if-then-else or try-catch we are. However, the intruder can always contrast this knowledge with the observations about incoming and outgoing messages: if an observed sent message does not fit with one branch of the transaction, then the intruder knows that branch was not taken, and thus they also learn something about the truth value of the corresponding conditions. In other cases, the intruder may know what is in a received message and thus know the truth value of some condition. The intruder thus performs a symbolic execution of the transaction, leaving open what they do not know, keeping a list of possibilities, and in fact the semantics of transactions formally models this symbolic execution by the intruder.

Let a *possibility* be a tuple $(P, \phi, \mathit{struct}, \delta)$, where P is the transaction being executed, ϕ is the conditions under which this possibility was reached, *struct* is a frame that we call the structural knowledge about the messages in this possibility and δ is a sequence of memory updates. A *state* is a tuple $(\alpha, \beta_0, \gamma, \mathcal{P})$ where α, β_0 and γ are Σ_0-formulas, and $\mathcal{P}$ is a non-empty finite set of possibilities $\mathcal{P} = \{(P_1, \phi_1, \mathit{struct}_1, \delta_1), \ldots, (P_n, \phi_n, \mathit{struct}_n, \delta_n)\}$, where exactly one of the possibilities is actually the case in the real execution (but the intruder does not know which one, in general). The formula γ in a state is the ground *truth* that interprets every privacy variable.

In this thesis, we consider only well-formed states, where a *state* is *well-formed* iff all struct_i have the same domain, $\gamma \models \gamma_0$ describes a unique model of $\alpha \wedge \beta_0$ and the ϕ_i both are mutually exclusive, i.e., $\models \neg(\phi_j \wedge \phi_k)$, for $j \neq k$, and cover all models, i.e., $\alpha \wedge \beta_0 \models \bigvee_{i=1}^{n} \phi_i$. Recall that γ_0 is a formula specified by the protocol that fixes the interpretation of all relation symbols, it is a requirement for well-formedness that $\gamma \models \gamma_0$. Well-formedness of states will be preserved by the state transitions.

In a well-formed state, the truth γ identifies a unique possibility $(P_j, \phi_j, \mathit{struct}_j, \delta_j)$, since γ models the disjunction of the ϕ_j and the ϕ_j are mutually exclusive. This unique possibility is the one that really corresponds to the concrete execution. We define the concrete frame *concr* as the instantiation of the $struct_j$ from the possibility such that $\gamma \models \phi_j$.

Definition 2.2.2 (Multi message-analysis problem)**.** *Given a well-formed state* $S = (\alpha, \beta_0, \gamma, \mathcal{P})$*, let* $concr = \gamma(struct_j)$ *where* $(P_j, \phi_j, struct_j, \delta_j)$ *is the unique possibility in* $\mathcal{P}$ *such that* $\gamma \models \phi_j$*. Define*

$$\beta(S) = \alpha \wedge \beta_0 \wedge \bigvee_{i=1}^{n} \phi_i \wedge concr \sim struct_i \ .$$

We say that S satisfies privacy *iff* $(\alpha, \beta(S))$*-privacy holds.*

The semantics models that the intruder can symbolically execute transactions, but at branchings they do not know in general whether the condition is true; the possibilities reflect the different possible truth values of the conditions and what would have happened in that case. The unique possibility identified by γ is the one that really happened. In the rest of the thesis, we refer to this true possibility as the *underlined* possibility. Contrasting *concr* and each $struct_i$, the intruder may be able to exclude some possibility (because *concr* and $struct_i$ are distinguishable) and thus learn about the truth value of conditions ϕ_i. Vice-versa, what the intruder knows about variables may allow them to exclude some possibilities.

To model the symbolic execution of a transaction P, we start in a state where all possibilities contain that process P. The semantics defines an evaluation relation $\rightarrow$ on states that works off the processes in each possibility until all processes are 0. We call such a state *finished.* The branching of $\rightarrow$ represents the non-deterministic choices of the process as well as choices of messages by the intruder.

To give a gentle introduction to (α, β)-privacy in transition systems, we present the symbolic execution at hand of the running example from Example 2.2.1. The complete definition of the rules is in Table 2.1, where the changes to the state are highlighted with the color red. Notation: $\delta_{|\mathsf{cell}}$ is the largest subsequence of δ that only contains memory updates on cell and $\uplus$ is the disjoint union of sets.

As a starting point for the symbolic execution, we use the singleton set of possibilities $\{\underline{(P, \mathsf{true}, [], [])}\}$ where P is the process from the running example; [] denotes the empty frame and empty memory. Let α and β_0 be true, and γ be γ_0. This state means that initially the intruder knows nothing; optionally, one could give them the knowledge of a private key by having some messages instead of the empty frame. We list in Table 2.2 the successive states that are reached when executing P and we explain below the transitions.

Non-deterministic choice

The first steps in the running example are two non-deterministic choices of privacy variables. In general, there are several successor states, one for each possible choice of the variables. Let us follow the case where $x = \mathsf{a}$ and $y = \mathsf{yes}$; this is added to γ, and we add to α that $x \in \mathsf{Agent}$ and $y \in \{\mathsf{yes}, \mathsf{no}\}$. We reach states 2 then 3. Note that x and y are not replaced in the remaining process—this is a symbolic execution by the intruder. Also note that the general rule assumes that all possibilities start with the same mode; this is ensured since this choice can only occur in the left part of the transaction, before any branching on conditions and tries can occur.

Receive

The next step is now $\mathsf{rcv}(M)$. Again the construction ensures that every process in the possibilities starts with a receive step (with the same variable). Here, the intruder can

Table 2.1: Semantics of the execution for states

Choice	$(\alpha, \beta_0, \gamma, \{(\mathsf{mode}\ x \in D.P_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \ldots, n\}\})$ $\rightarrow (\alpha', \beta'_0, \gamma \wedge x \doteq c, \{(P_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \ldots, n\}\})$ for every $c \in D$, where $\alpha' = \alpha \wedge x \in D$ and $\beta'_0 = \beta_0$ if $\mathsf{mode} = \star$, $\alpha' = \alpha$ and $\beta'_0 = \beta_0 \wedge x \in D$ if $\mathsf{mode} = \diamond$
Receive	$(\alpha, \beta_0, \gamma, \{(\mathsf{rcv}(X).P_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \ldots, n\}\})$ $\rightarrow (\alpha, \beta_0, \gamma, \{(P_i[X \mapsto \mathit{struct}_i(r)], \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \ldots, n\}\})$ for every recipe r over the domain of the struct_i
Cell read	$(\alpha, \beta_0, \gamma, \{(X := \mathsf{cell}[s].P, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P})$ $\rightarrow (\alpha, \beta_0, \gamma, \{(P', \phi, \mathit{struct}, \delta)\} \cup \mathcal{P})$ where $\delta_{\vert\mathsf{cell}} = \mathsf{cell}[s_1] := t_1. \cdots .\mathsf{cell}[s_k] := t_k$, the ground context for initial value of cell is $C[\cdot]$ and $P' = \mathsf{if}\ s \doteq s_1\ \mathsf{then}\ P[X \mapsto t_1]\ \mathsf{else}\ \ldots$ $\mathsf{if}\ s \doteq s_k\ \mathsf{then}\ P[X \mapsto t_k]\ \mathsf{else}\ P[X \mapsto C[s]]$
Cell write	$(\alpha, \beta_0, \gamma, \{(\mathsf{cell}[s] := t.P, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P})$ $\rightarrow (\alpha, \beta_0, \gamma, \{(P, \phi, \mathit{struct}, \mathsf{cell}[s] := t.\delta)\} \cup \mathcal{P})$
Conditional	$(\alpha, \beta_0, \gamma, \{(\mathsf{if}\ \psi\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P})$ $\rightarrow (\alpha, \beta_0, \gamma, \{(P_1, \phi \wedge \psi, \mathit{struct}, \delta), (P_2, \phi \wedge \neg\psi, \mathit{struct}, \delta)\} \cup \mathcal{P})$
Release	$(\alpha, \beta_0, \gamma, \{(\star\ \psi.P, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P}) \rightarrow (\alpha', \beta_0, \gamma, \{(P, \phi, \mathit{struct}, \delta)\} \cup \mathcal{P})$ where $\alpha' = \alpha \wedge \psi$ if $\gamma \models \phi$, and $\alpha' = \alpha$ otherwise
Eliminate	$S = (\alpha, \beta_0, \gamma, \{(P, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P}) \rightarrow (\alpha, \beta_0, \gamma, \mathcal{P})$ if $\beta(S) \models_\Sigma \neg\phi$
Send	$(\alpha, \beta_0, \gamma, \{(\mathsf{snd}(t_i).P_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \ldots, n\}\} \uplus \mathcal{P})$ $\rightarrow (\alpha, \beta_0 \wedge \bigvee_{i=1}^{n} \phi_i, \gamma, \{(P_i, \phi_i, \mathit{struct}_i.l \mapsto t_i, \delta_i) \mid i \in \{1, \ldots, n\}\})$ if $\gamma \models \bigvee_{i=1}^{n} \phi_i$ and every process in $\mathcal{P}$ is 0, where l is a fresh label
Terminate	$(\alpha, \beta_0, \gamma, \{(0, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \ldots, n\}\} \uplus \mathcal{P})$ $\rightarrow (\alpha, \beta_0 \wedge \bigvee_{i=1}^{n} \phi_i, \gamma, \{(0, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \ldots, n\}\})$ if $\gamma \models \bigvee_{i=1}^{n} \phi_i$, $\mathcal{P} \neq \emptyset$ and every process in $\mathcal{P}$ starts with $\mathsf{snd}(\cdot)$

choose an arbitrary recipe r (over the domain of the struct_i) for the message that should be received as M. In fact, in general, we have here infinitely many possible r and thus infinitely many successors. (Our decision procedure uses a constraint-based approach to handle this in a finite way.) Note that the message that is being received depends on the possibility: it is $\mathit{struct}_i(r)$ in the ith possibility, i.e., whatever the recipe r yields in the respective intruder knowledge struct_i. As the intruder knowledge at this point is empty in the example, r can only be a recipe built from public constants and functions; r cannot contain any labels since it is applied to the empty frame, and thus the message generated is identical to the recipe. Let us consider $r = \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{a}, \mathsf{h}(\mathsf{a}))$, which then replaces M in the process. This leads to state 4.

Cell read and write

The running example does not use memory cells, and we describe briefly the rules here. When reading from the memory, we add conditional statements to substitute in the process the appropriate value from the memory (that depends on the argument term). When writing to the memory, the update is simply prepended to the sequence of memory updates δ.

Table 2.2: States of the running example

State	α	β_0	γ	$\mathcal{P}$
1	true	true	γ_0	$\{\underline{(\star\ x \in \mathsf{Agent}\ldots, \mathsf{true}, [], [])}\}$
2	$x \in \mathsf{Agent}$	true	$\gamma_1 \wedge x \doteq \mathsf{a}$	$\{\underline{(\star\ y \in \{\mathsf{yes}, \mathsf{no}\}\ldots, \mathsf{true}, [], [])}\}$
3	$\alpha_2 \wedge y \in \{\mathsf{yes}, \mathsf{no}\}$	true	$\gamma_2 \wedge y \doteq \mathsf{yes}$	$\{\underline{(\mathsf{rcv}(M)\ldots, \mathsf{true}, [], [])}\}$
4	α_3	true	γ_3	$\{\underline{(\mathsf{try}\ N := \ldots, \mathsf{true}, [], [])}\}$
5	α_3	true	γ_3	$\{(0, \mathsf{false}, [], []), \underline{(\mathsf{if}\ y \doteq \mathsf{yes}\ldots, \mathsf{true}, [], [])}\}$
6	α_3	true	γ_3	$\{\underline{(\mathsf{if}\ y \doteq \mathsf{yes}\ldots, \mathsf{true}, [], [])}\}$
7	α_3	true	γ_3	$\{\underline{(\mathsf{snd}(\ldots\mathsf{pair}(\mathsf{yes}, \mathsf{a})\ldots).0, y \doteq \mathsf{yes}, [], [])},$ $(\mathsf{snd}(\ldots\mathsf{no}\ldots).0, y \not\doteq \mathsf{yes}, [], [])\}$
8	α_3	$y \doteq \mathsf{yes}$ $\vee y \not\doteq \mathsf{yes}$	γ_3	$\{\underline{(0, y \doteq \mathsf{yes}, l \mapsto \ldots\mathsf{pair}(\mathsf{yes}, \mathsf{a})\ldots, [])},$ $(0, y \not\doteq \mathsf{yes}, l \mapsto \ldots\mathsf{no}\ldots, [])\}$

Conditional statement

The process is now $\mathsf{try}\ N := \ldots\ \mathsf{in}\ \mathsf{if}\ldots\ \mathsf{catch}\ 0$. For the sake of this semantics, we can just consider $\mathsf{try}\ X := t\ \mathsf{in}\ P_1\ \mathsf{catch}\ P_2$ as syntactic sugar for $\mathsf{if}\ t \doteq \mathbb{ff}\ \mathsf{then}\ P_2\ \mathsf{else}\ P_1[X \mapsto t]$. (For the decision procedure it is important that destructors only occur in this try-catch form, however.) We have to evaluate the condition $\mathsf{dcrypt}(\mathsf{inv}(\mathsf{pk}(\mathsf{s})), \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{a}, \mathsf{h}(\mathsf{a}))) \doteq \mathbb{ff}$, which we can simplify to false, i.e., the intruder knows that the received message will decrypt correctly. We thus have in state 5 two possibilities of which the second is underlined and where N has been substituted with the content of the encryption, i.e., the constant a. The **Eliminate** rule allows removing possibilities with the condition false, so we reach state 6. The underlined possibility is what really happened (which is here obvious). We apply a second time the **Conditional** rule, again splitting into two possibilities and leading to state 7. The first possibility is what really happens (as stated by γ) and is thus underlined, and here the intruder does not know which one is the case.

Fresh constants

The ν operator can be implemented by replacing the placeholder with a fresh non-public constant, say r_1. We can in fact do this as a preparation before executing the transaction.

Send

When all the rules for the other constructs have been applied as far as possible, each of the remaining processes must be either a send or 0. If γ models the condition ϕ_i for one of the possibilities that is sending, then it means that in the concrete execution a message is indeed sent. (The counterpart is the transition for **Terminate**, applicable if γ models instead a possibility that is terminating and not sending.) The intruder observes that a message is sent, and this rules out all possibilities where the remaining process is 0. For all others, each $struct_i$ is augmented by the message sent in the respective possibility. Moreover, β_0 is updated with the disjunction of the ϕ_i that are consistent with the intruder observations (i.e., we only keep the possibilities that were sending). In our example, we reach state 8, where $concr(l) = \mathsf{crypt}(\mathsf{pk}(\mathsf{a}), \mathsf{pair}(\mathsf{yes}, \mathsf{a}), \mathsf{r}_1)$. We have reached a finished state, and the intruder has thus completed the symbolic execution of this transaction.

Example 2.2.2. Let us point out a few more interesting features of our running example. At the finished state, without further knowledge, the intruder is unable to tell which of the two possibilities is the case. This would be different if the encryption were not randomized: suppose we drop the third argument of crypt. Then the intruder could now construct $\mathsf{crypt}(\mathsf{pk}(x'), \mathsf{no})$ for each value $x' \in \mathsf{Agent}$ and compare the result with the observed message. Since this does not succeed in any case, the intruder learns that the second possibility is excluded, thus $y \doteq \mathsf{yes}$, violating (α, β)-privacy. Even worse, if we look at the state where the non-deterministic choice was $y = \mathsf{no}$, the intruder would find out x because exactly one of the guesses succeeds.

Reverting to randomized encryption, suppose that there had been an earlier transaction where the intruder observed $l' \mapsto \mathsf{crypt}(\mathsf{pk}(z), \mathsf{no}, r_2)$ for some privacy variable $z \in \mathsf{Agent}$. If the intruder uses this as input for the next transaction, then the decryption works iff $z \doteq \mathsf{s}$. Thus, we have a third possibility at the final sending step, namely $(0, z \not\doteq \mathsf{s}, l' \mapsto \mathsf{crypt}(\mathsf{pk}(z), \mathsf{no}, r_2), [])$. Then from the fact that a message was sent, the intruder can rule out this third possibility and thus deduce that $z \doteq \mathsf{s}$, again violating (α, β)-privacy. $\triangleleft$

Release

This construct is used to declare information that the intruder is now allowed to learn, for instance in some cases we may let the intruder learn the true value of a privacy variable. In our running example, we did not use the release, because so far we have not considered whether the intruder is one of the agents from which x is chosen. Let us now consider that there is $\mathsf{i} \in \mathsf{Agent}$ where i is the name of a dishonest agent representing the intruder, and we have the private key $\mathsf{inv}(\mathsf{pk}(\mathsf{i}))$ as part of the initial knowledge. In case the server is replying to the intruder, then the intruder can decrypt the message and observe what was the decision. Thus they would learn both the value of x (i.e., the agent was the intruder) and y (i.e., they know the server's decision), which would violate (α, β)-privacy. We could thus add a release if x is the intruder to allow this deduction.

We consider it a specification error if when applying the **Release** rule, the formula ϕ released contains symbols which are in $\Sigma \setminus \Sigma_0$ and variables not in $fv(\alpha)$. Thus, the specification can use symbols from the technical level in a release *as long as* the evaluated terms use only symbols in Σ_0 and $fv(\alpha)$ (i.e., the payload level) when executing the protocol. This means that releasing technical information in the payload is not allowed. Additionally for our decision procedure below, the same requirement applies to a relational formula $R(t_1, \dots, t_n)$ in the symbolic execution of conditional statements. This kind of specification error can be detected during the symbolic execution and means that insufficient checks are made over the terms before the release or conditional statement.

2.3 Privacy as reachability

We have defined in Table 2.1 the relation $\rightarrow$ that works off the steps of a transaction, modeling an intruder's symbolic execution of a transaction P. We now define a transition relation $\longrightarrow$ on finished states such that $S \longrightarrow S'$ iff there is a transaction P such that $start(P, S) \rightarrow^* S'$, where $start(P, S)$ denotes replacing the 0-process in every possibility of S with process P. Then one transition with relation $\longrightarrow$ means executing exactly one transaction. Let the initial state be $S_0 = (\mathsf{true}, \mathsf{true}, \gamma_0, \{\underline{(0, \mathsf{true}, [], [])}\})$.

Definition 2.3.1 (The reachability problem)**.** *Let $k \geq 0$. A protocol specification* satisfies privacy until bound *k iff (α, β)-privacy holds for every S and $i \in \{0, \dots, k\}$ such that $S_0 \longrightarrow^i S$. A protocol specification* satisfies privacy *iff for every $k \geq 0$, it satisfies privacy until bound k.*

The first contribution of the present thesis is a procedure to solve the reachability problem given a bound k, i.e., whether a violation is reachable in at most k transitions, under the restriction of the algebraic properties to constructor/destructor theories of Definition 3.4.1.

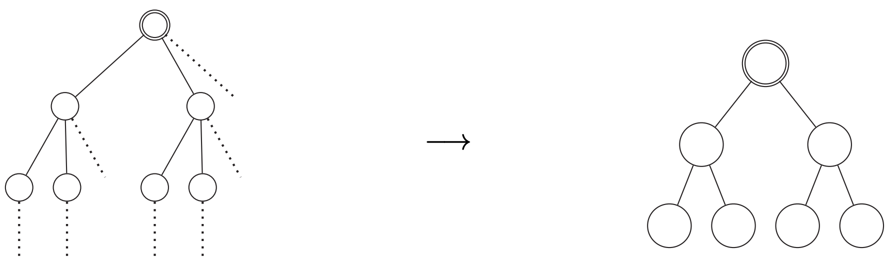
Chapter 3

Decision procedure

This chapter is based on [44, 43].

(α, β)-privacy shifts the problem from a notion of equivalence (that is a challenge for automation) to a simple reachability problem where however the privacy check for each reached state is more involved. The previous work [40] was a first step towards automating verification of (α, β)-privacy, with several restrictions applied to make matters simple. That work considers a solution to checking a given state in (α, β)-privacy, however it is only applicable to specifications without conditional branching and it is based on an exploration of all concrete messages that the intruder can send, which are infinitely many unless one bounds the intruder.

Our first contribution in this thesis is a decision procedure for the full notion of transaction processes defined in Chapter 2, which is based on [47], for constructor/destructor theories [16, 17, 25, 4, 27]. This notion in fact entails that the intruder performs a symbolic execution of the transaction that in general yields several possibilities (due to conditional branching if the intruder does not know the truth value of the condition) and the intruder can then contrast this with all observations and experiments (constructing different messages and comparing them) to potentially rule out some possibilities. The core of our work is in a procedure to model this intruder analysis without bounding the number of steps that the intruder can make in this process. To that end, we use a popular constraint-based technique to represent the intruder symbolically, i.e., without exploring infinite sets of possibilities, which we call here the *lazy intruder*. In fact, we use several layers of symbolic representation to make the approach feasible.



Infinite branching: intruder choices of recipes
Infinite depth: another transaction can always be executed

Finite branching: constraint solving (lazy intruder)
Finite depth: bound on the number of transactions executed

Figure 3.1

Fig. 3.1 illustrates the objective of this chapter. On the left, we have a tree generated by the semantics of Table 2.1 (which we refer to as the *ground model*): each node is a reachable state, and from a single initial state we have in general infinitely many reachable states. On the right, we have a tree that is generated by the semantics defined later in this chapter in Table 3.2 (which we refer to as the *symbolic model*): each node is a symbolic representation of several states.

The tree generated by the semantics in Table 2.1 is both infinitely branching, because there are infinitely many different messages the intruder can construct and send to honest agents, and has infinite depth because there is no bound on how many steps can be executed. If we however bound the number of steps of honest agents, then the problem becomes decidable (for the considered class of algebraic theories allowed by Definition 3.4.1). In fact, we give in this chapter a decision procedure that considers a semantics with symbolic states (Table 3.2) where each node has only finitely many successors and thus for a bounded depth the entire tree can be explored. We show that this symbolic semantics correctly represents the original tree of reachable states.

Our decision procedure tells us whether from a given state we can reach a state that violates privacy for a fixed bound on the number of transitions. Our procedure is limited to such a bound on transitions, corresponding to the restriction to a bounded number of sessions in many approaches [69]. This is similar to the bounds needed in tools like APTE [22], AKiSs [21], SPEC [70, 71] and DeepSec [25]. In fact, this chapter draws from the techniques used in these approaches, such as the symbolic representation of the intruder, a notion of an analyzed intruder knowledge, and methods for deciding the equivalence of frames. There are, however, several basic differences and generalizations. In particular, we use a symbolic handling of privacy variables (that in the equivalence-based approaches are simply one binary choice) and this is linked to logical formulas about relations between elements of the considered universe. In fact, in the prototype implementation of our decision procedure that we provide as a further contribution, we employ the SMT solver cvc5 [12] to handle these logical evaluations. Moreover, we have multiple frames with constraints for the different possibilities resulting from conditional branching and we analyze if the intruder can rule out any possibilities in any instance.

In contrast, the tools ProVerif [16] and Tamarin [60] do handle unbounded sessions but require the restriction to so-called *diff-equivalence* [36, 27], which drastically limits the use of branching in security protocols, though [28] recently relaxes this restriction. It seems thus in general that one has to choose between expressive power and unbounded sessions, and our approach is decidedly on the side of expressive power.

The chapter is organized as follows. In Section 3.1, we define how we symbolically represent messages sent by the intruder and how to solve constraints with the *lazy intruder* rules. In Section 3.2, we introduce the notion of symbolic states with their semantics. In Section 3.3, we explain how the intruder can perform experiments and make logical deductions relevant for privacy by comparing messages in their knowledge. In Section 3.4, we define the algebraic theories supported and how they are handled in the procedure. In Section 3.5, we discuss the prototype tool that we have developed as a further contribution and the case studies we have applied it to. Finally we conclude in Section 3.6 with the discussion of related work.

3.1 FLICs: Framed Lazy Intruder Constraints

The semantics of the transition system says that, in a state where the processes are receiving a message, the intruder can choose any recipe built on the domain of *concr* (respectively, the $struct_i$: they all have the same domain). The problem is that there are in general infinitely many recipes the intruder can choose from. A classic technique for deciding such infinite spaces of intruder possibilities is a constraint-based approach that we call the *lazy intruder* [61, 69, 13]: it is lazy in that it avoids, as long as possible, instantiating the variables of receive steps like $\mathsf{rcv}(X)$. The concrete intruder choice at this point does not matter; only when we check a condition that depends on X, we consider possible instantiations of X as far as needed to determine the outcome of the condition. Note that this is one symbolic layer of our approach: a symbolic state with variable X represents all concrete states where X is replaced with a message that the intruder can construct. In fact what the intruder can construct depends on the messages the intruder knew at the time when the message represented by X was sent. Due to the symbolic execution, in a state there are in general several struct_i, and thus we need to represent not only the messages sent by the intruder with variables but also the recipes that they have chosen, because a given recipe can produce different messages in each struct_i.

To keep track of this, we define an extension of frames called *framed lazy intruder constraints (FLICs)*: the entries of a standard frame represent messages that the intruder received and we write them now with a minus sign: $-l \mapsto t$. We extend this by also writing entries for messages the intruder sends with a plus sign: $+R \mapsto t$, where R is a *recipe variable* (disjoint from privacy and intruder variables). When solving the constraints, R may be instantiated with a more concrete recipe, but only using labels that occurred in the FLIC before this receive step; the order of the entries is thus significant. The messages like t can contain variables representing intruder choices that we have not yet made concrete. We require that the intruder variables first occur in positive entries as they represent intruder choices made when sending a message.

Since we deal with several possibilities in parallel, we will have several FLICs in parallel, replacing the struct_i in the ground model. Each FLIC has the same sequence of incoming labels and outgoing recipes. The intruder does not know in general which possibility is the case, but knows how they constructed messages from their knowledge, i.e., the same recipe may result in a different message in each possibility.

A FLIC is a constraint, namely that the intruder can indeed produce messages of the form needed to reach a particular state of the execution. We show that we can *solve* such FLICs, i.e., find a finite representation of all solutions (as said before, there are in general infinitely many possible concrete choices) using the lazy intruder technique, similarly to other works doing constraint-based solving with frames such as [23, 25]. In the rest of this section, we focus on defining and solving constraints by considering just one FLIC and not the rest of the possibilities, and we will explain in Section 3.2 how the lazy intruder is used for the transition system with several possibilities.

3.1.1 Defining constraints

Definition 3.1.1 (FLIC). *A framed lazy intruder constraint (FLIC) $\mathcal{A}$ is a sequence of mappings of the form $-l \mapsto t$ or $+R \mapsto t$, where each label l and recipe variable R occurs at most once, each term t is built from function symbols, privacy variables, and intruder variables. The first occurrence of each intruder variable must be in a message sent.*

We write $-l \mapsto t \in \mathcal{A}$ if $-l \mapsto t$ occurs in $\mathcal{A}$, and similarly $+R \mapsto t \in \mathcal{A}$. The domain *$\mathit{dom}(\mathcal{A})$ is the set of labels of $\mathcal{A}$ and $\mathit{vars}(\mathcal{A})$ are the privacy and intruder variables that occur in $\mathcal{A}$; similarly, we write $\mathit{rvars}(\mathcal{A})$ for the recipe variables.*

The message $\mathcal{A}(r)$ produced *by a recipe r in $\mathcal{A}$ is defined as:*

$$\begin{aligned} \mathcal{A}(l) &= t && \text{if } -l \mapsto t \in \mathcal{A}\,, \\ \mathcal{A}(R) &= t && \text{if } +R \mapsto t \in \mathcal{A}\,, \\ \mathcal{A}(f(r_1,\dots,r_n)) &= f(\mathcal{A}(r_1),\dots,\mathcal{A}(r_n))\,. \end{aligned}$$

For recipes that use labels or recipe variables not defined in the FLIC, the result is undefined. We also define an ordering between recipes and labels: $r <_{\mathcal{A}} l$ iff every label l' in r occurs before l in $\mathcal{A}$.

Example 3.1.1. Consider the transaction from Example 2.2.1, step $\mathsf{rcv}(M)$. Suppose that we have a FLIC that represents all the messages the intruder has sent and received before this execution of the transaction. Then we simply append $+R \mapsto M$ to the FLIC, where R is a fresh recipe variable. As discussed later, before executing a transaction we also rename all variables in that transaction with fresh variables, so M in this case is also fresh, i.e., it did not occur before this transaction. We are lazy in the sense that we do not explore at this point what R and M might be, because any value would do. Now the server checks whether M can be decrypted with the private key $\mathsf{inv}(\mathsf{pk}(\mathsf{s}))$. This is the case iff M has the form $\mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \cdot, \cdot)$ (we will show in Section 3.2 how exactly the try-catch construct works in the presence of the lazy intruder.) In the positive case, M is instantiated with $\mathsf{crypt}(\mathsf{pk}(\mathsf{s}), X, Y)$ for two fresh intruder variables X and Y, thus requiring that R indeed yields a message of this form. The constraint solving in Section 3.1.2 computes a finite representation of all solutions for R.

We also need to consider the negative case, i.e., when the intruder chose to use a recipe that does not satisfy the decryption; this will yield the negated equality $M \neq \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \cdot, \cdot)$. ◁

Definition 3.1.2 (Semantics of FLICs). *Let $\mathcal{A}$ be a FLIC such that $vars(\mathcal{A}) = \emptyset$, i.e., the messages in $\mathcal{A}$ are ground, so $\mathcal{A}$ has only recipe variables. $\mathcal{A}$ is* constructable *iff there exists a ground substitution of recipe variables ρ_0 such that $\mathcal{A}_1(\rho_0(R)) \approx t$ for every recipe variable R where $\mathcal{A} = \mathcal{A}_1.{+R} \mapsto t.\mathcal{A}_2$. (This implies that only labels from $dom(\mathcal{A}_1)$ can occur in $\rho_0(R)$.) We then say that ρ_0* constructs *$\mathcal{A}$.*

Let $\mathcal{A}$ be an arbitrary FLIC and $\mathcal{I}$ be an interpretation of all privacy and intruder variables. We say that $\mathcal{I}$ is a model *of $\mathcal{A}$, written $\mathcal{I} \models \mathcal{A}$, iff $\mathcal{I}(A)$ is constructable. $\mathcal{A}$ is* satisfiable *iff it has a model.*

A FLIC is thus satisfiable if there exist a suitable interpretation for the variables in messages and a suitable intruder choice for the variables in recipes such that all the constraints are satisfied.

Example 3.1.2. Suppose that Alice sent a signed message m to the intruder i and the constraint is to send some signed message to Bob. This is recorded in the following FLIC $\mathcal{A}$:

$$\begin{aligned}&-l_1 \mapsto \mathsf{inv}(\mathsf{pk}(\mathsf{i})).-l_2 \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{i}), \mathsf{sign}(\mathsf{inv}(\mathsf{pk}(\mathsf{a})), \mathsf{m})).\\&+R \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{b}), \mathsf{sign}(\mathsf{inv}(\mathsf{pk}(X)), Y))\ .\end{aligned}$$

Here $\mathcal{I}_1 = [X \mapsto \mathsf{a}, Y \mapsto \mathsf{m}]$ is a model, because $\mathcal{I}_1(\mathcal{A})$ is constructable using the recipe $R = \mathsf{crypt}(\mathsf{pk}(\mathsf{b}), \mathsf{dcrypt}(l_1, l_2))$. For every ground recipe r over $dom(\mathcal{A})$ also $\mathcal{I}_r = [X \mapsto \mathsf{i}, Y \mapsto \mathcal{A}(r)]$ is a model, using $R = \mathsf{crypt}(\mathsf{pk}(\mathsf{b}), \mathsf{sign}(l_1, r))$; note there are infinitely many such r. ◁

3.1.2 Solving constraints

We now present how to solve constraints when the intruder does not have access to destructors, i.e., as if all destructors were private functions and thus cannot occur in recipes. Hence the only place where destructors can occur are in transactions using try-catch. This allows us to work in the free algebra *for now* and with only destructor-free terms. To achieve the correctness of our procedure for the full intruder model with access to destructors, we show in Section 3.4 how all intruder applications of destructors can be handled by special analysis steps. This allows us to keep the core of the method free of destructors and algebraic reasoning.

Definition 3.1.3 (Simple FLIC). *A FLIC $\mathcal{A}$ is called* simple *iff every message sent is an intruder variable, and each intruder variable is sent only once, i.e., every message sent is of the form $+R_i \mapsto X_i$ and the X_i are pairwise distinct.*

Simple FLICs are always satisfiable, since there are no more constraints on the messages, and the intruder can choose any recipes they want. In order to solve constraints in a non-simple FLIC, we instantiate privacy, intruder and recipe variables until we reach a simple FLIC. Computing a finite representation of all solutions is then done by keeping track of the substitutions applied to instantiate the variables.

Definition 3.1.4. *Let σ be a substitution that does not contain recipe variables. We define $\sigma(-l \mapsto t.\mathcal{A}) = -l \mapsto \sigma(t).\sigma(\mathcal{A})$ and $\sigma(+R \mapsto t.\mathcal{A}) = +R \mapsto \sigma(t).\sigma(\mathcal{A})$.*

However, we cannot directly define the instantiation of recipe variables for an arbitrary FLIC, because we always need to make sure we instantiate both the recipe and the intruder variables according to the constraints. We thus define how to apply a substitution of recipe variables for simple FLICs.

Definition 3.1.5 (Choice of recipes). *A* choice of recipes *for a simple FLIC $\mathcal{A}$ is a substitution ρ mapping recipe variables to recipes, where $dom(\rho) \subseteq rvars(\mathcal{A})$.*

Let $[R \mapsto r]$ be a choice of recipes for $\mathcal{A}$ that maps only one recipe variable, where $\mathcal{A} = \mathcal{A}_1.{+R} \mapsto X.\mathcal{A}_2$. Let $R_1, \dots, R_n$ be the fresh variables in r, i.e., $\{R_1, \dots, R_n\} = rvars(r) \setminus rvars(\mathcal{A})$, taken in a fixed order (e.g., the order in which they first occur in r). Let $X_1, \dots, X_n$ be fresh intruder variables. The application of $[R \mapsto r]$ to the FLIC $\mathcal{A}$ is defined as $[R \mapsto r](\mathcal{A}_1.{+R} \mapsto X.\mathcal{A}_2) = \mathcal{A}'.\sigma(\mathcal{A}_2)$ where $\mathcal{A}' = \mathcal{A}_1.{+R_1} \mapsto X_1.\cdots.{+R_n} \mapsto X_n$ and $\sigma = [X \mapsto \mathcal{A}'(r)]$.

For the general case, let ρ be a choice of recipes for $\mathcal{A}$. We define $\rho(\mathcal{A})$ recursively where one recipe variable is substituted at a time, and we follow the order in which the recipe variables occur in $\mathcal{A}$: if $\rho = [R \mapsto r]\rho'$, where R occurs in $\mathcal{A}$ before any $R' \in dom(\rho')$, then $\rho(\mathcal{A}) = \rho'([R \mapsto r](\mathcal{A}))$. Every application $[R \mapsto r](\mathcal{A})$ corresponds to a substitution $\sigma = [X \mapsto \mathcal{A}'(r)]$ (as defined above), and we denote with $\sigma_\rho^{\mathcal{A}}$ the idempotent substitution aggregating all these substitutions σ from applying ρ to $\mathcal{A}$.

Note that if ρ is a choice of recipes for a simple FLIC $\mathcal{A}$, then $\rho(\mathcal{A})$ is simple, because the fresh recipe variables added in $\rho(\mathcal{A})$ map to fresh intruder variables.

Example 3.1.3. Consider the FLIC

$$\mathcal{A} = -l_1 \mapsto \mathsf{inv}(\mathsf{pk}(\mathsf{i})).{+R_1} \mapsto X_1.{-l_2} \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), X_1, \mathsf{r}).{+R_2} \mapsto X_2 \ .$$

This represents the situation where the intruder knows the private key of agent i, they have sent some message X_1 (using recipe R_1), after that they have observed a message encrypted for the server s containing X_1, and finally the intruder has sent another message X_2 (using recipe R_2).

Let ρ be the choice of recipes $[R_1 \mapsto \mathsf{pair}(R_3, R_4), R_2 \mapsto l_2]$. This expresses the fact that to send X_1, the intruder has composed themselves a message that is a pair, and to send X_2, they have simply reused a message observed previously. In order to apply ρ to $\mathcal{A}$, we first consider the choice for R_1: we remove the mapping $+R_1 \mapsto X_1$, and we introduce new mappings with fresh intruder variables X_3, X_4, so we have

$$\begin{aligned}\mathcal{A}' = {} & -l_1 \mapsto \mathsf{inv}(\mathsf{pk}(\mathsf{i})).{+R_3} \mapsto X_3.{+R_4} \mapsto X_4.{-l_2} \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{pair}(X_3, X_4), \mathsf{r}). \\ & +R_2 \mapsto X_2 \ .\end{aligned}$$

Table 3.1: Lazy intruder rules

Unification	$(\rho, \mathcal{A}_1.{-l} \mapsto s.\mathcal{A}_2.{+R} \mapsto t.\mathcal{A}_3, \sigma)$ $\rightsquigarrow (\rho', \sigma'(\mathcal{A}_1.{-l} \mapsto s.\mathcal{A}_2.\mathcal{A}_3), \sigma')$ where $\rho' = [R \mapsto l]\rho$ and $\sigma' = mgu(\sigma \wedge s \doteq t)$	if $\mathcal{A}_1.{-l} \mapsto s.\mathcal{A}_2$ is simple, $s, t \notin \mathcal{V}$ and $\sigma' \neq \bot$
Composition	$(\rho, \mathcal{A}_1.{+R} \mapsto f(t_1, \dots, t_n).\mathcal{A}_2, \sigma)$ $\rightsquigarrow (\rho', \mathcal{A}_1.{+R_1} \mapsto t_1. \cdots .{+R_n} \mapsto t_n.\mathcal{A}_2, \sigma)$ where the R_i are fresh recipe variables and $\rho' = [R \mapsto f(R_1, \dots, R_n)]\rho$	if $\mathcal{A}_1$ is simple, $f \in \Sigma_{pub}$ and $\sigma \neq \bot$
Guessing	$(\rho, \mathcal{A}_1.{+R} \mapsto x.\mathcal{A}_2, \sigma) \rightsquigarrow (\rho', \sigma'(\mathcal{A}_1.\mathcal{A}_2), \sigma')$ where $\rho' = [R \mapsto c]\rho$ and $\sigma' = mgu(\sigma \wedge x \doteq c)$	if $\mathcal{A}_1$ is simple, $c \in dom(x)$ and $\sigma' \neq \bot$
Repetition	$(\rho, \mathcal{A}_1.{+R_1} \mapsto X.\mathcal{A}_2.{+R_2} \mapsto X.\mathcal{A}_3, \sigma)$ $\rightsquigarrow (\rho', \mathcal{A}_1.{+R_1} \mapsto X.\mathcal{A}_2.\mathcal{A}_3, \sigma)$ where $\rho' = [R_2 \mapsto R_1]\rho$	if $\mathcal{A}_1.{+R_1} \mapsto X.\mathcal{A}_2$ is simple and $\sigma \neq \bot$

This first choice of recipes induces the substitution $[X_1 \mapsto \mathsf{pair}(X_3, X_4)]$, which is applied to the rest of the FLIC: the X_1 inside the encryption has been replaced with the pair.

Now we consider the choice for R_2: we remove the mapping $+R_2 \mapsto X_2$, and there is no new mappings to introduce since the recipe used is label l_2. Finally, we get the FLIC

$$\rho(\mathcal{A}) = -l_1 \mapsto \mathsf{inv(pk(i))}.{+R_3} \mapsto X_3.{+R_4} \mapsto X_4.{-l_2} \mapsto \mathsf{crypt(pk(s)}, \mathsf{pair}(X_3, X_4), \mathsf{r}) \ ,$$

and the substitution $\sigma_\rho^{\mathcal{A}} = [X_1 \mapsto \mathsf{pair}(X_3, X_4), X_2 \mapsto \mathsf{crypt(pk(s)}, \mathsf{pair}(X_3, X_4), \mathsf{r})]$. $\lhd$

We denote with $mgu(s_1 \doteq t_1 \wedge \cdots \wedge s_n \doteq t_n)$ the result, called *most general unifier (mgu)*, of unifying the s_i and t_i, which is either some substitution or $\bot$ whenever no unifier exists. Slightly abusing notation, we consider a substitution $[x_1 \mapsto t_1, \dots, x_n \mapsto t_n]$ as the formula $x_1 \doteq t_1 \wedge \cdots \wedge x_n \doteq t_n$ and $\bot$ as false. We modify the standard mgu algorithm so that when we have to unify an intruder variable X with a privacy variable x, it will always result in $[X \mapsto x]$ (i.e., privacy variables are never substituted with intruder variables).

Moreover, we add a postprocessing step to mgu: if the resulting σ contains $x \mapsto c$ where x is a privacy variable chosen from a domain D that does not contain c, then the result is $\bot$.

In order to solve the constraints, we define a reduction relation $\rightsquigarrow$ on FLICs: $\rightsquigarrow$ is Noetherian and a FLIC that cannot be further reduced is either simple or unsatisfiable. Moreover, $\rightsquigarrow$ is not confluent, but rather is meant to explore different ways for the intruder to satisfy constraints, and thus we will consider the set of all simple FLICs that are reachable from a given one: the simple FLICs together will be equivalent to the given FLIC. Since $\rightsquigarrow$ is not only Noetherian, but also finitely branching, the set of reachable simple FLICs is always finite by Kőnig's lemma.

Definition 3.1.6 (Lazy intruder rules). *The relation $\rightsquigarrow$ is a relation on triples $(\rho, \mathcal{A}, \sigma)$ of a choice of recipes ρ, a FLIC $\mathcal{A}$ and a substitution σ, where ρ and σ keep track of all variable substitutions performed in the reduction steps so far and are such that $dom(\rho) \cap rvars(\mathcal{A}) = \emptyset$ and $dom(\sigma) \cap vars(\mathcal{A}) = \emptyset$. The rules are defined in Table 3.1.*

Below, we explain the meaning of the rules and we use the variable identifiers from Table 3.1.

Unification. When the intruder has to send a message t, they can use any message s previously received and that unifies ($\sigma' = mgu(\sigma \wedge s \doteq t) \neq \bot$), choosing the label l that maps to s for instantiating the recipe variable R. There is one less message to send, but the unifier might make other constraints non-simple (application of σ' to the rest of the constraints). This rule is *not* applicable if t is a variable: we do not explore how an arbitrary message can be constructed—this is where the intruder is *lazy*. Similarly, the rule is not applicable if s is a variable, because that means that it is a message the intruder constructed earlier (or that the intruder can guess, for a privacy variable); this is why we require this part of the FLIC to be already simple: s is a message that the intruder already knew how to construct earlier.

Composition. When the intruder has to send a composed message $f(t_1, \dots, t_n)$, they can produce it themselves if f is public and they can produce the t_i. The intruder thus chooses to compose the message themselves, so the recipe R is the application of f to other recipes: we have $R = f(R_1, \dots, R_n)$ and we add the new constraints that each R_i produces the respective t_i.

Guessing. When the intruder has to send a privacy variable x, they can guess the actual value of x, say c. In fact, this is a guessing attack as we let the privacy variables range over small domains of public constants. This rule represents the case that the intruder guesses correctly, and the variable x is replaced with the guessed value c. Note that using the **Guessing** rule does *not* yet mean that the intruder knows that c is the correct guess: in the rest of the procedure, whenever there is such a guess we model both the right and wrong guesses, and the intruder may not be able to tell what is the case.

Repetition. If the intruder has to send an intruder variable X that they have already sent earlier, i.e., there is a mapping $R_1 \mapsto X$ before the constraint $R_2 \mapsto X$, then they use the same recipe, i.e., $R_2 = R_1$. Since there may be several ways to produce the same message, one may wonder if this is actually complete: could there be an attack where constructing the same message in two different ways would tell the intruder anything more? In fact, for what concerns the behavior of the honest agents, it cannot make a difference, and comparing different ways to construct the same message is covered in the intruder experiments in Section 3.3.

We now define the lazy intruder results as the set of choices of recipes ρ that solve the constraints.

Definition 3.1.7 (Lazy intruder results)**.** *Let $\mathcal{A}$ be a FLIC and σ be a substitution. Let ε be the identity substitution. We define*

$$LI(\mathcal{A}, \sigma) = \{\rho \mid (\varepsilon, \sigma(\mathcal{A}), \sigma) \rightsquigarrow^* (\rho, \mathcal{A}', \_) \text{ and } \mathcal{A}' \text{ is simple}\} \,.$$

Example 3.1.4. Following Example 3.1.1, let us assume that the intruder has already observed a message encrypted for the server from another agent x', and is now symbolically executing the transaction. With the constraint induced by the decryption from the server, the FLIC is now $-l \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), x', r).{+R} \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), X, Y)$. Since pk is public, the lazy intruder returns two choices of recipes: $\rho_1 = [R \mapsto l]$, meaning the intruder replays the message from the knowledge (since it unifies), and $\rho_2 = [R \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), R_1, R_2)]$, meaning the intruder composes the message themselves where R_1 and R_2 stand for arbitrary recipes. $\lhd$

Definition 3.1.8 (Representation of choice of recipes)**.** *Let $\mathcal{A}$ be a FLIC, $\mathcal{I} \models \mathcal{A}$, ρ_0 be a ground choice of recipes and ρ be a choice of recipes. We say that ρ* represents ρ_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$ *iff there exists ρ_0' such that ρ_0' is an instance of ρ and for every $R \in rvars(\mathcal{A})$, $\mathcal{I}(\mathcal{A})(\rho_0'(R)) = \mathcal{I}(\mathcal{A})(\rho_0(R))$ and:*

- *If $\rho(R) \in dom(\mathcal{A})$, then $\rho_0(R) \in dom(\mathcal{A})$ and either $\rho_0'(R) = \rho_0(R)$ or $\rho_0'(R) <_{\mathcal{A}} \rho_0(\mathcal{A})$.*
- *If $\rho(R)$ is a composed recipe and $\rho_0(R) \in dom(\mathcal{A})$, then $\rho_0'(R) <_{\mathcal{A}} \rho_0(R)$.*

This notion of representation gives the lazy intruder some liberties, namely to be lazy in not instantiating recipe variables that do not matter, and to replace subrecipes with equivalent ones (that may be smaller according to our ordering between recipes and labels).

Example 3.1.5. Consider the FLIC

$$\mathcal{A} = +R_1 \mapsto X_1.+R_2 \mapsto X_2.-l_1 \mapsto f(X_1, X_2).+R_3 \mapsto X_3.-l_2 \mapsto X_3.+R_4 \mapsto f(\mathsf{a}, \mathsf{b}) ,$$

where f is a private function and a, b are public constants. Let $\mathcal{I} = [X_1 \mapsto \mathsf{a}, X_2 \mapsto \mathsf{b}, X_3 \mapsto f(\mathsf{a}, \mathsf{b})]$. Note that $\mathcal{I}$ is a model, because $\mathcal{I}(\mathcal{A})$ can be constructed with, e.g., $\rho_0 = [R_1 \mapsto \mathsf{a}, R_2 \mapsto \mathsf{b}, R_3 \mapsto l_1, R_4 \mapsto l_2]$. During constraint solving with the lazy intruder, we would never apply **Unification** between l_2 and R_4, because l_2 maps to intruder variable X_3 (thus, whatever X_3 is, it is a message that the intruder can already generate from their knowledge up to l_1). Instead, we would get the intruder result $\rho = [R_1 \mapsto \mathsf{a}, R_2 \mapsto \mathsf{b}, R_4 \mapsto l_1]$. This ρ represents ρ_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$, because we have $\rho_0' = [R_1 \mapsto \mathsf{a}, R_2 \mapsto \mathsf{b}, R_3 \mapsto l_1, R_4 \mapsto l_1]$ as an instance of ρ and ρ_0' constructs $\mathcal{I}(\mathcal{A})$, albeit in a slightly different way than ρ_0. While $\rho(R_4) = l_1 \neq l_2 = \rho_0(R_4)$, we still have $\rho_0'(R_4) = l_1 <_{\mathcal{A}} l_2 = \rho_0(R_4)$. In other words, the representation does not follow exactly the same choices of recipes as ρ_0, because we found a "simpler" recipe for R_4: there is no point in using l_2 when l_1 already gives the same message.

For another example, consider the FLIC

$$\mathcal{A} = +R_1 \mapsto X_1.-l \mapsto X_1.+R_2 \mapsto \mathsf{a} .$$

Let $\mathcal{I} = [X_1 \mapsto \mathsf{a}]$. Then $\mathcal{I}$ is a model because $\mathcal{I}(\mathcal{A})$ can be constructed with, e.g., $\rho_0 = [R_1 \mapsto \mathsf{a}, R_2 \mapsto l]$. Here again, the lazy intruder would not apply **Unification** between l and R_2 because l maps to an intruder variable. However, since a is public, we would get the lazy intruder result $\rho = [R_2 \mapsto \mathsf{a}]$. Then ρ represents ρ_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$, because we have $\rho_0' = [R_1 \mapsto \mathsf{a}, R_2 \mapsto \mathsf{a}]$ as an instance of ρ and ρ_0' constructs $\mathcal{I}(\mathcal{A})$. Note that $\rho(R_2) = \mathsf{a} <_{\mathcal{A}} l = \rho_0(R_2)$, i.e., the composed recipe a is "simpler" than label l because it can be constructed before receiving the message. $\lhd$

In the completeness proof we show that every solution of the constraint is represented by some choice of recipes that the lazy intruder finds. The lazy intruder rules are sound, complete and terminating.

Theorem 3.1.1 (Lazy intruder correctness). *Let $\mathcal{A}$ be a FLIC, σ be a substitution, $\mathcal{I} \models \mathcal{A}$ such that $\mathcal{I} \models_\Sigma \sigma$ and let ρ_0 be a ground choice of recipes. Then ρ_0 constructs $\mathcal{I}(\mathcal{A})$ iff there exists $\rho \in LI(\mathcal{A}, \sigma)$ such that ρ represents ρ_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$. Moreover, $LI(\mathcal{A}, \sigma)$ is finite.*

3.2 Symbolic states

Our approach explores a transition system on symbolic states, where each symbolic state represents an infinite set of ground states. In the rest of the thesis, we denote symbolic states by $\mathcal{S}$, $\mathcal{S}'$, etc., and ground states by S, S', etc.

A ground state (defined in Chapter 2) may actually contain privacy variables, representing the possible uncertainty of the intruder in this state, but each variable has one concrete value that represents *the truth* in that state, which is expressed by a formula γ that the intruder does not have access to (and the frame *concr* is an instance of one of the *struct$_i$* under γ). This is the reason why we call it a ground state, even though it contains variables. A symbolic state includes actually two symbolic layers. For the first symbolic layer, we define a symbolic state to merge all those ground states that differ only in the concrete γ and thus the concrete frame *concr*, i.e., where the intruder has the same uncertainty. Therefore, a symbolic state does not contain γ and *concr*, and has no underlined possibility. Thus, we need to keep track of the released formula α_i for each possibility separately. A second symbolic layer is to use intruder variables and FLICs to avoid enumerating the infinite choices that the intruder has when sending messages, thus the frames $struct_i$ are generalized to FLICs $\mathcal{A}_i$ in symbolic states. We will introduce in Section 3.3 the intruder experiments, and a symbolic state $\mathcal{S}$ contains a record of all experiments the intruder has already performed.

Definition 3.2.1 (Symbolic state). *A* symbolic state *is a tuple* $(\alpha_0, \beta_0, \mathcal{P}, \mathit{Checked})$ *such that:*

- α_0 *is a* Σ_0*-formula, the* common payload*;*
- β_0 *is a* Σ_0*-formula, the* intruder reasoning *about possibilities and privacy variables;*
- $\mathcal{P}$ *is a set of* possibilities*, which are each of the form* $(P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)$*, where* P *is a process,* ϕ *is a* Σ_0*-formula,* $\mathcal{A}$ *is a FLIC,* $\mathcal{X}$ *is a disequalities formula (in the grammar below),* α *is a* Σ_0*-formula called* partial payload*, and* δ *is a sequence of memory updates of the form* $\mathsf{cell}[s] := t$ *for messages* s *and* t*; and*
- *Checked is a set of pairs* (l, r)*, where* l *is a label and* r *is a recipe.*

where disequalities formulas are of the following form:

$$\begin{array}{rcll} \mathcal{X} & := & \mathcal{X} \wedge \mathcal{X} \mid \forall \vec{X}.\ \neg \mathcal{X}_0 & \textit{Disequalities formula} \\ \mathcal{X}_0 & := & \mathcal{X}_0 \wedge \mathcal{X}_0 \mid t \doteq t & \textit{Equalities formula} \end{array}$$

A symbolic state is finished *iff all the processes in* $\mathcal{P}$ *are* 0*.*

We may write $\mathcal{S}[e \leftarrow e']$ to denote the symbolic state identical to $\mathcal{S}$ except that e is replaced with e'.

We have augmented the FLICs $\mathcal{A}_i$ here with disequalities $\mathcal{X}_i$, i.e., negated equality constraints, which allows us to restrict the choices of the intruder in a symbolic state. This is needed when we want to make a split between the case that the intruder makes a particular choice and the case that they choose anything else. This is formalized in the following definition of applying a recipe substitution which is only possible when all the respective $\mathcal{X}_i$ are consistent with it:

Definition 3.2.2 (Choice of recipes for a symbolic state). *Let* $\mathcal{S} = (\_, \_, \mathcal{P}, \mathit{Checked})$ *be a symbolic state and* ρ *be a recipe substitution. We say that* ρ *is a* choice of recipes for $\mathcal{S}$ *iff* ρ *is a choice of recipes for all FLICs in* $\mathcal{P}$ *and for every FLIC* $\mathcal{A}$ *and associated disequalities* $\mathcal{X}$ *in* $\mathcal{P}$*, the formula* $\sigma_\rho^{\mathcal{A}}(\mathcal{X})$ *is satisfiable, i.e., the disequalities are satisfiable under the substitution induced by* ρ *in* $\mathcal{A}$ *(Definition 3.1.5). Moreover, we define*

$$\begin{aligned} \rho(\mathcal{P}) &= \{(\sigma_\rho^{\mathcal{A}}(P), \phi, \rho(\mathcal{A}), \sigma_\rho^{\mathcal{A}}(\mathcal{X}), \alpha, \sigma_\rho^{\mathcal{A}}(\delta)) \mid (P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta) \in \mathcal{P}\}\ , \\ \rho(\mathit{Checked}) &= \{(l, \rho(r)) \mid (l, r) \in \mathit{Checked}\}\ , \\ \rho(\mathcal{S}) &= \mathcal{S}[\mathcal{P} \leftarrow \rho(\mathcal{P}), \mathit{Checked} \leftarrow \rho(\mathit{Checked})]\ . \end{aligned}$$

Example 3.2.1. Continuing Example 3.1.1 where the intruder has sent a message M that cannot be decrypted by the server: we will define precisely the transitions for symbolic states in Table 3.2, but for now it suffices to say that the execution of the transaction from Example 2.2.1 can lead to a symbolic state with a single possibility $(0, \mathsf{true}, +R \mapsto$

$M, \forall X, Y.\ M \not\doteq \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), X, Y), \mathsf{true}, [])$. By Definition 3.1.5, the substitution $\rho = [R \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{a}, \mathsf{h}(\mathsf{a}))]$ is a choice of recipes for the FLIC $\mathcal{A} = +R \mapsto M$. However, this ρ is now excluded by this symbolic state, because it is inconsistent with the disequalities formula: by applying ρ to $\mathcal{A}$, we would get the substitution $\sigma_\rho^{\mathcal{A}} = [M \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{a}, \mathsf{h}(\mathsf{a}))]$, and the formula $\forall X, Y.\ \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{a}, \mathsf{h}(\mathsf{a})) \not\doteq \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), X, Y)$ would not be satisfiable. The point here is that we consider such a choice of recipes in a separate symbolic state, and we are here in the case that the intruder chose something else that was not decrypted by the server. $\lhd$

When writing $\rho(\mathcal{S})$ in the following, we implicitly assume that all disequalities in $\mathcal{S}$ are satisfiable under ρ, and that $\rho(\mathcal{S})$ is discarded otherwise. To decide whether disequality $\mathcal{X}$ is satisfiable it suffices to replace the free variables with distinct fresh constants and check that the corresponding unification problems have no solution. Moreover, we will always use the lazy intruder in the context of a symbolic state, so we further assume that $LI(\cdot, \cdot)$ only returns choices of recipes for the current symbolic state, i.e., excluding any ρ that would contradict a disequality.

From a symbolic state we can define all the choices of recipes (instantiations of the recipe and intruder variables) for the messages sent by the intruder and all the concrete executions (instantiations of privacy variables) that the intruder considers possible. A symbolic state represents a set of ground states, where each ground state corresponds to one multi message-analysis problem. For every ground state, the common payload α_0 is augmented with the partial payload α_i released in the corresponding possibility. Moreover, every model γ of the privacy variables needs to be augmented with the fixed interpretation γ_0 of the relation symbols (recall that this Σ_0-formula γ_0 is part of the protocol specification).

Meta-notation. In the specification of transactions, we allow in the release steps $\star\ \phi$ the use of the *meta-notation* $\gamma(t)$ for a message t. Recall that in every ground state, the real values of privacy variables is defined by a ground interpretation γ. Thus, for instance, releasing $\star\ x \doteq \gamma(x)$ means allowing the intruder to learn the true value of x. In the symbolic execution for ground states, the meta-notation can be resolved by using γ as a substitution before adding the formula to α.

Example 3.2.2. In Example 2.2.1, in case the agent x is actually the intruder i, i.e., $x \doteq \mathsf{i}$, then the intruder can decrypt the message and observe what was the decision. Thus they would learn both that $x \doteq \mathsf{i}$ as well as the value of y (i.e., they know the server's decision). This leads to a privacy violation, unless we declassify x and y by releasing, if x is the intruder, the formula $\star\ x \doteq \gamma(x) \land y \doteq \gamma(y)$. Releasing this information is still not enough because in case $x \not\doteq \mathsf{i}$ the intruder can also deduce that; so we additionally need to release $\star\ x \not\doteq \mathsf{i}$ in that case to remove the privacy violation. $\lhd$

In a symbolic state, however, there is no γ since the symbolic state represents all possible γ at once. Hence, in order to define the semantics, we need to resolve the meta-notation that we allow in the α_i. Given α_i and the truth γ, let $[\alpha_i]^\gamma$ be the instantiation of the meta-notation in α_i, i.e., replacing every occurrence of a term $\gamma(x)$ in α_i (for a variable x) with the actual value of x in the given γ. For instance, if $\gamma(x) = \mathsf{i}$, then $[x \doteq \gamma(x)]^\gamma = x \doteq \mathsf{i}$.

Definition 3.2.3 (Semantics of symbolic states). *Let $\mathcal{S} = (\alpha_0, \beta_0, \mathcal{P}, \_)$ be a finished symbolic state. The ground states represented by $\mathcal{S}$ are given by*

$$\begin{aligned} [\![\mathcal{S}]\!] = \{(\alpha_0 \land [\alpha_i]^\gamma, \beta_0, \gamma, \rho(\mathcal{P})) \mid\ & (0, \phi_i, \mathcal{A}_i, \_, \alpha_i, \_) \in \mathcal{P}, \\ & \rho \textit{ is a ground choice of recipes for } \mathcal{S}, \\ & \gamma \textit{ is a } \Sigma_0\textit{-interpretation of } \alpha_0 \land \beta_0 \land \gamma_0 \land \phi_i\} \end{aligned}$$

where $\rho(\mathcal{P})$ returns possibilities of the form $(0, \phi_j, struct_j, \delta_j)$, i.e., the additional components of symbolic possibilities are dropped because they are irrelevant for ground states (note that the α_i have already been used as part of the payload α).

We say that a symbolic state $\mathcal{S}$ satisfies privacy *iff every ground state $S \in [\![\mathcal{S}]\!]$ satisfies privacy.*

When computing the mgu between messages or solving constraints with the lazy intruder rules, we may deal with substitutions that contain both privacy and intruder variables. However, it is important to remember that the instantiation of privacy variables does not depend on the intruder, it is actually the goal of the intruder to learn about the privacy variables. On the other hand, intruder variables are instantiated according to the recipes chosen by the intruder. Thus, we distinguish substitutions that only substitute privacy variables (we will compute substitutions as mgus so we also have to consider $\perp$ when no mgu exists).

Definition 3.2.4 (Privacy substitution). *Given a substitution σ, the predicate isPriv is defined as: isPriv(σ) iff dom(σ) $\subseteq \mathcal{V}_{privacy}$. We also define $\underline{\sigma}$ as the privacy part of σ, i.e., $\underline{\sigma}(x) = \sigma(x)$ if $x \in \mathcal{V}_{privacy}$, and $\underline{\sigma}(x) = x$ otherwise. Moreover, define isPriv($\perp$) =* false *and $\underline{\perp} = \perp$.*

Table 3.2 defines the transitions on symbolic states to evaluate processes (with the lazy intruder) as a relation $\mathcal{S} \Rightarrow \mathcal{C}$ between one symbolic state $\mathcal{S}$ and a set $\mathcal{C}$ of symbolic states. We use sets here to explicitly gather all successors of one symbolic state $\mathcal{S}$: a singleton means that there is exactly one successor, and a set of higher cardinality means that there is branching. We extend the definition to relate sets of symbolic states: $\mathcal{C} \Rightarrow \mathcal{C}'$ iff $\mathcal{S} \in \mathcal{C}$, $\mathcal{S} \Rightarrow \mathcal{C}_{\mathcal{S}}$ and $\mathcal{C}' = \mathcal{C} \setminus \{\mathcal{S}\} \cup \mathcal{C}_{\mathcal{S}}$. We summarize here this semantics and we discuss in Appendix A.2 the correctness of the correspondence to Table 2.1.

The non-deterministic choice is quite simple: instead of splitting into one successor state for each value in the domain, all these are handled in one symbolic state where we only add the domain constraint to α_0 or β_0, respectively. For a receiving step $\mathsf{rcv}(X)$, recall that the ground model has here an infinite branching over all the recipes that the intruder could use. This is the very reason for introducing the FLICs in the symbolic model: we simply choose a fresh recipe variable R and augment every FLIC with $+R \mapsto X$, saying that the intruder can choose any recipe R (over the labels of the FLIC so far) to form the input message X.

Due to the two symbolic representations of privacy choices and intruder choices, treating try-catch is quite complicated. Consider a symbolic state $\mathcal{S}$ where one possibility has process $\mathsf{try}\ X := d(t_1, t_2)\ \mathsf{in}\ P_1\ \mathsf{catch}\ P_2$. The class of algebraic theories we support (we give the precise definition in Definition 3.4.1) ensures that there is a unique rewrite rule for destructor d. Let $d(k, c(k', X_1, \dots, X_n)) \to X_i$ be a fresh instance of this rule (all variables renamed to fresh intruder variables). The destructor succeeds iff the formula $\psi = t_1 \doteq k \wedge t_2 \doteq c(k', X_1, \dots, X_n)$ is satisfied. Let $\sigma = mgu(\psi)$; note that this mgu may not exist ($\sigma = \perp$). Note also that σ may instantiate both privacy variables (that the intruder cannot control) and all the other variables (that the intruder can, at least indirectly, control by the choice of recipes for messages received). We now have several symbolic states for the different cases.

Destructor (1.1) is the case that there is a solution ($\sigma \neq \perp$) and the intruder makes a choice of recipes $\rho \in LI(\mathcal{A}, \sigma)$ that may satisfy σ. Note that ρ can only determine intruder variables: it induces a substitution $\sigma_\rho^{\mathcal{A}}$ in the given possibility (Definition 3.1.5). It is now guaranteed that $\sigma_\rho^{\mathcal{A}}(t_2)$ yields a term of the form $c(s_0, \dots, s_n)$, because we had required this in ψ and t_2 cannot be a privacy variable. Thus we can now extract the exact term s_i in case the destructor works. Whether it works depends however on the privacy variables. For

Table 3.2: Semantics of the execution for symbolic states

Choice	$(\alpha_0, \beta_0, \{(\mathsf{mode}\ x \in D.P_i, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \mathit{Checked})$ $\Rightarrow \{(\alpha'_0, \beta'_0, \{(P_i, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \mathit{Checked})\}$ where $\alpha'_0 = \alpha_0 \wedge x \in D$ and $\beta'_0 = \beta_0$ if $\mathsf{mode} = \star$, $\alpha'_0 = \alpha_0$ and $\beta'_0 = \beta_0 \wedge x \in D$ if $\mathsf{mode} = \diamond$
Receive	$(\alpha_0, \beta_0, \{(\mathsf{rcv}(X).P_i, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \mathit{Checked})$ $\Rightarrow \{(\alpha_0, \beta_0, \{(P_i, \phi_i, \mathcal{A}_i.{+R} \mapsto X, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \mathit{Checked})\}$ where R is a fresh recipe variable
Cell read	$(\alpha_0, \beta_0, \{(X := \mathsf{cell}[s].P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P}, \mathit{Checked})$ $\Rightarrow \{(\alpha_0, \beta_0, \{(P', \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \cup \mathcal{P}, \mathit{Checked})\}$ where $\delta_{\mid\mathsf{cell}} = \mathsf{cell}[s_1] := t_1. \cdots .\mathsf{cell}[s_k] := t_k$, the ground context for initial value of cell is $C[\cdot]$ and $P' = \mathsf{if}\ s \doteq s_1\ \mathsf{then}\ P[X \mapsto t_1]\ \mathsf{else}\ \dots$ $\mathsf{if}\ s \doteq s_k\ \mathsf{then}\ P[X \mapsto t_k]\ \mathsf{else}\ P[X \mapsto C[s]]$
Cell write	$(\alpha_0, \beta_0, \{(\mathsf{cell}[s] := t.P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P}, \mathit{Checked})$ $\Rightarrow \{(\alpha_0, \beta_0, \{(P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \mathsf{cell}[s] := t.\delta)\} \cup \mathit{Checked})\}$
	Let $P = \mathsf{try}\ X := d(t_1, t_2)\ \mathsf{in}\ P_1\ \mathsf{catch}\ P_2$, consider a fresh instance $d(k, c(k', X_1, \dots, X_n)) \to X_i$ of the rewrite rule for d, let $\psi = t_1 \doteq k \wedge t_2 \doteq c(k', X_1, \dots, X_n)$ and $\sigma = \mathit{mgu}(\psi)$. The rules for $\mathsf{try\text{-}catch}$ are:
Destructor (1.1)	$(\alpha_0, \beta_0, \{(P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P}, \mathit{Checked})$ $\Rightarrow \{\rho((\alpha_0, \beta_0, \{(P_1[X \mapsto s_i], \phi \wedge \underline{\sigma'}, \mathcal{A}, \mathcal{X}, \alpha, \delta),$ $(P_2, \phi \wedge \neg\underline{\sigma'}, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \cup \mathcal{P}, \mathit{Checked}))$ $\mid \rho \in \mathit{LI}(\mathcal{A}, \sigma)$, let $c(s_0, \dots, s_n) = \sigma^{\mathcal{A}}_{\rho}(t_2)$ and $\sigma' = \mathit{mgu}(\sigma^{\mathcal{A}}_{\rho}(\psi))\}$
Destructor (1.2)	$\cup \begin{cases} \{(\alpha_0, \beta_0, \{(P_2, \phi, \mathcal{A}, \mathcal{X}', \alpha, \delta)\} \cup \mathcal{P}, \mathit{Checked})\} & \text{if } \sigma(\mathcal{A}) \text{ is not simple} \\ \emptyset & \text{otherwise} \end{cases}$ if $\sigma \neq \bot$, where $\mathcal{X}' = \mathcal{X} \wedge \forall \vec{Y}.\ \neg\sigma$ and $\vec{Y} = \mathit{ivars}(\sigma) \setminus \mathit{ivars}(\mathcal{A})$
Destructor (2)	$(\alpha_0, \beta_0, \{(P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P}, \mathit{Checked})$ $\Rightarrow \{(\alpha_0, \beta_0, \{(P_2, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \cup \mathcal{P}, \mathit{Checked})\}$ if $\sigma = \bot$
Conditional	$(\alpha_0, \beta_0, \{(\mathsf{if}\ R(t_1, \dots, t_n)\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P}, \mathit{Checked})$ $\Rightarrow \{(\alpha_0, \beta_0, \{(P_1, \phi \wedge R(t_1, \dots, t_n), \mathcal{A}, \mathcal{X}, \alpha, \delta),$ $(P_2, \phi \wedge \neg R(t_1, \dots, t_n), \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \cup \mathcal{P}, \mathit{Checked})\}$ For $\mathsf{if}\ s \doteq t\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$, the transitions are like **Destructor** (1) and **Destructor** (2), but with $P = \mathsf{if}\ s \doteq t\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ and $\psi = s \doteq t$
Release	$(\alpha_0, \beta_0, \{(\star\ \psi.P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P}, \mathit{Checked})$ $\Rightarrow \{(\alpha_0, \beta_0, \{(P, \phi, \mathcal{A}, \mathcal{X}, \alpha \wedge \psi, \delta)\} \cup \mathcal{P}, \mathit{Checked})\}$
Eliminate	$(\alpha_0, \beta_0, \{(P, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta)\} \uplus \mathcal{P}, \mathit{Checked}) \Rightarrow \{(\alpha_0, \beta_0, \mathcal{P}, \mathit{Checked})\}$ if $\alpha_0 \wedge \beta_0 \models \neg\phi$
Send	$(\alpha_0, \beta_0, \{(\mathsf{snd}(t_i).P_i, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \dots, n\}\} \uplus \mathcal{P}, \mathit{Checked})$ $\Rightarrow \{(\alpha_0, \beta_0 \wedge \bigvee_{i=1}^{n} \phi_i, \{(P_i, \phi_i, \mathcal{A}_i.{-l} \mapsto t_i, \mathcal{X}_i, \alpha_i, \delta_i)$ $\mid i \in \{1, \dots, n\}\}, \mathit{Checked}),$ $(\alpha_0, \beta_0 \wedge \bigwedge_{i=1}^{n} \neg\phi_i, \mathcal{P}, \mathit{Checked})\}$ if every process in $\mathcal{P}$ is 0, where l is a fresh label

instance if the destructor is asymmetric description with key $\mathsf{inv}(\mathsf{pk}(\mathsf{a}))$ and the intruder chooses a message encrypted with $\mathsf{pk}(x)$, this succeeds iff $x \doteq \mathsf{a}$. We thus need to compute a unifier σ' for $\sigma_\rho^{\mathcal{A}}(\psi)$, i.e., the condition under the current choice of recipes, and $\underline{\sigma'}$ is now the substitution of privacy variables under which the destructor succeeds. We thus split the possibility in two: one where we continue with P_1 where X is bound to the result s_i of the destructor and condition $\underline{\sigma'}$, and one with P_2 and condition $\neg\underline{\sigma'}$.

Destructor (1.2) is the case that there is a solution ($\sigma \neq \bot$) and it depends on intruder choices ($\sigma(\mathcal{A})$ is not simple), but the intruder chooses any recipe that does not satisfy σ. For this we simply add the disequality $\forall\vec{Y}.\neg\sigma$ to $\mathcal{X}$, where $\vec{Y}$ are the intruder variables not bound by $\mathcal{A}$. (If $\sigma(\mathcal{A})$ is simple, then there is only one trivial choice of recipes, the identity substitution, so there is no symbolic state for excluding this choice.) **Destructor** (2) finally is the case that there is no unifier σ, so we necessarily end up in P_2.

The if-then-else conditional is handled in a very similar way when the condition is an equality $s \doteq t$, obtaining a most general unifier $\sigma = mgu(s \doteq t)$ under which the condition is true. When the condition is a relation applied to some terms, we simply split on whether the relation holds. For a condition with negation, we swap the branches: if $\neg\phi$ then P else Q is rewritten into if ϕ then Q else P. For conjunction, we nest the branches: if $\phi \wedge \psi$ then P else Q is rewritten into if ϕ then if ψ then P else Q else Q.

For releasing α information, recall that we have an α_i in each possibility that we can augment with the formula released. For sending or terminating, compared to Table 2.1 we merge the two rules so one symbolic state yields in general two symbolic states, one where we only keep the possibilities that send a message and one where we only keep those already terminated.

During the symbolic execution, if a symbolic state has an empty set of possibilities, then this state is discarded since it does not represent any ground state (e.g., in the **Send** rule, if $\mathcal{P} = \emptyset$ then the second state yielded by the rule is discarded). Moreover, if several rules are applicable at the same time, then it does not matter which one is applied first so the procedure fixes an arbitrary order for applying the rules.

The symbolic executions transform a symbolic state into a set of finished symbolic states. When all symbolic executions have terminated, we shall check whether the reached symbolic states satisfy privacy.

3.3 Intruder experiments

An *experiment* is to compare pairs of recipes and the messages they produce in every frame: in a ground state, the intruder can check whether two messages are equal in the frame *concr*. In a symbolic state, each possibility considered by the intruder contains a different simple FLIC. When doing the comparison on a FLIC, the intruder may find out equalities that must hold (constraints on privacy and intruder variables) for messages to be equal. It thus may depend on the intruder's choices of recipes whether the outcome is positive, as well as on the privacy variables. In this section, we show how to extract all these conclusions and obtain a set of symbolic states in which every experiment either gives the same result in all FLICs or different results in all FLICs. This is formalized in the following equivalence relation between recipes:

Definition 3.3.1. *Let $\mathcal{S} = (\alpha_0, \beta_0, \mathcal{P}, \_)$ be a symbolic state, where the possibilities have conditions $\phi_1, \ldots, \phi_n$ and FLICs $\mathcal{A}_1, \ldots, \mathcal{A}_n$. Let r_1 and r_2 be two recipes and $\sigma_i = mgu(\mathcal{A}_i(r_1) \doteq \mathcal{A}_i(r_2))$ for $i \in \{1, \ldots, n\}$. We define $r_1 \simeq r_2$ iff $r_1 \square r_2$ or $r_1 \bowtie r_2$,* *where*

$$\begin{aligned} r_1 \square r_2 \quad & \textit{iff} \quad \textit{for every } i \in \{1, \ldots, n\}, \ \textit{isPriv}(\sigma_i) \textit{ and } \alpha_0 \wedge \beta_0 \wedge \phi_i \models \sigma_i \ , \\ r_1 \bowtie r_2 \quad & \textit{iff} \quad \textit{for every } i \in \{1, \ldots, n\}, \ \textit{LI}(\mathcal{A}_i, \sigma_i) = \emptyset \\ & \qquad\qquad \textit{or } (\textit{isPriv}(\sigma_i) \textit{ and } \alpha_0 \wedge \beta_0 \wedge \phi_i \models \neg\sigma_i) \ . \end{aligned}$$

Intuitively, $r_1 \square r_2$ means that the two recipes produce the same message in every FLIC. Conversely, $r_1 \bowtie r_2$ means that the two recipes produce different messages in every FLIC, under any possible instantiation of the variables: either the unifier depends on at least one intruder variable but the intruder cannot solve the constraints in any way, or the unifier depends only on privacy variables and its instances are already excluded by the intruder reasoning.

Example 3.3.1. Based on Example 2.2.1, suppose that we reached a symbolic state containing two possibilities with $\phi_1 = y \doteq \mathsf{yes}$, $\phi_2 = y \not\doteq \mathsf{yes} \wedge x \not\doteq \mathsf{a}$ and

$$\begin{aligned} \mathcal{A}_1 &= +R \mapsto N.-l \mapsto \mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, N)) \ , \\ \mathcal{A}_2 &= +R \mapsto N.-l \mapsto \mathsf{crypt}(\mathsf{pk}(x), \mathsf{no}) \ . \end{aligned}$$

Here we again assume non-randomized encryption for the sake of the example. Then we have $l \bowtie \mathsf{crypt}(\mathsf{pk}(\mathsf{a}), \mathsf{no})$, because in $\mathcal{A}_1$ there is no unifier and in $\mathcal{A}_2$ the unifier $[x \mapsto \mathsf{a}]$ is excluded by ϕ_2. $\lhd$

We now define well-formed symbolic states. Among other things, all the pairs of recipes in *Checked* are experiments that have already been done by the intruder and do not distinguish the possibilities.

Definition 3.3.2 (Well-formed symbolic state)**.** *Let $\mathcal{S} = (\alpha_0, \beta_0, \mathcal{P}, \mathit{Checked})$ be a symbolic state, with the possibilities $\mathcal{P} = \{(\_, \phi_1, \mathcal{A}_1, \mathcal{X}_1, \alpha_1, \_), \ldots, (\_, \phi_n, \mathcal{A}_n, \mathcal{X}_n, \alpha_n, \_)\}$. We say that $\mathcal{S}$ is* well-formed *iff*

- *the ϕ_i are such that $\models \neg(\phi_i \wedge \phi_j)$ for $i \neq j$, $\mathit{fv}(\phi_i) \subseteq \mathit{fv}(\alpha_0) \cup \mathit{fv}(\beta_0)$ and $\alpha_0 \wedge \beta_0 \models \bigvee_{i=1}^{n} \phi_i$;*
- *the $\mathcal{A}_i$ are simple FLICs with the same labels and same recipe variables, occurring in the same order;*
- *the disequalities $\mathcal{X}_i$ are satisfiable;*
- *the α_i are such that $\mathit{fv}(\alpha_i) \subseteq \mathit{fv}(\alpha_0)$ and $\alpha_0 \wedge \beta_0 \wedge \gamma_0 \wedge \phi_i \models \alpha_i$; and*
- *for every $(l, r) \in \mathit{Checked}$, we have $l \simeq r$.*

Recipe variables can only occur in the FLICs $\mathcal{A}_i$. Since $\mathit{dom}(\mathcal{A}_1) = \cdots = \mathit{dom}(\mathcal{A}_n)$, we may write $\mathit{dom}(\mathcal{S})$ for the domain of the symbolic state.

In the rest of the chapter, we only consider well-formed symbolic states (and well-formedness is preserved by the procedure).

In general, an experiment can be done by comparing two arbitrary recipes. We will however show that it suffices to check, for every message t received by the intruder, all the ways that the intruder can produce t. Therefore, our experiments are of the form (l, r) where l is a label and $r \neq l$ is any recipe that produces the same message as label l in some FLIC. We now define a set of experiments $\mathit{Pairs}(\mathcal{S})$ that are relevant to perform: for every label l in the state and every FLIC $\mathcal{A}$, we try any other way to construct $\mathcal{A}(l)$ (except l) with the lazy intruder constraint $\mathcal{A}.{+R} \mapsto \mathcal{A}(l)$, where R is a fresh recipe variable. For each solution ρ, the recipes to compare are l and $\rho(R)$. Initially, the set *Checked* is empty, and each experiment that has been performed is added to this set.

Definition 3.3.3 (Pairs and normal symbolic state)**.** *Let $\mathcal{S} = (\_, \_, \mathcal{P}, \mathit{Checked})$ be a symbolic state with FLICs $\mathcal{A}_1, \ldots, \mathcal{A}_n$ in $\mathcal{P}$. The set of pairs of recipes to compare in $\mathcal{S}$ is*

$$\begin{aligned}\mathit{Pairs}(\mathcal{S}) = \{(l, \rho(R)) \mid l \in \mathit{dom}(\mathcal{S}), i \in \{1, \ldots, n\}, \rho \in \mathit{LI}(\mathcal{A}_i.{+}R \mapsto \mathcal{A}_i(l), \varepsilon), \rho(R) \neq l\}\\ \setminus \mathit{Checked}\ .\end{aligned}$$

We say that $\mathcal{S}$ is normal *iff $\mathcal{S}$ is finished and $\mathit{Pairs}(\mathcal{S}) = \emptyset$.*

We define a reduction relation $\rightarrowtail$ that, given a symbolic state $\mathcal{S}$ and a pair $(l, r) \in \mathit{Pairs}(\mathcal{S})$, yields a set of symbolic states after the experiment of comparing l and r, and a symbolic state that cannot be reduced further is normal. We call these experiments *compose-checks.*

Definition 3.3.4 (Compose-checks)**.** *Let $\mathcal{S}$ be a symbolic state $(\_, \beta_0, \mathcal{P}, \mathit{Checked})$, with possibilities $\mathcal{P} = \{(0, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \ldots, n\}\}$, $(l, r) \in \mathit{Pairs}(\mathcal{S})$ and $\sigma_i = \mathit{mgu}(\mathcal{A}_i(l) \doteq \mathcal{A}_i(r))$ for $i \in \{1, \ldots, n\}$. Then the set of symbolic states after the compose-check w.r.t. (l, r) is as follows.*

Privacy split *If for every $i \in \{1, \ldots, n\}$, $\mathit{isPriv}(\sigma_i)$ or $\mathit{LI}(\mathcal{A}_i, \sigma_i) = \emptyset$: Then $\mathcal{S} \rightarrowtail \{\mathcal{S}_1, \mathcal{S}_2\}$, where*

$$\begin{aligned}
\mathcal{S}_1 = \mathcal{S}[\beta_0 \leftarrow \beta_0 \wedge \bigwedge_{i=1}^{n} \left(\phi_i \Rightarrow \begin{cases}\sigma_i & \text{if } \mathit{isPriv}(\sigma_i)\\ \mathsf{false} & \text{otherwise}\end{cases}\right)\\
\mathcal{P} \leftarrow \{(0, \phi_i \wedge \sigma_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \ldots, n\}, \mathit{isPriv}(\sigma_i)\}\\
\mathit{Checked} \leftarrow \mathit{Checked} \cup \{(l, r)\}]\ ,\\
\mathcal{S}_2 = \mathcal{S}[\beta_0 \leftarrow \beta_0 \wedge \bigwedge_{i=1}^{n} \left(\phi_i \Rightarrow \begin{cases}\neg\sigma_i & \text{if } \mathit{isPriv}(\sigma_i)\\ \mathsf{true} & \text{otherwise}\end{cases}\right)\\
\mathcal{P} \leftarrow \{(0, \phi_i \wedge \neg\sigma_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \ldots, n\}, \mathit{isPriv}(\sigma_i)\}\\
\cup \{(0, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \ldots, n\}, \text{not } \mathit{isPriv}(\sigma_i)\}\\
\mathit{Checked} \leftarrow \mathit{Checked} \cup \{(l, r)\}]\ .
\end{aligned}$$

Recipe split *If there exists $i \in \{1, \ldots, n\}$ such that not $\mathit{isPriv}(\sigma_i)$ and $\mathit{LI}(\mathcal{A}_i, \sigma_i) = \{\rho_1, \ldots, \rho_k\}$ $(k \geq 1)$: Then*

$$\mathcal{S} \rightarrowtail \{\rho_1(\mathcal{S}), \ldots, \rho_k(\mathcal{S}), \mathcal{S}[\mathcal{X}_i \leftarrow \mathcal{X}_i \wedge \neg\sigma_i]\}\ .$$

Similarly to the symbolic execution, we extend the definition to relate sets of finished symbolic states: $\mathcal{C} \rightarrowtail \mathcal{C}'$ iff $\mathcal{S} \in \mathcal{C}$, $\mathcal{S} \rightarrowtail \mathcal{C}_{\mathcal{S}}$ and $\mathcal{C}' = \mathcal{C} \setminus \{\mathcal{S}\} \cup \mathcal{C}_{\mathcal{S}}$.

Note that given a pair (l, r), either **Privacy split** is applicable, or some **Recipe split** is. In the case of **Privacy split**, the outcome of the experiment is independent of the intruder choices (the unifiers between messages produced by l and r are only using privacy variables or require a choice of recipes that has already been excluded). We split into two symbolic states. The first represents the case that the outcome of the experiment is positive ($l \mathrel{\square} r$), ruling out possibility i if $\mathit{isPriv}(\sigma_i)$ does not hold. The second represents the case that the outcome is negative ($l \bowtie r$), where in possibility i, if $\mathit{isPriv}(\sigma_i)$ then the intruder now knows that $\neg\sigma_i$ holds.

In the case of **Recipe split**, for at least one FLIC $\mathcal{A}_i$, the unifier σ_i depends on what the intruder has chosen (i.e., the unifier substitutes intruder variables), and there are possible choices of recipes $\rho_1, \ldots, \rho_k$ under which the experiment may succeed (it may still depend on the privacy variables). Then we consider each of these choices in separate symbolic states, as well as the case that the intruder chose anything else for the recipes (so we have one symbolic state where we augment the disequalities formula $\mathcal{X}_i$ with $\neg\sigma_i$).

Example 3.3.2. Let us consider again Example 2.2.1, where for now we assume that encryption is not randomized. Let $\mathcal{S} = (\alpha_0, \beta_0, \mathcal{P}, \emptyset)$ be the symbolic state such that:

$$\begin{aligned}
&\alpha_0 = x \in \mathsf{Agent} \land y \in \{\mathsf{yes}, \mathsf{no}\}\ , \qquad \beta_0 = y \doteq \mathsf{yes} \lor y \not\doteq \mathsf{yes}\ ,\\
&\mathcal{P} = \{(0, y \doteq \mathsf{yes}, \mathcal{A}_1, \mathsf{true}, \mathsf{true}, []), (0, y \not\doteq \mathsf{yes}, \mathcal{A}_2, \mathsf{true}, \mathsf{true}, [])\}\ ,\\
&\mathcal{A}_1 = +R \mapsto N.-l \mapsto \mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, N))\ ,\\
&\mathcal{A}_2 = +R \mapsto N.-l \mapsto \mathsf{crypt}(\mathsf{pk}(x), \mathsf{no})\ .
\end{aligned}$$

$\mathcal{S}$ is not normal since, e.g., $(l, \mathsf{crypt}(\mathsf{pk}(\mathsf{a}), \mathsf{no})) \in \mathit{Pairs}(\mathcal{S})$. We can perform a compose-check, in this case by applying the **Privacy split** rule. In $\mathcal{A}_1$ we have to unify the messages $\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, N))$ and $\mathsf{crypt}(\mathsf{pk}(\mathsf{a}), \mathsf{no})$, which is not possible. In $\mathcal{A}_2$ we have to unify $\mathsf{crypt}(\mathsf{pk}(x), \mathsf{no})$ and $\mathsf{crypt}(\mathsf{pk}(\mathsf{a}), \mathsf{no})$, which gives the mgu $\sigma = [x \mapsto \mathsf{a}]$. Then we get two symbolic states $\mathcal{S}_1$ and $\mathcal{S}_2$, which have the same α_0 as $\mathcal{S}$ but we update β_0 and $\mathcal{P}$. Moreover, in both $\mathcal{S}_1$ and $\mathcal{S}_2$ we have $\mathit{Checked} = \{(l, \mathsf{crypt}(\mathsf{pk}(\mathsf{a}), \mathsf{no}))\}$.

$$\begin{aligned}
\text{In } \mathcal{S}_1 : \quad &\beta_0 = (y \doteq \mathsf{yes} \lor y \not\doteq \mathsf{yes}) \land (y \doteq \mathsf{yes} \Rightarrow \mathsf{false}) \land (y \not\doteq \mathsf{yes} \Rightarrow x \doteq \mathsf{a})\ ,\\
&\mathcal{P} = \{(0, y \not\doteq \mathsf{yes} \land x \doteq \mathsf{a}, \mathcal{A}_2, \mathsf{true}, \mathsf{true}, [])\}\ .\\
\text{In } \mathcal{S}_2 : \quad &\beta_0 = (y \doteq \mathsf{yes} \lor y \not\doteq \mathsf{yes}) \land (y \doteq \mathsf{yes} \Rightarrow \mathsf{true}) \land (y \not\doteq \mathsf{yes} \Rightarrow x \not\doteq \mathsf{a})\ ,\\
&\mathcal{P} = \{(0, y \doteq \mathsf{yes}, \mathcal{A}_1, \mathsf{true}, \mathsf{true}, []), (0, y \not\doteq \mathsf{yes} \land x \not\doteq \mathsf{a}, \mathcal{A}_2, \mathsf{true}, \mathsf{true}, [])\}\ . \quad \lhd
\end{aligned}$$

Using the compose-checks, we can transform a symbolic state into a set of normal symbolic states, since by definition a symbolic state is normal when there are no more pairs to compare. Moreover, the compose-checks preserve the semantics of symbolic states by partitioning the ground states represented.

Theorem 3.3.1 (Compose-check correctness)**.** *Let $\mathcal{S}$ be a finished symbolic state, $(l, r) \in$ $\mathit{Pairs}(\mathcal{S})$ and $\mathcal{S}_1, \dots, \mathcal{S}_n$ be the symbolic states such that $\mathcal{S} \rightarrowtail \{\mathcal{S}_1, \dots, \mathcal{S}_n\}$ w.r.t. (l, r). Then $[\![\mathcal{S}]\!] = \biguplus_{i=1}^{n} [\![\mathcal{S}_i]\!]$. Moreover, there does not exist any infinite sequence $(\mathcal{S}_i)_{i \geq 1}$ where for every i, there exists $\mathcal{C}_i$ such that $\mathcal{S}_i \rightarrowtail \mathcal{C}_i$ and $\mathcal{S}_{i+1} \in \mathcal{C}_i$.*

In a normal symbolic state, there are no more pairs of recipes that could distinguish the possibilities (they have all been checked). Thus, given a ground choice of recipes, all the concrete instantiations of frames are statically equivalent. This means that in a normal symbolic state, the FLICs do not contain any more insights for the intruder, and all remaining violations of (α, β)-privacy can only result from any other information β_0 that the intruder has gathered. We thus define that a symbolic state is consistent iff β_0 cannot lead to violations either.

Definition 3.3.5 (Consistent symbolic state)**.** *We say that a finished symbolic state $\mathcal{S}$ is* consistent *iff (α, β_0)-privacy holds for every $(\alpha, \beta_0, \_, \_) \in [\![\mathcal{S}]\!]$.*

Even though $[\![\mathcal{S}]\!]$ is infinite, we need to consider only finitely many (α, β_0) pairs. This is because the corresponding α and β_0 in $\mathcal{S}$ do not contain intruder variables and we only need to resolve the meta-notation if present. For truth γ, we also have only to consider finitely many instances of the privacy variables (as they range over finite domains). Our construction ensures that β_0 only contains symbols in Σ_0, and for each α and β_0, the Σ_0-models are computable as we show in Appendix A.1. While that algorithm is based on an enumeration of models as a simple means to prove we are in a decidable fragment, our prototype tool uses the SMT solver cvc5 [12] to check consistency more efficiently.[1]

[1]In a nutshell, for each possibility i, we make the following check. We first assert the formula $\alpha_0 \land \beta_0 \land \gamma_0 \land \phi_i$ where every occurrence of a variable x is replaced with a fresh variable x_γ. Then we assert the formula α

Example 3.3.3. In the symbolic state $\mathcal{S}$ from Example 3.3.2, we have $\alpha_0 = x \in \mathsf{Agent} \land y \in \{\mathsf{yes}, \mathsf{no}\}$ and $\beta_0 = y \doteq \mathsf{yes} \lor y \not\doteq \mathsf{yes}$. Since (α_0, β_0)-privacy holds, $\mathcal{S}$ is consistent.

In the symbolic state $\mathcal{S}_1$ that results from comparing recipes l and $\mathsf{crypt}(\mathsf{pk}(\mathsf{a}), \mathsf{no})$, we have the same α_0 but now $\beta_0 = y \not\doteq \mathsf{yes} \land x \doteq \mathsf{a}$. Here (α_0, β_0)-privacy does not hold anymore because β_0 rules out a model of α_0, namely $[x \mapsto \mathsf{a}, y \mapsto \mathsf{yes}]$. Thus $\mathcal{S}_1$ is not consistent. $\lhd$

To verify whether a symbolic state satisfies privacy, we perform all *compose-checks* to get a set of normal symbolic states, and then in each of these normal states it suffices to verify consistency.

Theorem 3.3.2. *Let $\mathcal{S}$ be a normal symbolic state. Then $\mathcal{S}$ satisfies privacy iff $\mathcal{S}$ is consistent.*

3.4 Algebraic properties

So far, our procedure consists in (i) executing a transaction with the rules of Table 3.2, (ii) normalizing symbolic states with intruder experiments, and (iii) checking consistency in the reached normal symbolic states. The above gives us a decision procedure for (α, β)-privacy (under a bound k on the number of transitions) as long as the intruder has no access to destructors. Note that transactions can apply destructors already. This allows for a very convenient and economical way to extend the intruder model with destructors as well without painfully extending all the above machinery to destructors: we define a set of special transactions called *destructor oracles*, one for each destructor. They receive a term and decryption key candidate, and send back the result of applying the destructor unless it fails. Note that calling these oracles does not count towards the bound on the number of transitions, but rather we apply them to a reached symbolic state until they yield no further results.

3.4.1 The supported algebraic theories

We give in Fig. 2.1 a concrete example theory, but our result can be quite easily used for many similar theories. For instance, many modelers prefer for asymmetric cryptography that private keys are defined as atomic constants and the corresponding public key is obtained by a public function pub. We like, in contrast, to start with public keys and have a private function inv to obtain the respective public key. This allows us to define a public function from agent names to public keys, which can be convenient in reasoning about privacy when the public-key infrastructure is fixed. Similarly, one may want to define further functions, in particular transparent functions like pair, i.e., functions that describe

where every $\gamma(x)$ is replaced with x_γ. Finally we assert $\forall y_1 \in dom(y_1), \dots, y_n \in dom(y_n).\ \neg\beta_0$ where y_i are the privacy variables that only occur in β_0 (i.e., the variables chosen with mode $\diamond$). There is a model satisfying all these assertions iff for some truth γ (i.e., interpretation of all privacy variables according to their domains) and some model of α, no compatible model of β_0 exists. If there is no model for these assertions, then it means that every model of α can be extended to a model of β_0, i.e., (α, β_0)-privacy holds. For instance, following Example 3.2.2 we can consider the situation where the server has sent a positive response to intruder i. This corresponds to a reachable symbolic state with the payload $\alpha_0 = x \in \mathsf{Agent} \land y \in \{\mathsf{yes}, \mathsf{no}\}$, intruder deductions $\beta_0 = x \doteq \mathsf{i} \land y \doteq \mathsf{yes}$, and a possibility with condition $\phi = x \doteq \mathsf{i} \land y \doteq \mathsf{yes}$ where the formula $x \doteq \gamma(x) \land y \doteq \gamma(y)$ has been released. Thus, the intruder has found out the name of the agent and the server's decision, but both have been released so it is not a violation of privacy. First we assert $x_\gamma \in \mathsf{Agent} \land y_\gamma \in \{\mathsf{yes}, \mathsf{no}\} \land x_\gamma \doteq \mathsf{i} \land y_\gamma \doteq \mathsf{yes}$. Then we assert $x \in \mathsf{Agent} \land y \in \{\mathsf{yes}, \mathsf{no}\} \land x \doteq x_\gamma \land y \doteq y_\gamma$. Finally, we assert $\neg(x \doteq \mathsf{i} \land y \doteq \mathsf{yes})$. There is no model satisfying all these assertions, so every model of α can be extended to a model of β_0 (there is a unique model of α and there are no variables chosen with $\mathsf{mode} = \diamond$).

message serialization and where the intruder can extract every subterm. Finally, in some cases it is convenient to model some private extractor functions when we are dealing with messages where the recipient has to perform a small guessing attack. For instance, in a protocol like Basic Hash [20] (modeled in Appendix B.2) the tag reader actually needs to try out every shared key with a tag to find out which tag it is. Rather than describing transitions that iterate over all tags and try to decrypt, it is convenient to model a private $\mathsf{extract}$ function that "magically" extracts the name of the tag, if the message is of the correct form, and returns false otherwise. This extraction must be a private function since the intruder should not be able to observe this unless they know the respective shared keys; if they do, then the experiments in our method automatically allow the intruder to perform the guessing attack. We thus distinguish three kinds of algebraic properties of destructors that can be used arbitrarily in our approach:

Definition 3.4.1 (Algebraic theory). *A* destructor rule *is a rewrite rule of one of the following forms:*

- *Decryption: $d(k, c(k', X_1, \ldots, X_n)) \to X_i$ where d is a destructor, c is a constructor, $fv(k) = fv(k')$, k, k' are destructor-free, the X_j are variables and $i \in \{1, \ldots, n\}$.*
- *Projection: $d_i(c(X_1, \ldots, X_n)) \to X_i$ where $i \in \{1, \ldots, n\}$, d_i is a public destructor called a* projector*, c is a constructor of arity n, the X_j are variables. There must be such a rule for every $i \in \{1, \ldots, n\}$ and c is then called* transparent.
- *Private extraction: $d(c(t_1, \ldots, t_n)) \to t_0$ where d is a private destructor called a* private extractor*, c is a constructor, the t_i are destructor-free and t_0 is a subterm of one of the t_i.*

Let E be a set of such rules and $=_E$ be the reflexive, symmetric and transitive closure of the rewrite relation $\to_E$ induced by E. We require the following:

- *Each destructor d occurs in exactly one rule of E.*
- *Each constructor c cannot occur both in decryption and projection rules.*
- *For every decryption rule $d(k, c(k', X_1, \ldots, X_n)) \to X_i$, we have that $k = k'$ or $k =_E f(k')$ or $k' =_E f(k)$ for some public function f.*

Let $t \downarrow_E$ denote the normal form *of term t. Define $\approx$ to be the least congruence that includes $=_E$ such that $t \approx \mathbb{ff}$ if $t \downarrow_E$ contains destructors.*

The normal form, and thus also the congruence, is well-defined because our requirements ensure the convergence of the term rewriting system E.

Lemma 3.4.1. *Let E be a term rewriting system satisfying the requirements of Definition 3.4.1. Then E is convergent.*

The requirement $k = k'$ or $k =_E f(k')$ or $k' =_E f(k)$ for some public f means that, given the decryption key k one can derive the encryption key k', or the other way around. In particular, in most asymmetric encryption schemes, the public key can be derived from the private key; for signatures the private key takes the role of the "encryption key". This requirement forces us to define in our example theory the rule $\mathsf{pubk}(\mathsf{inv}(k)) \to k$. Suppose that we omitted this rule, denying the intruder to derive the public key to a given private key. Suppose further that the intruder has received two messages $l_1 \mapsto \mathsf{inv}(\mathsf{pk}(x))$ and $l_2 \mapsto \mathsf{pk}(y)$ and is wondering whether maybe $x \doteq y$. Then they could make the experiment whether $\mathsf{dcrypt}(l_1, \mathsf{crypt}(l_2, r_1, r_2)) \approx \mathbb{ff}$ (for arbitrary recipes r_1, r_2) and this would be the case iff $x \not\approx y$. For our method, we want however to ensure that the intruder never needs to decrypt messages that they encrypted themselves. In the example, with the public-key extraction rule, the intruder can instead derive $\mathsf{pubk}(\mathsf{inv}(\mathsf{pk}(x))) \approx \mathsf{pk}(x)$ and now directly compare this with l_2. The requirement allows us to show that the intruder cannot learn anything new from decrypting terms that they have encrypted themselves.

The rewrite rules corresponding to the standard cryptographic operators of Fig. 2.1 are as follows:

$$\begin{aligned}
\mathsf{dcrypt}(\mathsf{inv}(X), \mathsf{crypt}(X, Y, Z)) &\to Y \\
\mathsf{dscrypt}(X, \mathsf{scrypt}(X, Y, Z)) &\to Y \\
\mathsf{open}(X, \mathsf{sign}(\mathsf{inv}(X), Y)) &\to Y \\
\mathsf{pubk}(\mathsf{inv}(X)) &\to X \\
\mathsf{proj}_1(\mathsf{pair}(X, Y)) &\to X \\
\mathsf{proj}_2(\mathsf{pair}(X, Y)) &\to Y
\end{aligned}$$

3.4.2 Destructor oracles

Since transactions can already apply destructors, we can model oracles that provide decryption services for the intruder, namely the intruder has to send a term to decrypt and the proposed decryption key and the oracle gives back the result of applying the destructor.

Definition 3.4.2 (Destructor oracle). *Given the decryption rule $(d(k, c(k', X_1, \ldots, X_n)) \to X_i) \in E$, its* destructor oracle *is the transaction:*

$$\mathsf{rcv}(X).\mathsf{rcv}(Y).\mathsf{try}\ Z := d(Y, X)\ \mathsf{in}\ \mathsf{snd}(Z).\mathsf{snd}(Y).0\ \mathsf{catch}\ 0\ .$$

Given the projection rules $(d_i(c(X_1, \ldots, X_n)) \to X_i) \in E$, we define a single oracle:

$$\begin{aligned}
&\mathsf{rcv}(X).\mathsf{try}\ Z_1 := d_1(X)\ \mathsf{in}\ \ldots\ \mathsf{try}\ Z_n := d_n(X)\ \mathsf{in} \\
&\mathsf{snd}(Z_1).\cdots.\mathsf{snd}(Z_n).0\ \mathsf{catch}\ 0\ \ldots\ \mathsf{catch}\ 0\ .
\end{aligned}$$

For transparent functions, there is no need for a key and for each $i \in \{1, \ldots, n\}$, the ith subterm can be retrieved with destructor d_i, so we define one oracle returning all subterms. There are no oracles for private extractors since these functions are not available to the intruder.

Obviously, the destructor oracles are redundant if the intruder has access to the destructors and also it is sound to add such transactions. Also redundant is the output $\mathsf{snd}(Y)$, because Y is already an input, but this ensures that different ways of composing the key will be considered by our compose-checks.

The reader may wonder why we do not do the same also for constructors, e.g., using transactions of the form $\mathsf{rcv}(X_1).\cdots.\mathsf{rcv}(X_n).\mathsf{snd}(c(X_1, \ldots, X_n)).0$, so we could use an intruder who neither encrypts nor decrypts and just uses oracles for both jobs. The reason is that constructors give rise to an infinite set of terms that can be produced and it is difficult to limit that—this is why we use the lazy intruder technique as a way to represent the infinitely many choices in a finite and yet complete way. For destructors on the other hand, we do not have the same problem since it is limited what we can achieve here. In particular there is no need for the intruder to destruct terms that they have constructed themselves, thus allowing us to limit the use of destructors, respectively the destructor oracles, in a simple way.

3.4.3 Analysis strategy

In general the destructor oracles are applicable without boundary. We use a strategy in which to apply them that does not lead into non-termination, but covers all applications that are necessary for any attack. Note also that the application of oracles does not count towards the bound on the number of transitions.

Definition 3.4.3 (Term marking). *All received terms and subterms in a FLIC shall be marked with one of three possible markings: $\star$ for terms that might be decrypted but have not been so far; $+$ for terms that cannot be decrypted at the given intruder knowledge for any instance of the variables; and $\checkmark$ for terms that either have already been decrypted or have been composed by the intruder themselves (so the intruder knows already the subterms that may result from a decryption).*

The default initial marking is $\star$. The exceptions are privacy and intruder variables, as well as functions that do not have a public destructor; all such terms (and subterms if they have) are marked with $\checkmark$. Markings are only changed according to Definition 3.4.4. In particular, when a variable gets instantiated, the resulting term keeps its $\checkmark$ marking.

Our strategy applies the destructor oracles to a given symbolic state $\mathcal{S}$ to obtain a finite set of analyzed symbolic states $\mathcal{S}_1, \ldots, \mathcal{S}_n$ that are together equivalent to $\mathcal{S}$ except that the FLICs are augmented with the results of decryptions (or projections), which we call *shorthands*.

Definition 3.4.4 (Analysis strategy). *Let $\mathcal{S}$ be a normal symbolic state. (Recall that in $\mathcal{S}$ all FLICs are simple, and thus intruder variables represent messages the intruder composed; and $\mathcal{S}$ is normal, i.e., all compose-checks have been made.)*

The following strategy is applied as long as there exists a label l that maps to a $\star$-marked term. Let l be the first label (in the order of the domain) that maps to a term $c(s_0, \ldots, s_n)$ which is $\star$-marked in some FLIC; note that by construction, it can only be a constructor term, since variables are marked $\checkmark$. If $c(s_0, \ldots, s_n)$ is an encryption, i.e., c occurs in a decryption rule (the intruder can decrypt iff they can produce the appropriate key), then we apply the oracle for that rewrite rule under the specialization that the recipe for X (the oracle input for the constructor term) must be the label l. If c is a transparent function, then we use the appropriate oracle that applies all its projectors and returns all subterms.

Executing the oracle transaction and performing experiments leads to a finite number of successor states $\mathcal{S}_1, \ldots, \mathcal{S}_m$ (there is at least one, so $m \geq 1$) that are again normal and have simple FLICs. In each $\mathcal{S}_i$ the decryption has either worked in every FLIC, or failed in every FLIC. We now update the marks in the $\mathcal{S}_i$ as follows.

If in $\mathcal{S}_i$ decryption has failed, assuming that c is the constructor for which we had executed the corresponding oracle, then in every FLIC where $l \mapsto c(t_0, \ldots, t_n)$ that is marked $\star$, we change to mark $+$ because it is (with the current knowledge) not decipherable. If it was already marked $\checkmark$, we do not change the label. (Note that in some FLIC, l may map to a term with a different constructor c'; if that term is marked $\star$, it maintains this marking, so that one of the next analysis steps will be to check if the respective destructor for c' can be applied.)

If in $\mathcal{S}_i$ decryption has worked, then we update and introduce markings in each FLIC as follows. In case of a decryption rule, and thus in a given FLIC, l maps to some term $c(k', t_1, \ldots, t_n)$, the result of the analysis is bound to a new label $l' \mapsto t_i$ (for some $i \in \{1, \ldots, n\}$); the decryption key is bound to new label $l'' \mapsto k$. If m_i is the mark of t_i in l, then the new occurrence of t_i at l' shall also be marked with m_i. In turn, $c(k', t_1, \ldots, t_n)$ are now all marked $\checkmark$, because they are fully analyzed. Similarly the key term $l'' \mapsto k$ and all its subterms receive the $\checkmark$ mark, because they have been produced by the intruder *already (and are thus taken from another label that is already analyzed, or composed by the intruder and thus not interesting for decryption). All terms that were marked $+$ are changed with marking $\star$, because the newly analyzed term may allow for some decryption that was impossible before. In case of projection rules, the marking is similar for the new subterms.*

We repeat this process of attempting to decrypt the first $\star$-marked term until there are no more $\star$-marks. A symbolic state is analyzed *iff it does not contain any $\star$-marked terms.*

We call a label l in a symbolic state $\mathcal{S}$ a shorthand *iff there exists a recipe r over labels before l such that $\mathcal{A}(l) \approx \mathcal{A}(r)$ for every FLIC $\mathcal{A}$ in $\mathcal{S}$.*

The analysis strategy augments FLICs only by shorthands and thus does not change what is derivable for an intruder who can decompose.

Theorem 3.4.1 (Analysis correctness). *For a symbolic state $\mathcal{S}$, the analysis strategy produces in finitely many steps a set $\{\mathcal{S}_1, \dots, \mathcal{S}_n\}$ of symbolic states that are analyzed. Further, for every ground state $S \in [\![\mathcal{S}]\!]$ there exists $S' \in [\![\mathcal{S}_i]\!]$, for some $i \in \{1, \dots, n\}$, such that S and S' are equivalent except that the frames in S' may contain further shorthands; and vice versa, for every $S' \in [\![\mathcal{S}_i]\!]$ there exists $S \in [\![\mathcal{S}]\!]$ such that S' is equivalent to S except for shorthands.*

Example 3.4.1. In the symbolic state reached after executing the transaction from Example 2.2.1, there is one FLIC that contains $-l \mapsto \mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, N), r)$ (marked $\star$) as well as another mapping $-l_0 \mapsto \mathsf{inv}(\mathsf{pk}(\mathsf{i}))$ modeling that the intruder knows their own private key. Then the strategy will execute the asymmetric decryption oracle for label l. This gives two states: $\mathcal{S}_1$ where for this possibility we unify $x \doteq \mathsf{i}$ and the intruder has a new label $-l_1 \mapsto \mathsf{pair}(\mathsf{yes}, N)$, and $\mathcal{S}_2$ where we have $x \not\doteq \mathsf{i}$ and the intruder cannot decrypt l (given the intruder knows no other private keys $\mathsf{inv}(\mathsf{pk}(\cdot))$). The encrypted message at label l is now marked $\checkmark$ in $\mathcal{S}_1$ and $+$ in $\mathcal{S}_2$. $\lhd$

Let $\mathcal{S}_0 = (\mathsf{true}, \mathsf{true}, \{(0, \mathsf{true}, [], \mathsf{true}, \mathsf{true}, [])\}, \emptyset)$ be the initial symbolic state. Given a transaction P and a finished symbolic state $\mathcal{S}$, let $start(P, \mathcal{S})$ denote the symbolic state identical to $\mathcal{S}$ but where the 0-process in every possibility of $\mathcal{S}$ is replaced with process P. Our decision procedure defines how to symbolically execute processes, perform intruder experiments and apply the analysis strategy such that, given a finished symbolic state $\mathcal{S}$ and a bound k on the number of transitions, we compute a finite set $\{\mathcal{S}_1, \dots, \mathcal{S}_n\}$ of normal, analyzed symbolic states that can be reached after executing at most k transactions (not counting the destructor oracles). In normal analyzed states, the intruder does not need any destructors anymore, because we can show that for every recipe, there exists a destructor-free one (possibly using the shorthands added during analysis), and then there are also no relevant experiments that could be made with recipes using destructors. It then suffices to verify consistency in the reached normal analyzed symbolic states to decide whether (α, β)-privacy holds. Fig. 3.2 summarizes the different steps of this procedure.

We can now conclude the correctness of our decision procedure. All the proofs are in Appendix A.3. Note that we need a bound on the number of transitions, and this bound is restricting the number of transactions that are executed. All "internal" transitions taken by our compose-checks and analysis steps do not count towards that bound.

Theorem 3.4.2 (Procedure correctness). *Given a protocol specification for (α, β)-privacy, a bound on the number of transitions and an algebraic theory allowed by Definition 3.4.1, our decision procedure is sound, complete and terminating.*

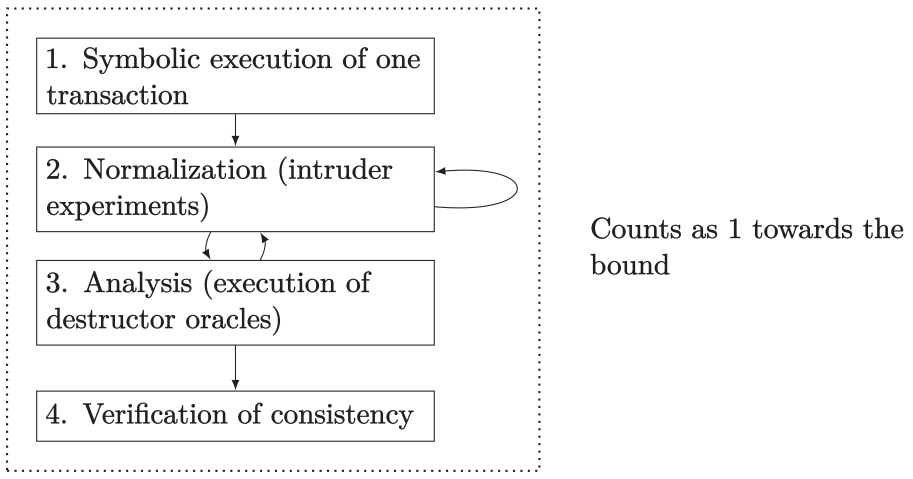

1. Symbolic execution of one transaction
2. Normalization (intruder experiments)
3. Analysis (execution of destructor oracles)
4. Verification of consistency
Counts as 1 towards the bound

Figure 3.2: Summary of the decision procedure

3.5 Case studies

We have developed a prototype tool called `noname` [39] implementing our decision procedure. The tool is a proof-of-concept showing that automation for (α, β)-privacy is achievable and practical. The user must provide as input the protocol specification, consisting of the transactions that can be executed, and a bound on the number of transactions to execute. For the cryptographic operators, we make available by default primitives for asymmetric encryption/decryption, symmetric encryption/decryption, signatures and pairing (Fig. 2.1). The user can define custom operators with the restriction to constructor/destructor theories given in Definition 3.4.1. We have also implemented an interactive running mode (the default is automatic, i.e., exploring all reachable states) in which the user is prompted whenever there are multiple successor states, so that one can manually explore the symbolic transition system.

In case there is a privacy violation, the tool provides an attack trace that includes the sequence of atomic transactions executed and steps taken by the intruder (i.e., the recipes they have chosen) to reach an attack state, as well as a countermodel proving that the privacy goals in that state do not hold, i.e., a witness that the intruder has learned more in that state than what is allowed by the payload.

As case studies, we have focused on unlinkability goals in the following protocols: our running example, Basic Hash [74], OSK [66] (which is particularly challenging as it is a stateful protocol), and several variants of BAC [57] and Private Authentication [1] (the (α, β)-privacy specifications of these last two protocols are given in [41]). For each protocol, we describe briefly our results on whether it satisfies (α, β)-privacy. The models, together with more details, are in Appendix B.

3.5.1 Running example

As explained in Example 3.2.2, if the server is replying to a compromised agent, then there is a privacy violation because the intruder is able to learn the identity of that agent, and the tool finds this attack. When permitting that the intruder learns the compromised agent's identity, the tool discovers another problem: if the agent is not compromised, then the intruder is also able to learn this fact. When releasing also that information, no more violations are found. This illustrates how the tool can help to discover all private information that is leaked, and thus either fix the protocol or permit that leak, and then finally verify that no additional information is leaked (given the bound on the number of transitions).

3.5.2 Basic Hash

In this RFID protocol, a tag sends to a reader a pair of messages containing a nonce and a MAC, using a secret key shared between tag and reader. Then the reader tries to recompute the MAC with every secret key they know to identify the tag (this behavior of the reader is modeled with a private extractor that retrieves the tag name from the MAC). We have verified that the Basic Hash protocol satisfies unlinkability, but fails to provide forward privacy [20].

3.5.3 OSK

This is also an RFID protocol with tags and readers. We have modeled two variants where, respectively, no desynchronization and one desynchronization step is tolerated. For both versions the tool finds the known linkability flaws [11].

3.5.4 BAC

This standard RFID protocol is used to read data from passports. A tag and a reader perform a challenge-response, where the tag sends a nonce and an encrypted message containing that nonce, and the reader receives both and verifies that the nonces match. The tool finds the known problems in some implementations [6, 30, 45].

3.5.5 Private Authentication

This protocol specifies communication between agents that encrypt messages using a public-key infrastructure. The initiator sends a message containing their name and a nonce, and the responder either sends back a message with a fresh nonce or sends a decoy message. We consider several variants: AF0 denotes the situation where agents always want to talk to other agents, while AF denotes the situation where agents might not want to talk to some other agents. For AF0, there is one model with a privacy violation due to insufficient release in α and another model fixing this issue; AF builds on top of the fixed AF0 and adds a binary relation to model whether agents want to talk.

3.5.6 Discussion of the results

Finding a privacy violation is usually fast, because the tool stops as soon as it finds one without exploring the rest of the transition system. Most protocols take a few seconds to analyze, but when incrementing the bound on the number of transitions we can notice a steep increase in the verification time. Indeed, in our model, transactions can always be executed so there is in general a large number of possible interleavings. The tool seems thus to be limited by the substantial size of the search space, like earlier tools for deciding equivalence such as APTE [22]. In our decision procedure, we are not deciding static equivalence between frames, but the experiments made by the intruder to try and distinguish the different possibilities seem to have a comparable complexity. For unlinkability goals, in particular, our tool and others (for bounded sessions) essentially provide similar privacy guarantees. We share the challenges and techniques such as symbolic representation of constraints for the unbounded intruder. Thus, we believe that optimizations implemented in tools such as DeepSec [26], e.g., forms of symmetries and partial order reductions, could be adapted to our procedure.

3.6 Related work

It is a striking parallel between (α, β)-privacy and equivalence-based privacy models that the vast amount of possibilities leads to very high complexity for procedures, as mentioned in, e.g., [36]. In equivalence-based approaches, the underlying problem is the static equivalence of (concrete) frames, representing two possible intruder knowledges. In (α, β)-privacy, we have instead the multi message-analysis problem: there is just one concrete frame *concr*, the observed messages, and one or more $struct_i$ that result from a symbolic execution of the transactions by the intruder, where the privacy variables are not instantiated. Each possibility has a corresponding condition ϕ_i, exactly one of which is actually true, and the intruder knows that *concr* is an instance of the corresponding $struct_i$, i.e., under the true instance of the privacy variables, $concr \sim struct_i$ for the true ϕ_i. Thus, evaluating the static equivalence can exclude several instantiations of privacy variables (even if there is just one *struct*) or rule out an entire possibility ϕ_i. The methods for solving these two problems bear many similarities, in particular one essentially in both cases looks for a pair of recipes that distinguishes the frames, i.e., the experiments that the intruder can do on their knowledge.

Like many other tools for a bounded number of sessions such as APTE [22] and DeepSec [25], we also use the symbolic representation of the lazy intruder, using variables for messages sent by the intruder that are instantiated only in a demand-driven way when solving intruder constraints, turning frames into FLICs. This makes the frame distinction problems a magnitude harder (e.g., [14]). In recipes we have to also take into account variables that represent what the intruder has sent earlier and the actual choice may allow for different experiments now. We tackle this problem by first considering a model where the intruder cannot use destructors. It suffices then to check only if any message in any $struct_i$ can be composed in a different way, which in turn can be solved with intruder constraint solving. This is the idea behind the notion of a *normal* state, i.e., where all said experiments have been done, and we can thus check if the results of the experiments exclude any model of α.

What makes the handling of destructors relatively easy is our requirement that all destructors yield a subterm or $\mathtt{ff}$, which the intruder and honest agents can observe. Thus we have no problem with "garbage terms" like decryption of a nonce. This allows us to show that it is sufficient that the intruder has applied destructors as far as possible to their knowledge using the oracles—the notion of an *analyzed* state: for any recipe that contains destructors, there is an equivalent recipe that uses the result of an oracle.

One restriction of our procedure is the class of algebraic theories we support. In the future, we would like to extend the procedure to handle more general theories. In particular, we plan to include unification modulo theories like AC to support Diffie-Hellman exponentiation. Moreover, for voting protocols it would be nice to allow arithmetic expressions in the formulas for the payload (e.g., the tally may be a sum of votes) and the information gathered by the intruder. We believe the integration with SMT solvers is a promising direction, as we can benefit from built-in support of arithmetic.

Another objective for future work is to obtain a full-fledged tool that is user-friendly, with an easier to read output, and more performant, as the prototype is not optimized.

One may wonder if a procedure for an unbounded number of transitions is possible. If we look at the equivalence-based approaches, it seems the best option for this is the notion of diff-equivalence [36, 27] as used in ProVerif [16] and Tamarin [60], or the observational and may-testing relations of [28] that are less restrictive than diff-equivalence. Roughly speaking, diff-equivalence sidesteps the problem of the intruder's uncertainty in branching by requiring that the conditions are either true in both executions or both false. This seems to correspond to the restriction in (α, β)-privacy that the intruder can always observe whether a condition was true or false, and we thus have just one $struct_i$ in each state. We are investigating whether this can allow for an unbounded-step procedure similar to ProVerif for (α, β)-privacy. Again it is a striking similarity with equivalence-based approaches that one may either need a tight bound on the number of transitions or substantial restrictions on the processes one can model.

The main difference with other tools is in the properties being verified. Our tool looks at the reachable states from an (α, β)-privacy specification of a protocol, and the privacy goals are constructed by the tool when exploring the transition system. Instead of verifying whether a number of properties hold, we thus verify whether the intruder is ever able to learn more than the information allowed (payload α). One advantage is that, in case of successful verification, we ensure that the intruder cannot learn *anything more* (about the privacy variables) than what the protocol is intentionally releasing.

One may wonder how fair the comparison between privacy in trace-equivalence models and (α, β)-privacy actually is. Gondron, Mödersheim, and Viganò [47] give an argument how trace-equivalence properties can be translated into (α, β)-privacy problems and vice-versa. Nonetheless there are several substantial differences in the models. (α, β)-privacy assumes that the intruder can always observe which transaction is executed, and may be just unclear about the concrete values like privacy variables, and which branch of a conditional is taken. In contrast, the trace-equivalence approaches are focused on a trace of messages that the intruder sent or received, thus the intruder is a priori unable to tell which position in the considered process has produced a particular output, and where a particular input was received. Thus, the intruder gets a little more information in (α, β)-privacy than typically in the model in other approaches; this can actually be often justified in practice since the intruder can know which inputs and outputs belong to the same session, and they are a substantial simplification for automated reasoning.

For some protocols such as Private Authentication, we believe that a characterization of the privacy goals with (α, β)-privacy can give a better understanding of what guarantees the protocol actually provides, as we do not see an obvious way of expressing all the privacy goals with equivalences between processes.

Chapter 4

Typing

This chapter is based on [43].

Type-flaw attacks occur when a security protocol uses several messages that have different meaning but have a similar shape so that an intruder can exploit it and send a message of one type where a message of another type is expected. For example, one message of the protocol is a signature on a nonce for challenge-response, say $\mathsf{sign}(\mathsf{inv}(\mathsf{pk}(x)), N)$, and another message is a signature on an encrypted message like $\mathsf{sign}(\mathsf{inv}(\mathsf{pk}(y)), \mathsf{crypt}(\mathsf{pk}(z), M, R))$. It is actually easy to prevent type-flaw attacks by good protocol design: agents should not sign or encrypt raw data, but rather include a few bits of information that specify the meaning of the message. In the example, the signatures should contain at least some kind of tag that distinguishes the different types of signed statements. Such a countermeasure is not only almost for free, it is completely in line with prudent engineering principles [2, 50].

Formal verification of security protocols generally gets easier if we can rule out type-flaw attacks and analyze everything in a *typed model* where the intruder is restricted to sending well-typed messages. Then, many security problems become decidable (and, e.g., one can guarantee termination of tools like ProVerif [18]).

This motivates a relative soundness result of the form: "if a protocol that obeys certain type-flaw resistance requirements has an attack, then it has a well-typed attack." It is then sound to verify such a protocol in the typed model. This is particularly relevant in practice, if many existing protocols without modification already satisfy type-flaw requirements.

Most of the existing typing results, e.g., [7, 3, 52, 51, 32], use a constraint-based method for analyzing security protocols that is based on a symbolic representation like we did with the lazy intruder in Chapter 3. We remind the reader that this technique avoids exploring all the messages that the intruder could generate at a given point, but instead uses a variable with the constraint that this variable represents any message that the intruder can generate from their current knowledge. This variable is only instantiated when the choice matters for the attack.

One can then show that in a type-flaw resistant protocol, these instantiations are always well-typed, and that all remaining variables (that do not matter for the attack in the end) can be instantiated with something well-typed as well. Thus, if an attack exists, there exists a well-typed one. Although the decision procedure obtained in Chapter 3 using the lazy intruder method works only for a bounded number of transitions, since the argument applies to an attack of arbitrary length, the typing result is not bounded to a fixed number of transitions and can be used in approaches/tools that do not use the lazy intruder (like ProVerif).

A trend in protocol verification is the support for privacy-type properties such as unlinkability or vote-secrecy, i.e., secrecy of a choice over a small domain of intruder-known values. This is challenging for verification tools and thus many tools require a restriction like diff-equivalence where, roughly speaking, conditions—and thus control flow—cannot depend on the private choice. It is thus very desirable to simplify the tools' lives by a typing result, but that is harder to obtain for privacy as well. For instance, a typing result needs to exclude that the intruder can gain any insight about a condition (and thus possibly private choices) by sending an ill-typed message. This is in fact related again to the problem of control flow (that classical diff-equivalence sidesteps): the intruder may not know in general what exactly is happening in the protocol, while in standard protocol verification the intruder is only unclear about the concrete value of some cryptographically strong secrets.

Our second contribution in this thesis is a typing result for (α, β)-privacy: "if there is an attack, then there is a well-typed one." We define a set of requirements for protocols and algebraic theories we can support, and prove that under these requirements the procedure performs only well-typed instantiations of variables and well-typed intruder experiments. As in previous typing results, this is independent of the number of transitions considered. This result is, to our knowledge, not only more general than previous typing results for privacy, since the requirements are less restrictive and a larger class of protocols is considered, but it also has a more declarative proof.

The chapter is organized as follows. In Section 4.1, we define the class of *type-flaw resistant* protocols that our typing result supports. In Section 4.2, we present the typing result for an unbounded number of transitions. In Section 4.3, we look at type-flaw resistance in case study protocols. Finally we conclude in Section 4.4 with the discussion of related work.

4.1 The typed model

We now define a simple type system and the protocol specification includes the type of every constant and variable, e.g., agent or $\mathsf{h}(\mathsf{nonce})$. This is first a mere annotation: we specify the *intended* type. The intruder is of course able to send messages of any type and an honest agent in general cannot check if a received message is of the intended type. We then develop a notion of *type-flaw resistant* protocols and show in the following section a typing result for them: if there is an attack, then there is a well-typed attack. Thus, given a type-flaw resistant protocol, it is sound to restrict the intruder model to well-typed messages.

To that end, we show that our procedure of the previous chapter will never perform an ill-typed substitution, when it is applied to a type-flaw resistant protocol. This is in fact proved for an arbitrary reachable state, i.e., our typing result holds without any bound on the number of transitions. We will make two adaptations to the procedure: we reduce the destructor applications with try to pattern matching and conditions, and we formulate analysis as built-in transitions instead of special transactions (destructor oracles); we also show that these transformations are correct.

4.1.1 Type system

Types are defined similarly to terms. Instead of a set of variables, we use a set of *atomic types*, e.g., $\{\mathsf{agent}, \mathsf{nonce}, \dots\}$. The *composed types* are defined using the functions in Σ, with the restriction to constructors of non-zero arity, i.e., we forbid destructors and constants in composed types. The type system assigns an atomic or composed type to every message with the following requirements:

Definition 4.1.1 (Typing function). *A typing function Γ is such that:*

- *$\Gamma(c)$ is atomic for every function $c \in \Sigma$ of arity* 0.
- *$\Gamma(f(t_1, \ldots, t_n)) = f(\Gamma(t_1), \ldots, \Gamma(t_n))$ for every constructor $f \in \Sigma$ of arity $n > 0$.*
- *$\Gamma(x)$ is a type (atomic or composed) for every variable $x \in \mathcal{V}$.*

Our type system does not include terms containing destructors, because they represent terms that need to be evaluated and we rather want to give a type to the result. Recall that, in a protocol specification, destructors can only occur as part of a destructor application of the form $\mathsf{try}\ Y := d(k, X)\ \mathsf{in}\ \ldots$ where either the result is $\mathtt{ff}$ and the transaction stops (for the typing result, we will require that all catch branches are empty), or Y is bound to the respective subterm of X, and thus shall have the respective (destructor-free) type.

The fact that instantiations of variables are well-typed is defined with the notion of a substitution being well-typed.

Definition 4.1.2 (Well-typed substitution). *A substitution σ is* well-typed *iff for every $x \in dom(\sigma)$, we have $\Gamma(x) = \Gamma(\sigma(x))$.*

We need to ensure that the intruder is always able to make a well-typed choice, therefore they must be able to compose arbitrarily many messages of each type, even before receiving any message from honest agents. Hence, we require that, for each *atomic* type, there is an infinite set of public constants of that type, i.e., the intruder initially knows an unbounded number of constants of each atomic type. Suppose all function symbols were public, then the intruder would also immediately have access to an unbounded number of terms of every composed type. In fact, [51] observes that, even if all functions are public, one can still model a private function f of arity n by a public function f' of arity $n+1$, where the additional argument is filled with a distinct secret constant. Thus, private functions like f are just syntactic sugar. We adopt this suggestion and, for the rest of the thesis, continue to use public and private functions, with the subset $\Sigma_{pub} \subseteq \Sigma$ to identify the public functions.

We first define the precise class of algebraic theories that our typing result supports. We consider *linear terms*, i.e., terms in which every variable occurs at most once. Compared to Definition 3.4.1, we include the requirement of linearity for the constructor terms in the rewrite rules and we exclude constants in encryption/decryption keys (they may still use private functions). This will be used when proving the typing result for state transitions (in particular in proofs relying on Definition 4.2.2).

Definition 4.1.3 (Algebraic theory for typing result). *Let E be a set of rewrite rules satisfying Definition 3.4.1. For every decryption rule $(d(k, c(k', X_1, \ldots, X_n)) \rightarrow X_i) \in E$, we require that the constructor term $c(k', X_1, \ldots, X_n)$ is linear and that neither k nor k' contains a constant. For every projection and private extraction rule $(d(c(t_1, \ldots, t_n)) \rightarrow t_0) \in E$, we require that $c(t_1, \ldots, t_n)$ is linear.*

In a protocol specification, we write type annotations with a colon, i.e., $t : \tau$ specifies that $\Gamma(t) = \tau$. We further define what it means for a protocol specification to *type check*. This does not yet include all the requirements for type-flaw resistance but simply ensures that the type annotations are consistent throughout the specification.

Definition 4.1.4 (Type checking). *For every constant c, one has to specify $\Gamma(c)$, i.e., the type of that constant. For every memory cell $\mathsf{cell}[\cdot]$, one has to specify $\Gamma(\mathsf{cell})$ which is the type of the argument for cell reads. The type annotations of constants and memory cells are global to the specification, while type checking a transaction uses local type annotations for the variables bound in that transaction. Every transaction must satisfy the following:*

- *For every choice $x \in D$, we have that D is a set of public constants of the same atomic type τ, and we then set $\Gamma(x) = \tau$.*

- *For every message received* $\mathsf{rcv}(X : \tau)$*, we have that τ is a type and we then set $\Gamma(X) = \tau$.*

- *For every destructor application* $\mathsf{try}\ Y := d(t, X)$*, consider a fresh instance of the rewrite rule for d: $d(k, c(k', X_1, \dots, X_n)) \to X_i$, where the variables in k, k' and the X_i do not have a type yet. Let $\tau_k = \Gamma(t)$. $\Gamma(X)$ must be of the form $c(\tau_{k'}, \tau_1, \dots, \tau_n)$ for some types $\tau_{k'}, \tau_i$ and there must exist types for the variables in k, k' and the X_i such that $\tau_k = \Gamma(k)$ and $\tau_{k'} = \Gamma(k')$. We then set $\Gamma(Y) = \tau_i$.*

- *For every cell read $X :=$* $\mathsf{cell}[s]$*, we have $\Gamma(s) = \Gamma(\mathsf{cell})$ and we then set $\Gamma(X) = \Gamma(C[s])$, where $C[\cdot]$ is the ground context for the initial value of* $\mathsf{cell}[\cdot]$*. For every cell write* $\mathsf{cell}[s] := t$*, we have $\Gamma(s) = \Gamma(\mathsf{cell})$ and $\Gamma(t) = \Gamma(C[s])$.*

- *For every equality $s \doteq t$ in a formula, we have $\Gamma(s) = \Gamma(t)$.*

- *For every step $\nu X_1 : \tau_1, \dots, X_k : \tau_k$, the τ_i are atomic types and we then set $\Gamma(X_i) = \tau_i$.*

In the rest of this chapter, we will only consider protocol specifications such that the type checking requirements above are satisfied.

In Definition 2.2.1, there can be a process for handling failure, e.g., sending an error message, while for the typing result we only support transactions that silently stop in case of destructor failure. For notation, we now omit writing catch 0, else 0 and trailing 0's in processes. Moreover, try in Definition 2.2.1 is part of the center process, while we now require it before branching, so that any destructor failure means that the entire transaction goes directly to 0. We also introduce additional requirements on protocols, which we use to ensure that the intruder knows the types of the messages in their knowledge and to control the shapes of messages that can occur during the protocol execution.

Definition 4.1.5 (Requirements). *Consider the tree that is induced by the conditionals of the transactions (i.e., every* if-then-else *is a node with the respective subprocesses as children). We say two execution paths are* statically distinguishable *for the intruder, iff a different number of messages are sent along the paths.*[1] *Every transaction must satisfy the following:*

1. *Every destructor application occurs before any cell read or conditional statement, and every* catch *branch is empty, i.e., it only contains the nil process.*

2. *For any two execution paths that are not statically distinguishable (and thus have the same number of sent messages), and under any instantiation of the intruder variables (including ill-typed instantiations), the ith message sent in either path has the same type.*

3. *In every cell write* $\mathsf{cell}[s] := t$*, the term t does not contain intruder variables.*

4. *When a decryption destructor is applied to a variable, this variable does not occur in other destructor applications.*

5. *If several projectors or private extractors are applied to the same variable, then the rewrite rules for these destructors are defined over the same constructor term.*

[1]One could use here a finer distinction criterion, but with a coarser relation one errs on the safe side as it excludes more protocols from being admitted.

6. *For every message sent* $\mathsf{snd}(t)$ *and every subterm* t' *of* t*, if* t' *is composed with a constructor* c *occurring in a decryption rule* $d(k, c(k', X_1, \dots, X_n)) \to X_i$*, we have that* t' *is an instance of* $c(k', X_1, \dots, X_n)$*.*

Our requirements ensure the following invariant: the intruder always knows what type every message has; so revealing the type of a message is never an issue. Requirement 1 ensures that if the intruder sends an ill-typed message such that a destructor application in the recipient transaction fails, then the recipient does nothing on this input; thus the intruder could at most learn from this transaction that the input was ill-typed. Since our approach ensures that the intruder knows the type of every message, this is completely pointless for the intruder, and an attack that contains such steps (i.e., a try that fails) can be simplified by omitting the respective transaction. Said another way, without this restriction, we would have the problem of an ill-typed attack that runs into some catch-process and that we cannot simulate by any well-typed attack.

An example for a protocol that does not satisfy Requirement 2 immediately, i.e., that messages on two paths either are statically distinguishable or have the same type, is the model of Private Authentication found in [41]: here an agent B receives a message and performs a check on it. If the check succeeds, then B sends an encrypted reply as an answer. Otherwise, B sends a random nonce as a decoy to hide whether the check succeeded. Thus there are two paths where the messages sent have different types, and indeed the point is to hide from the intruder which message was really sent. In the original model by Abadi and Fournet [1], however, B instead of a random nonce as decoy sends an encrypted message with a fresh key and random contents of the same type as the positive case. In that formulation, the protocol satisfies our requirement. The only example we can think of that would resist a similar transformation are onion-routing protocols where the intruder should not be able to tell the number of encryption layers of a given message. For protocols that do not rely on hiding the taken branch from the intruder, one can of course easily make the messages of the branches statically distinguishable and thus can also use messages of different types.

Requirement 3 is a significant restriction on cell writes, because it essentially means that we cannot update the memory with an arbitrary message sent by the intruder. Indeed, if the intruder was able to send some message to a transaction that writes this message in memory without doing any checks on it, then we could not maintain the desired invariant that the intruder always knows the types of the messages they observe.

Requirements 4 and 5 are not directly about the intruder knowing the types of the messages. We consider the requirements on the use of destructors as a reasonable restriction that ensures compatible destructor applications: whenever a variable is decomposed, we can instantiate the variable with a unique corresponding constructor term, because for this decryption there is a unique rewrite rule or for these projections/private extractions all rewrite rules are defined over the same term.

Requirement 6 on the use of constructor terms in messages sent will be useful when proving the well-typedness of analysis: if a subterm in a message sent is composed with a constructor that can be decomposed, it should be an instance of the constructor term in the corresponding rewrite rule. For instance, if we model signatures with the rewrite rule $\mathsf{open}(K, \mathsf{sign}(\mathsf{inv}(K), M)) \to M$, then signatures sent by honest agents must have a key starting with inv and cannot use a variable in this place, e.g., we do not allow sending $\mathsf{sign}(X, m)$, because $\mathsf{sign}(X, m)$ is not an instance of $\mathsf{sign}(\mathsf{inv}(K), M)$.

Using Definition 4.1.5, we can maintain the following invariant: in a reachable state, a given label maps to the same type in every possibility, i.e., the intruder always knows the types of the messages they have observed. Thus whatever recipe they use when sending a message, it corresponds to the same type in every possibility.

Lemma 4.1.1. *Let S be a reachable state in a protocol satisfying Definition 4.1.5, $struct_1, \dots, struct_n$ be the frames in S and r be a recipe over the domain of the $struct_i$. Then $\Gamma(struct_1(r)) = \dots = \Gamma(struct_n(r))$.*

4.1.2 Message patterns

To show the typing result, it is convenient to replace the try mechanism for handling destructors with pattern matching. In fact, the (α, β)-privacy semantics does not have a notion of pattern matching, because in a general untyped model, it is unclear how to define such a construct in a suitable way. However, for a specification that satisfies Definition 4.1.5, the intruder knows the type of every message, and thus also knows whether a given message will agree with a given pattern. Hence, we make a conservative extension of the receive construct with pattern matching (under the restrictions of Definition 4.1.5).

Instead of $\mathsf{rcv}(X)$ for an intruder variable X, we now allow also $\mathsf{rcv}(t)$ where t is a *linear pattern term*: it contains fresh intruder variables, where each intruder variable can only occur once, and no constants. The meaning is that the agent only accepts an incoming ground message m, if m is an instance of t and then binds the variables of t with the respective subterms of m. (This ignores how an agent would be able to check that m is an instance of t.) This is a conservative extension of the semantics on ground states (Table 2.1).

Definition 4.1.6. *We extend the* ***Receive*** *rule of Table 2.1 with the following:*

$$(\alpha, \beta_0, \gamma, \{(\mathsf{rcv}(t).P_i, \phi_i, struct_i, \delta_i) \mid i \in \{1, \dots, n\}\})$$
$$\to (\alpha, \beta_0, \gamma, \{(P'_i, \phi_i, struct_i, \delta_i) \mid i \in \{1, \dots, n\}\})$$

for every recipe r over the domain of the $struct_i$, where for every $i \in \{1, \dots, n\}$, $\sigma_i = mgu(t \doteq struct_i(r))$, and $P'_i = \begin{cases} \sigma_i(P_i) & \text{if } \sigma_i \neq \bot \\ 0 & \text{otherwise} \end{cases}$.

Note for protocols satisfying Definition 4.1.5, following Lemma 4.1.1 either the unifier exists in every possibility or every possibility goes to 0. Now we can replace try with pattern matching and a condition, because the intruder already knows the type of every message in their knowledge and thus knows whether the messages they send will have the correct structures for every destructor application to succeed. (Of course, a message with the correct structure can still fail, e.g., if it does not have the right key.) Consider, for instance,

$$\mathsf{rcv}(X).\mathsf{try}\ Y := \mathsf{dscrypt}(k, X)\ \mathsf{in}\ \mathsf{try}\ Z := \mathsf{proj}_1(Y)\ \mathsf{in}\ P\,.$$

If the intruder sends for X any term that is not of the form $\mathsf{scrypt}(K, \mathsf{pair}(M, N), R)$ (for some K, M, N and R), then the destructors are going to fail. Thus we can split the try's into a structural check that we can describe with a linear pattern like $\mathsf{scrypt}(K, \mathsf{pair}(M, N), R)$ and a condition on the pattern variables. This transformation allows us to get rid of destructors in processes entirely. The transformed example is

$$\mathsf{rcv}(\mathsf{scrypt}(K, \mathsf{pair}(M, N)), R).\mathsf{if}\ k \doteq K\ \mathsf{then}\ P[Y \mapsto \mathsf{pair}(M, N), Z \mapsto M]\,.$$

Definition 4.1.7 (Removing destructors)**.** *Let P be a transaction, from a protocol satisfying Definition 4.1.5, that contains a destructor application for decryption, i.e., $P = C[\mathsf{try}\ Y := d(t, X)\ \mathsf{in}\ P']$ for some process context $C[\cdot]$ that does not contain any destructor applications. Let $d(k, c(k', X_1, \dots, X_n)) \to X_i$ be the corresponding rewrite rule with all variables freshly renamed. Let $\sigma = [X \mapsto c(k', X_1, \dots, X_n), Y \mapsto X_i]$. Then we replace the transaction P with $\sigma(C[\mathsf{if}\ t \doteq k\ \mathsf{then}\ P'])$.*

In case of projectors or private extractors, X may appear in m destructor applications: we have try $Y^j := d^j(X)$ in $\dots$, $j \in \{1, \dots, m\}$*, and rewrite rules of the form* $d^j(c(t_1, \dots, t_n)) \to t^j$*. Then we remove all destructor applications for X since there are no keys, and we apply the substitution* $[X \mapsto c(t_1, \dots, t_n), Y^1 \mapsto t^1, \dots, Y^m \mapsto t^m]$ *to the transaction.*

This transformation is repeated until the transaction does not contain any destructor application anymore. The result is denoted P_{pat}.

Example 4.1.1. We adapt Example 2.2.1, where now the agent sends a message containing two nonces to the server and the server includes one of these nonces in their reply. We assume that we have three atomic types agent, decision and nonce. Every constant in the set Agent and the constant s are of type agent, and the constants yes, no are of type decision. We now apply the transformation to remove the destructors occurring in the following transaction P:

$$
\begin{aligned}
&\star\ x \in \mathsf{Agent}.\star\ y \in \{\mathsf{yes}, \mathsf{no}\}.\\
&\mathsf{rcv}(M : \mathsf{crypt}(\mathsf{pk}(\mathsf{agent}), \mathsf{pair}(\mathsf{nonce}, \mathsf{nonce}), \mathsf{nonce})).\\
&\mathsf{try}\ N := \mathsf{dcrypt}(\mathsf{inv}(\mathsf{pk}(\mathsf{s})), M)\ \mathsf{in}\\
&\mathsf{try}\ N_1 := \mathsf{proj}_1(N)\ \mathsf{in}\\
&\mathsf{try}\ N_2 := \mathsf{proj}_2(N)\ \mathsf{in}\\
&\mathsf{if}\ y \doteq \mathsf{yes}\ \mathsf{then}\\
&\qquad \nu r : \mathsf{nonce}.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, N_1), r))\\
&\mathsf{else}\ \nu r : \mathsf{nonce}.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{no}, N_2), r))
\end{aligned}
$$

The first step is to remove try $N :=$ dcrypt$(\dots)$ with the application of substitution $[M \mapsto \mathsf{crypt}(X, Y, Z), N \mapsto Y]$, and the second step is to remove both projections with the substitution $[Y \mapsto \mathsf{pair}(Y_1, Y_2), N_1 \mapsto Y_1, N_2 \mapsto Y_2]$. We now have P_{pat}:

$$
\begin{aligned}
&\star\ x \in \mathsf{Agent}.\star\ y \in \{\mathsf{yes}, \mathsf{no}\}.\\
&\mathsf{rcv}(\mathsf{crypt}(X : \mathsf{pk}(\mathsf{agent}), \mathsf{pair}(Y_1 : \mathsf{nonce}, Y_2 : \mathsf{nonce}), Z : \mathsf{nonce})).\\
&\mathsf{if}\ \mathsf{inv}(\mathsf{pk}(\mathsf{s})) \doteq \mathsf{inv}(X)\ \mathsf{then}\\
&\quad \mathsf{if}\ y \doteq \mathsf{yes}\ \mathsf{then}\\
&\qquad \nu r : \mathsf{nonce}.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, Y_1), r))\\
&\quad \mathsf{else}\ \nu r : \mathsf{nonce}.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{no}, Y_2), r)) \qquad \triangleleft
\end{aligned}
$$

Lemma 4.1.2. *A protocol satisfying Definition 4.1.5 and its transformation to use pattern matching according to Definition 4.1.7 yield the same set of reachable ground states (up to logical equivalence of the contained formulas α and β).*

We now define how to compute the message patterns from a protocol specification using the P_{pat} version of transactions.

Definition 4.1.8 (Protocol message patterns). *For a protocol transaction P, we define patterns(P) as the set of terms occurring in P_{pat}. For a memory cell* cell$[\cdot]$*, we define the message pattern* cell$_{pat}$ *as the message $C[X]$, where $C[\cdot]$ is the ground context for the initial value of* cell$[\cdot]$ *and X is a variable of type* Γ(cell)*, i.e., the argument type for the cell. For a protocol Spec, we define patterns(Spec) as the union of the patterns(P) for every transaction P and of the* cell$_{pat}$ *for every* cell$[\cdot]$ *in the specification (up to α-renaming of variables so they are distinct in each transaction/cell).*

Example 4.1.2. Continuing Example 4.1.1, for the messages patterns of transaction P, we consider the terms occurring in P_{pat} and this gives the following:

$$\begin{aligned}\mathit{patterns}(P) = {} & \mathsf{Agent} \cup \{x, y, r, \mathsf{yes}, \mathsf{no}, \mathsf{inv}(\mathsf{pk}(\mathsf{s})), \mathsf{inv}(X), \mathsf{crypt}(X, \mathsf{pair}(Y_1, Y_2), Z)\} \\ & \cup \{\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, Y_1), r), \mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{no}, Y_2), r)\}\end{aligned}$$

where the type of x is agent, y is decision, X is pk(agent) and r, Y_1, Y_2, Z are nonce. $\lhd$

4.1.3 Type-flaw resistance

The core part in the proof of our typing result is that variables can always be instantiated with messages of the same type. We first define the set of sub-message patterns, which includes all subterms, well-typed instantiations and key terms. To prove our result we will use the fact that every message in the symbolic execution of the protocol is in this set of sub-message patterns.

Definition 4.1.9 (Sub-message patterns). *The set of* sub-message patterns*, $\mathit{SMP}(M)$, of a set of terms M is the least set closed under the following rules:*

1. *If $t \in M$, then $t \in \mathit{SMP}(M)$.*
2. *If $t \in \mathit{SMP}(M)$ and t' is a subterm of t, then $t' \in \mathit{SMP}(M)$.*
3. *If $t \in \mathit{SMP}(M)$ and σ is a well-typed substitution, then $\sigma(t) \in \mathit{SMP}(M)$.*
4. *If $t \in \mathit{SMP}(M)$, k and t' are terms such that for some destructor d we have $d(k, t) \to t'$ as an instance of the rewrite rule for d, then $k \in \mathit{SMP}(M)$.*

With rule 4, we ensure that relevant decryption keys are in $\mathit{SMP}(M)$, because they may occur in the symbolic constraints when performing analysis steps.

We have now everything in place to formally define type-flaw resistance, which ensures that composed messages of different types cannot be unified.

Definition 4.1.10 (Type-flaw resistance). *A set of terms M is* type-flaw resistant *iff for all $s, t \in \mathit{SMP}(M) \setminus \mathcal{V}$ we have that $\Gamma(s) = \Gamma(t)$ if s and t are unifiable.*

A protocol Spec is type-flaw resistant *iff it satisfies Definition 4.1.5 and the set $\mathit{patterns}(\mathit{Spec})$ is type-flaw resistant.*

Example 4.1.3. The protocol from Example 4.1.2 is not type-flaw resistant, because in the message patterns we have the input pattern $\mathsf{crypt}(X, \mathsf{pair}(Y_1, Y_2), Z)$ and an output pattern $\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, Y_1), r)$, which can be unified even though they have different types: the mgu $[X \mapsto \mathsf{pk}(x), Y_1 \mapsto \mathsf{yes}, Y_2 \mapsto \mathsf{yes}, Z \mapsto r]$ is not well-typed since $\Gamma(Y_i) \neq \Gamma(\mathsf{yes})$.

One can make the protocol type-flaw resistant by using *formats*, for instance by replacing the function pair with f_1 in the input and with f_2 in the outputs, where the f_i are transparent functions (this requires also replacing the projectors in the process). $\lhd$

4.2 Typing result

4.2.1 Well-typedness of the constraint solving

We now turn back to the lazy intruder rules (Table 3.1) and show that, for a type-flaw resistant protocol, none of the rules perform ill-typed instantiations of intruder variables. In particular, recall that for the **Unification** rule, we have to unify two terms s and t

which are not variables. If the protocol is type-flaw resistant and s and t are unifiable, then they must have the same type and their most-general unifier thus by well-typed.

Using type-flaw resistance, we can show that every lazy intruder rule preserves the well-typedness of the substitution σ of variables performed in previous lazy intruder reduction steps. Let $\mathit{terms}(\mathcal{A}) = \{t \mid -l \mapsto t \in \mathcal{A} \text{ or } +R \mapsto t \in \mathcal{A}\}$ be the set of terms occurring in a FLIC $\mathcal{A}$. We extend the notion of well-typed substitutions to well-typed choices of recipes.

Definition 4.2.1 (Well-typed choice of recipes). *Let $\mathcal{A}$ be a simple FLIC and ρ be a choice of recipes for $\mathcal{A}$. We say that ρ is* well-typed w.r.t. *$\mathcal{A}$ iff for every $+R \mapsto X \in \mathcal{A}$, we have $\Gamma(X) = \Gamma(\rho(\mathcal{A})(\rho(R)))$.*

We can now conclude that the lazy intruder results are doing only well-typed instantiations.

Theorem 4.2.1 (Lazy intruder well-typedness). *Let Spec be a type-flaw resistant protocol, $\mathcal{A}$ be a simple FLIC such that $\mathit{terms}(\mathcal{A}) \subseteq \mathit{SMP}(\mathit{patterns}(\mathit{Spec}))$ and let σ be a well-typed substitution. Then every $\rho \in \mathit{LI}(\mathcal{A}, \sigma)$ is well-typed w.r.t. $\mathcal{A}$.*

4.2.2 Well-typedness of state transitions

We have defined in Table 3.2 the relation $\Rightarrow$ that models the execution of processes for symbolic states. Since we have made an extension for pattern matching for ground states, we now define how to handle this construct for symbolic states, extending the relation $\Rightarrow$ and proving correctness for this case. To that end, we use the lazy intruder to consider all choices of recipes producing a linear pattern: the intruder can either use a label that produces a message of the same type, or compose the pattern themselves. We want to ensure that if a label is a solution in one FLIC, then it is a solution in every FLIC. This is why the linearity requirement in rewrite rules is crucial, since the type information cannot distinguish variables of the same type. For instance, if a message $\mathsf{rcv}(f(X, X))$ was expected, it might be that in one FLIC a label l maps to $f(t, t)$ for some message t of type τ, and in another FLIC the label l maps to $f(t, s)$ where $s \not\approx t$ but s is still of type τ. Then the choice of using label l for receiving this message would be a solution in the first FLIC but not the second. We instead consider only linear patterns, so in the example we might have $\mathsf{rcv}(f(X, Y))$ (the transaction could still check whether $X \doteq Y$ after the receive). Similarly, we forbid constants in patterns because otherwise we cannot, using only the type information, know which value matches a pattern.

Definition 4.2.2 (Receiving message patterns). *We extend the semantics of receive steps to support linear patterns, with the following transition if $t \notin \mathcal{V}_{\mathit{intruder}}$:*

$$\begin{aligned}
&(\alpha_0, \beta_0, \{(\mathsf{rcv}(t).P_i, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \ldots, n\}\}, \mathit{Checked})\\
&\Rightarrow \{\rho((\alpha_0, \beta_0, \{(P_i, \phi_i, \mathcal{A}_i.{+R} \mapsto X, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \ldots, n\}\}, \mathit{Checked}))\\
&\qquad \mid \rho \in \mathit{LI}(\mathcal{A}_1.{+R} \mapsto X, [X \mapsto t]), \textit{ where } R \textit{ is a fresh recipe variable and}\\
&\qquad X \textit{ is a fresh intruder variable}\}\\
&\quad \cup \{(\alpha_0, \beta_0, \{(0, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \ldots, n\}\}, \mathit{Checked})\}
\end{aligned}$$

*In case $t \in \mathcal{V}_{\mathit{intruder}}$, the transition is the same as **Receive** in Table 3.2.*

Example 4.2.1. Continuing Examples 4.1.1 and 4.1.3, the transaction P_{pat} starts by receiving the linear pattern $\mathsf{rcv}(\mathsf{crypt}(X, \mathsf{f}_1(Y_1, Y_2), Z))$. The intruder may compose that message themselves with the recipe $\mathsf{crypt}(R_X, \mathsf{f}_1(R_{Y_1}, R_{Y_2}), R_Z)$. Another solution, assuming they have observed earlier a message $-l \mapsto \mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{f}_1(\mathsf{n}_1, \mathsf{n}_2), \mathsf{r})$ sent by an honest agent, is to use the label l to instantiate the pattern (if there are multiple possibilities, this label could map in other FLICs to other messages of the same type, e.g., where the nonces are different, and the substitutions are done accordingly in each process). $\lhd$

This conservative extension means that even when receiving patterns, we keep FLICs simple, and the representation with symbolic states is still correct when using pattern matching.

Lemma 4.2.1. *Given a type-flaw resistant specification, then the set of reachable states in the symbolic semantics represents exactly the reachable states of the ground semantics.*

The lazy intruder is used in two ways for the experiments: (i) to compute recipes that can be compared to labels, and (ii) to solve constraints whenever the outcome of an experiment depends on messages sent earlier. Since we have already shown that the lazy intruder results are well-typed, we have the guarantee that in a experiment with a pair (l, r), the label l and the recipe r produce messages of the same type and all transitions to determine the outcome of an experiment are only doing well-typed instantiations. Thus, there is nothing more to show for intruder experiments.

It remains to show that analysis also never introduces ill-typed instantiations. In a normal symbolic state, the intruder can perform analysis by decrypting messages in their knowledge, if they know the appropriate key. The analysis is always done in *normal* states, i.e., after the experiments. In Definition 3.4.4, the analysis is performed through *destructor oracles*, which are defined as transactions available to the intruder. These destructor oracles do not work directly with the typing result: since they are defined as transactions, the computation of the sub-message patterns set SMP would need to include the patterns from the destructor oracles. This prevents us from achieving type-flaw resistance even for reasonable protocols and when formats are used. By *formats* we refer to transparent functions that are used to express the meaning of messages. This is a slight generalization of the usual tagging schemes where the messages of different meaning contain a tag to tell them apart; rather formats like $\mathsf{f}_1, \mathsf{f}_2$ represent an arbitrary way to implement the message formats (e.g., XML or JSON) and we just assume that they are unambiguous and pairwise disjoint [62]. For instance, consider a protocol that uses several times crypt but with contents of different types, e.g., $\mathsf{crypt}(\mathsf{pk}(\mathsf{agent}), \mathsf{f}_1(\mathsf{agent}), \mathsf{nonce})$ and $\mathsf{crypt}(\mathsf{pk}(\mathsf{agent}), \mathsf{f}_2(\mathsf{nonce}), \mathsf{nonce})$. To compute the message patterns, we have to consider the transformed destructor oracle that uses pattern matching instead of try; for dcrypt, this would yield the transaction $\mathsf{rcv}(\mathsf{crypt}(X, Y, Z)).\mathsf{rcv}(K).\mathsf{if}\ K \doteq \mathsf{inv}(X)\ \mathsf{then}\ \mathsf{snd}(Y).\mathsf{snd}(K)$. We have the pattern $\mathsf{crypt}(X, Y, Z)$, because the decryption does not care about the actual content of the message but just about whether the key is correct. If we assume that there are multiple instances of this transaction where only the type annotations change (to cover all possible types), we would have $\mathsf{crypt}(X_1, Y_1, Z_1)$ and $\mathsf{crypt}(X_2, Y_2, Z_2)$ in SMP, with for instance $\Gamma(X_i) = \mathsf{pk}(\mathsf{agent})$, $\Gamma(Z_i) = \mathsf{nonce}$, $\Gamma(Y_1) = \mathsf{f}_1(\mathsf{agent})$ and $\Gamma(Y_2) = \mathsf{f}_2(\mathsf{nonce})$. These two message patterns are unifiable but have different types, so type-flaw resistance is not achieved.

However, the procedure does not unconditionally apply destructor oracles but always restricts the step $\mathsf{rcv}(X)$ to using a label l as recipe for message X, where l maps to a message composed with the top-level constructor corresponding to the oracle. Therefore, we can be more precise and specialize the processes coming out of the destructor oracles: instead of a general pattern like $\mathsf{crypt}(X, Y, Z)$, we only consider instances of that pattern with messages that the intruder has observed, e.g., $\mathsf{crypt}(\mathsf{pk}(\mathsf{a}), \mathsf{f}_1(A), R)$.

We define analysis steps as part of the transition system instead of special transactions. We will show that, for a type-flaw resistant protocol, this alternative way of performing analysis is equivalent to using destructor oracles. The benefit of this formulation of analysis is that we ensure all messages are instances of the protocol message patterns, and thus we can obtain the typing result.

Definition 4.2.3 (Analysis transition)**.** *Let $\mathcal{S}$ be a reachable symbolic state in a type-flaw resistant protocol. The transition for analysis is:*

$$\begin{aligned}&(\alpha_0, \beta_0, \{(0, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \mathit{Checked})\\ &\Rightarrow^{\bullet}(\alpha_0, \beta_0, \{(P_i, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \mathit{Checked})\end{aligned}$$

if $\mathcal{S}$ is normal and there exist a label $l \in dom(\mathcal{S})$ and a public destructor $d \in \Sigma_{pub}$ such that l may be analyzed with d, i.e., for every $i \in \{1, \dots, n\}$, $-l \mapsto c(k_i', t_i^1, \dots, t_i^m) \in \mathcal{A}_i$ where $d(k_i, c(k_i', t_i^1, \dots, t_i^m)) \to t_i^j$ (for some $j \in \{1, \dots, m\}$) is an instance of the rewrite rule for d and for every $i \in \{1, \dots, n\}$, let $P_i = \mathsf{rcv}(Y).\mathsf{if}\ Y \doteq k_i\ \mathsf{then}\ \mathsf{snd}(t_i^j).\mathsf{snd}(k_i)$. In case c is transparent, we define $P_i = \mathsf{snd}(t_i^1).\cdots.\mathsf{snd}(t_i^m)$.

Note that the processes P_i that we put in each possibility are exactly the instances of the corresponding destructor oracle, after transformation to pattern matching and substitution of the message to analyze with the respective message that the label maps to in each FLIC.

Example 4.2.2. Continuing Examples 4.1.1 and 4.1.3, suppose the intruder is an agent with their own private key, they have sent the message $\mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{f}_1(X_1, X_2), X_3)$ to the server and received a reply. The intruder knowledge is represented with two possibilities, where one contains the following FLIC:

$$-l_1 \mapsto \mathsf{inv}(\mathsf{pk}(\mathsf{i})).{+R_1} \mapsto X_1.{+R_2} \mapsto X_2.{+R_3} \mapsto X_3.{-l_2} \mapsto \mathsf{crypt}(\mathsf{pk}(x), \mathsf{f}_2(\mathsf{yes}, X_1), \mathsf{r}_1)\ .$$

The other possibility contains a similar FLIC with $\mathsf{f}_2(\mathsf{no}, X_2)$ in the reply. Applying the transition for analysis means that we execute the transaction $\mathsf{rcv}(Y).\mathsf{if}\ Y \doteq \mathsf{inv}(\mathsf{pk}(x))\ \mathsf{then}\ \mathsf{snd}(\mathsf{f}_2(\mathsf{yes}, X_1)).\mathsf{snd}(\mathsf{inv}(\mathsf{pk}(x))$ (respectively $\mathsf{snd}(\mathsf{f}_2(\mathsf{no}, X_2))$). Assuming the intruder does not know any other private key, the lazy intruder would return the label l_1 for instantiating the message Y, which means $Y = \mathsf{inv}(\mathsf{pk}(\mathsf{i}))$.

This yields two symbolic states, one in which decryption succeeded and one in which it failed. If it succeeded, the intruder would learn $x \doteq \mathsf{i}$ and receive the decrypted pair; projecting the pair, the intruder would learn whether $y \doteq \mathsf{yes}$. If it failed, they would learn $x \neq \mathsf{i}$ but nothing about y. ◁

We now consider two transition relations. $\Longrightarrow$ is a relation on finished symbolic states induced by the transitions of Table 3.2 for executing a transaction with the evaluation of the process and by the normalization through experiments, where the transactions include the destructor oracles. More formally, $\mathcal{S} \Longrightarrow \mathcal{S}'$ iff there is a transaction P (including destructor oracles) such that $\{\mathit{start}(P, \mathcal{S})\} \Rightarrow^{*}\rightarrowtail^{*} \mathcal{C}$ and $\mathcal{S}' \in \mathcal{C}$. In contrast, we define now a relation $\Longrightarrow^{\bullet}$ that replaces the destructor oracles with the analysis transitions: $\mathcal{S} \Longrightarrow^{\bullet} \mathcal{S}'$ iff either (there is a transaction P (excluding destructor oracles) such that $\{\mathit{start}(P, \mathcal{S})\} \Rightarrow^{*}\rightarrowtail^{*} \mathcal{C}$ and $\mathcal{S}' \in \mathcal{C}$) or ($\mathcal{S} \Rightarrow^{\bullet} \mathcal{S}_a$, $\{\mathcal{S}_a\} \Rightarrow^{*}\rightarrowtail^{*} \mathcal{C}$ and $\mathcal{S}' \in \mathcal{C}$).[2]

We have made two changes to the procedure of Section 3.4: first, we have replaced explicit destructor applications with pattern matching (Definitions 4.1.7 and 4.2.2) and second, we have replaced destructor oracles with analysis transitions (Definition 4.2.3).

[2]In this form, both transition systems admit infinite sequences of analysis steps (e.g., attempting repeatedly to decrypt the same message), but we showed that there is a terminating strategy to saturate the intruder knowledge after every transaction. This strategy can here be applied in the same way and is orthogonal to our typing result.

In Lemmas 4.1.2 and 4.2.1, we have already shown that, for type-flaw resistant protocols, using pattern matching instead of explicit destructor applications is correct. Thus, for every transaction P in the protocol specification, we now consider that the transaction P_{pat} is executed instead of P. That way, we ensure that the messages in the symbolic constraints are always in the set of sub-message patterns SMP of the protocol. For type-flaw resistant protocols, we can show that the analysis transitions are equivalent to destructor oracles, and thus the two transition relations are actually the same: $\Longrightarrow = \Longrightarrow^{\bullet}$. Moreover, for type-flaw resistant protocols, all instantiations performed by the relation $\Longrightarrow^{\bullet}$ are well-typed. Hence, we conclude and obtain our main typing result, which holds for an unbounded number of transitions.

Theorem 4.2.2 (Typing result). *Given a type-flaw resistant protocol, it is correct to restrict the intruder model to well-typed recipes/messages for verifying privacy.*

4.3 Case studies

We consider again the protocols already studied in Section 3.5, namely: our running example, Basic Hash, OSK, BAC and Private Authentication. For each protocol, we explain how to achieve the type-flaw resistance requirements or why that is not possible in a reasonable way in case of the OSK protocol.

4.3.1 Running example

We explained how to make the protocol type-flaw resistant using formats in Example 4.1.3.

4.3.2 Basic Hash

Basic Hash is type-flaw resistant, where for the type annotations, we consider that we have the following atomic types: tag, used for the names of the tags and the privacy variable representing some tag name; nonce, used for the fresh number created by the tag; and ok, used for the reply from the reader when identification succeeds.

4.3.3 OSK

This protocol is out of the scope of our typing result. In OSK, similarly to Basic Hash, a reader tries to identify a tag. However, in OSK, both the tag and the reader use memory cells as ratchets (initialized with a shared secret), instead of a MAC. The processes contain steps like $S := \mathsf{cell}[x].\mathsf{cell}[x] := \mathsf{h}(S)$ representing a turn of the ratchet with the application of a hash function, and thus the updates change the type of the content stored in memory, which is not allowed by Definition 4.1.4.

4.3.4 BAC

In the model from [41], there is a non-empty catch branch and thus it violates our typing requirements. However, the interesting aspect of the try in this case—namely that it can reveal whether it is the right agent—is independent of whether it is an encryption in the first place (which the intruder knows). Thus with the pattern matching notation introduced in Section 4.1, we can equivalently formulate this as a pattern match and an if condition with a non-empty else branch and achieve the requirements of type-flaw resistance.[3]

[3]Currently the noname tool does not support the pattern matching notation, but it can be simulated using private extractors.

4.3.5 Private Authentication

The models of [41] violate our typing requirements in three regards. First, the decoy message is a fresh nonce, while a normal reply is an encrypted message. It is intended that the intruder in general cannot tell which one is the case, violating our requirement that in this case the messages must have the same type. However, the original model from [1] actually ensures that the decoy message is of the same type as the regular message: it is an encryption of a fresh nonce with a fresh key. Following this, the requirement is actually met, as the intruder now in each case knows the type of each message (just not whether its content and key are decoys or regular). Second, there are non-empty catch branches which however can now be solved using our pattern matching notation as in the case of BAC. Third, the message from the initiator and the reply from the responder are unifiable but do not have the same type. This is a type-flaw similar to Example 4.1.3, and we can use formats to solve this third issue and thus achieve type-flaw resistance.

For BAC and Private Authentication, we have been able to solve the issue of non-empty catch branches by using pattern matching. Thus, one may wonder if we could not do that in general and drop some restrictions on our typing result. In fact here is an example that we would not be able to transform to pattern matching: $\mathsf{rcv}(X).\mathsf{try}\ Y := \mathsf{dscrypt}(k, X)\ \mathsf{in}\ \mathsf{snd}(\mathsf{h}(Y))\ \mathsf{catch}\ \mathsf{snd}(\mathsf{sign}(\mathsf{inv}(pk), X))$, where the message in the catch is a signed (error) message on the input X. Even if the intruder knows a priori that a particular message is not decipherable, obtaining the signature on it may be relevant in an attack that cannot be done in a well-typed way.

4.4 Related work

There are in our view four benefits to typing results: robust engineering, efficiency, decidability, and simplifying interactive theorem proving. First, *robust engineering*: we spend a few extra bits (if not already present) to explicitly say what messages *mean* and thereby "solve" type-flaw attacks. In fact, the intruder can still take an encrypted message and send it in a place where a nonce is expected (thus still sending an "ill-typed" message), but due to the clear annotation of the meaning, every honest agent will always treat this bitstring as a nonce and never try to decrypt it. Hence, if there is an attack, the same attack would work if the intruder had indeed sent a random nonce, and it is thus sound to consider an intruder model with only well-typed messages.

This leads to *efficiency.* The first typing result was by Heather, Lowe, and Schneider [50] and supports the Casper tool based on the model checker FDR2 to explore the state space. This requires bounds on the number of steps honest agents and the intruder can perform; restricting the intruder to well-typed messages drastically cuts down the search space. Similarly, the model checker SATMC of the AVISPA Tool and AVANTSSAR Platform requires a typed model [8, 10, 9]. The result of Heather, Lowe, and Schneider [50] and several that followed are based on inserting tags into messages. This has a disadvantage when we consider existing protocols, say TLS, that do use some tagging but do not follow the precise tagging scheme of the typing result in question—then that result is simply not applicable. We follow the approach of Hess and Mödersheim [51] and model the concrete formatting of messages in a protocol implementation by using transparent functions, where different functions represent disjoint formats in the implementation. This style of typed model is compatible, e.g., with TLS 1.2 [68]. Several papers have shown how to apply typing results to larger classes of protocols and properties, e.g., Arapinis and Duflot [7] show how to extend beyond secrecy goals, and Hess and Mödersheim [51] how to extend to stateful protocols.

For several typing results, including the present one, the proof is based on a constraint-based representation of protocol executions. The core of the proof is to show that the constraint-solving procedure for the lazy intruder constraints never performs an ill-typed substitution when applied to a constraint that originates from a type-flaw resistant protocol. Originally, the lazy intruder was however devised not as a proof technique but as a symbolic model-checking technique, namely in tools like OFMC and CL-Atse of AVISPA and AVANTSSAR [13, 64, 72, 10, 9], and the `noname` tool [39] for (α, β)-privacy implementing our decision procedure. It is in the nature of the matter that these tools, for a type-flaw resistant protocol, will not consider any ill-typed messages, so the restriction to a typed model does not further cut down the size of the state space they explore.

The mentioned model-checking approaches are concerned with bounded number of steps of the honest agents. However, for verifying protocol security without such bounds, one of the most popular tools is ProVerif [16], based on abstract interpretation, basically abstracting the fresh messages into a coarser set of abstract values, while maintaining unbounded steps of both honest agents and intruder. This is in general still undecidable, but with a typed model, it becomes decidable as shown by Blanchet and Podelski [18]: essentially, we will have finitely many equivalence classes and thereby a finite set of well-typed messages that can occur in the saturation of Horn clauses that represent "what can happen". A similar tool based on abstract interpretation is PSPSP [54], which relies on a typed model and computes a finite fixpoint for stateful protocols. While the abstraction is in general an over-approximation, PSPSP implements a decision procedure for the resulting abstraction under a typed model.

There are currently no tools and methods for (α, β)-privacy that perform verification for an unbounded number of sessions; therefore we currently cannot demonstrate how a typed model can help here and possibly allow for a decision procedure here, as well, but this seems very likely.

Finally, concerning interactive theorem proving, the first results in Isabelle/HOL by Paulson [67] in fact use a typed model (without any typing result). It underlines how the typed model allows for easier reasoning than dealing with ill-typed messages in manual proofs. Similarly, the compositionality result of Hess, Mödersheim, and Brucker [53] in Isabelle relies on typed model. Our compositionality result for (α, β)-privacy in Chapter 5 also largely benefits from a typed model.

A major challenge, and in fact the focus of our second contribution, is to give a typing result for privacy-type properties, where the most common approaches work with models based on trace-equivalence. [32, 31] are, to our knowledge, the only major results for this question, and thus also the related work closest to ours. Since our approach is based on (α, β)-privacy, it plays a quite different game but results in [47, §V] suggest that the two notions have similar expressive power.

Our work is more general than [32, 31] in the following three regards. First, they require that protocols are deterministic and they do not support if-then-else branching. In contrast, we allow non-deterministic choice of privacy variables by honest agents and if-then-else with conditions that can refer to all messages in scope (including privacy variables). This generalization is significant because it allows for protocols where the privacy also depends on the control flow, e.g., where the intruder does not know whether a recipient accepted a message (and sent a legitimate answer) or not (and sent a decoy answer). Note also that a common restriction for verification tools is the notion of diff-equivalence which (at least in its original form) forbids dependence on conditions.

A second generalization is the handling of constructors and destructors. [31] does not model destructors in the processes (only in the intruder model) and rather obtains decryption by pattern matching. We instead support the explicit application of destructors by honest agents that (α, β)-privacy uses in try-catch statements, where we only require the catch branch to be the nil process, i.e., honest agents just abort when decryption fails. We in fact turn this into a pattern-matching problem, but it is part of the method (and its soundness proof) rather than being part of the model. Note that we assume that failure of a destructor is detectable; this is significant as an intruder may learn something from this failure. It seems reasonable to assume for the constructor/destructor theories supported here as most standard cryptographic implementations of primitives like AES and RSA indeed reveal if decryption failed. An interesting question is how to handle more algebraic properties like those of exponentiation with inverses that does not allow to detect failure to "decrypt" in general. However, such algebraic properties are not supported by any of the mentioned typing results.

A final generalization is that our approach supports protocols with long-term state (the memory cells). An interesting aspect of this is that there are several results concerning decidability based on typing and bounding the number of fresh nonces; one may wonder if this is also applicable in our case. However, there is an obstacle since our argument requires an infinite supply of constants of all types for the intruder to solve the disequalities that arise, among other things, from handling the long-term state. We thus leave this question to future work.

Another closely related problem is that of compositionality, which is also regarded as a relative soundness result: given that several protocols are secure in isolation, can we show that also their composition is secure? It works indeed also by similar methods, namely transforming an attack against the composed system to an attack against one component. Since here one of the key problems is when the attacker can use messages from one component in another component, and a solution can be similarly some form of tagging, one could call compositionality a form of "typing" for a family of protocols. In fact, typing can thus be a stepping stone for a compositionality result [51] and we investigate this for (α, β)-privacy in Chapter 5.

Chapter 5

Compositionality

This chapter is based on [42].

Using a communication medium like the Internet where a variety of protocols run in parallel, sharing a public-key infrastructure, begs the question of protocol composition: whether any attacks can arise even if each protocol in isolation is secure. It is challenging to verify the composition directly, not the least because any new protocol and any protocol update would require one to start verification from scratch. There are compositionality results that show: this composition is secure, if the messages of the component protocols are sufficiently disjoint [49], which can be achieved, e.g., using tags [34, 33, 7].

Beyond protocols that only share network and public-key infrastructure, there are some compositionality results that allow for some interaction between the component protocols, e.g., one protocol P_1 negotiates a key that is then used by another protocol P_2. There needs to be an interface between the components; for instance, P_1 gives guarantees of secrecy, authentication, and freshness of the negotiated keys, and P_2 may guarantee that it does not leak those keys to another party. This also allows one to verify that components can be replaced with ones that offer the same interface to the other protocol: for P_2 it does not matter how exactly P_1 achieves its goals and vice-versa.

For non-privacy properties like secrecy and authentication that can easily be expressed as trace properties, the compositionality argument consists in demonstrating that any attack trace against the composed protocol can be split into traces for the component protocols in isolation such that the goals of at least one component are violated. This reasoning is substantially more complicated when considering privacy properties; this is especially true for the common approaches to privacy based on trace-equivalence notions. Thus, there exist but a few compositionality results for privacy-type goals and they have rather restrictive assumptions; this is discussed in detail in the final section.

We now investigate how to obtain a compositionality result based on (α, β)-privacy. Since (α, β)-privacy allows for turning privacy into a reachability problem without losing the expressive power of equivalence-based notions, we can thus resort to the standard approach for compositional reasoning, i.e., showing that an attack trace against the composed protocol can be mapped into traces for the component protocols, and in fact obtain a general result:

- The component protocols do not need to be disjoint: they can have a set of shared secrets, and these secrets can also be declassified in the course of the protocol.
- A component protocol can be used as a subprotocol called by another protocol, e.g., to query a key server and proceed with the result from the key server.
- The component protocols can include long-term state (that lasts beyond a single

session) and this long-term state can also be shared.

In this chapter, we assume that the cryptographic operators are modeled with constructors and destructors according to Definition 3.4.1. Moreover, we assume a typed model, where the intruder is only sending messages of the correct types. This makes the reasoning about attack traces much easier and typing results for similar protocol models show that this is without loss of attacks for well-tagged protocols [7, 52, 51, 31, 53, 43]. Note that in this chapter the fact that all messages are well-typed is an assumption. The result of Chapter 4 is one way to ensure that this assumption is sound (for type-flaw resistant protocols) and there may be other ways to achieve this.

Our third contribution in this thesis is a compositionality result for (α, β)-privacy: we start by extending (α, β)-privacy to generalize the class of protocols that can be modeled, with new constructs useful for protocol composition, and we identify requirements for *composable protocols*, which allow for the modular verification of privacy goals with an "assume-guarantee" approach, i.e., we verify one component with the interface of the other components.

The chapter is organized as follows. In Section 5.1, we introduce protocol composition through an example based on Needham-Schroeder. In Section 5.2, we define some extensions of the protocol specifications and their semantics. In Section 5.3, we define the class of composable protocols that can be composed securely. In Section 5.4, we present our compositionality result. In Section 5.5, we summarize how to apply our result and discuss some limitations. Finally we conclude in Section 5.6 with the discussion of related work. All the proofs and intermediate results are provided in Appendix A.5, and we give a larger example of composition based on a simplified model of TLS in Appendix C.

5.1 Running example

As a simple example we play a bit with the famous Needham-Schroeder-Lowe public-key protocol (NSL) [65, 59]. In this protocol, the initiator A communicates with the responder B over an insecure network in order to establish a key. Privacy was not a concern in the original protocol, but we can easily add this as a requirement: when A contacts B, nobody else but A and B should learn who is communicating here. In fact, A and B may not know each other in advance and need to first get each other's public key from a key server S who thus also learns their identities. In the original protocol, the key-exchange with the server is not encrypted, therefore everybody can observe that A is looking up B's public key and vice-versa. We are going to encrypt this exchange. As an example for compositionality, we actually split the protocol into two components: P_1 the three-way handshake between A and B and P_2 the lookup protocol for the key server, so we can verify handshake and key server protocol separately.

We give the specification of P_1 the handshake protocol in Fig. 5.1 (on the left); the notation is adapted from the transaction language used in the previous chapters, with some extensions for protocol composition, procedure calls and sequencing of transactions. We give in the next sections the precise syntax and semantics. We use the color red to highlight the parts in a protocol that the other protocol must know about.

In the first component, we have two roles, Initiator and Responder, which each consist of a number of *transactions*. Recall that a transaction is a sequence of steps that will be executed atomically (without interleaving by other transactions). One of our extensions is to express a role with transactions separated by semicolons. This is relevant for privacy as we assume that the intruder knows the protocol and can observe when an agent reaches the end of a transaction in our semantics, because typically the agent is inactive and waiting for another input message.

```
Role $Initiator$ :
$\star\ x_A \in \mathsf{Honest}.$
$\star\ x_B \in \mathsf{Agent}.$
$PKB := lookup(x_A, x_B)$
;
$\nu N_A : \mathsf{nonce}, R : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{crypt}(PKB, \mathsf{f}_1(N_A, x_A), R)).$
if $x_B \in \mathsf{Honest}$ then
   $\star\ x_B \in \mathsf{Honest}$
else
   $\star\ x_A \doteq \gamma(x_A) \wedge x_B \doteq \gamma(x_B).$
   $\mathsf{snd}(N_A)$
;
$\mathsf{rcv}(\mathsf{crypt}(\mathsf{pk}(x_A),$
   $\mathsf{f}_2(N_A, N_B : \mathsf{nonce}, x_B), \_ : \mathsf{nonce})).$
$\nu R' : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{crypt}(PKB, \mathsf{f}_3(N_B), R'))$

Role $Responder$ :
$\mathsf{rcv}(\mathsf{crypt}(\mathsf{pk}(B : \mathsf{agent}),$
   $\mathsf{f}_1(N_A : \mathsf{nonce}, A : \mathsf{agent}), \_ : \mathsf{nonce})).$
if $B \notin \mathsf{Honest}$ then
   stop
;
$PKA := lookup(B, A)$
;
$\nu N_B : \mathsf{nonce}, R : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{crypt}(PKA, \mathsf{f}_2(N_A, N_B, B), R)).$
if $A \notin \mathsf{Honest}$ then
   $\mathsf{snd}(N_B)$
;
$\mathsf{rcv}(\mathsf{crypt}(\mathsf{pk}(B), \mathsf{f}_3(N_B), \_ : \mathsf{nonce}))$
```

```
Procedure $lookup(A : \mathsf{agent}, B : \mathsf{agent})$ :
$\nu N : \mathsf{nonce}, R : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{scrypt}(\mathsf{sk}(A, \mathsf{s}), \mathsf{req}(B, N), R))$
;
$\mathsf{rcv}(\mathsf{scrypt}(\mathsf{sk}(A, \mathsf{s}),$
   $\mathsf{resp}(B, PKB : \mathsf{pk}(\mathsf{agent}), N), \_ : \mathsf{nonce})).$
$\mathsf{assert}(PKB \doteq \mathsf{pk}(B)).$
$\mathsf{return}(\mathsf{pk}(B))$

Role $Server$ :
$\mathsf{rcv}(\mathsf{scrypt}(\mathsf{sk}(A : \mathsf{agent}, \mathsf{s}),$
   $\mathsf{req}(B : \mathsf{agent}, N : \mathsf{nonce}), \_ : \mathsf{nonce})).$
$\nu R : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{scrypt}(\mathsf{sk}(A, \mathsf{s}), \mathsf{resp}(B, \mathsf{pk}(B), N), R))$
```

Figure 5.1: Specification based on the Needham-Schroeder-Lowe public-key protocol

The first transaction in the initiator role is that we choose two *privacy variables* x_A and x_B from the sets Honest and Agent. These can be any finite sets (that we do not specify here) where $\mathsf{Honest} \subseteq \mathsf{Agent}$ and the intruder can control a number of dishonest agents in $\mathsf{Agent} \setminus \mathsf{Honest}$. The symbol $\star$ means here that the intruder is allowed to learn the domains of x_A and x_B. More precisely every state will contain a formula α, also called the *payload*, which contains all the information that was deliberately released to the intruder; in this case the conjunct $x_A \in \mathsf{Honest} \wedge x_B \in \mathsf{Agent}$. Note that the variables will be freshly renamed for every session, i.e., instantiation of the role. One may wonder why $x_A \in \mathsf{Honest}$ rather than Agent. This is because x_A would be the agent executing this role; since a dishonest agent may not follow the protocol, their behavior is described by the intruder rules (that includes the possibility of following the protocol), and thus it is convenient to restrict the execution here to the honest x_A (while x_B may well be a dishonest agent). The first thing that x_A really does is to look up the public key of x_B. Whatever happens here will be the job of the other protocol P_2, so we just consider here that this results in a key PKB.

In its second transaction, x_A generates nonces N_A and R and sends out the first real message of the protocol: $\mathsf{crypt}(PKB, \mathsf{f}_1(N_A, x_A), R)$ where R is a randomization value to avoid deterministic encryption, and f_1 is abstracting the data format of the first message of the protocol.

Note that in this way, the protocol would be violating privacy: if x_B is honest, then the intruder would learn this fact from the outgoing message, assuming the intruder knows the private keys of all dishonest agents. (This is the worst-case assumption behind the Dolev-Yao model that all dishonest agents work together and we can thus think of them as one single intruder.) Moreover, if the intruder is dishonest, then they learn at this point both the values of x_A and x_B. Both deductions are hardly avoidable, and so we just specify as part of this transaction that we *release* this information, again to be stored in the formula α. Recall that $x_A \doteq \gamma(x_A)$ refers to a third formula γ that is present besides α and β in every state: it represents the truth, mapping every privacy variable to its true value. Thus if $\gamma(x_A) = \mathsf{a}$, then we release here the formula $x_A \doteq \mathsf{a}$. Our compositionality model by default treats all fresh values like nonces as secrets, and it counts as an attack (called *leakage*) if the intruder learns these values, unless we explicitly *declassify* them. Therefore, the nonce N_A is initially a shared secret of the two protocols, but in the dishonest x_B case, the intruder is now able to learn N_A since x_B is the intended recipient for it. Thus the nonce has to be declassified, while in the honest x_B case it would count as a leakage if the intruder finds out N_A as it is then shared between two honest agents.

The third and final transaction consists of both receiving the answer from x_B and sending the reply. Here, the process for receiving uses pattern matching expressing that x_A only accepts the incoming message iff: it is encrypted with the public key $\mathsf{pk}(x_A)$ of x_A, it contains the format f_2 of the second step of the protocol, the first item under f_2 is nonce N_A that x_A created earlier, and the third item under f_2 is the name x_B chosen before. Moreover, the variable N_B is bound at this point to whatever (apparently) x_B has sent as their nonce. (The underscore corresponds to variable that is not used.) The role will just abort when receiving a message that does not fit; this also prevents the intruder from sending several messages attempting an online-guessing attack, since the role will stop at the first failure. We will later explain in more detail that this pattern matching is syntactic sugar for receiving a linear pattern (as defined in Definition 4.1.6) and making equality checks with an $\mathsf{if\text{-}then\text{-}else}$ statement.

Let us turn now to the responder role. Here, we start with a receive message that contains again a pattern with variables B, N_A and A; similarly as before, we stop if the input has not the right form, but the variables B and N_A and A can be arbitrary and this is their binding occurrence. Thus, in this model, B "learns" its name from this message, meaning that the encrypted message reaches it intended recipient. Since we want that this is only executed by an honest agent, we stop here if $B \notin \mathsf{Honest}$. The stop means that in this case the following transactions of this role are not going to be executed. (In fact, this is the same stop that occurs when the pattern of a receive is not matched.) Our semantics assumes that the intruder can observe such a stop, because the agent ceases to communicate.[1] The rest of the responder role is similar to the initiator role.

Let us now describe the protocol P_2 (given in Fig. 5.1, on the right). Here the first role is the procedure *lookup*, i.e., this role cannot spontaneously start or be triggered by a received message, but rather this is started by an invocation from P_1, binding the agent names A and B. Here we assume that each honest agent has a shared key $\mathsf{sk}(A, \mathsf{s})$ with the trusted key server s. Note that s is a constant of type agent i.e., this represents a fixed honest server that cannot be played by the intruder. The first message is an encrypted request (message format req) for the public key of B, including a fresh nonce N.

For the sake of this running example, we consider that the public-key infrastructure is fixed but *not* publicly known: agents do not initially know the keys of other agents (hence the lookup), but each agent x has their own fixed key $\mathsf{pk}(x)$.[2] When the lookup process receives the answer from the server, we have the assertion $PKB \doteq \mathsf{pk}(B)$. This reflects a protocol goal: it should never happen that the server returns another key than $\mathsf{pk}(B)$ as answer to a request for public key of B. If such a mismatch were to happen, then it would count as an attack; in this way we make the assertion part of the verification of the security of the lookup protocol. Finally, if the assertion has succeeded, the lookup process returns the key it has received from the server. Since we can at this point be sure that $PKB \doteq \mathsf{pk}(B)$, we can simply return $\mathsf{pk}(B)$. Thus *lookup* is guaranteed to always give back the correct key, or fail with an attack before returning something.

5.1.1 The composition

Now the composition is in some sense obvious: the calls to *lookup* in the initiator and responder roles shall be replaced with the procedure's body under the respective instantiation of the formal parameters, and replacing PKB and PKA in the initiator and responder roles with the return values $\mathsf{pk}(B)$ and $\mathsf{pk}(A)$, respectively. This is formalized in Definition 5.2.3. The composed protocol consists then of this expanded initiator and responder roles of P_1 and the server role of P_2 (because the server is triggered by incoming messages, not a procedure call).

This gives the entire protocol as a monolith, and we rather want to verify the two protocols as components in isolation. However, in complete isolation, neither component really makes sense. This is because P_1 cannot really do anything without some form of mechanism to get the key of the intended recipient. While P_2 can work sort of independently, it really gets its actual goals from the context with P_1, namely that it is transporting the

[1]One may wonder whether the if could break privacy: since the intruder observes the stop, they also observe that B was not honest. However, it is easy to notice that the intruder must already know this in any case, because either the intruder has composed this message themselves (and thus knows that they put the name of a dishonest agent in it) or this was sent by an honest initiator, so the information about sender and receiver was already released earlier. Thus the protocol is not revealing any information about the agents at this point. In general, if the modeler accidentally forgets to release something that the intruder can indeed learn, this means erring on the safe side: the verification of (α, β)-privacy will just fail and demonstrate how the intruder can learn more than was deliberately released. In this case the modeler can review and adapt the model either by changing the protocol to strengthen privacy, or by releasing more information.

[2]In contrast to previous chapters, we assume here that pk is not a public function, so the intruder also does not know this public-key infrastructure.

names and public keys of agents that are privacy variables chosen in P_1; we need this context to make clear that P_2 has the obligation to keep these values private.

We thus need to define an interface between the two protocols, and we do so by highlighting in each protocol which steps are relevant to the other protocol. In [53], the interface is made explicit with the $\star$ symbol; for now we use highlighting to avoid confusion with the choices of privacy variables. Let us now discuss the highlighting in Fig. 5.1. Basically this is saying: when verifying either protocol, one needs to consider at least the highlighted steps from the other protocol, because these are relevant things happening in the other protocol. Most notably this includes the non-deterministic choices like $x_A \in \mathsf{Honest}$, because here a privacy-critical information is introduced, as well as all released information that is released in α, like $x_A \doteq \gamma(x_A)\ldots$. Procedure calls are replaced with the body of procedures so we did not highlight them in Fig. 5.1, but they are obviously relevant to the other protocol. Creation of fresh nonces is also highlighted, because it introduces fresh secrets that can in principle reach either protocol. As part of the protocol composition, it is necessary to specific a set *Secrets* of messages that are initially secret and may be declassified during the protocol execution. For our running example, we have to specific all nonces as secrets, as well as all ground messages of the form $\mathsf{crypt}(\mathsf{pk}(a), \mathsf{f}_1(n_a, a), r)$ (and their subterms) where a is a concrete agent name and n_a, r are nonces. These messages need to be part of the set *Secrets* because we will require in Definition 5.3.4 that all terms in the interface are either public or declared as secrets.

All other kinds of steps are not necessarily highlighted: it depends on whether this step is deemed relevant for the other protocol. We will have below some requirements on the highlighting, but besides these requirements it is the choice of the modeler what to highlight. Naturally, we want as few highlighted steps as possible, because the more details can be hidden, the easier the verification task gets.

As can be seen in the specification, we have highlighted all the conditionals: in the initiator role, releases depend on the condition; in the first conditional of the responder role, it depends on the condition whether we proceed in the first place, and the second conditional is again containing a release. Of all the protocol messages we have only highlighted the first message of P_1. This is because the message is binding several of the variables in the responder role upon reception, in particular A and B that are again relevant for highlighted steps. Thus this binding occurrence is also relevant.[3]

Highlighting a $\mathsf{snd}(t)$ step has another important meaning in our composition: it means that the message t is now *declassified*, i.e., the intruder may know it. We use declassification in the initiator role with $\mathsf{snd}(N_A)$ in the case x_B is dishonest: the intruder is just allowed to know N_A in this case, because one of the dishonest agents is the intended recipient of N_A.

We thus regard as a component protocol the individual steps of that component protocol plus all the highlighted steps of the other component protocol. In a nutshell, the compositionality result below is that an attack trace against the composed protocol can be transformed into an attack trace against one of the components. Thus it suffices to verify the security of the components to show the security of their composition—provided they satisfy the requirements of our compositionality theorem.

[3]In Appendix C, we discuss an alternative model that avoids highlighting the protocol-specific message.

5.2 Extensions of specification and semantics

5.2.1 Protocol specification

We follow the same notations as in Chapters 2 and 4 with additional constructs for protocol composition.

Definition 5.2.1 (Protocol specification)**.** *A* protocol specification *consists of*

- *a number of* roles *and* procedures *according to the syntax below;*
- *a number of* memory cells*, e.g.,* $\mathsf{cell}[\cdot]$*, together with a ground context* $C[\cdot]$ *for each memory cell defining the initial value of the memory, so that initially* $\mathsf{cell}[t] = C[t]$*;*
- *a set* Γ_0 *of* Σ_0*-interpretations of the relations occurring in the roles and procedures; and*
- *an assignment of every constant and memory cell to a type.*

$$
\begin{array}{lll l}
R & & & \textit{Role} \\
 & ::= & P_l; R & \textit{Sequence} \\
 & \mid & P_l & \textit{Transaction} \\
P & ::= & \mathit{proc}(X_1 : \tau_1, \ldots, X_n : \tau_n) : R & \textit{Procedure} \\
P_l & & & \textit{Left process} \\
 & ::= & \star\ x \in D.P_l & \textit{Non-deterministic choice} \\
 & \mid & \mathsf{rcv}(t : \tau).P_l & \textit{Receive} \\
 & \mid & \mathsf{let}\ X = t.P_l & \textit{Let statement} \\
 & \mid & X := \mathit{proc}(t, \ldots, t) & \textit{Procedure call} \\
 & \mid & P_c & \textit{Center process} \\
P_c & & & \textit{Center process} \\
 & ::= & X := \mathsf{cell}[t].P_c & \textit{Cell read} \\
 & \mid & \mathsf{if}\ \phi\ \mathsf{then}\ P_c\ \mathsf{else}\ P_c & \textit{Conditional statement} \\
 & \mid & \mathsf{let}\ X = t.P_c & \textit{Let statement} \\
 & \mid & \mathsf{stop} & \textit{Stop} \\
 & \mid & \nu X_1 : \tau_1, \ldots, X_k : \tau_k.P_r & \textit{Fresh constants} \\
P_r & & & \textit{Right Process} \\
 & ::= & \mathsf{snd}(t).P_r & \textit{Send} \\
 & \mid & \mathsf{cell}[t] := t.P_r & \textit{Cell write} \\
 & \mid & \star\ \phi.P_r & \textit{Release} \\
 & \mid & \mathsf{assert}(\phi).P_r & \textit{Assertion} \\
 & \mid & \mathsf{let}\ X = t.P_r & \textit{Let statement} \\
 & \mid & \mathsf{return}(t) & \textit{Return} \\
 & \mid & 0 & \textit{Terminate (nil process)}
\end{array}
$$

where D is a finite set of public constants, t ranges over destructor-free messages (that do not contain $\mathtt{ff}$*), τ, τ_i range over types and ϕ ranges over quantifier-free Herbrand logic formulas.*

The non-deterministic choices, receives, let statements, procedure calls, cell reads and fresh constants are binding.

We require that every role is closed. Every procedure has a number of parameters, which are the free variables (each associated with a type) of the procedure's body, and every procedure call must give the appropriate number of arguments. Moreover, the last step of a *procedure must be either a* return *statement or* stop. *Finally, there must be no cycle in the graph of procedure calls.*

In the previous chapters, a protocol specifies a fixed interpretation γ_0 of all relations. To consider several interpretations, we would have to apply our decision procedure several times. Here instead, the specification defines a set Γ_0 of interpretations for the relations that are used in the processes, and we consider that the intruder knows that in every concrete execution, the true interpretation is in Γ_0 (but they do not know a priori which one). Note that Γ_0 is finite since we only allow relations over the finite alphabet Σ_0. Relations can model some context about the agents participating in the protocol. For instance, in our model of NSL, we could have a relation $\mathit{talk}(\cdot,\cdot)$ over agent names that represents whether some agent wants to talk to another one. Then the set of interpretations could contain, e.g., some interpretations where Alice (denoted a) wants to talk to Bob (denoted b), i.e., where $\mathit{talk}(\mathsf{a},\mathsf{b})$ holds, and some other interpretations where it is not the case, modeling that the intruder does not know whether Alice actually wants to talk to Bob. We could then require that all interpretations agree on $\mathit{talk}(x,y)$ whenever x is dishonest, modeling that initially the intruder knows who dishonest agents want to talk to.

Definition 5.2.2 (Syntactic sugar)**.** *For ease of notation, we allow the following in specifications:*

- *Trailing* 0 *and* else 0 *can be omitted.*
- *Variables bound in a message received but otherwise never used can be written with a wildcard* _ *instead of a variable name.*
- *A step of a right process can be written in the left or center part with the meaning that this step is executed in every branch.*

As in Chapter 4, the semantics of receiving a message is only defined when the message is linear with only fresh variables and no constants. In order to ensure this, we transform a message received, that may contain variables bound earlier or some constants, into a linear term with only fresh variables and no constants. We then insert a conditional statement to check that the freshly introduced variables have the expected values.

Formally, for a message received of the form $R; P_1.\mathsf{rcv}(t).P_2.P_3$*, where* R *is a sequence of transactions,* $P_2.P_3$ *is a left process and* P_3 *is a center process, we obtain a linear term* t' *by replacing in* t *every variable already bound in* R *or* P_1 *and every constant with fresh intruder variables, and we replace* $\mathsf{rcv}(t).P_2.P_3$ *with* $\mathsf{rcv}(t').P_2.\mathsf{if}\ t \doteq t'\ \mathsf{then}\ P_3\ \mathsf{else\ stop}$.

Example 5.2.1. $\mathsf{snd}(t).\mathsf{if}\ \phi\ \mathsf{then}\ P\ \mathsf{else}\ Q$ is short for $\mathsf{if}\ \phi\ \mathsf{then}\ \mathsf{snd}(t).P\ \mathsf{else}\ \mathsf{snd}(t).Q$.

The process $\star\ x \in \mathsf{Agent}.\mathsf{rcv}(\mathsf{scrypt}(\mathsf{sk}(x,\mathsf{s}),M,\_)).P$, where P is a center process, is short for $\star\ x \in \mathsf{Agent}.\mathsf{rcv}(\mathsf{scrypt}(\mathsf{sk}(X,S),M,\_)).\mathsf{if}\ X \doteq x \wedge S \doteq \mathsf{s}\ \mathsf{then}\ P\ \mathsf{else\ stop}$. Here we have simplified the formula since only X and S have expected values. $\lhd$

We have already presented informally most of the steps in processes when describing the example in Section 5.1. Moreover, many constructs are explained in Chapter 2 and in this chapter, these constructs have the same semantics as in Table 2.1, except that we extend the definition to support roles. We explain below the parts of protocol specification which are newly introduced in this chapter. The details of the semantics for processes are given in Definition 5.2.7.

Roles

The behavior of agents can be modeled as *roles*, which are defined as sequences of transactions. The benefit of using roles is that we can express the behavior of agents in a single role, where the variables bound in some transaction in the role are still in scope in every transaction coming later in the role. For instance, we can express that an agent generates a fresh value, sends a message containing that value, and then later receives a message that they can check contains the fresh value generated earlier. Without the notion of roles, we would need to use memory cells or message passing to communicate the local state of the agent from one transaction to the next. The execution of a sequence $P_1; \ldots ; P_n$ is not on the whole atomic, but each transaction P_i is executed atomically, in order. Between each of these P_i, other transactions from the protocol may be executed.

Let

In a let statement, a fresh variable can be bound to a message, where the scope of the binding is the rest of the role or procedure (and not just the current transaction). Let statements are part of the semantics and not just syntactic sugar: we will consider interleavings of transactions and the substitution has to be applied to all transactions that follow the let statement.

Procedure call and return

A role can call a *procedure*, which is also a sequence of transactions but takes formal parameters and returns a value. When the procedure is called, its parameters are instantiated with the arguments of the call, and the returned value is bound to a fresh variable that can be used in the rest of the role. Procedure call is one of the constructs that can be used for modeling protocol composition.

Stopping

The execution of a role or procedure can be stopped, which terminates not only the current transaction but also the rest of that role or procedure. In our semantics, the intruder can distinguish possibilities that are stopping and the ones that are not stopping (e.g., sending a message).

Assertion

A transaction can contain *assertions*, and if during the protocol execution, there is an assertion that does not hold, then it is considered as an attack on the protocol. Assertions are useful for our modular verification using an "assume-guarantee" approach: for a component that contains an assertion, the abstract interface typically does not contain the assertion and from the point of view of other components, the assertion can be assumed to hold; the assertion is however checked when verifying the component itself.

5.2.2 Semantics

The grammar of specifications allows for procedure calls, where one protocol is calling a process that may be specified by another protocol. Procedure calls can be used to define the interface between two protocols, e.g., one protocol obtains a key that is established by another protocol. The semantics of a procedure call is essentially to inline the body of the procedure. Formally, we define the replacement of procedure calls with the processes they represent as *procedure call expansion.*

Definition 5.2.3 (Procedure call expansion). *Let R_1, R_2 be sequences of transactions such that R_1 does not contain any procedure call and let $P.X := proc(t_1, \ldots , t_n)$ be a transaction,* *where proc is a procedure with parameters $X_1, \dots, X_n$ and body R. Define*

$$\mathit{expand}(R_1; P.X := \mathit{proc}(t_1, \dots, t_n); R_2) = R_1; \mathit{expand}(P.\sigma(R'); R_2)$$

where $\sigma = [X_1 \mapsto t_1, \dots, X_n \mapsto t_n]$ and R' is the same as R except that every $\mathsf{return}(t)$ *is replaced with* $\mathsf{let}\ X = t$*, i.e., the return value is bound to variable X.*

The semantics of $\mathsf{let}\ X = t$ (defined in Table 5.1) is that the variable X will be substituted with t in the rest of the sequence of transactions when this step is executed. Since the graph of procedure calls is required to be acyclic, the expansion terminates.

Example 5.2.2. Let us consider again our NSL running example from Section 5.1. In the initiator role, there is a procedure call to look up the public key of the responder. After procedure call expansion, we get the following sequence of transactions:

```
$\star\ x_A \in \mathsf{Honest}.$
$\star\ x_B \in \mathsf{Agent}.$
$\nu N : \mathsf{nonce}, R'' : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{scrypt}(\mathsf{sk}(x_A, \mathsf{s}), \mathsf{req}(x_B, N), R''))$
;
$\mathsf{rcv}(\mathsf{scrypt}(\mathsf{sk}(x_A, \mathsf{s}), \mathsf{resp}(x_B, \mathit{PKB}' : \mathsf{pk}(\mathsf{agent}), N), \_ : \mathsf{nonce})).$
$\mathsf{assert}(\mathit{PKB}' \doteq \mathsf{pk}(x_B)).$
$\mathsf{let}\ \mathit{PKB} = \mathsf{pk}(x_B)$
;
$\nu N_A : \mathsf{nonce}, R : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{crypt}(\mathit{PKB}, \mathsf{f}_1(N_A, x_A), R)).$
$\mathsf{if}\ x_B \in \mathsf{Honest}\ \mathsf{then}$
    $\star\ x_B \in \mathsf{Honest}$
$\mathsf{else}$
    $\star\ x_A \doteq \gamma(x_A) \wedge x_B \doteq \gamma(x_B).$
    $\mathsf{snd}(N_A)$
;
$\mathsf{rcv}(\mathsf{crypt}(\mathsf{pk}(x_A), \mathsf{f}_2(N_A, N_B : \mathsf{nonce}, x_B), \_ : \mathsf{nonce})).$
$\nu R' : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{crypt}(\mathit{PKB}, \mathsf{f}_3(N_B), R'))$
```

Similarly, before executing the responder role, the lookup procedure call needs to be expanded. $\triangleleft$

In this chapter, we extend the notion of state with additional information that is used for compositionality. Every state contains a sequence ρ called *choice of recipes* that stores the recipes used by the intruder whenever the protocol was receiving a message[4], and a boolean *flag* that tracks whether some assertion was broken.

Definition 5.2.4 (State). *A* state *S is a tuple $S = (\alpha, \gamma, \mathcal{P}, \rho, \mathit{flag})$ such that:*

- *α is a Σ_0-formula, the* payload*;*

[4]In contrast to the previous chapters, here a choice of recipes is a sequence of recipes and not a substitution, because we only deal with ground recipes and we simply need to record which recipes were used.

- *γ is a Σ_0-formula, the* truth*;*
- *$\mathcal{P}$ is a set of* possibilities*, which are each of the form $(R, \phi, struct, \delta)$, where R is a sequence of processes, ϕ is a Σ-formula, struct is a frame and δ is a sequence of memory updates of the form* $\mathsf{cell}[s] := t$ *for messages s and t;*
- *ρ is a sequence of recipes, the* choice of recipes *made so far for every message sent by the intruder; and*
- *flag is either true or false, where the flag set to true represents that some assertion did not hold.*

Let $\mathcal{P} = \{(R_1, \phi_1, struct_1, \delta_1), \ldots, (R_n, \phi_n, struct_n, \delta_n)\}$ be the possibilities in S. Then S is well-formed *iff*

- *$\alpha \models \bigvee_{\gamma_0 \in \Gamma_0} \gamma_0$;*
- *$\gamma \models \alpha \wedge \bigvee_{i=1}^{n} \phi_i$;*
- *the ϕ_i are such that $\models_\Sigma \neg(\phi_i \wedge \phi_j)$ for $i \neq j$ and $fv(\phi_i) \subseteq fv(\alpha)$;*
- *the $struct_i$ are frames with the same labels occurring in the same order; and*
- *the recipes in ρ are over the domain of the $struct_i$.*

S is a milestone *iff every sequence R_i starts with the nil process, and S is an* intermediate state *otherwise.*

In the rest of the chapter, we only consider well-formed states (our semantics ensures that the reachable states of a protocol are well-formed).

Note that, in contrast to Chapter 2, we do not need the Σ_0-formula β_0 anymore, because for compositionality we only support non-deterministic choices with $\mathsf{mode} = \star$, i.e., privacy variables that become part of the payload α (that is, we do not have any variables with $\mathsf{mode} = \diamond$). Another difference is that, since a protocol now specifies a set Γ_0 of interpretations, the requirements of well-formedness are updated accordingly with the disjunction over Γ_0 instead of a single interpretation γ_0. This models the fact that the intruder a priori does not know exactly which interpretation of relations is true; they just know that one among Γ_0 holds.

In a given state, the privacy goals are expressed through the payload α. The intruder knowledge is expressed through a formula β, which is defined based on the possibilities in that state. The intruder has made concrete observations, recorded in a frame *concr*, they consider a set of possibilities containing ϕ_i, $struct_i$ such that, if condition ϕ_i holds, then the concrete instances of $struct_i$ are statically equivalent with *concr*.

Definition 5.2.5 (Intruder knowledge). *Given a well-formed state $S = (\alpha, \gamma, \mathcal{P}, \_, \_)$, let $concr = \gamma(struct_j)$ where $(\_, \phi_j, struct_j, \_) \in \mathcal{P}$ is the unique possibility such that $\gamma \models \phi_j$. The* intruder knowledge *in state S is defined as*

$$\beta(S) = \alpha \wedge \bigvee_{i=1}^{n} \phi_i \wedge concr \sim struct_i \ .$$

We say that S satisfies privacy *iff $(\alpha, \beta(S))$-privacy holds.*

5.2.3 State transition system

A protocol specification defines a set of roles, which are sequences of transactions, and a concrete protocol execution corresponds to an interleaving of such transactions. We later want to consider any number of instances of the roles of the protocol, and call them *threads*. Each such thread will get a unique *thread ID*. We allow any interleaving of the transactions of the different threads: as before, each transaction is atomic, but for instance after the first transaction of thread TID_1, can come a transaction of thread TID_2, and then another of thread TID_1. In this way, we get any sequence of transactions corresponding to unbounded "sessions" of the protocol: when projecting to one particular thread ID, we get (a prefix of) an instance of a role. This will be made precise in Definition 5.2.9 below, but for now we define a protocol execution as a sequence of transactions, where each transaction is annotated with a thread ID. All possibilities in a state will contain such a sequence of transactions and we will ensure the invariant that in every possibility, the sequence has the same length and corresponding transactions have the same thread ID.

Definition 5.2.6 (Thread filtering)**.** *Let* $\underbrace{P_1}_{TID_1}; \ldots ; \underbrace{P_n}_{TID_n}$ *be a sequence of transactions (after expansion), where every transaction is annotated with a* thread ID*.*

We define $filter_{=TID}(\underbrace{P_1}_{TID_1}; \ldots ; \underbrace{P_n}_{TID_n})$ *as the largest subsequence that only contains transactions with thread ID equal to TID. Conversely,* $filter_{\neq TID}(\underbrace{P_1}_{TID_1}; \ldots ; \underbrace{P_n}_{TID_n})$ *is the largest subsequence that does not contain any transaction with thread ID equal to TID.*

In the rest of the chapter, we will omit the braces with thread IDs whenever they are not relevant and we just need to consider the transactions themselves.

The symbolic execution is defined through a set of rules that are transitions between states. Each transition evaluates one step of the processes.

Definition 5.2.7 (Semantics)**.** *The semantics of the symbolic execution is given in Table 5.1. The changes to the state are highlighted in red. The semantics defines a relation* $\rightarrow$ *on states. Let* S, S' *be two milestones. We define* $\longrightarrow$ *as the relation such that* $S \longrightarrow S'$ *iff* $S \rightarrow^* S'$ *and with the requirement that whenever the* **Eliminate** *rule is applicable, then it must be applied.*

Compared to Table 2.1, we have the following new rules:

- **Let**: We use let statements to bind variables to messages. Note that this is part of the semantics and not purely syntactic sugar because we have to substitute a variable in every transaction that remains in the sequence being executed.
- **Assert**: We check an assertion in the unique possibility that is the case (according to the ground truth γ) and update the assertion flag if the formula asserted does not hold.
- **Stop**: The process stop aborts an entire role (sequence of transactions), so at this point we remove all transactions having the same thread ID from the transactions that remain to be executed. Note that we have made the stop observable to the intruder, so they can distinguish possibilities that are not stopping and rule them out.
- **Milestone**: This is the updated version of the **Terminate** rule, where now every process may be stopping, sending or just 0 and the intruder only keeps the possibilities that are 0 to reach a milestone.

Table 5.1: Semantics of the symbolic execution

Choice	$(\alpha, \gamma, \{(\star\ x \in D.P_i; R_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \rho, \mathit{flag})$ $\to (\alpha \wedge x \in D, \gamma \wedge x \doteq c, \{(P_i; R_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \rho, \mathit{flag})$ for every $c \in D$
Receive	$(\alpha, \gamma, \{(\mathsf{rcv}(t).P_i; R_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \rho, \mathit{flag})$ $\to (\alpha, \gamma, \{(\sigma_i(P_i; R_i), \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \rho.r, \mathit{flag})$ for every recipe r over the domain of the struct_i, where $\sigma_i = \mathit{mgu}(t \doteq \mathit{struct}_i(r))$
Let	$(\alpha, \gamma, \{(\mathsf{let}\ X = t.P; R, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P}, \rho, \mathit{flag})$ $\to (\alpha, \gamma, \{((P; R)[X \mapsto t], \phi, \mathit{struct}, \delta)\} \cup \mathcal{P}, \rho, \mathit{flag})$
Cell read	$(\alpha, \gamma, \{(X := \mathsf{cell}[s].P; R, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P}, \rho, \mathit{flag})$ $\to (\alpha, \gamma, \{(P'; R, \phi, \mathit{struct}, \delta)\} \cup \mathcal{P}, \rho, \mathit{flag})$ where $\delta_{\mid\mathsf{cell}} = \mathsf{cell}[s_1] := t_1. \cdots .\mathsf{cell}[s_k] := t_k$, the ground context for initial value of cell is $C[\cdot]$ and $P' = \mathsf{if}\ s \doteq s_1\ \mathsf{then\ let}\ X = t_1.P\ \mathsf{else} \dots$ $\mathsf{if}\ s \doteq s_k\ \mathsf{then\ let}\ X = t_k.P\ \mathsf{else\ let}\ X = C[s].P$
Cell write	$(\alpha, \gamma, \{(\mathsf{cell}[s] := t.P; R, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P}, \rho, \mathit{flag})$ $\to (\alpha, \gamma, \{(P; R, \phi, \mathit{struct}, \mathsf{cell}[s] := t.\delta)\} \cup \mathcal{P}, \rho, \mathit{flag})$
Conditional	$(\alpha, \gamma, \{((\mathsf{if}\ \psi\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2); R, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P}, \rho, \mathit{flag})$ $\to (\alpha, \gamma, \{(P_1; R, \phi \wedge \psi, \mathit{struct}, \delta), (P_2; R, \phi \wedge \neg\psi, \mathit{struct}, \delta)\} \cup \mathcal{P}, \rho, \mathit{flag})$
Release	$(\alpha, \gamma, \{(\star\ \psi.P; R, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P}, \rho, \mathit{flag})$ $\to (\alpha', \gamma, \{(P; R, \phi, \mathit{struct}, \delta)\} \cup \mathcal{P}, \rho, \mathit{flag})$ where $\alpha' = \alpha \wedge \psi$ if $\gamma \models \phi$ and $\alpha' = \alpha$ otherwise
Assert	$(\alpha, \gamma, \{(\mathsf{assert}(\psi).P; R, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P}, \rho, \mathit{flag})$ $\to (\alpha, \gamma, \{(P; R, \phi, \mathit{struct}, \delta)\} \cup \mathcal{P}, \rho, \mathit{flag}')$ where $\mathit{flag}' = \mathsf{true}$ if $\gamma \models \phi \wedge \neg\psi$ and $\mathit{flag}' = \mathit{flag}$ otherwise
Eliminate	$S = (\alpha, \gamma, \{(P; R, \phi, \mathit{struct}, \delta)\} \uplus \mathcal{P}, \rho, \mathit{flag}) \to (\alpha, \gamma, \mathcal{P}, \rho, \mathit{flag})$ if $\beta(S) \models_\Sigma \neg\phi$
Send	$(\alpha, \gamma, \{(\mathsf{snd}(t_i).P_i; R_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \dots, n\}\} \uplus \mathcal{P}, \rho, \mathit{flag})$ $\to (\alpha, \gamma, \{(P_i; R_i, \phi_i, \mathit{struct}_i.l \mapsto t_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \rho, \mathit{flag})$ if $\gamma \models \bigvee_{i=1}^{n} \phi_i$ and every process in $\mathcal{P}$ starts with stop or 0, where l is a fresh label
Stop	$(\alpha, \gamma, \{(\underbrace{\mathsf{stop};}_{TID} R_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \dots, n\}\} \uplus \mathcal{P}, \rho, \mathit{flag})$ $\to (\alpha, \gamma, \{(\underbrace{\ 0\ }_{TID}; \mathit{filter}_{\neq TID}(R_i), \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \rho, \mathit{flag})$ if $\gamma \models \bigvee_{i=1}^{n} \phi_i$ and every process in $\mathcal{P}$ starts with $\mathsf{snd}(\cdot)$ or 0
Milestone	$(\alpha, \gamma, \{(0; R_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \dots, n\}\} \uplus \mathcal{P}, \rho, \mathit{flag})$ $\to (\alpha, \gamma, \{(0; R_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \rho, \mathit{flag})$ if $\gamma \models \bigvee_{i=1}^{n} \phi_i$, $\mathcal{P} \neq \emptyset$ and every process in $\mathcal{P}$ starts with stop or $\mathsf{snd}(\cdot)$
Next	$(\alpha, \gamma, \{(0; R_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \rho, \mathit{flag})$ $\to (\alpha, \gamma, \{(R_i, \phi_i, \mathit{struct}_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \rho, \mathit{flag})$

- **Next**: Once we have reached a milestone, if there is still a transaction to execute, then we can start it.

The symbolic execution maintains some invariants, assuming that initially every possibility starts with the same sequence of transactions, with the same thread IDs:

- The non-deterministic choices happen at the same time in every possibility, because these steps occur before any branching.
- The receives also happen at the same time in every possibility, because these steps occur before any branching and also because we are only considering well-typed instantiations and messages received are linear with only fresh intruder variables and no constants, so in every frame, the unification problem between the received pattern and the actual message has a solution.
- The thread ID of the transaction being executed is the same in every possibility, because the possibilities are synchronized when starting the next transaction in the sequence.
- There is always a single possibility with condition ϕ such that $\gamma \models \phi$ in the current state, which means that when every possibility is either stopping, sending a message or reaching a milestone, then exactly one of the rules **Stop**, **Send** and **Milestone** is applicable. If any possibility starts with a different step, then another rule must be applied.
- The messages in the frames can be considered destructor-free without loss of generality, even though they may contain privacy variables. The intruder can always compare the outcome of a destructor with what happens in the concrete frame, since we allow destructors in recipes. The **Eliminate** rule allows for eliminating all frames that are not statically equivalent to the concrete frame observed by the intruder. Therefore, for every message received by a process, even if a recipe chosen by the intruder contains a destructor, either the destructor gives a subterm in every frame, or the constant $\mathbb{ff}$ in every frame.

We are able to maintain these invariants thanks to the structure of processes: the distinction between left, center and right processes ensures that during symbolic execution, the processes are progressing all together (for the left part) or reducing one at a time, until we reach a milestone.

Before the symbolic execution, we perform a number of preparatory steps on the roles that are specified by the protocol.

Definition 5.2.8 (Fresh role instance)**.** *Let R be a role. A* fresh instance *R' of R is obtained by*

1. *Renaming all variables apart in the role R and in the procedures it calls.*
2. *Expanding the role.*
3. *Removing every step $\nu X_1, \dots, X_k$ by instantiating the X_i with distinct fresh constants.*
4. *Giving every transaction in the resulting sequence R' the same fresh thread ID.*

We now have all the necessary concepts to define the overall state transition system and the traces of a protocol. A trace consists of the sequence of transactions executed, the interpretation of all relations and privacy variables chosen in those transactions, and the recipes chosen for the messages sent by the intruder (i.e., messages received in the transactions). Recall that the intruder does not know a priori what is the true interpretation of relations: the protocol specifies a set Γ_0 of possible interpretations, and for the symbolic execution we need to fix this interpretation. Thus, for the set of reachable states we consider several initial states where for each, we fix some $\gamma_0 \in \Gamma_0$.

Definition 5.2.9 (State transition system). *A sequence of transactions $P_1; \ldots; P_n$ is* valid *iff there exist role instances $R_1, \ldots, R_m$, with thread IDs $TID_1, \ldots, TID_m$, such that for every $j \in \{1, \ldots, m\}$, $filter_{=TID_j}(P_1; \ldots; P_n)$ is a prefix of R_j.*

Let γ be a Σ_0-formula such that $\gamma \models \bigvee_{\gamma_0 \in \Gamma_0} \gamma_0$ and $P_1; \ldots; P_n$ be a valid sequence of transactions. Then the initial state w.r.t. γ and $P_1; \ldots; P_n$ *is*

$$init(\gamma, P_1; \ldots; P_n) = (\bigvee_{\gamma_0 \in \Gamma_0} \gamma_0, \gamma, \{(0; P_1; \ldots; P_n, \mathsf{true}, [], [])\}, [], \mathsf{false})$$

where [] denotes the empty frame, empty memory and empty choice of recipes.

A tuple $(P_1; \ldots; P_n, \gamma, \rho)$ is a trace *iff $P_1; \ldots; P_n$ is valid and there exist $\gamma_0 \in \Gamma_0$ and a milestone $S = (\_, \gamma, \_, \rho, \_)$ such that $\gamma \models \gamma_0$ and $init(\gamma_0, P_1; \ldots; P_n) \longrightarrow S$. The milestone S is then called a* reachable state.

5.3 Composition and composability

5.3.1 Composition

We consider a composed protocol as the composition of two smaller specifications (the definitions and results can be generalized to an arbitrary number of specifications, we use two here for convenience of notation). Our main result is that, for the class of *composable* protocols (Definition 5.3.4), an attack trace on the composed protocol can be projected to an attack on one smaller protocol. To achieve this, every protocol specifies its *interface* and our projection corresponds to an individual protocol composed with the interface of the other protocol.

In the following we adopt the marking convention from [53] where all protocol steps are marked as 1, 2, or $\star$ for individual steps of the component protocols P_1 and P_2 and the interface, respectively. The $\star$ mark for process steps replaces here the highlighting we had in Section 5.1.

Definition 5.3.1 (Protocol composition). *Let $Spec_1$ and $Spec_2$ be protocols. The* composition *$Spec_1 \parallel Spec_2$ is the protocol defined with:*

- *the union of the roles, procedures, memory cells and algebraic theories from $Spec_1$ and $Spec_2$; and*
- *the set of interpretations Γ_0, where we assume that both protocols specify the same set.*

If there are shared memory cells, then both protocols must specify the same ground contexts for the initial values of these cells.

In the context of a composed protocol, we consider that some steps of the processes specified are marked with either a protocol-specific index or the symbol $\star$. Let $i \in \{1, 2\}$ and P be a transaction from $Spec_i$. In the specification of P, the steps for receives, let statements, procedure calls, conditional statements, cell reads or writes, sends and assertions are marked with either i or $\star$. However, the steps for non-deterministic choices, stops, *fresh constants, releases and nil processes are not marked because they are always relevant for the composition.*

During procedure call expansion, every statement let $X = t$ *that replaces a* return(t) *is marked with the same mark as the procedure call that is being expanded.*

In frames, every mapping has the mark of the corresponding snd$(\cdot)$ *step.*

Our main result is that we can *project* an attack on a composed protocol to an attack on a smaller protocol. In the context of composition, every step in a process has a mark that says whether it is part of the protocol's interface, i.e., it must always be present in the projection, or it is protocol-specific and can be abstracted away in the projection.

Definition 5.3.2 (Projection to one component). *Let $i, j \in \{1, 2, \star\}$. The projection of a transaction (after expansion) is defined below.*

$$
\begin{aligned}
(\star\ x \in D.P)|_j &= \star\ x \in D.P|_j \\
(i : \mathsf{rcv}(t).P)|_j &= \begin{cases} \mathsf{rcv}(t).P|_j & \textit{if } i \in \{j, \star\} \\ P|_j & \textit{otherwise} \end{cases} \\
(i : \mathsf{let}\ X = t.P)|_j &= \begin{cases} \mathsf{let}\ X = t.P|_j & \textit{if } i \in \{j, \star\} \\ P|_j & \textit{otherwise} \end{cases} \\
(i : X := \mathsf{cell}[s].P)|_j &= \begin{cases} X := \mathsf{cell}[s].P|_j & \textit{if } i \in \{j, \star\} \\ P|_j & \textit{otherwise} \end{cases} \\
(i : \mathsf{if}\ \phi\ \mathsf{then}\ P\ \mathsf{else}\ Q)|_j &= \begin{cases} \mathsf{if}\ \phi\ \mathsf{then}\ P|_j\ \mathsf{else}\ Q|_j & \textit{if } i \in \{j, \star\} \\ P|_j & \textit{otherwise} \end{cases} \\
(\mathsf{stop})|_j &= \mathsf{stop} \\
(\nu X_1, \dots, X_k.P)|_j &= \nu X_1, \dots, X_k.P|_j \\
(i : \mathsf{snd}(t).P)|_j &= \begin{cases} \mathsf{snd}(t).P|_j & \textit{if } i \in \{j, \star\} \\ P|_j & \textit{otherwise} \end{cases} \\
(i : \mathsf{cell}[s] := t.P)|_j &= \begin{cases} \mathsf{cell}[s] := t.P|_j & \textit{if } i \in \{j, \star\} \\ P|_j & \textit{otherwise} \end{cases} \\
(\star\ \phi.P)|_j &= \star\ \phi.P|_j \\
(i : \mathsf{assert}(\phi).P)|_j &= \begin{cases} \mathsf{assert}(\phi).P|_j & \textit{if } i \in \{j, \star\} \\ P|_j & \textit{otherwise} \end{cases} \\
(0)|_j &= 0
\end{aligned}
$$

The notion of projection is extended to sequences of transactions and protocols. For a frame F, we define $F|_i$ as the projection of the frame to mappings marked with i or $\star$.

The reader may wonder at this point why for a protocol-specific conditional statement, we only keep the interface of the then branch, and completely disregard the else branch. This projection is correct because of our composability requirements listed in Definition 5.3.4; in particular, we have Requirement 1, which ensures that for protocol-specific branching, the else branch is simply the process stop.

Example 5.3.1. In Example 5.2.2, we showed the initiator role after procedure call expansion. We now add marks to denote which process steps are part of the abstract interface of the protocols and which steps are protocol-specific (and in this case, from which protocol they are). We mark with 1 the steps from the NSL initiator, and with 2 the steps from the lookup procedure.

```
$\star$ $x_A \in \mathsf{Honest}$.
$\star$ $x_B \in \mathsf{Agent}$.
    $\nu N : \mathsf{nonce}, R'' : \mathsf{nonce}$.
2   $\mathsf{snd}(\mathsf{scrypt}(\mathsf{sk}(x_A, \mathsf{s}), \mathsf{req}(x_B, N), R''))$
    ;
2   $\mathsf{rcv}(\mathsf{scrypt}(\mathsf{sk}(x_A, \mathsf{s}), \mathsf{resp}(x_B, PKB' : \mathsf{pk}(\mathsf{agent}), N), \_ : \mathsf{nonce}))$.
2   $\mathsf{assert}(PKB' \doteq \mathsf{pk}(x_B))$.
$\star$ $\mathsf{let}\ PKB = \mathsf{pk}(x_B)$
    ;
    $\nu N_A : \mathsf{nonce}, R : \mathsf{nonce}$.
$\star$ $\mathsf{snd}(\mathsf{crypt}(PKB, \mathsf{f}_1(N_A, x_A), R))$.
$\star$ $\mathsf{if}\ x_B \in \mathsf{Honest}\ \mathsf{then}$
$\star$     $x_B \in \mathsf{Honest}$
$\star$ $\mathsf{else}$
$\star$     $x_A \doteq \gamma(x_A) \wedge x_B \doteq \gamma(x_B)$.
$\star$     $\mathsf{snd}(N_A)$
    ;
1   $\mathsf{rcv}(\mathsf{crypt}(\mathsf{pk}(x_A), \mathsf{f}_2(N_A, N_B : \mathsf{nonce}, x_B), \_ : \mathsf{nonce}))$.
    $\nu R' : \mathsf{nonce}$.
1   $\mathsf{snd}(\mathsf{crypt}(PKB, \mathsf{f}_3(N_B), R'))$
```

Let us project this to component 1, i.e., we keep all the steps that are part of NSL plus the interface of lookup. The projection corresponds to removing the three lines marked with 2. Note that one of the messages received is using pattern matching on privacy variables and a nonce: this is syntactic sugar for receiving a message with only fresh variables and a conditional statement such that the process stops if the message received does not contain the expected values (Definition 5.2.2). Thus removing this step involves projecting a protocol-specific conditional statement.

The projection of the initiator is effectively abstracting the behavior of the lookup procedure: assuming that the lookup does not lead to any attack (which would be checked when verifying the lookup component together with the interface of NSL), we can simply consider that the variable PKB is bound to the public key of agent x_B.

Note also that in this example of projection, we have the step $\nu N : \mathsf{nonce}, R'' : \mathsf{nonce}$ from the lookup procedure that remains in the projection, even though neither N nor R'' are used anymore (they were only present in messages specific to protocol 2). While in this example the generation of these nonces could be soundly removed from the projection, it makes no difference to the symbolic execution, and in general we need to keep such steps since we consider them as part of the interface. This is because in our approach, all freshly generated values like nonces are initially declared secrets: these values should not be accessible to the intruder unless they are explicitly declassified. In this particular example, it is clear that N and R'' cannot be leaked since they are never used. $\lhd$

5.3.2 Composability

In general, the composition of secure protocols is not necessarily secure. In this section, we identify a set of requirements that form the class of protocols called *composable*. These requirements control the interface between protocols and the messages they share.

As mentioned in previous sections, in this chapter we only consider well-typed instantiations. In an execution of the protocol, the concrete messages observed are ground instances of the terms that occur in the specification. We define below the notion of ground sub-message patterns and we will express our requirements using these ground patterns. Recall that $SMP(\cdot)$ (Definition 4.1.9) is closed under well-typed substitutions, and it also includes the key terms of encrypted messages.

Definition 5.3.3 (Ground sub-message patterns). *The set of* ground sub-message patterns*, $GSMP(M)$, of M is $\{t \in SMP(M) \mid t \text{ is ground}\}$.*

Let $\mathcal{T}_{pub} = \mathcal{T}(\Sigma_c \cap \Sigma_{pub}, \emptyset)$ denote the set of destructor-free public ground terms. Let $Spec = Spec_1 \parallel Spec_2$ be a composed protocol. Our result is parameterized over a set of secret terms that has to be given with the composed protocol. It can be convenient to have protocols that share terms, where the terms themselves are not secrets but contain secrets as subterms. For instance, some protocols may use the same public key certificates signed with a trusted private key. We support this by specifying such terms as secrets, and when messages are sent they can be marked as *declassified* so that the intruder is now allowed to learn those messages. Formally, let *Secrets* be a set of ground terms disjoint from $\mathcal{T}_{pub}$. Let $GSMP_i$ denote the ground sub-message patterns of the terms occurring in $Spec|_i$ for $i \in \{1, 2\}$, and let $GSMP_\star = GSMP_1 \cap GSMP_2$. The intersection $GSMP_\star$ corresponds to all ground messages (and their subterms) that are shared by the protocols.

We now list our requirements on composed protocols.

Definition 5.3.4 (Composability). *Spec is* composable *w.r.t. Secrets iff*

1. *For every conditional statement* if ϕ then P else Q*, if the branching is not marked with $\star$, then $Q =$* stop.
2. *For every cell read or cell write, if the memory cell is shared (i.e., it occurs in both $Spec_1$ and $Spec_2$), then the step is marked with $\star$.*
3. *If a process calls a procedure from the other protocol, then the procedure call is marked with $\star$.*
4. *In every formula released ϕ, all the variables $fv(\phi)$ are privacy variables chosen earlier in that role or procedure.*
5. *In every transaction, if there are two branches that send the same number of messages, then the marks of the messages sent must match.*
6. *Every role in $Spec|_1$ and $Spec|_2$ is well-formed (in the sense of Definition 5.2.1).*
7. *$GSMP_\star \subseteq \mathcal{T}_{pub} \cup Secrets$.*

Let us provide an intuitive explanation for the requirements:

1. If the branching is protocol-specific, i.e., not marked with $\star$, then we want the projection to abstract it away. Given an attack trace on the composed protocol, we will argue why it is not necessary for the intruder to reach the second branch given that it stops immediately, so that we only have to consider the projection of the first branch.

2. Shared memory cells are a way for the two protocols to interact, and in order for the projection to faithfully represent the behavior of the original role or procedure, all reads and writes on shared memory cells must also be present in the projection.

3. Procedure calls are another way for the two protocols to interact, and the values returned by these calls must also be present in the projection.

4. We exclude releasing information on variables that are bound in messages received, procedure calls or cell reads, i.e., variables that are not necessarily privacy variables. This is a strong restriction that seems necessary to obtain an intermediate result, which in short means that whenever a protocol is receiving a message, we can consider that the intruder sends a message from that same protocol (and not the other one) without loss of generality. Intuitively, since releases are changing the information allowed about privacy variables, it makes sense that an agent only releases information about the choices they have made themselves and not arbitrary messages that may or may not contain privacy variables chosen in other transactions.

5. In every reachable state, the frames in the different possibilities are indistinguishable and we must further have that a given label has the same mark in every possibility. Intuitively, the intruder should know whether a message is protocol-specific (and then from which protocol it comes from) or shared (and then it is always present in the projections).

6. When projecting to 1 or 2, the resulting (expanded) roles must be well-formed so that the semantics is well-defined for the projected protocols; in particular, the sequences of transactions after projection must be closed.

7. Messages shared by the two protocols must be either public or specified as secrets. We will have to verify that the components do not leak any secret, and if that is the case then we can show (also using Item 4) that the intruder never needs to use a protocol-specific message, say from protocol 1, when executing protocol 2.

5.4 Compositionality result

5.4.1 Compositionality on the frame level

In the rest of the chapter, we consider a composable protocol *Spec*. Before concluding on the compositionality for the protocol itself, we present results on frames. We give here informal proof sketches and all detailed proofs and intermediate results are provided in Appendix A.5.

Definition 5.4.1. *A ground frame F is* well-formed *iff for every mapping $i : l \mapsto t$ in F, we have $i \in \{1, 2, \star\}$ and $t \in GSMP_i$.*

The set of declassified terms *of F is $declassified(F) = \{t \mid \exists r.\ F|_\star(r) \approx t\}$. We say that F* leaks a secret from *Secrets iff there exist $t \in Secrets \setminus declassified(F)$, $i \in \{1, 2\}$ and r such that $F|_i(r) \approx t$. The frame F is* leakage-free *iff F does not leak any secrets.*

We will only consider well-formed frames.

We define that a secret is leaked when it is leaked in one projection of the frame instead of the full frame. This is crucial, because leakage-freeness is a requirement of the composition; due to our construction it is only necessary to check that the component protocols do not leak (rather than having to check their composition for leakage).

We introduce the notion of *homogeneous* recipes to describe recipes that can be used in a projection of the composed protocol, i.e., recipes that are not using protocol-specific messages from two different protocols.

Definition 5.4.2. *Let F be a ground frame and $i \in \{1, 2, \star\}$. A recipe r over $dom(F)$ is* i-homogeneous *iff every label in r is marked in F with i or $\star$. A pair of recipes (r_1, r_2) is i-homogeneous iff every label in r_1 and every label in r_2 is marked with i or $\star$.*

The results in this section are formulated for ground frames, and we will later use them with frames from a reachable state. Thus we only consider frames that have the same domain, with the same marks for the labels.

Definition 5.4.3. *Two ground frames F_1 and F_2 are* comparable *iff they have the same domain and for every label, they agree on its mark.*

The first result is that, if a frame does not leak any secret, then for every message occurring in the protocol execution, we can obtain a homogeneous recipe to produce that message.

Lemma 5.4.1. *Let F be a leakage-free frame, $i \in \{1, 2, \star\}$ and r be a recipe such that $F(r) \in GSMP_i$. Then there exists an i-homogeneous recipe r' such that $F(r) \approx F(r')$.*

Proof sketch. If r is not i-homogeneous, then there is a subterm of $F(r)$ that is in $GSMP_\star$ and thus this subterm must be either public or a secret. If it is a secret, then it must be declassified since the frame is leakage-free. Then the protocol-specific label producing the subterm can be replaced with either a public term or a recipe containing only labels marked with $\star$ (for the declassified case). $\triangleleft$

The second result is that, if both frames do not leak any secret and are not statically equivalent, then we can find a witness against static equivalence that is homogeneous. Recall that the intruder knowledge in a state is defined through static equivalence between the frames that the intruder considers possible in that state. Thus the lemma is useful to prove that a violation of privacy in a reachable state of the composed protocol can be mapped to a violation of privacy in a state of a projection of the protocol.

Theorem 5.4.1. *Let F_1, F_2 be leakage-free comparable frames. If for every $i \in \{1, 2\}$, $F_1|_i \sim F_2|_i$, then $F_1 \sim F_2$.*

Proof sketch. We proceed by contraposition and assume that $F_1 \not\sim F_2$, so there exists a witness against static equivalence. Then we apply a reduction strategy that successively replaces some labels in the witness with homogeneous recipes until the witness is homogeneous. During these reduction steps, we may find another simpler homogeneous witness, e.g., if two recipes for key terms suffice to distinguish the frames without needing to apply decryption. The main argument in the reduction is that if a recipe is not homogeneous, then there is a term that is shared between protocols and since the frames are leakage-free, we can find a homogeneous recipe for that term. In the end we get a homogeneous witness, which means that we can project the frames such that the witness is still well-defined (i.e., using only labels available in the projected frames). $\triangleleft$

5.4.2 Compositionality on the state level

Definition 5.4.4 (Leakage-free state and protocol)**.** *A state S is* leakage-free *iff the concrete frame concr in that state is leakage-free. A protocol is* leakage-free *iff every reachable state is leakage-free.*

The properties that we are verifying for a protocol are *reachability* properties, even privacy, so a protocol has an attack iff some reachable state has an attack. Besides the main focus on (α, β)-privacy properties, we also need to check that no assertion is broken and that no secrets are leaked.

Definition 5.4.5 (Attack state and attack trace). *A milestone S is an* attack state *iff at least one of the following is true:*

- *The flag in S is set to true.*
- *S is not leakage-free.*
- *S does not satisfy privacy.*

Let $(P_1; \ldots; P_n, \gamma, \rho)$ be a trace and $S_0, \ldots, S_n$ be the milestones such that for every $i \in \{0, \ldots, n-1\}$, executing P_{i+1}, starting from S_i, leads to S_{i+1}, following the truth γ and using the recipes in ρ (recall that by Definitions 5.2.7 and 5.2.9 a trace determines a unique sequence of milestones, because γ indicates how the ***Choice*** *transitions, and others depending on the true possibility, are taken, and ρ the* ***Receive*** *transitions).*
$(P_1; \ldots; P_n, \gamma, \rho)$ is an attack trace *iff*

- *For every $i \in \{0, \ldots, n-1\}$, S_i is not an attack state.*
- *S_n is an attack state.*

Finally we can state our main result: if no projection of the protocol has an attack, then also the composed protocol has no attack.

Theorem 5.4.2. *If for every $i \in \{1, 2\}$, $Spec|_i$ has no attack, then Spec has no attack.*

Proof sketch. We proceed by contraposition and assume that there exists an attack trace on the composed protocol. Our goal is to prove the existence of an attack trace on a projection of the protocol, i.e., to show that there is an attack on one protocol composed with the interface of the other protocol.

The first step is to show that we can assume the intruder chooses homogeneous recipes without loss of generality, i.e., whenever a protocol is receiving a message, then the intruder uses labels from that same protocol (or declassified messages). Since we only consider well-typed instantiations, we can use our result of Lemma 5.4.1 to show that whenever a process is receiving a message, then the intruder can use a homogeneous recipe to produce that message. This step is actually quite difficult because we need to show that the change to homogeneous recipe does not significantly alter the reached state.

As a second step, we consider transactions that went into an else branch for a conditional that is not marked $\star$, i.e., that is protocol specific. Here the problem is that such a branching is not possible in the projection to the other protocol, but we can use our requirement that the else branch for protocol-specific conditionals must be stop. Thus, the transaction can only have the effect to show that the respective condition is false. So either that gives an attack on privacy, or we can remove this transaction from the trace.

Finally, we make a case distinction on the kind of attacks: if some assertion was broken or some secret was leaked, then the projected attack trace also leads to a broken assertion or secret leaked. If the attack is instead a violation of privacy, i.e., there is a model of privacy variables allowed by the payload α that the intruder can actually rule out given their knowledge, then we show that either we find a "simpler" attack that is just a secret leaked or we also have a violation of privacy from the projected trace. $\triangleleft$

5.5 Application of the result and limitations

In order to apply our compositionality result, the workflow is the following:

1. The modeler has to specify a composed protocol; part of the specification is the assignment of types to constants and variables, the explicit marking of process steps that form the abstract interface of components and also the set of shared secrets.

2. The composability requirements of Definition 5.3.4 have to be checked. Since one requirement is that every projected role be closed, at this point we can find specification errors. For instance, if we have a variable used in a shared message (part of the interface) and the step binding that variable is marked protocol-specific, then the projection to other components would miss the binding. Therefore, the modeler might have to adapt the markings for the interfaces, but we can give feedback on the specification error to help the modeler.

3. For each component, the specification is projected to that component (keeping the interface of other components) and the goals of the resulting specification have to be verified (i.e., we check the assertions, leakage of secrets and (α, β)-privacy for each component, not for the entire composed protocol).

The composability requirements we list in Definition 5.3.1 are sufficient, not necessary: we are not claiming that every protocol *has* to meet our requirements in order to enable compositional reasoning. However, we have justified why we have each requirement, i.e., why we failed to find a feasible proof argument without these restrictions. In particular, we have Requirement 1 that says else branches must be just stop if the conditional statement is not part of the interface. The motivation is to be able to ignore such branching, when the condition is local to the process and not relevant for other components. In our examples, we make use of this with syntactic sugar when a message received is marked as protocol-specific: most pattern-matching examples in Fig. 5.1 stand for receiving a fresh message and then performing equality checks; in case any value is not equal to the expectation, the process stops. The point of compositionality is that we can indeed ignore steps such as receiving protocol-specific messages, and checks on them, from the point of view of the other components.

Let us return to the running example of Fig. 5.1. We now consider a variant where, in the lookup procedure, we omit the random number N that should be included in a request and the corresponding response. The component is given in Fig. 5.2. We show how the omission of the nonce induces a privacy violation that we find when verifying the lookup component with the abstract interface of the initiator role. In order to verify the security of this lookup component, we still have to consider the steps in the initiator and responder roles that were marked as part of the interface (highlighted in Fig. 5.1). In particular, the choices of privacy variables are present in this component.

The first time that the initiator role (its projected version) is executed, two privacy variables x_A and x_B are chosen, and there is a call to the lookup procedure. After receiving the request, the server responds with a message containing the public key of x_B. At this point, the intruder has observed the messages $l_1 \mapsto \mathsf{scrypt}(\mathsf{sk}(x_A, \mathsf{s}), \mathsf{req}(x_B), \mathsf{r}_1)$ and $l_2 \mapsto \mathsf{scrypt}(\mathsf{sk}(x_A, \mathsf{s}), \mathsf{resp}(x_B, \mathsf{pk}(x_B)), \mathsf{r}_2)$.

Now we consider that the initiator is started a second time. This leads to choices of new privacy variables x'_A and x'_B and the intruder observing two messages for the new request and response: $l_3 \mapsto \mathsf{scrypt}(\mathsf{sk}(x'_A, \mathsf{s}), \mathsf{req}(x'_B), \mathsf{r}_3)$ and $l_4 \mapsto \mathsf{scrypt}(\mathsf{sk}(x'_A, \mathsf{s}), \mathsf{resp}(x'_B, \mathsf{pk}(x'_B)), \mathsf{r}_4)$. Finally, we resume the process from the first call to *lookup*, but the intruder uses message labeled l_4 instead of the expected l_2. That is to say, the intruder forwards the second

```
Procedure $\mathit{lookup}(A : \mathsf{agent}, B : \mathsf{agent})$ :
$\nu R : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{scrypt}(\mathsf{sk}(A, \mathsf{s}), \mathsf{req}(B), R))$
;
$\mathsf{rcv}(\mathsf{scrypt}(\mathsf{sk}(A, \mathsf{s}), \mathsf{resp}(B, \mathit{PKB} : \mathsf{pk}(\mathsf{agent})), \_ : \mathsf{nonce})).$
$\mathsf{assert}(\mathit{PKB} \doteq \mathsf{pk}(B)).$
$\mathsf{return}(\mathsf{pk}(B))$

Role $\mathit{Server}$ :
$\mathsf{rcv}(\mathsf{scrypt}(\mathsf{sk}(A : \mathsf{agent}, \mathsf{s}), \mathsf{req}(B : \mathsf{agent}), \_ : \mathsf{nonce})).$
$\nu R : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{scrypt}(\mathsf{sk}(A, \mathsf{s}), \mathsf{resp}(B, \mathsf{pk}(B)), R))$
```

Figure 5.2: Insecure variant of the lookup component

response from the server instead of the first one. In the process that receives this response, the message is expected to be encrypted with the shared key $\mathsf{sk}(x_A, \mathsf{s})$ and to contain the public key of x_B, however in the execution we are considering, the message is encrypted with the shared key $\mathsf{sk}(x'_A, \mathsf{s})$ and contains the public key of x'_B. The matching on x_A and x_B is actually syntactic sugar for equality checks, which means that the process continues only if $x_A \doteq x'_A \wedge x_B \doteq x'_B$, and simply stops otherwise. Since the fact that a process stops is observable by the intruder, at this point they can deduce whether $x_A \doteq x'_A \wedge x_B \doteq x'_B$. This is a privacy violation since the payload α only specifies in which domains the variables are but no relation between them.

The example of privacy violation that we have described happens in the composed protocol because the lookup component is insecure, and we have explained how the verification of the lookup component (instead of the entire composed protocol) is enough to find this attack.

Since our result supports stateful protocols that read from and write to memory cells, the protocol from the running example could potentially be developed further to allow agents to update their private/public key pairs: instead of having a fixed public-key infrastructure, the server could read from a database of public keys, and additional transactions could model an agent write to their own memory cell for updating the key. In such a protocol, the memory cells containing the public keys would be shared between several components, thus in the model we should mark the steps of reading from and writing to memory as part of the abstract interface.

For our compositionality result, when verifying a composed protocol, we need to consider each component together with the abstract interface of the other components. This means that we are not verifying each component completely in isolation. This can be seen as a limitation, since we cannot derive the security of the composed protocol “for free” from the security of the components. However, there is also benefit in the specification of protocol interfaces: if one wants to swap a component with another implementation that has the same interface, then we only need to verify that new component with the interface of the others.

For generality, we have considered components that can interact through messages, memory cells and procedure calls. A special case of composition is when the different components are executed concurrently but without any interaction: it is typical that several protocols having nothing to do with one another run on the same machine. This suggests a "disjoint" case where there are no shared messages, no shared memory cells and no procedure calls from one protocol to the other. By Definitions 5.3.1 and 5.3.2, there are some steps in processes such as non-deterministic choices and generation of fresh constants that always remain in every projection. Thus, even for protocols that are disjoint enough, we would have that the projection to one component contains some steps from the other components because interfaces are basically never empty. However, it may be possible to detect cases such as, e.g., when the privacy variables from one protocol are used in a way such that they can never be reused by the intruder in other protocols. This would enable complete modular verification, for these special disjoint cases.

An important limitation of our compositionality result is the restriction to constructor/destructor theories according to Definition 3.4.1. This is used in the proofs showing that for composable protocols, when executing transactions from one component, the intruder never needs to use messages coming from other components. We can still model many standard cryptographic operators, but we cannot model for instance operators such as exponentiation or blind signatures, which means that our compositionality result is not applicable to protocols using, e.g., Diffie-Hellman key exchange or commitments.

5.6 Related work

There is a number of works that show compositionality for standard trace-based properties in the symbolic (Dolev-Yao-style) model [48, 29, 5, 24, 53]. The closest to ours is [53] which shows a compositionality result for stateful protocols. Here the state is represented by a family of sets of messages, e.g., a set of key registered as valid at a server. These sets can be shared between the component protocols. This is similar to our work where we have instead a family of memory cells which can only hold one message and also in our case these can be shared between the components. This is because previous work on (α, β)-privacy [47] uses such memory cells, but we want to investigate in future work whether we can also support sets. From [53] we adopted the idea that the protocols do not need to be completely disjoint but can share a set of messages that either are public or initially secret.

As already mentioned there are very few works on symbolic compositionality for privacy-type goals. The standard model for privacy goals is based on a notion of indistinguishability, i.e., the intruder cannot tell two processes apart. In a nutshell, for every trace that one process can exhibit, one has to show that the other process can have a similar trace in the sense that the intruder frames are indistinguishable. This is particularly challenging in case of conditionals, because the intruder may in general not be able to tell whether the condition was true and thus the then or the else branch was the case. Note that in (α, β)-privacy this is handled by maintaining a number of possibilities $\mathcal{P}$: the rule **Conditional** in Table 5.1 says that a possibility with a conditional statement is split into two possibilities for the positive and negative branch. Moreover, the rule **Eliminate** says that the intruder can then discard possibilities that are not compatible with observations, and keeps all that are. As this is a major difficulty in approaches for unbounded session verification of trace equivalence, a restriction is often considered: bi-processes and diff-equivalence [36, 27]. Here, the two processes that should be indistinguishable for the intruder differ only in the concrete terms which come in two variants, and one requires that all conditionals are satisfied for both or for neither variant, so that it is always either the then or the else case. This allows to easily turn the problem into a trace problem that is much easier to deal with.

The most advanced work on compositionality for trace-equivalence that we are aware of, [5], also uses the restriction to bi-processes and diff-equivalence. (They do also have a result for the standard trace properties that does not need this restriction, of course.) They show results for both the parallel case and a sequential case where one protocol generates a key and another protocol uses it. Our compositionality result is significantly more general than this: we do not need the restriction to bi-processes, i.e., we consider the standard model of (α, β)-privacy where all branches of conditionals are possible and maintained. For what concerns different types of composition, the generality of our result blurs the boundaries between concepts like parallel and sequential composition: in our parallel composition the components can actually communicate with each other, this just has to be part of the interface, either via memory cells and via shared messages (and declassification). Communication via memory cells actually allows for sequential composition: for instance, the first protocol could establish a key and write it to a shared memory cell, while the second protocol reads the value of that memory cell to use as a key in the rest of the communication. Moreover, we can employ components as subprotocols that can be invoked by other components.

We believe that the reason we can make such a generalization with complicated but still manageable proofs is due to the technically more simple concepts of (α, β)-privacy: we deal with a simple reachability problem where each state carries enough information to model the intruder reasoning, namely which models of the relations and privacy variables are still compatible with all observations (and messages sent by the intruder). This allows us to show that any trace of the composed system, when projected to the steps of one component and the shared ($\star$-marked) steps, still works with that component alone; thus the security of every component implies the security of the composition.

There are however also some aspects where [5] is more general. First, they allow for one component to use Diffie-Hellman (however it cannot be shared between components), which is valuable for key-exchange protocols. We plan to investigate as future work how to support more algebraic properties. Moreover, we require a typed model while the requirements of [5] seem less restrictive—except for Diffie-Hellman where they essentially have a strictly typed model.

Chapter 6

Tool support

Sections 6.2 and 6.3 in this chapter are based on [41].

When writing a protocol specification for (α, β)-privacy, the modeler needs to specify what information is released in α, but does not specify β directly. Rather, the modeler specifies a set of *transactions* describing how honest agents send and receive messages and update their local state. The β of every reached state is then defined by the semantics of the transaction language and reflects all observations and deductions the intruder can make. Thus the modeler only needs to specify the protocol itself and what information is released in α and that is a positive specification, i.e., what the intruder is allowed to know. Thus, in the worst case the modeler errs on the safe side: if one forgets to specify something that the intruder can actually derive, then (α, β)-privacy is violated and the `noname` tool presents a violation. Then the modeler can decide whether the information that the intruder can derive is indeed acceptable, and specify that by adding an appropriate α release, or otherwise strengthen the protocol. In this way one can even *explore* what privacy guarantees a protocol gives by starting with α containing only what is obviously released and then incrementing the release until no more violations occur.

Our fourth contribution in this thesis is the implementation of the decision procedure from Chapter 3 in the `noname` tool. Our fifth contribution is the detailed case studies for the BAC [57] and Private Authentication [1] protocols, with the discussion of several variants of the protocols, explanations on how to apply the tool to the models and how to understand attack traces.

The chapter is organized as follows. In Section 6.1, we explain how to write an (α, β)-privacy specification and analyze it with the `noname` tool. In Sections 6.2 and 6.3, we go into details about the modeling and analysis results for the case studies of BAC and Private Authentication, respectively.

6.1 Brief introduction to `noname`

6.1.1 Writing a specification

The reference for the tool is the release of `noname` version 0.3 [39]. The rest of the chapter applies to this specific version and in case of a different version one should check the documentation provided with the tool.

We consider again the simple protocol from Example 2.2.1, with the transaction:

$$\begin{array}{l}\star\ x \in \mathsf{Agent}.\star\ y \in \{\mathsf{yes},\mathsf{no}\}.\\ \mathsf{rcv}(M).\\ \mathsf{try}\ N := \mathsf{dcrypt}(\mathsf{inv}(\mathsf{pk}(\mathsf{s})), M)\ \mathsf{in}\\ \quad \mathsf{if}\ y \doteq \mathsf{yes}\ \mathsf{then}\\ \qquad \nu R.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{pair}(\mathsf{yes}, N), R)).0\\ \quad \mathsf{else}\ \nu R.\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(x), \mathsf{no}, R)).0\\ \mathsf{catch}\ 0\end{array}$$

We now present how to express the protocol specification as a text file parsed by the `noname` tool. The first thing to write is the declaration of the symbols used in the protocols:

```
Sigma0: public yes/0 no/0 a/0 b/0 i/0 s/0
```

Here we only have to declare a few public constants: `yes` and `no` represent the possible decisions of the server; `a`, `b`, `i` and `s` are four agent names, where `a` and `b` will be used as the honest agents, `i` the intruder (a dishonest agent) and `s` the server receiving the message and sending a reply. These constants are all declared as part of Σ_0: they are concrete values that the privacy variables are chosen from.

In general, the syntax for declarations is `f/n` where `f` is a function name and `n` its arity. Private functions can be declared after the public functions: one has to write the keyword `private` followed by the declarations. Similarly, we can declare symbols in Σ, i.e., technical symbols that are used in the processes but not part of Σ_0, by writing `Sigma:` followed by the function declarations as above.

Next, one may define rewrite rules for the algebraic properties of operators, in the form: `Algebra: d(k',c(k,X1,...,Xn))->Xi ....` For instance, the behavior of symmetric encryption is written as `dscrypt(X,scrypt(X,Y,Z))->Y`. By default, `noname` includes the standard set of cryptographic operators from Fig. 2.1, with the addition of the public function `pk`: these functions and their algebraic properties do not need to be written explicitly. For our running example, there is no algebraic property to write in the specification because we are only using standard operators.

Now comes the specification of transactions: each transaction is given a name, followed by the process for that transaction. Let us define two transactions:

```
Transaction ReceivePrivateKey:
send inv(pk(i)). nil

Transaction Server:
* x in {a,b,i}.
* y in {yes,no}.
receive M.
try N:=dcrypt(inv(pk(s)),M) in
  if y=yes then
    new R.
    send crypt(pk(x),pair(yes,N),R). nil
  else
    new R.
    send crypt(pk(x),no,R). nil
catch nil
```

The `Server` transaction corresponds exactly to the process in this running example. We have added the `ReceivePrivateKey` transaction here to model the compromise of the dishonest agent. This transaction simply sends the private key `inv(pk(i))` of agent `i` over the network. Given our intruder model, this message can be observed by the intruder and they are then able to use this private key in the rest of the execution.

This transaction sending a private key is just one way of modeling the compromise of dishonest agents. One alternative would be to have built-in the tool this special agent `i` as the intruder with their own private key. Another option would be to support the declaration of initial knowledge, i.e., a set of messages that are added before the protocol is started, so that in the initial state the frame is not empty but contains these messages. One advantage of our modeling with a transaction is that the compromise does not have to happen at the beginning, but may be executed later in the protocol. This may be relevant when verifying protocols such that parties are initially trusting each other and only later on one of them becomes compromised. In our semantics of the symbolic execution, any transaction can always be executed, thus the disadvantage of having a transaction sending the private key is that it may happen multiple times. When setting the bound to 2, we will for instance consider traces where `ReceivePrivateKey` is executed twice in a row, which is unnecessary. This suggests the specification could allow the modeler to give some hints about the relevant traces to verify, for example we could mark a particular transaction to be executed at most n times, while other transactions can always be executed; another possible improvement would be to give a partial order between transactions.

Finally, the specification should provide a bound on the number of transitions (if the bound is not declared, the default is 0). One may write any integer here; for our running example, 2 is enough because we know that there is a violation of privacy (Example 3.2.2).

```
Bound: 2
```

Recall that the tool stops as soon as it finds a violation of privacy, so even if we were to write a greater number, the result would be exactly the same.

For the syntax of specifications, we have omitted here the declarations of relation symbols, interpretation of these relations, and ground context for the initial values of memory cells. Examples for these are provided in the models of case studies in Appendix B.

6.1.2 Analyzing a specification

We assume that the specification described in the previous section is the content of a file named `runex.nn`. In order to verify whether this protocol satisfies (α, β)-privacy, we execute the command `noname runex.nn`. We get the following output[1]:

```
Privacy violation found after 2 transactions.
alpha = x2∈{yes,no}∧x1∈{a,b,i}
beta_0 = (x1=i)∧(x2≠yes)
(alpha, beta_0)-privacy does not hold.
Model found: x1=i∧x2=yes
State: Executed = ReceivePrivateKey.Server
Recipe choice = [R1->l1,R2->crypt(pk(s),R4,R5),R3->pk(s),R6->s,R7->l3,R8->
    l1]
alpha_0 = x2∈{yes,no}∧x1∈{a,b,i}
beta_0 = (x1=i)∧(x2≠yes)
gamma_0 = ⊤
Possibilities = {(nil,(x1=i)∧(x2≠yes),[-l1->inv(pk(i)).-l2->pk(i).+R4->X11
    .+R5->X13.-l3->crypt(pk(x1),no,n2).-l4->no.-l5->inv(pk(i))],{},⊤,[])}
Checked = {(l4,no),(l1,l5),(l2,pk(i))}
```

[1]We omit here the marks for analysis of messages and additional lines about the exact model returned by the SMT solver, because these are mostly useful for debugging purposes.

The first line tells use that there is a violation of privacy, as expected. The issue is that the recipient of the server's message may be the dishonest agent `i`, in which case the intruder learns the value of both privacy variables chosen by the server. In the output, we have that two transactions were executed: first `ReceivePrivateKey` and then `Server`. The privacy variable `x1` corresponds to the agent name (`x` in the transaction) and `x2` to the server's decision (`y` in the transaction). During the symbolic execution, there were at some point two possibilities, depending on whether `x2=yes`. After the server transaction finished, the messages observed by the intruder were analyzed, and here we considered the states where `x1=i`, i.e., the intruder used the private key `inv(pk(i))` to decrypt the server's message and the decryption was successful. This added a label `l4` mapping to the content of the message, either `pair(yes,x1)` or `no` depending on the server's decision. Finally, among the intruder experiments that were performed, there is the comparison between label `l4` and constant `no`. Here we considered the states where the two recipes produced the same message, which ruled out the case that `x2=yes`. The intruder deductions are summarized in the formula β_0. There is a violation of privacy because the intruder has learned more than what is allowed by the payload α.

Note that the output also contains a model that interprets the privacy variables. We find a violation of privacy by checking whether (α, β_0)-privacy holds, i.e., whether the symbolic state is consistent (Definition 3.3.5). The countermodel is an interpretation that is a model of α but that cannot be extended to a model of β_0. Here the countermodel is simple: α only discloses information about the domain of privacy variables, while in β_0 the intruder has deduced that one particular interpretation must hold.

If we update the specification to fix the issue, i.e., in case of a dishonest recipient the server releases the values of the privacy variables and in case of an honest agent the server releases that the agent is indeed honest and not the intruder (Example 3.2.2), then we get the following output:

```
Bound reached, no privacy violation found after 2 transactions.
```

In addition to the file containing the protocol specification, `noname` takes several command-line arguments, for instance to indicate a path to write the output to or to enable the interactive mode. The user may also specify the bound as a command-line argument. In this case, the argument overrides the bound specified in the text file. The different options can be found by running `noname --help` or in the `README.org` provided along with the source files for the tool [39].

6.2 Case study: BAC

Our first case study is the unlinkability in the Doc 9303 BAC protocol for passport readers [57], where we replicate known problems in some implementations [6, 30, 45]. The BAC protocol is an RFID protocol used for passports with RFID card readers. We simplify matters for two reasons: it allows us to focus on the actual issue, avoiding complicated and irrelevant modeling challenges, and moreover, the `noname` tool quickly runs into state space explosions otherwise. The full specification is found in [39].

The first step is that an RFID tag `x` starts a new session, with a new random nonce `N` that it sends to the tag reader. There is a unique key `sk(x)` for each tag `x` that is derived from the passport data (that the reader obtains using OCR). The reader uses `sk(x)` to encrypt `N` as a response to the challenge, as well as a challenge `N'` of its own. We omit `N'` here for simplicity. Also for simplicity, not to have to model the exchange of `sk(x)`, we put these two steps into one atomic transaction.

```
Transaction Challenge:
```

```
2  * x in {t1,t2}.
3  new N. send session(x,N). send N.
4  send scrypt(sk(x),N). nil
```

Here, Line 2 means that x is non-deterministically chosen from a set of two tags {t1,t2} (we have to bound the number of tags tightly for the tool performance). The * symbol indicates that the information is added to α: at this point the intruder is allowed to know that x in {t1,t2}, but nothing more. If this transaction is executed repeatedly, then the variables will all be freshly renamed each time, the intruder obtains an α formula like x1 in {t1,t2},...,xn in {t1,t2}, saying that there have been n transactions performed by some tags x1,...,xn and the intruder is not allowed to learn anything more than that they are tags. In particular the intruder is not allowed to know whether or not x1=x2 holds. This is indeed all that needs to be specified to formulate unlinkability as a goal in (α, β)-privacy.

Line 3 creates a new random number, sends it out on the network (so the intruder can observe it), and we send also a special *pseudo-message* session(x,N), i.e., a message that only exists in our model: it formalizes the session state of the tag x were session is a private function that the intruder cannot apply. Line 4 represents the answer that the server sends: a symmetric encryption (scrypt) with key sk(x) and content N. (In this case study, we use scrypt as a binary function to model deterministic encryption; in Appendix B.4, the third argument for the randomness is set to a constant.) The nil command finishes the transaction.

The next step is that the tag receives the messages from the reader, tries to decrypt it, check that it contains the number N challenged to the server, and sends an error message otherwise:

```
1  Transaction Response:
2  receive Session.
3  receive M.
4  try X:=sfst(Session) in
5  try N:=ssnd(Session) in
6  try NN:=dscrypt(sk(X),M) in
7    State:=noncestate[N].
8    if N=NN and State=fresh then
9      noncestate[N]:=spent.
10     send ok. nil
11   else send nonceErr. nil # nonce check failed
12 catch send formatErr. nil # decryption failed
13 catch nil
14 catch nil
```

In Line 2, the tag actually tries to retrieve its session state (that we had sent as a pseudo-message in the other transaction). This is supposed to be of the form session(X,N) consisting of the tag name and challenge of that session. To check and extract this information, we have two private functions (i.e., functions not accessible to the intruder) called sfst and ssnd with the property that they return X and N respectively if the given message has the form session(X,N), and give an error otherwise. The try-in-catch construct accordingly continues with the in part if there is no error, and to the catch part otherwise; here that would be the last two lines where the transaction does nothing.

In Lines 3 and 6, the tag receives the actual message that (supposedly) the tag reader has sent and tries to decrypt it with its key sk(X) (where X is the value just retrieved from its session state). The function dscrypt is public, i.e., also the intruder can apply it with known keys, but the intruder in this example does not know any sk(X). (One could model that the intruder has its own passport with tag tI and knows the key sk(tI).) Again the decryption function either returns the content if this is a symmetric encryption with the given key, or an error otherwise. In the error case (Line 12) the tag responds with a `formatErr` code.

The next step is to compare the received nonce `NN` with the actual nonce `N` from the session state. Here we have however a slight challenge in modeling: since the session state is modeled here as a pseudo-message that was sent in the first transaction and received back in the second, an intruder can replay an old session state that was actually already processed by the tag, and we cannot prevent that in our model. However, (α, β)-privacy transactions have a notion of long-term state that we can use. Here we use an (infinite) array of memory cells `noncestate[.]` that is initialized with `fresh`. In Line 7 we retrieve the state of the nonce `N` in question, check that the state is still `fresh` in Line 8, and then set it to `spent` in Line 9, effectively blocking a nonce from being used twice. Finally the tag responds with an `ok` message or a `nonceErr`.

6.2.1 The attack

A sequence of three transactions is sufficient to get to a violation of (α, β)-privacy. We start with two Challenge transactions giving `alpha = x1,x2 in {t1,t2}` and respective messages observed by the intruder:

```
l1 -> session(x1,N1)          l3 -> session(x2,N2)
l2 -> scrypt(sk(x1),N1)       l4 -> scrypt(sk(x2),N2)
```

Next, we execute the Response transaction where the intruder chooses for `Session` the message `l1` and for `M` the message `l4`. Now there are two possibilities for what can happen: either `x1=x2`, then the decryption works (Line 6 of the Response transaction), but the nonce check fails (Line 8), or `x1/=x2` and then already the decryption fails. The error message by the tag is `nonceErr` in the first case, and `formatErr` in the second case, so the intruder now knows whether or not `x1=x2`. Since this does not follow from α, the observations β of the intruder violate (α, β)-privacy, and we can find this attack with the `noname` tool.

This attack was first reported by Arapinis et al. [6] and it is interesting that the French implementation of the protocol was vulnerable to this attack, while the British implementation was not. The Doc 9303 standard [57] requires the tag to send error messages when receiving an ill-formed or incorrect message from the reader, however this standard does not prescribe what the error message should be. In the French implementation, two different error messages were used (represented here with `nonceErr` and `formatErr`), while the British implementation uses the same error message in both cases, and in fact, doing so we can verify the specification with the `noname` tool (for up to four transactions).

6.2.2 Another problem

Filimonov et al. [45] actually pointed out that the protocol has another problem that is here (and in several other models) buried by the fact that the first exchange between tag and reader, namely the nonce `N` from the tag and the response `scrypt(sk(x),N)` from the reader is merged into one transaction. This does not allow for a possible confusion that can arise when multiple tags in parallel are shown to the same or different readers.

We thus split the Challenge transaction into two transactions.

```
Transaction InitSession:
* x in {t1,t2}.
new N.
send session(x,N).
send N. nil

```

```
Transaction Challenge:
receive Session.
receive N.
try X:=sfst(Session) in
  send scrypt(sk(X),N). nil
catch nil
```

Here the `InitSession` is the part of the tag creating a new nonce and session state, and this is where the only augmentation of α occurs, thus one may not learn more than that `x` is a tag.

The `Challenge` transaction now receives the pseudo-message `Session`, which simply models that the reader receives the (claimed) identity `X` of the card and can compute the key `sk(X)`. Note that we would be "cheating" if the server also received the nonce `N` from the session state, because that is actually transmitted over a public channel that the intruder can access and manipulate. This `Challenge` transaction now allows for the confusion that the reader gets the claimed identity and shared key from one passport, and the nonce from another.

Now suppose the following transactions: two tags `x1` and `x2` (possibly the same) perform the transaction `InitSession` and the intruder sends the session message from `x1` and the nonce `N2` from `x2` to the server in the `Challenge` transaction, who thus produces `scrypt(sk(x1),N2)`. The intruder feeds this message to the second tag, i.e., executing the `Response` transaction with the session state of `x2`. This will give the `ok` message if and only if `x1=x2`.

This attack can be found by the `noname` tool, however due to complexity, we introduced a "guardrail", guiding the tool to first execute two `InitSession`, followed by a `Challenge` and a `Response`. With this guidance we prune other interleavings from the search tree, and it is then small enough to find the attack in a reasonable amount of time. We also verified with the tool under this guardrail that the attack does not exist when encrypting the responses from the tag. The particular attack trace can also be found by running the tool in interactive mode, where the user can select which transactions are executed and which messages the intruder sends.

We observe here a clear advantage of (α, β)-privacy: the attack description in [45] requires one page of explanation and an understanding of their particular notion of bi-similarity between processes. It may be impossible to make that intuitive to non-mathematical readers because it refers to a world in which only one tag exists, so that the above strategy of the intruder cannot lead to an error. In contrast our specification of the privacy goal is rather simple (the intruder may not find out more about the involved tags other than that they are tags) and also the violation is: confusing the steps of two parallel sessions leads to an observable error message unless the two sessions are with the same tag.

Finally, one may wonder if this is really an issue, because RFID tags do not actually really perform multiple sessions at the same time. If every tag works strictly sequentially, i.e., a new session can only be started once the current session is finished or timed out, then the attack is also prevented. However, this situation is also troublesome: since the intruder also knows that tags cannot participate in two sessions at the same time, the successful completion of two interleaved sessions means that distinct tags are involved. The encryption of all response messages also prevents this attack.

6.3 Case study: Private Authentication

Our second case study is the Private Authentication by Abadi and Fournet [1] that hides as far as possible the identity of participants as well as the fact which participant is willing to talk to which other participants. To our knowledge this is the tightest characterization of the privacy goals that this protocol enjoys.

We denote the Private Authentication protocol with AF for short. The protocol contains two roles, initiator and responder. The initiator sends an encrypted message containing a nonce to the responder, who either replies with a nonce for authentication or with a decoy message. The purpose of the decoy is to hide the fact that the responder does not want to talk to the initiator, or that the incoming message does not have the right format. The intruder should not be able to tell the difference between a decoy message and a properly encrypted reply. AF is parameterized over a specification of which agent is willing to talk which other agents. We first look at simple variant AF0 where everybody is willing to talk to everybody.

6.3.1 AF0: initial attempt

We consider three agents `a,b,i` in this specification where `a` and `b` are honest, and `i` is the intruder, all of which can play as participants here. Each participant `x` has a public key `pk(x)` and the corresponding private key `inv(pk(x))`. The intruder knows all public keys (because `pk` is a public function, and agent names are public constants) and their own private key `inv(pk(i))`.[2]

The first transaction describes how an honest agent `xA` initiates communication with a (possibly dishonest) agent `xB` (the case of a dishonest `xA` is already covered by the intruder model).

```
1  Transaction Initiator:
2  * xA in {a,b}.
3  * xB in {a,b,i}.
4  if xB=i then
5    new NA,R.
6    send crypt(pk(xB),pair(xA,NA),R).
7    * xA=gamma(xA) and xB=gamma(xB). nil
8  else
9    new NA,R.
10   send crypt(pk(xB),pair(xA,NA),R).
11   * xB in {a,b}. nil
```

Like in the previous case study, Lines 2 and 3 specify the non-deterministic choices of agent names from the respective domains, and thus specify that the intruder so far is only allowed to know that `xA` and `xB` are chosen from these domains. The initiator sends an encrypted message to the recipient, containing a pair of their name and a fresh nonce `NA`. (`R` is also a nonce for randomized encryption.) If the recipient is dishonest, then the intruder will learn the values of `xA` and `xB` from this message (knowing the private key to decrypt it). Thus we get in this case a violation of (α,β)-privacy unless we release this information, which we do in Line 7. Here the formula `x=gamma(x)` means that we release the true value `gamma(x)` of `x`: `gamma(x)` will be replaced with the true value of `x` when reaching this point. Also in the case that the recipient is honest, the intruder can learn something from the fact that they cannot decrypt the message: that `xB` is honest, which we release in Line 11. Releasing means that we *allow* this information to be known by the intruder, so that it does not count as an attack if the intruder finds this out. If we had forgotten any of these releases, the `noname` tool would have notified us with an attack. Note that, except for the α-release, the `then` and `else` branches are identical: they reflect the steps that `xA` indeed performs, while the `if-then-else` and α-releases are only for specifying the privacy goal.

The response is now described from the perspective of an honest `xB`.

[2]As in Section 6.1, we model this with an additional transaction that simply sends the private key.

```
1  Transaction Responder:
2  * xB in {a,b}.
3  receive M.
4  try DEC:=dcrypt(inv(pk(xB)),M) in
5    try A:=proj1(DEC) in
6      if A=i then
7        new NB,R.
8        send crypt(pk(A),NB,R).
9        * xB=gamma(xB). nil
10     else
11       new NB,R.
12       send crypt(pk(A),NB,R). nil
13   catch new NB. send NB. nil
14 catch new NB. send NB. nil
```

It may be surprising that `xB` is here also non-deterministically chosen. The example protocol actually leaves the communication model abstract and just models that a message may arrive at *any* participant, since this may be caused by the intruder. `xB` receives a message `M` that they try to decrypt with their private key. The operator `dcrypt` satisfies the equation `dcrypt(inv(K),crypt(K,M,R))=M`. If the decryption succeeds, the result `DEC` must be a pair of a sender name `A` and a nonce. To obtain `A`, the responder uses `proj1` which satisfies the equation `proj1(pair(X,Y))=X`. If this succeeds, `xB` sends an answer encrypted for `A` containing a fresh nonce `NB` and randomization `R` (Lines 8 and 12). As before, we must take into account what the intruder can learn if `A=i`: since then the answer is decipherable for them, they learn the name of `xB` (in case they did not know already). If anything goes wrong (if decryption fails or the content is not a pair) then the recipient sends a decoy message, a random nonce `NB` that is not distinguishable from an encrypted message that the intruder does not have the decryption key for.

The attack

The `noname` tool reports an attack on this specification. In this attack, only the `Responder` transaction was executed where the intruder provided as input for message `M` the following recipe: `crypt(pk(a),pair(i,R8),R5)` where `R8` and `R5` are recipe variables that stand for arbitrary recipes. The intruder has thus sent a message to recipient `a` under their true name `i`. We have thus two cases. First, `xB=a` and thus the message is correctly decrypted by `xB` and the intruder receives as an answer `crypt(pk(i),NB,R)` for some fresh values `NB` and `R`. Second, `xB/=a` and the decryption fails and `xB` sends a decoy message `NB`. This is of course observable for the intruder, since they can decrypt in the first case, but not in the second. The concrete state that the `noname` tool presents is the latter case, and the intruder thus learns `xB/=a` without that being released.

This illustrates how we may forget something that the intruder might find out and we may then decide that this is completely benign: the intruder here acts under their real name and just finds out that `xB` who did answer the message was not the intended recipient. Without a basic change of communication model, this information release is unavoidable and the solution is thus to release just this information in this case.

6.3.2 AF0: corrected release

We update the responder transaction to add the information being released in the case that the decryption fails, i.e., the `catch` branch following Line 14.

```
Transaction Responder:
* xB in {a,b}.
receive M.
```

```
4  try DEC:=dcrypt(inv(pk(xB)),M) in
5    try A:=proj1(DEC) in
6      if A=i then
7        new NB,R.
8        send crypt(pk(A),NB,R).
9        * xB=gamma(xB). nil
10     else
11       new NB,R.
12       send crypt(pk(A),NB,R). nil
13   catch new NB. send NB. nil
14 catch
15   try C:=recipient(M) in
16     try DEC:=dcrypt(inv(pk(C)),M) in
17       try A:=proj1(DEC) in
18         if A in {a,b,i} and C in {a,b} then
19           new NB.
20           send NB.
21           * not (C=xB and A=i). nil
22         else new NB. send NB. nil
23       catch new NB. send NB. nil
24     catch new NB. send NB. nil
25   catch new NB. send NB. nil
```

The case where the actual recipient `xB` could not decrypt the given message is complicated. To even talk about who is the intended recipient `C` (if the message is even an encryption) we need to model a function that agents in reality cannot perform, namely extracting the name of the recipient from the message, if it exists (Line 15). That is the purpose of the private function `recipient` which satisfies the equation `recipient(crypt(pk(B),M,R))=B`. These steps do not correspond to actions an agent would do and which we only need in order to determine whether the intruder is allowed to learn something. This is the case if the message is indeed encrypted for some agent `C` and contains a pair of an agent name `A` (the claimed sender) and a nonce. We can even restrict this to an honest `C`, as the intruder can otherwise already decrypt the given message and learn `A` and `C`. If `A` is an agent and `C` is honest, then the intruder learns that at least one of two things must be true: `C` is not the actual recipient `xB` or `A` is honest, for if `C=xB` and `A` dishonest, then the intruder could decipher the answer.

In the case where the first specification had the attack, the intruder knew that `A=i` and `C=a` by construction. The released formula α in the updated specification is thus `not(a=xB and i=i)` or simply `a/=xB`. We can indeed verify with the `noname` tool that there are no more violations of (α, β)-privacy under a bound of three transactions.

6.3.3 AF

We now lift the simplification of AF0 that everybody is willing to talk to everybody. We define a binary relation `talk`, where `talk(x,y)` means that `x` is willing to talk to `y`. The `noname` tool requires to give a fixed interpretation of such a relation. We choose for our experiments the interpretation: `talk: (a,b),(a,i),(b,a)` which specifies that `talk` is true for the listed tuples and false otherwise. The aim of the protocol is that privacy holds for every interpretation of `talk`, but we cannot encode this in the `noname` tool and in fact the definition of (α, β)-privacy requires a fixed interpretation of all relation symbols [47].

The initiator now checks that the given `xA` really wants to talk to the given `xB` in Line 4:

```
1  Transaction Initiator:
2  * xA in {a,b}.
3  * xB in {a,b,i}.
4  if talk(xA,xB) then
5    if xB=i then
```

```
    new NA,R.
    send crypt(pk(xB),pair(xA,NA),R).
    * talk(xA,xB) and xA=gamma(xA) and xB=gamma(xB). nil
  else
    new NA,R.
    send crypt(pk(xB),pair(xA,NA),R).
    * talk(xA,xB) and xB in {a,b}. nil
else * not talk(xA,xB). nil
```

In the positive cases the intruder learns that `talk(xA,xB)` (in case `xB=i` the intruder learns also the concrete agent names, in case `xB/=i` the intruder learns that `xB` is honest), in the negative case, the intruder learns `not talk(xA,xB)`. This case is a bit artificial, because if `xA` does not want to talk to `xB`, they would not even start this transaction in reality, but here we need to non-deterministically choose the agent names first and then abort if `not talk(xA,xB)`, and then the intruder learns that because no message goes out.

This has an interesting consequence: suppose we are in a state where the intruder, as part of α, has learned that agent `a` talks to every other agent, and now observes `not talk(xA,xB)`. From `xA in {a,b}` follows that `xA=b`. This is not a violation of (α, β)-privacy, since this `xA=b` holds in every model of α. In general, it is completely acceptable that the intruder learns the value of privacy variables like `xA`, as long as this is a consequence of α.

The responder role is now getting more complex.

```
1  Transaction Responder:
2  * xB in {a,b}.
3  receive M.
4  try DEC:=dcrypt(inv(pk(xB)),M) in
5    try A:=proj1(DEC) in
6      if A=i then
7        if talk(xB,A) then
8          new NB,R.
9          send crypt(pk(A),NB,R).
10         * xB=gamma(xB) and talk(xB,A). nil
11       else
12         new NB.
13         send NB.
14         * not talk(xB,A). nil
15     else if A in {a,b} then
16       if talk(xB,A) then
17         new NB,R.
18         send crypt(pk(A),NB,R). nil
19       else new NB. send NB. nil
20     else new NB. send NB. nil
21   catch new NB. send NB. nil
22 catch
23   try C:=recipient(M) in
24     try DEC:=dcrypt(inv(pk(C)),M) in
25       try A:=proj1(DEC) in
26         if A in {a,b,i} and C in {a,b} then
27           new NB.
28           send NB.
29           * not (C=xB and A=i and talk(xB,A)). nil
30         else new NB. send NB. nil
31       catch new NB. send NB. nil
32     catch new NB. send NB. nil
33   catch new NB. send NB. nil
```

The main change here is the case split on whether `talk(xB,A)` in Lines 7 and 16: if not, we get into the decoy cases. Observe that only in the dishonest cases the intruder learns whether `talk(xB,A)` or not in Lines 10 and 14. In case the message is a valid message to some different agent `C`, the intruder learns a bit less in Line 29 when compared to the AF0 version: it basically says that now the reason for not being able to decipher the reply could be that `not talk(xB,A)`.

We have verified privacy with the `noname` tool up to four transactions, both for the above interpretation of talk, and when everybody talks to everybody.

Chapter 7

Conclusion

In this thesis, we have developed three main results for (α, β)-privacy in the following topics: automated verification, typing and compositionality. Moreover, we have applied our results to several example protocols.

7.1 Decision procedure

In Chapter 3, we have defined a decision procedure for a bounded number of transitions. We focused first on the symbolic execution of protocols and in particular on the constraint solving using the lazy intruder. This showed how we can represent, in a finite way, the intruder choices of recipes in the absence of destructors. Then we have defined rules for normalizing a symbolic state by performing all relevant intruder experiments that could distinguish possibilities in that state. Finally, we have explained how to support algebraic properties of constructor/destructor theories by following an analysis strategy that decrypts messages as far as possible, so that our representation is sound and complete also in the presence of destructors.

There are two important limitations. First, our restriction of algebraic theories is significant: primitives such as Diffie-Hellman exponentiation or blind signatures for commitment schemes are not supported. Hence, one opportunity for future work is to investigate whether the requirements on primitives can be relaxed in order to obtain a procedure that can used with more protocols. Second, the verification is done for a bounded number of transitions: another area for future work is the design of a procedure for unbounded steps. This would require a significant change in the model of protocol execution and could involve computing an over-approximation of the intruder knowledge.

7.2 Typing

In Chapter 4, we have introduced a typed model, where a protocol specification includes the intended type of every message. We have also extended the semantics of the symbolic execution to support pattern matching. Then we proved that, for protocols that satisfy some requirements, all the destructor applications in transactions can be replaced with pattern matching. After that, we defined the notion of *type-flaw resistance*, which in short ensures that messages of different types cannot be confused. Finally, we have shown that, given a type-flaw resistant protocol, each part of the procedure of Chapter 3 only performs well-typed substitutions. (This required some change in the formalization of the analysis procedure.) We thus obtained a typing result of the form: “if a type-flaw resistant protocol has an attack, then it has a well-typed attack.” This result is more general than related work on typing for privacy properties, because we allow non-deterministic choices to influence the control flow of processes, we fully support if-then-else branches, and we also support stateful protocols with memory cells.

We have discussed the benefits of such typing results. In particular, it is good practice for protocol design to rule out all attacks relying on type-flaws, and working a typed model can significantly help to achieve additional results such as compositionality. Since our method for obtaining the typing result is essentially to prove well-typedness of the decision procedure, we again have a main limitation for the supported algebraic theories. However, our typing result holds without any bounds on the number of transitions.

7.3 Compositionality

In Chapter 5, we studied protocol composition. We started by extending the grammar of protocols and the semantics of the symbolic execution to support new constructs such as procedure calls used for modeling composition. We assumed a typed model and only considered well-typed executions. The typing result of Chapter 4 is one way to ensure that this assumption is sound, but it would require some extension to include the constructs added for composition. We also introduced the notion of *roles*, which are sequences of transactions allowing for interleaving with other transactions between each atomic execution. We illustrated our definitions with an example based on the Needham-Schroeder-Lowe protocol together with procedure calls to a key server. We defined the projection of a composed protocol to an individual component as the restriction to transactions from that component plus the abstract interface of the other components. Then we defined the notion of *composability*, which in short ensures that every projection of the composed protocol is well-defined and can be executed. Finally, we presented our compositionality results, focusing on frames and recipes before obtaining our main result of the form: “given a composable protocol, if every component is secure, then the composed protocol is secure.”

Our main limitation is again the restriction to constructor/destructor theories. Moreover, when verifying one component of the composed system, we still need to consider the abstract interface of other components, so we are not verifying each component completely in isolation. (We discussed in Section 5.5 why this is the case in our approach.) There are very few compositionality results that target privacy properties and support stateful protocols. Our result is more general than related work in several aspects because we are less restrictive in the way protocols are composed: we support shared messages, declassification of secrets, composition through memory cells and procedure calls.

7.4 Tool support

In Chapter 6, we have given a brief introduction to the `noname` tool implementing our decision procedure and presented in details the analysis of two protocols, BAC and Private Authentication. The other protocols of the case studies (presented in Chapter 3) are the simple running example protocol (Example 2.2.1), Basic Hash and OSK. We have focused on unlinkability goals, but some models like for Private Authentication go beyond this. The case studies show the benefits of the declarative approach to privacy goals. In particular, we find it useful to iterate on specifications: if the protocol has an attack, the tool reports it to the user who can then decide whether the information learned by the intruder constitutes a real violation of privacy. In case the intruder deductions are deemed acceptable, one can then update the specification with precise releases of information in the transactions and apply the tool again to check whether there are other attacks.

The case studies with the `noname` tool show that automated analysis of (α, β)-privacy is practical, at least within small bounds on the number of transactions. While this is a prototype tool, there is still much room for improvement, but on the other hand we do obtain a massive state explosion with an increase of bounds. Similar effects are indeed observed with other tools like DeepSec that, like us, focus on expressive power. It is striking that, although (α, β)-privacy is such a different approach, we seem to hit the similar technical limits, so one could argue that they are truly inherent in the problem of verifying privacy.

7.5 Future work

There are several directions to explore for automated verification of (α, β)-privacy:

- Development of tool support: the procedure from Chapter 3 has been implemented in the prototype tool `noname`, but it can be improved for optimization and presentation of results to the user. Moreover, the verification of type-flaw resistance and composability requirements is amenable to automation, since they are syntactic requirements. Even though the set of messages patterns of a protocol is infinite in general, the condition for type-flaw resistance can be checked automatically using a finite representation [51]. While the decision procedure from Chapter 3 does not consider protocol composition, it seems that it could be extended to support the constructs introduced in Chapter 5 without any theoretical challenge.

- Modeling and analysis: studying more protocols, both older protocols with known attacks and new protocols that have not been verified, can provide insights to extend or refine the (α, β)-privacy specification language, and it would also be beneficial to benchmark and showcase how (α, β)-privacy works. For instance, it would be useful to identify and model different composed protocols in order to understand how applicable our compositionality result from Chapter 5 is in practice.

- Relaxation of requirements, in particular for algebraic theories: the handling of destructors may be generalized to support more primitives; Diffie-Hellman exponentiation would be particularly relevant. One example is the verification of the EMV protocol for contactless payment card transactions [38]. Even though our tool does not support the blinded Diffie-Hellman key exchange used in this protocol, we can find (using manual exploration) a linkability attack in case of an active attacker [56], because this particular attack does not require the algebraic property of blinding. However, we currently cannot prove that the extended version presented in [56] is unlinkable.

- Support for probabilistic (α, β)-privacy: [46] defines semantics where privacy variables can be sampled with probabilities, which we have not considered in this thesis.

- Design of procedure for unbounded steps: can we prove that a protocol satisfies (α, β)-privacy without putting a bound on the number of transitions?

Bibliography

[1] M. Abadi and C. Fournet. “Private Authentication”. In: *Theor. Comput. Sci.* 322.3 (2004), pp. 427–476. DOI: 10.1016/j.tcs.2003.12.023.

[2] M. Abadi and R. Needham. “Prudent engineering practice for cryptographic protocols”. In: *IEEE Trans. Softw. Eng.* 22.1 (1996), pp. 6–15. DOI: 10.1109/32.481513.

[3] O. Almousa, S. Mödersheim, P. Modesti, and L. Viganò. “Typing and Compositionality for Security Protocols: A Generalization to the Geometric Fragment”. In: *ESORICS 2015*. Vol. 9327. LNCS. Springer, 2015, pp. 209–229. DOI: 10.1007/978-3-319-24177-7_11.

[4] D. Aparicio-Sánchez, S. Escobar, R. Gutiérrez, and J. Sapiña. “An Optimizing Protocol Transformation for Constructor Finite Variant Theories in Maude-NPA”. In: *ESORICS 2020*. Vol. 12309. LNCS. Springer, 2020, pp. 230–250. DOI: 10.1007/978-3-030-59013-0_12.

[5] M. Arapinis, V. Cheval, and S. Delaune. “Composing Security Protocols: From Confidentiality to Privacy”. In: *POST 2015*. Vol. 9036. LNCS. Springer, 2015, pp. 324–343. DOI: 10.1007/978-3-662-46666-7_17.

[6] M. Arapinis, T. Chothia, E. Ritter, and M. Ryan. “Analysing Unlinkability and Anonymity Using the Applied Pi Calculus”. In: *CSF 2010*. IEEE, 2010, pp. 107–121. DOI: 10.1109/CSF.2010.15.

[7] M. Arapinis and M. Duflot. “Bounding messages for free in security protocols – extension to various security properties”. In: *Inf Comput* 239 (2014), pp. 182–215. DOI: 10.1016/j.ic.2014.09.003.

[8] A. Armando, R. Carbone, and L. Compagna. “SATMC: A SAT-Based Model Checker for Security-Critical Systems”. In: *TACAS 2014*. Vol. 8413. LNCS. Springer, 2014, pp. 31–45. DOI: 10.1007/978-3-642-54862-8_3.

[9] A. Armando et al. “The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures”. In: *TACAS 2012*. Vol. 7214. LNCS. Springer, 2012, pp. 267–282. DOI: 10.1007/978-3-642-28756-5_19.

[10] A. Armando et al. “The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications”. In: *CAV 2005*. Vol. 3576. LNCS. Springer, 2005, pp. 281–285. DOI: 10.1007/11513988_27.

[11] D. Baelde, S. Delaune, and S. Moreau. “A Method for Proving Unlinkability of Stateful Protocols”. In: *CSF 2020*. IEEE, 2020, pp. 169–183. DOI: 10.1109/CSF49147.2020.00020.

[12] H. Barbosa et al. “cvc5: A Versatile and Industrial-Strength SMT Solver”. In: *TACAS 2022*. Vol. 13243. LNCS. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

[13] D. Basin, S. Mödersheim, and L. Viganò. “OFMC: A Symbolic Model Checker for Security Protocols”. In: *Int. J. Inf. Secur.* 4.3 (2005), pp. 181–208. DOI: 10.1007/s10207-004-0055-7.

[14] M. Baudet. “Deciding Security of Protocols Against Off-line Guessing Attacks”. In: *CCS 2005.* ACM, 2005, pp. 16–25. DOI: 10.1145/1102120.1102125.

[15] D. Bernhard et al. “Adapting Helios for Provable Ballot Privacy”. In: *ESORICS 2011.* Vol. 6879. LNCS. Springer, 2011, pp. 335–354. DOI: 10.1007/978-3-642-23822-2_19.

[16] B. Blanchet. “An Efficient Cryptographic Protocol Verifier Based on Prolog Rules”. In: *CSFW 2001.* IEEE, 2001, pp. 82–96. DOI: 10.1109/CSFW.2001.930138.

[17] B. Blanchet, M. Abadi, and C. Fournet. “Automated Verification of Selected Equivalences for Security Protocols”. In: *J Log Algebr Program* 75.1 (2008), pp. 3–51. DOI: 10.1016/j.jlap.2007.06.002.

[18] B. Blanchet and A. Podelski. “Verification of cryptographic protocols: tagging enforces termination”. In: *Theor. Comput. Sci.* 333.1 (2005), pp. 67–90. DOI: 10.1016/j.tcs.2004.10.018.

[19] A. Boutet et al. *Contact Tracing by Giant Data Collectors: Opening Pandora’s Box of Threats to Privacy, Sovereignty and National Security.* Tech. rep. EPFL, Switzerland; Inria, France; JMU Würzburg, Germany; University of Salerno, Italy; base23, Switzerland; TU Darmstadt, Germany, 2020. URL: https://hal.inria.fr/hal-03116024.

[20] M. Brusó, K. Chatzikokolakis, and J. den Hartog. “Formal Verification of Privacy for RFID Systems”. In: *CSF 2010.* IEEE, 2010, pp. 75–88. DOI: 10.1109/CSF.2010.13.

[21] R. Chadha, V. Cheval, Ş. Ciobâcă, and S. Kremer. “Automated Verification of Equivalence Properties of Cryptographic Protocols”. In: *ACM Trans. Comput. Logic* 17.4 (2016), pp. 1–32. DOI: 10.1145/2926715.

[22] V. Cheval. “APTE: An Algorithm for Proving Trace Equivalence”. In: *TACAS 2014.* Vol. 8413. LNCS. Springer, 2014, pp. 587–592. DOI: 10.1007/978-3-642-54862-8_50.

[23] V. Cheval, H. Comon-Lundh, and S. Delaune. “A Procedure for Deciding Symbolic Equivalence Between Sets of Constraint Systems”. In: *Inf Comput* 255 (2017), pp. 94–125. DOI: 10.1016/j.ic.2017.05.004.

[24] V. Cheval, V. Cortier, and B. Warinschi. “Secure Composition of PKIs with Public Key Protocols”. In: *CSF 2017.* IEEE, 2017, pp. 144–158. DOI: 10.1109/CSF.2017.28.

[25] V. Cheval, S. Kremer, and I. Rakotonirina. “DEEPSEC: Deciding Equivalence Properties in Security Protocols Theory and Practice”. In: *S&P 2018.* IEEE, 2018, pp. 529–546. DOI: 10.1109/SP.2018.00033.

[26] V. Cheval, S. Kremer, and I. Rakotonirina. “Exploiting Symmetries When Proving Equivalence Properties for Security Protocols”. In: *CCS 2019.* ACM, 2019, pp. 905–922. DOI: 10.1145/3319535.3354260.

[27] V. Cheval, S. Kremer, and I. Rakotonirina. “The Hitchhiker’s Guide to Decidability and Complexity of Equivalence Properties in Security Protocols”. In: *Logic, Language, and Security.* Vol. 12300. LNCS. Springer, 2020, pp. 127–145. DOI: 10.1007/978-3-030-62077-6_10.

[28] V. Cheval and I. Rakotonirina. “Indistinguishability Beyond Diff-Equivalence in ProVerif”. In: *CSF 2023.* IEEE, 2023, pp. 184–199. DOI: 10.1109/CSF57540.2023.00036.

[29] C. Chevalier, S. Delaune, S. Kremer, and M. Ryan. “Composition of password-based protocols”. In: *Form Methods Syst Des* 43.3 (2013), pp. 369–413. DOI: 10.1007/s10703-013-0184-6.

[30] T. Chothia and V. Smirnov. “A Traceability Attack against e-Passports”. In: *FC 2010*. Vol. 6052. LNCS. Springer, 2010, pp. 20–34. DOI: 10.1007/978-3-642-14577-3_5.

[31] R. Chrétien, V. Cortier, A. Dallon, and S. Delaune. “Typing Messages for Free in Security Protocols”. In: *ACM Trans. Comput. Logic* 21.1 (2020), pp. 1–52. DOI: 10.1145/3343507.

[32] R. Chrétien, V. Cortier, and S. Delaune. “Typing Messages for Free in Security Protocols: The Case of Equivalence Properties”. In: *CONCUR 2014*. Vol. 8704. Springer, 2014, pp. 372–386. DOI: 10.1007/978-3-662-44584-6_26.

[33] S. Ciobâca and V. Cortier. “Protocol Composition for Arbitrary Primitives”. In: *CSF 2010*. IEEE, 2010, pp. 322–336. DOI: 10.1109/CSF.2010.29.

[34] V. Cortier and S. Delaune. “Safely composing security protocols”. In: *Form Methods Syst Des* 34.1 (2009), pp. 1–36. DOI: 10.1007/s10703-008-0059-4.

[35] V. Cortier et al. “Machine-Checked Proofs of Privacy for Electronic Voting Protocols”. In: *S&P 2017*. IEEE, 2017, pp. 993–1008. DOI: 10.1109/SP.2017.28.

[36] S. Delaune and L. Hirschi. “A Survey of Symbolic Methods for Establishing Equivalence-based Properties in Cryptographic Protocols”. In: *J. Log. Algebraic Methods Program.* 87 (2017), pp. 127–144. DOI: 10.1016/j.jlamp.2016.10.005.

[37] S. Doghmi, J. Guttman, and F. J. Thayer. “Searching for Shapes in Cryptographic Protocols”. In: *TACAS 2007*. Vol. 4424. LNCS. Springer, 2007, pp. 523–537. DOI: 10.1007/978-3-540-71209-1_41.

[38] *EMV Contactless Specifications for Payment Systems*. Books A–E. https://www.emvco.com/specifications/.

[39] L. Fernet. *noname: Formal Verification of (Alpha, Beta)-Privacy in Security Protocols*. Version 0.3. Nov. 2024. DOI: 10.5281/zenodo.14198336. URL: https://doi.org/10.5281/zenodo.14198336.

[40] L. Fernet and S. Mödersheim. “Deciding a Fragment of (alpha, beta)-Privacy”. In: *STM 2021*. Vol. 13075. LNCS. Springer, 2021, pp. 122–142. DOI: 10.1007/978-3-030-91859-0_7.

[41] L. Fernet and S. Mödersheim. “Private Authentication with Alpha-Beta-Privacy”. In: *OID 2023*. LNI. GI, 2023. DOI: 10.18420/OID2023_05. URL: https://people.compute.dtu.dk/samo/alphabeta.

[42] L. Fernet, S. Mödersheim, and L. Viganò. *A Compositionality Result for Alpha-Beta Privacy*. Tech. rep. DTU, Denmark; King’s, United Kingdom, 2024. URL: https://people.compute.dtu.dk/samo.

[43] L. Fernet, S. Mödersheim, and L. Viganò. “A decision procedure and typing result for alpha-beta privacy”. In: *J. Comput. Secur.* (2024). Submitted for review.

[44] L. Fernet, S. Mödersheim, and L. Viganò. “A Decision Procedure for Alpha-Beta Privacy for a Bounded Number of Transitions”. In: *CSF 2024*. Extended version at https://people.compute.dtu.dk/samo/alphabeta. IEEE, 2024, pp. 159–174. DOI: 10.1109/CSF61375.2024.00011.

[45] I. Filimonov, R. Horne, S. Mauw, and Z. Smith. “Breaking Unlinkability of the ICAO 9303 Standard for e-Passports Using Bisimilarity”. In: *ESORICS 2019*. Vol. 11735. LNCS. Springer, 2019, pp. 577–594. DOI: 10.1007/978-3-030-29959-0_28.

[46] S. Gondron. “Vertical Composition of Distributed Systems”. PhD thesis. Technical University of Denmark, 2021.

[47] S. Gondron, S. Mödersheim, and L. Viganò. “Privacy as Reachability”. In: *CSF 2022*. IEEE, 2022, pp. 130–146. DOI: 10.1109/CSF54842.2022.9919668.

[48] J. Guttman. “Cryptographic Protocol Composition via the Authentication Tests”. In: *FOSSACS 2009*. Vol. 5504. LNCS. Springer, 2009, pp. 303–317. DOI: 10.1007/978-3-642-00596-1_22.

[49] J. Guttman and F. J. Thayer. “Protocol independence through disjoint encryption”. In: *CSFW 2000*. IEEE, 2000, pp. 24–34. DOI: 10.1109/CSFW.2000.856923.

[50] J. Heather, G. Lowe, and S. Schneider. “How to prevent type flaw attacks on security protocols”. In: *J. Comput. Secur.* 11.2 (2003), pp. 217–244. DOI: 10.3233/JCS-2003-11204.

[51] A. Hess and S. Mödersheim. “A Typing Result for Stateful Protocols”. In: *CSF 2018*. IEEE, 2018, pp. 374–388. DOI: 10.1109/CSF.2018.00034.

[52] A. Hess and S. Mödersheim. “Formalizing and Proving a Typing Result for Security Protocols in Isabelle/HOL”. In: *CSF 2017*. IEEE, 2017, pp. 451–463. DOI: 10.1109/CSF.2017.27.

[53] A. Hess, S. Mödersheim, and A. Brucker. “Stateful Protocol Composition in Isabelle/HOL”. In: *ACM Trans. Priv. Secur.* 26.3 (2023), pp. 1–36. DOI: 10.1145/3577020.

[54] A. Hess, S. Mödersheim, A. Brucker, and A. Schlichtkrull. “Performing Security Proofs of Stateful Protocols”. In: *CSF 2021*. IEEE, 2021, pp. 1–16. DOI: 10.1109/CSF51468.2021.00006.

[55] T. Hinrichs and M. Genesereth. *Herbrand Logic*. Tech. rep. LG-2006-02. Stanford University, USA, 2006. URL: http://logic.stanford.edu/reports/LG-2006-02.pdf.

[56] R. Horne, S. Mauw, and S. Yurkov. “Unlinkability of an Improved Key Agreement Protocol for EMV 2nd Gen Payments”. In: *CSF 2022*. IEEE, 2022, pp. 364–379. DOI: 10.1109/CSF54842.2022.9919666.

[57] ICAO. *Machine Readable Travel Documents*. Doc Series, Doc 9303. https://www.icao.int/publications/pages/publication.aspx?docnum=9303.

[58] V. Iovino, S. Vaudenay, and M. Vuagnoux. *On the Effectiveness of Time Travel to Inject COVID-19 Alerts*. Cryptology ePrint Archive, Paper 2020/1393. 2020. DOI: 10.1007/978-3-030-75539-3_18.

[59] G. Lowe. “An attack on the Needham-Schroeder public-key authentication protocol”. In: *Inf Process Lett* 56.3 (1995), pp. 131–133. DOI: 10.1016/0020-0190(95)00144-2.

[60] S. Meier, B. Schmidt, C. Cremers, and D. Basin. “The TAMARIN Prover for the Symbolic Analysis of Security Protocols”. In: *CAV 2013*. Vol. 8044. LNCS. Springer, 2013, pp. 696–701. DOI: 10.1007/978-3-642-39799-8_48.

[61] J. Millen and V. Shmatikov. “Constraint Solving for Bounded-Process Cryptographic Protocol Analysis”. In: *CCS 2001*. ACM, 2001, pp. 166–175. DOI: 10.1145/502006.502007.

[62] S. Mödersheim and G. Katsoris. “A Sound Abstraction of the Parsing Problem”. In: *CSF 2014*. IEEE, 2014, pp. 259–273. DOI: 10.1109/CSF.2014.26.

[63] S. Mödersheim and L. Viganò. “Alpha-Beta Privacy”. In: *ACM Trans. Priv. Secur.* 22.1 (2019), pp. 1–35. DOI: 10.1145/3289255.

[64] S. Mödersheim and L. Viganò. “The Open-Source Fixed-Point Model Checker for Symbolic Analysis of Security Protocols”. In: *FOSAD 2007/2008/2009 Tutorial Lectures*. Vol. 5705. LNCS. Springer, 2009, pp. 166–194. DOI: 10.1007/978-3-642-03829-7_6.

[65] R. Needham and M. Schroeder. “Using encryption for authentication in large networks of computers”. In: *Commun. ACM* 21.12 (1978), pp. 993–999. DOI: 10.1145/359657.359659.

[66] M. Ohkubo, K. Suzuki, and S. Kinoshita. “Cryptographic Approach to ‘Privacy-Friendly’ Tags”. In: *RFID Privacy Workshop 2003*. 2003.

[67] L. Paulson. “The inductive approach to verifying cryptographic protocols”. In: *J. Comput. Secur.* 6.1 (1998), pp. 85–128. DOI: 10.3233/JCS-1998-61-205.

[68] E. Rescorla and T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. 2008. DOI: 10.17487/RFC5246. URL: https://www.rfc-editor.org/info/rfc5246.

[69] M. Rusinowitch and M. Turuani. “Protocol Insecurity with a Finite Number of Sessions and Composed Keys is NP-Complete”. In: *Theor. Comput. Sci.* 299.1 (2003), pp. 451–475. DOI: 10.1016/S0304-3975(02)00490-5.

[70] A. Tiu and J. Dawson. “Automating Open Bisimulation Checking for the Spi Calculus”. In: *CSF 2010*. IEEE, 2010, pp. 307–321. DOI: 10.1109/CSF.2010.28.

[71] A. Tiu, N. Nguyen, and R. Horne. “SPEC: An Equivalence Checker for Security Protocols”. In: *APLAS 2016*. Vol. 10017. LNCS. Springer, 2016, pp. 87–95. DOI: 10.1007/978-3-319-47958-3_5.

[72] M. Turuani. “The CL-Atse Protocol Analyser”. In: *RTA 2006*. Vol. 4098. LNCS. Springer, 2006, pp. 277–286. DOI: 10.1007/11805618_21.

[73] S. Vaudenay and M. Vuagnoux. *Analysis of SwissCovid*. Tech. rep. EPFL, Switzerland; base23, Switzerland, 2020. URL: https://lasec.epfl.ch/people/vaudenay/swisscovid/swisscovid-ana.pdf.

[74] S. A. Weis, S. E. Sarma, R. L. Rivest, and D. W. Engels. “Security and Privacy Aspects of Low-Cost Radio Frequency Identification Systems”. In: *Security in Pervasive Computing*. Vol. 2802. LNCS. Springer, 2004, pp. 201–212. DOI: 10.1007/978-3-540-39881-3_18.

Appendix A

Proofs

A.1 Decidability of fragment of Herbrand logic

We support the fragment such that:

- The alphabet Σ_0 is finite (in particular, there are finitely many constants).
- The equivalence class $[t]_E$ of every Σ_0-term t is computable (and thus finite).
- Every variable x (both bound and unbound) must range over a fixed domain of constants, $dom(x) \subseteq \Sigma_0^0$.

Before giving a decision procedure, we first need some definitions. Given the Herbrand universe U induced by Σ_0 and given α, we define the *relevant part of U for α* as follows:

$$U_0^\alpha = \{[\sigma(t_i)]_E \mid R(t_1,\dots,t_n) \text{ occurs in } \alpha \text{ and for all } x \in fv(t_1,\dots,t_n), \sigma(x) \in dom(x)\}$$

We say that θ is an *interpretation representation* (w.r.t. α) iff θ maps every $x \in fv(\alpha)$ to some element of $dom(x)$ and every n-ary relation symbol R to a subset of $(U_0^\alpha)^n$. We say that θ *represents* interpretation $\mathcal{I}$ iff $\theta(x) = \mathcal{I}(x)$ for every $x \in fv(\alpha)$ and $\vec{t} \in \theta(R)$ iff $\vec{t} \in \mathcal{I}(R)$ for every n-ary relation symbol R and $\vec{t} \in (U_0^\alpha)^n$.

We now describe an algorithm that, given α, returns the set of all interpretation representations that represent a model of α (which implies a decision procedure for the model relation). We first compute all interpretation representations for α. This is finite since there are only finitely many variables and they have finite domains; moreover, U_0^α is finite, since finitely many relations $R(t_1,\dots,t_n)$ are used, their variables can range over finitely many values, and the equivalence classes of every term is finite. Thus there are finitely many possible interpretations of every R over U_0^α. For a given interpretation representation θ, we can check the model relation with α as follows:

$$\begin{array}{rll}
\theta \models s \doteq t & \text{iff} & [\theta(s)]_E = [\theta(t)]_E \\
\theta \models R(t_1,\dots,t_n) & \text{iff} & ([\theta(t_1)]_E,\dots,[\theta(t_n)]_E) \in \theta(R) \\
\theta \models \phi \wedge \psi & \text{iff} & \theta \models \phi \text{ and } \theta \models \psi \\
\theta \models \neg\phi & \text{iff} & \text{not } \theta \models \phi \\
\theta \models \exists x.\ \phi & \text{iff} & \text{there exists } c \in dom(x) \text{ such that} \\
 & & \theta[x \mapsto c] \models \phi
\end{array}$$

A.2 Correctness of representation with symbolic states

[47] defines rules for the symbolic execution of transactions and explains how to define (α, β)-privacy as a reachability property, in a transition system with ground states. We follow a similar approach but we have two additional layers of symbolic representation, namely: the merging of ground states that differ only in the truth formula, and the intruder variables for the lazy intruder. We say that the rules from [47], given in Table 2.1, are working on *the ground level*, while the rules from this thesis given in Table 3.2, are working on *the symbolic level*.

Definition A.2.1 (Starting symbolic state). *Let $\mathcal{S} = (\_, \_, \mathcal{P}, \_)$ be a finished symbolic state, P be a transaction and σ be a substitution such that σ substitutes the variables $X_1, \ldots, X_k$ (from a $\nu X_1, \ldots, X_k.P_r$ specification) with fresh and distinct constants from $\Sigma \setminus \Sigma_0$ that do not occur elsewhere in $\mathcal{S}$ or P, and such that σ substitutes all other variables with fresh variables that do not occur elsewhere. The* starting symbolic state for P w.r.t. $\mathcal{S}$ and σ *is*

$$\mathcal{S}[\mathcal{P} \leftarrow \{(\sigma(P), \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta) \mid (0, \phi, \mathcal{A}, \mathcal{X}, \alpha, \delta) \in \mathcal{P}\}] .$$

We write $start(P, \mathcal{S})$ and omit σ to denote a starting symbolic state, because the point is that all variables are substituted with fresh and distinct constants or fresh variables, so the actual values are not relevant.

The definition is similar for ground states, thus we also write $start(P, S)$ for a state S.

On the ground level, there is always a unique possibility that is underlined. The underlined possibility corresponds to the concrete execution observed by the intruder. On the symbolic level, there is no underlined possibility because we actually represent together all different instantiations for the underlined possibility. Our rules work so that each possibility could be the underlined one, and which one is underlined is defined in the semantics of the symbolic states (Definition 3.2.3).

We now argue that our rules on the symbolic level are correct w.r.t. the ground level, i.e., the symbolic states generated by our rules represent the ground states generated by the rules on the ground level. In the following, we describe the differences in the rules from Tables 2.1 and 3.2. For simplicity, we ignore the **Eliminate** rules because the redundant possibilities do not change the semantics of symbolic states, and the elimination can be seen as a kind of optimization.

Non-deterministic choice

All possibilities have this choice step at the same time. On the ground level, the variable x is chosen non-deterministically from the values in the domain D. There is a transition for every value $c \in D$ that the variable can take. On the symbolic level, we have a single transition and we only update α_0 or β_0 with the formula $x \in D$. The semantics of the symbolic states includes all models of $\alpha_0 \wedge \beta_0 \wedge \gamma_0 \wedge \phi_i$, so we represent all models $\gamma \models x \doteq c$ for every $c \in D$ (and such that γ is consistent with the rest of the formulas).

Receive

By construction, every possibility starts with a receive step (with the same variable). On the ground level, there is a transition for every recipe r that the intruder can generate, and the variable standing for the message received is directly substituted with what the recipe produces in each structural frame. On the symbolic level, we have the lazy intruder representation. Thus, we have a single transition, where the recipe and the corresponding message are left as recipe and intruder variables, respectively. Note that there are in general infinitely many transitions at this point on the ground level, while on the symbolic level there is just one transition. The intruder variables are not instantiated, unless we need to consider different values in order to resolve the conditions. The semantics of the symbolic states includes all ground choices of recipes, so all instantiations for the recipe variable (which determine the instantiations of the intruder variable in each structural frame).

Cell read

On the ground level, the memory δ contains the sequence $\mathsf{cell}[s_1] := t_1.\cdots.\mathsf{cell}[s_k] := t_k$ for the given cell, and the initial value is given with ground context $C[\cdot]$. The process is updated with conditional statements and ensures that the most recent update is read. On the symbolic level, the rule is the same.

Cell write

On the ground level, a memory update is prepended to the sequence δ. On the symbolic level, the rule is the same.

Destructor application

On the ground level, try-catch is syntactic sugar around if-then-else, so there is only the rule for conditional statements. On the symbolic level, we handle in a specific way the destructor applications in try-catch using our assumptions that this is the only place in a specification where destructors are allowed, and maintaining this as an invariant. In case the constraints can be solve, we have one symbolic state for every choice of recipes computed by the lazy intruder in the possibility under consideration, and one more symbolic state for not taking any of these choices. We show the correctness of our representation of choices of recipes in the proof of Theorem 3.1.1, and the point is that we here partition the concrete instantiations of recipes variables. After applying a choice of recipes, the destructor, in the possibility considered, succeeds under a particular substitution of privacy variables, and we split the possibility in two to represent whether this substitution holds. In case the constraints cannot be solved, we have one symbolic state where the process simply goes to the catch branch.

Conditional statement

On the ground level, we split a possibility into two, one for the case that the condition is true and we go into the then branch, and one for the else branch. By construction, if the underlined possibility is split then there is only one branch that is consistent with the current truth γ and it is underlined accordingly. On the symbolic level, there are two base cases: when the condition is a relation $R(t_1, \dots, t_n)$ and when the condition is an equality $s \doteq t$. For an arbitrary formula, we can eliminate the negation by swapping the branches and eliminate the conjunction by nesting conditional statements; this does not change the semantics.

- When the condition is a relation: The possibility is split into two possibilities, just like on the ground level.
- When the condition is an equality $s \doteq t$: We first compute the unifier $\sigma = mgu(s \doteq t)$ and then the transitions are just like for destructor application, i.e., we partition the

ground choices of recipes and split on whether the substitution of privacy variables holds.

Release

On the ground level, the formula released by the underlined possibility is added to the payload, and formulas released by other possibilities are ignored. On the symbolic level, we have that each possibility could be the underlined one. Therefore, we do not update the common payload but rather the partial payload attached to the given possibility. The formula released should be consistent with all models of $\alpha_0 \wedge \beta_0 \wedge \gamma_0 \wedge \phi$, i.e., the truths that this possibility symbolically represents. If that is not the case, it counts as a privacy violation: it would mean that after the transaction for some ground state we have a pair (α, β) where the payload is inconsistent and thus (α, β)-privacy trivially holds. For the procedure we can either assume that all releases are consistent or, as an option, we can detect inconsistent releases and give a warning to the user, because this indicates that something is wrong in the model.

In the semantics of the symbolic states, we consider all payloads $\alpha_0 \wedge [\alpha_i]^\gamma$ that the intruder can observe so our rules cover the releases with $\star$. [47] actually allows releases with $\mathsf{mode} = \diamond$, with the meaning that the formula is added to the truth γ instead of the payload α. This kind of release can be useful for modeling, e.g., voting protocols with interpreted functions. We do *not* support releases with $\mathsf{mode} = \diamond$ in this thesis, as the procedure does not support interpreted functions and many protocol models do not require this construct. However, it might be possible to support such releases, for instance we could add a component for "partial truth" γ_i similarly to the partial payloads, that would be used in the semantics when defining the models.

Send

On the ground level, if the intruder observes that a message is sent, then they can rule out all possibilities where the remaining process is 0. Note that this rule can only be applied if all possibilities start either with $\mathsf{snd}(\cdot)$ or 0; otherwise another evaluation rule must be applied. For all others, each $struct_i$ is augmented by the message sent in the respective possibility. On the symbolic level, the rule is similar.

Terminate

On the ground level, the intruder observes that the execution has terminated because no messages are sent, so they can rule out all possibilities that are not terminated. On the symbolic level, the rule is actually merged with the **Send** rule. Note that on the ground level, eventually the underlined possibility either sends or terminates and the corresponding rule is applied. Other steps are done in different evaluation rules that must be applied before, so the processes that do not send are actually terminating (nil process). On the symbolic level both the send and terminate cases are in general applicable at the same time, which is why the symbolic **Send** rule yields two symbolic states.

Correctness

In Table 3.2, in general several rules may be applicable at the same time. However, the order does not matter and the procedure arbitrarily fixes an order for the rules of the symbolic execution.

- The rules **Cell read**, **Cell write**, **Destructor** (2), **Conditional** and **Release** only change one possibility without affecting the others, so the ordering is irrelevant.

- **Eliminate** and (**Destructor** (1) or **Conditional**): If a possibility can be eliminated and has a process starting with a destructor application or conditional statement, then the possibilities that replace it in the states yielded by **Destructor** (1) or **Conditional** can also be eliminated.

- **Eliminate** and **Send**: If a possibility can be eliminated, then the same possibility can be eliminated in one of the states yielded by **Send**, depending on whether the process in that possibility was sending or already terminated.

- **Eliminate** and any other rule: If a possibility can be eliminated before another rule is applied, then the same possibility can be eliminated after the other rule has been applied.

- The rules **Choice**, **Receive** and **Send** cannot be applicable at the same time as other rules, except with **Eliminate** which we have treated above.

In Definition 3.3.4, in general there may be several pairs of recipes to choose from for the next intruder experiment, and given a pair, there may be several instances of the **Recipe split** rule that are applicable at the same time. However, again it does not matter in which order the experiments are done because here we are only partitioning the set of ground states represented, i.e., the set of symbolic states yielded by one experiment are together semantically equivalent to the symbolic state before the experiment (Theorem 3.3.1). Thus the procedure arbitrarily fixes an order for the experiments.

In Definition 3.4.4, in general there may be several destructor oracles applicable at the same time. For example, we may have a symbolic state $\mathcal{S}$ such that, in two FLICs, the same label l maps to $\star$-marked terms with different constructors, say, $-l \mapsto c(t_1, \dots, t_n) \in \mathcal{A}$ and $-l \mapsto c'(t'_1, \dots, t'_m) \in \mathcal{A}'$ where $c \neq c'$. For instance, the next message to analyze is either an encryption or a pair. Then it does not matter which destructor oracle is applied first, i.e., the one for the rewrite rule of c or the one for c', because the symbolic states yielded by application of one destructor oracle are semantically equivalent to the state before applying the oracle. Note that in this example, if the oracle for c is applied first and the term could not be analyzed, then the marking of the $c'(\dots)$ term is not changed, and thus the oracle for c' can be applied afterwards. If the $c(\dots)$ term was successfully analyzed, then we only kept possibilities where analysis succeeded as well and removed the possibility with the $c'(\dots)$ term because the possibilities are distinguishable (there is no point in applying this oracle knowing in advance that it fails). Thus, in case several oracles are applicable at the same time, the procedure arbitrarily fixes an order for the application of these destructor oracles.

Our evaluation rules correspond to internal transitions for the symbolic execution of transactions, which is distinct from the overall transition system where the transactions are atomic. We have defined in Chapter 2 the relation $\longrightarrow$ between finished ground states that models the execution of some transaction, using the relation $\rightarrow$ of evaluation rules until all processes in $\mathcal{P}$ have terminated. Similarly, we have defined in Section 4.2.2 the relation $\Longrightarrow$ for symbolic states. We write $\longrightarrow_P$ and $\Longrightarrow_P$ to explicitly mention the transaction P that is executed. Moreover, we define $S_{\longrightarrow_P}$ to be the set of ground states that are reached by executing P, and similarly $\mathcal{S}_{\Longrightarrow_P}$ to be the set of symbolic states after executing P:

$$S_{\longrightarrow_P} = \{S' \mid \mathit{start}(P, S) \longrightarrow_P S'\}\ , \qquad \{\mathit{start}(P, \mathcal{S})\} \Longrightarrow_P \mathcal{S}_{\Longrightarrow_P}\ .$$

The transitions on the symbolic level are correct w.r.t. the transitions that can happen on the ground level.

Proposition A.2.1 (Reachability correctness)**.** *Let $\mathcal{S}$ be a finished symbolic state and P be a transaction. Let $[\![\mathcal{S}]\!]_{\longrightarrow_P}$ be the ground states after transitions between ground states and $[\![\mathcal{S}_{\Longrightarrow_P}]\!]$ be the ground states after transitions between symbolic states:*

$$[\![\mathcal{S}]\!]_{\longrightarrow_P} = \{S' \mid S \in [\![\mathcal{S}]\!] \text{ and } S' \in S_{\longrightarrow_P}\}\ , \qquad [\![\mathcal{S}_{\Longrightarrow_P}]\!] = \{S \mid \mathcal{S}' \in \mathcal{S}_{\Longrightarrow_P} \text{ and } S \in [\![\mathcal{S}']\!]\}\ .$$

Then we have $[\![\mathcal{S}]\!]_{\longrightarrow_P} = [\![\mathcal{S}_{\Longrightarrow_P}]\!]$.

A.3 Decision procedure

A.3.1 Lazy intruder correctness

We start with a lemma relating the recipes for FLICs before and after one lazy intruder application.

Lemma A.3.1. *Let $\mathcal{A}$ be a FLIC, ρ be a choice of recipes such that $dom(\rho) \cap rvars(\mathcal{A}) = \emptyset$ and let σ be a substitution such that $dom(\sigma) \cap vars(\mathcal{A}) = \emptyset$. Let $(\rho', \mathcal{A}', \sigma')$ such that $(\rho, \mathcal{A}, \sigma) \rightsquigarrow (\rho', \mathcal{A}', \sigma')$. Then for every recipe r, we have $\sigma'(\mathcal{A}(r)) = \sigma'(\mathcal{A}'(\rho'(r)))$.*

Proof. For a recipe variable that is changed by the rule application:

- **Unification**: $\mathcal{A} = \mathcal{A}_1.{-l} \mapsto s.\mathcal{A}_2.{+R} \mapsto t.\mathcal{A}_3$, $\mathcal{A}' = \sigma'(\mathcal{A}_1.{-l} \mapsto s.\mathcal{A}_2.\mathcal{A}_3)$, $\rho'(R) = l$ and $\sigma' \models_\Sigma s \doteq t$ so $\sigma'(\mathcal{A}'(\rho'(R))) = \sigma'(s) = \sigma'(t) = \sigma'(\mathcal{A}(R))$.
- **Composition**: $\mathcal{A} = \mathcal{A}_1.{+R} \mapsto f(t_1, \dots, t_n).\mathcal{A}_2$, $\mathcal{A}' = \mathcal{A}_1.{+R_1} \mapsto t_1.\cdots.{+R_n} \mapsto t_n.\mathcal{A}_2$ and $\rho'(R) = f(R_1, \dots, R_n)$ so $\mathcal{A}'(\rho'(R)) = f(t_1, \dots, t_m) = t = \mathcal{A}(R)$.
- **Guessing**: $\mathcal{A} = \mathcal{A}_1.{+R} \mapsto x.\mathcal{A}_2$, $\mathcal{A}' = \sigma'(\mathcal{A}_1.\mathcal{A}_2)$, $\rho'(R) = c$ and $\sigma' \models_\Sigma x \doteq c$ so $\sigma'(\mathcal{A}'(\rho'(R))) = \sigma'(c) = \sigma'(x) = \sigma'(\mathcal{A}(R))$.
- **Repetition**: $\mathcal{A} = \mathcal{A}_1.{+R_1} \mapsto X.\mathcal{A}_2.{+R_2} \mapsto X.\mathcal{A}_3$, $\mathcal{A}' = \mathcal{A}_1.{+R_1} \mapsto X.\mathcal{A}_2.\mathcal{A}_3$ and $\rho'(R_2) = R_1$ so $\mathcal{A}'(\rho'(R_2)) = X = \mathcal{A}(R_2)$.

For a recipe variable R that is not changed by the rule application, we also have $\sigma'(\mathcal{A}(R)) = \sigma'(\mathcal{A}'(\rho'(R)))$ and similarly for labels. For a composed recipe, this holds by induction on the structure of the recipe. □

The next four lemmas prove the soundness, completeness, correctness and termination of the lazy intruder rules defined in Table 3.1.

Lemma A.3.2 (Lazy intruder soundness)**.** *Let $\mathcal{A}$ be a FLIC, ρ be a choice of recipes such that $dom(\rho) \cap rvars(\mathcal{A}) = \emptyset$, let σ be a substitution such that $dom(\sigma) \cap vars(\mathcal{A}) = \emptyset$, let $\mathcal{I} \models \mathcal{A}$ such that $\mathcal{I} \models_\Sigma \sigma$ and let ρ_0 be a ground choice of recipes such that $\rho_0 \models_\Sigma \rho$. Let $(\rho', \mathcal{A}', \sigma')$ such that $(\rho, \mathcal{A}, \sigma) \rightsquigarrow (\rho', \mathcal{A}', \sigma')$, ρ' represents ρ_0 with ρ_0' w.r.t. $\mathcal{A}$ and $\mathcal{I}$, ρ_0' constructs $\mathcal{I}(\mathcal{A}')$ and $\mathcal{I} \models_\Sigma \sigma'$. Then ρ_0 constructs $\mathcal{I}(\mathcal{A})$.*

Proof. We start by showing that ρ_0' constructs $\mathcal{I}(\mathcal{A})$. Let R be a recipe variable such that $\mathcal{I}(\mathcal{A}) = \mathcal{A}_1.{+R} \mapsto t.\mathcal{A}_2$. First, we consider the case that $R \notin dom(\rho')$. Then $\mathcal{I}(\mathcal{A}') = \mathcal{A}_1'.{+R} \mapsto t.\mathcal{A}_2'$ and $\mathcal{A}_1'(\rho_0'(R)) = t$, so $\mathcal{A}_1(\rho_0'(R)) = t$.

Next, we consider the case that $R \in dom(\rho')$. We proceed by distinguishing which lazy intruder rule has been applied.

- **Unification**: Then $\mathcal{I}(\mathcal{A}) = \mathcal{A}_1.{-l} \mapsto t.\mathcal{A}_2.{+R} \mapsto t.\mathcal{A}_3$ and $\rho_0'(R) = l$ so $\mathcal{A}_1(\rho_0'(R)) = t$.

- **Composition**: Then

$$\begin{aligned}\mathcal{I}(\mathcal{A}) &= \mathcal{A}_1.{+}R \mapsto f(t_1,\ldots,t_n).\mathcal{A}_2 \ ,\\ \mathcal{I}(\mathcal{A}') &= \mathcal{A}'_1.{+}R_1 \mapsto t_1.\cdots.{+}R_n \mapsto t_n.\mathcal{A}'_3 \ ,\\ \rho'_0(R) &= f(\rho'_0(R_1),\ldots,\rho'_0(R_n)) \ .\end{aligned}$$

  Therefore $\mathcal{A}_1(\rho'_0(R)) = f(\mathcal{I}(\mathcal{A}')(\rho'_0(R_1)),\ldots,\mathcal{I}(\mathcal{A}')(\rho'_0(R_n))) = f(t_1,\ldots,t_n) = t$.

- **Guessing**: Then $\mathcal{I}(\mathcal{A}) = \mathcal{A}_1.{+}R \mapsto c.\mathcal{A}_2$ and $\rho'_0(R) = c$ so $\mathcal{A}_1(\rho'_0(R)) = c$.

- **Repetition**: Then $\mathcal{I}(\mathcal{A}) = \mathcal{A}_1.{+}R' \mapsto t.\mathcal{A}_2.{+}R \mapsto t.\mathcal{A}_3$, ${+}R' \mapsto t \in \mathcal{I}(\mathcal{A}')$ and $\rho'_0(R) = \rho'_0(R')$. Let $\mathcal{A}_0 = \mathcal{A}_1.{+}R' \mapsto t.\mathcal{A}_2$. Then $\mathcal{A}_0(\rho'_0(R)) = \mathcal{I}(\mathcal{A}')(\rho'_0(R')) = t$.

We have shown that ρ'_0 constructs $\mathcal{I}(\mathcal{A})$. Since ρ' represents ρ_0 with ρ'_0, for every $R \in$ *rvars*$(\mathcal{A})$, we have $\mathcal{I}(\mathcal{A})(\rho'_0(R)) = \mathcal{I}(\mathcal{A})(\rho_0(R))$. Thus ρ_0 constructs $\mathcal{I}(\mathcal{A})$. □

For completeness, we are given a ground choice of recipes ρ_0 that satisfies the constraints. We consider the first non-simple constraint and we show that there is one lazy intruder rule applicable. For the most part, we simply follow the recipe that ρ_0 gives, however in some cases the same recipe cannot be chosen using the lazy intruder rules, and this is where our representation of choice of recipes (Definition 3.1.8) comes in: we show that there is in fact some applicable rule such that the lazy intruder solution represents ρ_0.

Lemma A.3.3 (Lazy intruder completeness)**.** *Let $\mathcal{A}$ be a FLIC, ρ be a choice of recipes such that dom$(\rho) \cap$ rvars$(\mathcal{A}) = \emptyset$, let σ be a substitution such that dom$(\sigma) \cap$ vars$(\mathcal{A}) = \emptyset$, let $\mathcal{I} \models \mathcal{A}$ such that $\mathcal{I} \models_\Sigma \sigma$ and let ρ_0 be a ground choice of recipes such that $\rho_0 \models_\Sigma \rho$ and ρ_0 constructs $\mathcal{I}(\mathcal{A})$. Then either $\mathcal{A}$ is simple or there exists $(\rho', \mathcal{A}', \sigma')$ such that $(\rho, \mathcal{A}, \sigma) \rightsquigarrow (\rho', \mathcal{A}', \sigma')$, ρ' represents ρ_0 with ρ'_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$, ρ'_0 constructs $\mathcal{I}(\mathcal{A}')$ and $\mathcal{I} \models_\Sigma \sigma'$.*

Proof. Assume that $\mathcal{A}$ is not simple. Let ${+}R \mapsto t \in \mathcal{A}$ denote the first non-simple constraint, i.e., t is either a composed term, an intruder variable that occurred before, or a privacy variable. We proceed by case distinction on t:

- If $t \notin \mathcal{V}$, i.e., $t = f(t_1,\ldots,t_n)$: We consider different subcases depending on the recipe $\rho_0(R)$.
  - If $\rho_0(R) = l \in$ *dom*$(\mathcal{A})$ and $\mathcal{A}(l) = s \notin \mathcal{V}$: Then $\mathcal{A} = \mathcal{A}_1.{-}l \mapsto s.\mathcal{A}_2.{+}R \mapsto t.\mathcal{A}_3$. Therefore **Unification** is applicable, producing $(\rho', \mathcal{A}', \sigma')$ where $\rho' = [R \mapsto l]\rho$, $\mathcal{A}' = \sigma'(\mathcal{A}_1.{-}l \mapsto s.\mathcal{A}_2.\mathcal{A}_3)$ and $\sigma' = mgu(\sigma \wedge s \doteq t)$. Let $\rho'_0 = \rho_0$. Then ρ' represents ρ_0 with ρ'_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$, because the only recipe variable in *rvars*$(\mathcal{A}) \cap$ *dom*(ρ') is R, and $\rho'_0(R) = \rho_0(R) = l$. Moreover, since ρ_0 constructs $\mathcal{I}(\mathcal{A})$, we have ρ'_0 constructs $\mathcal{I}(\mathcal{A}')$ and also $\mathcal{I}(s) = \mathcal{I}(t)$, thus $\mathcal{I} \models_\Sigma \sigma'$.
  - If $\rho_0(R) = l \in$ *dom*$(\mathcal{A})$ and $\mathcal{A}(l) \in \mathcal{V}$: We cannot directly use **Unification** because the label l used in ρ_0 maps to a variable. We have two subcases, depending on whether we can find an earlier label suitable for **Unification**.
    - If there exists a label $l' <_\mathcal{A} l$ such that $\mathcal{I}(\mathcal{A})(l') = t$ and $\mathcal{A}(l') = s \notin \mathcal{V}$: Then $\mathcal{A} = \mathcal{A}_1.{-}l' \mapsto s.\mathcal{A}_2.{+}R \mapsto t.\mathcal{A}_3$. Therefore **Unification** is applicable, producing $(\rho', \mathcal{A}', \sigma')$ where $\rho' = [R \mapsto l']\rho$, $\mathcal{A}' = \sigma'(\mathcal{A}_1.{-}l' \mapsto s.\mathcal{A}_2.\mathcal{A}_3)$ and $\sigma' = mgu(\sigma \wedge s \doteq t)$. Let $\rho'_0(R) = l'$ and $\rho'_0(R') = \rho_0(R')$ for other recipe variables. Then ρ' represents ρ_0 with ρ'_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$, because the only recipe variable in *rvars*$(\mathcal{A}) \cap$ *dom*(ρ') is R, and $\rho'_0(R) = l' <_\mathcal{A} l = \rho_0(R)$. Moreover, since ρ_0 constructs $\mathcal{I}(\mathcal{A})$, we have ρ'_0 constructs $\mathcal{I}(\mathcal{A}')$ and also $\mathcal{I}(s) = \mathcal{I}(t)$, thus $\mathcal{I} \models_\Sigma \sigma'$.

    * Otherwise: Let $\mathcal{A} = \mathcal{A}_1.{+}R \mapsto f(t_1, \dots, t_n).\mathcal{A}_2$. Since all labels in $\mathcal{A}_1$ that produce the message t under model $\mathcal{I}$ are mapping to variables, we know that t can be composed by the intruder before they receive the message labeled with l (recall that all instances of privacy variables are public constants, so if a label maps to a privacy variable x then the constraints can just as well be solved using the recipe $\mathcal{I}(x)$). Thus $f \in \Sigma_{pub}$ and there exists a ground recipe $r <_{\mathcal{A}} l$ such that $\mathcal{I}(\mathcal{A})(r) = t$ and $r = f(r_1, \dots, r_n)$. Therefore, **Composition** is applicable, producing $(\rho', \mathcal{A}', \sigma')$ where $\rho' = [R \mapsto f(R_1, \dots, R_n)]\rho$, $\mathcal{A}' = \mathcal{A}_1.{+}R_1 \mapsto t_1.\cdots.{+}R_n \mapsto t_n.\mathcal{A}_2$ and $\sigma' = \sigma$, where the R_i are fresh recipe variables. Let $\rho'_0(R_i) = r_i$ for $i \in \{1, \dots, n\}$ and $\rho'_0(R') = \rho_0(R')$ for other recipe variables. Then ρ' represents ρ_0 with ρ'_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$, because the only recipe variable in $\mathit{rvars}(\mathcal{A}) \cap \mathit{dom}(\rho')$ is R and $\rho'_0(R) = r <_{\mathcal{A}} l = \rho_0(R)$. Moreover, since ρ_0 constructs $\mathcal{I}(\mathcal{A})$, we have ρ'_0 constructs $\mathcal{I}(\mathcal{A}')$.
  - If $\rho_0(R) = f(r_1, \dots, r_n)$: Then $\mathcal{A} = \mathcal{A}_1.{+}R \mapsto f(t_1, \dots, t_n).\mathcal{A}_2$ and $f \in \Sigma_{pub}$. Therefore, **Composition** is applicable, producing $(\rho', \mathcal{A}', \sigma')$ where $\rho' = [R \mapsto f(R_1, \dots, R_n)]\rho$, $\mathcal{A}' = \mathcal{A}_1.{+}R_1 \mapsto t_1.\cdots.{+}R_n \mapsto t_n.\mathcal{A}_2$ and $\sigma' = \sigma$, where the R_i are fresh recipe variables. Let $\rho'_0(R_i) = r_i$ for $i \in \{1, \dots, n\}$ and $\rho'_0(R') = \rho_0(R')$ for other recipe variables. Then ρ' represents ρ_0 with ρ'_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$, because the only recipe variable in $\mathit{rvars}(\mathcal{A}) \cap \mathit{dom}(\rho')$ is R, and $\rho_0(R) = \rho'_0(R) = f(r_1, \dots, r_n)$. Moreover, since ρ_0 constructs $\mathcal{I}(\mathcal{A})$, we have ρ'_0 constructs $\mathcal{I}(\mathcal{A}')$.

- If $t \in \mathcal{V}_{intruder}$: Then $\mathcal{A} = \mathcal{A}_1.{+}R' \mapsto t.\mathcal{A}_2.{+}R \mapsto t.\mathcal{A}_3$. Therefore, **Repetition** is applicable, producing $(\rho', \mathcal{A}', \sigma')$ where $\rho' = [R \mapsto R']\rho$, $\mathcal{A}' = \mathcal{A}_1.{+}R' \mapsto t.\mathcal{A}_2.\mathcal{A}_3$ and $\sigma' = \sigma$. Let $\rho'_0(R) = \rho_0(R')$ and $\rho'_0(R'') = \rho_0(R'')$ for other recipe variables. Then ρ' represents ρ_0 with ρ'_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$, because the only recipe variable in $\mathit{rvars}(\mathcal{A}) \cap \mathit{dom}(\rho')$ is R and $\mathcal{I}(\mathcal{A})(\rho_0(R)) = \mathcal{I}(\mathcal{A})(\rho'_0(R))$. Moreover, since ρ_0 constructs $\mathcal{I}(\mathcal{A})$, we have ρ'_0 constructs $\mathcal{I}(\mathcal{A}')$.

- If $t \in \mathcal{V}_{privacy}$: Then $\mathcal{A} = \mathcal{A}_1.{+}R \mapsto t.\mathcal{A}_2$ and $\mathcal{I}(t) = c$ for some $c \in \mathit{dom}(t)$. Therefore, **Guessing** is applicable, producing $(\rho', \mathcal{A}', \sigma')$ where $\rho' = [R \mapsto c]\rho$, $\mathcal{A}' = \sigma'(\mathcal{A}_1.\mathcal{A}_2)$ and $\sigma' = \mathit{mgu}(\sigma \wedge t \doteq c)$. If $\rho_0(R) = c$, let $\rho'_0 = \rho_0$. Otherwise let $\rho'_0(R) = c$ and $\rho'_0(R') = \rho_0(R')$ for other recipe variables. Then ρ' represents ρ_0 with ρ'_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$, because the only recipe variable in $\mathit{rvars}(\mathcal{A}) \cap \mathit{dom}(\rho')$ is R and either $\rho_0(R) = \rho'_0(R) = c$ or $\rho'_0(R) = c <_{\mathcal{A}} \rho_0(R)$. Moreover, since ρ_0 constructs $\mathcal{I}(\mathcal{A})$, we have ρ'_0 constructs $\mathcal{I}(\mathcal{A}')$ and also $\mathcal{I}(t) = c$, thus $\mathcal{I} \models_{\Sigma} \sigma'$. □

In the lemma below we simply use the soundness and completeness lemmas to combine both implications and obtain an equivalence.

Lemma A.3.4. *Let $\mathcal{A}$ be a FLIC, ρ be a choice of recipes such that $\mathit{dom}(\rho) \cap \mathit{rvars}(\mathcal{A}) = \emptyset$, let σ be a substitution such that $\mathit{dom}(\sigma) \cap \mathit{vars}(\mathcal{A}) = \emptyset$, let $\mathcal{I} \models \mathcal{A}$ such that $\mathcal{I} \models_{\Sigma} \sigma$ and let ρ_0 be a ground choice of recipes such that $\rho_0 \models_{\Sigma} \rho$. Then ρ_0 constructs $\mathcal{I}(\mathcal{A})$ iff there exists $(\rho', \mathcal{A}', \sigma')$ such that $(\rho, \mathcal{A}, \sigma) \rightsquigarrow^* (\rho', \mathcal{A}', \sigma')$, ρ' represents ρ_0 with ρ'_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$, ρ'_0 constructs $\mathcal{I}(\mathcal{A}')$ and $\mathcal{I} \models_{\Sigma} \sigma'$.*

Proof. By induction, using Lemmas A.3.2 and A.3.3. □

For termination, we prove that the lazy intruder rules really form a *reduction* relation by defining a weight for FLICs and showing that every rule decreases the weight. Intuitively, either the lazy intruder rule instantiates some intruder variable so the constraints are simpler or the total size of the messages to send decreases because there is one less message to send.

Lemma A.3.5 (Lazy intruder termination). *Let $\mathcal{A}$ be a FLIC, ρ be a choice of recipes such that $dom(\rho) \cap rvars(\mathcal{A}) = \emptyset$ and let σ be a substitution such that $dom(\sigma) \cap vars(\mathcal{A}) = \emptyset$. Then there is a finite number of $(\rho', \mathcal{A}', \sigma')$ such that $(\rho, \mathcal{A}, \sigma) \rightsquigarrow^* (\rho', \mathcal{A}', \sigma')$.*

Proof. We define the weight of a FLIC $\mathcal{A}$ to be the pair (v, s), where

- v is the number of intruder variables in the FLIC: $v = \#ivars(\mathcal{A})$; and
- s is the sum of the size of the messages sent: $s = \sum_{+R \mapsto t \in \mathcal{A}} size(t)$, where the size of a message is defined as 1 for a variable and $size(f(t_1, \dots, t_n)) = 1 + \sum_{i=1}^{n} size(t_i)$ for a composed message.

The weights with the lexicographic order form a well-founded ordering. Every rule decreases the weight.

- **Unification**: The mgu may instantiate intruder variables so v would decrease, and if not then v stays the same but one message sent is removed so s decreases.
- **Composition**: v stays the same, but the message is decomposed by removing the outermost function application so s decreases (by 1).
- **Guessing** and **Repetition**: v stays the same, but one message sent is removed so s decreases (by 1).

There cannot be an infinite sequence of decreasing weights so the lazy intruder terminates. □

The correctness of the lazy intruder is defined as the conjunction of soundness, completeness and termination.

Theorem 3.1.1 (Lazy intruder correctness). *Let $\mathcal{A}$ be a FLIC, σ be a substitution, $\mathcal{I} \models \mathcal{A}$ such that $\mathcal{I} \models_\Sigma \sigma$ and let ρ_0 be a ground choice of recipes. Then ρ_0 constructs $\mathcal{I}(\mathcal{A})$ iff there exists $\rho \in LI(\mathcal{A}, \sigma)$ such that ρ represents ρ_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$. Moreover, $LI(\mathcal{A}, \sigma)$ is finite.*

Proof. This follows directly from Lemmas A.3.4 and A.3.5. □

A.3.2 Compose-check correctness

In this section we prove results for our compose-checks, i.e., the intruder experiments where one label is compared to a recipe that may produce the same message.

The following lemma is used to prove the termination of the compose-checks in the next theorem. We have that given a FLIC, two recipes and the unifier for the messages produced by these recipes, either: the unifier only depends on privacy variables and then no matter the further choices of recipes applied to the FLIC, the unifier between the two messages will always only depends on privacy variables; or the unifier depends on at least one intruder variable and then for every lazy intruder result, after applying it to the FLIC the unifier will not depend on intruder variables anymore.

Lemma A.3.6. *Let $\mathcal{A}$ be a simple FLIC, r_1, r_2 be recipes and $\sigma = mgu(\mathcal{A}(r_1) \doteq \mathcal{A}(r_2))$.*

- *If $isPriv(\sigma)$, then for every choice of recipes ρ, we have $isPriv(\sigma')$, where $\sigma' = mgu(\rho(\mathcal{A})(\rho(r_1)) \doteq \rho(\mathcal{A})(\rho(r_2)))$.*

- *If not isPriv(σ), then for every $\rho \in LI(\mathcal{A}, \sigma)$, we have isPriv($\sigma'$), where $\sigma' = mgu(\rho(\mathcal{A})(\rho(r_1)) \doteq \rho(\mathcal{A})(\rho(r_2)))$.*

Proof. First we consider the case that *isPriv*(σ). Let ρ be a choice of recipes and $\sigma' = mgu(\rho(\mathcal{A})(\rho(r_1)) \doteq \rho(\mathcal{A})(\rho(r_2)))$. If $\mathcal{A}(r_1)$ contains an intruder variable as a subterm, then $\mathcal{A}(r_2)$ contains the same intruder variable in the same position; otherwise, the intruder variable would be substituted and we would not have *isPriv*(σ). The argument is similar if $\mathcal{A}(r_2)$ contains intruder variables. Since the intruder variables are not relevant for unifying the two messages, the intruder variables can be instantiated in any way. Then we have $\sigma(\rho(\mathcal{A})(\rho(r_1))) = \sigma(\rho(\mathcal{A})(\rho(r_2)))$, which means that σ is an instance of σ' and thus *isPriv*(σ').

Next we consider the case that not *isPriv*(σ). Let $\rho \in LI(\mathcal{A}, \sigma)$, $\sigma' = mgu(\rho(\mathcal{A})(\rho(r_1)) \doteq \rho(\mathcal{A})(\rho(r_2)))$ and $\mathcal{A}'$, σ'' be such that $(\varepsilon, \sigma(\mathcal{A}), \sigma) \rightsquigarrow^* (\rho, \mathcal{A}', \sigma'')$ and $\mathcal{A}'$ is simple. By definition of ρ and $\mathcal{A}'$, $\mathcal{A}'(\rho(r_1)) = \mathcal{A}'(\rho(r_2))$. Moreover, $\rho(\mathcal{A})$ and $\mathcal{A}'$ are the same up to renaming of intruder variables and up to substitution of privacy variables, so there exists a substitution τ such that *isPriv*(τ) and $\tau(\rho(\mathcal{A})(\rho(r_1))) = \tau(\rho(\mathcal{A})(\rho(r_2)))$. Note that we have *isPriv*(τ) because the fresh intruder variables, i.e., the ones that are introduced by ρ, are just renamed compared to the intruder variables in $\mathcal{A}'$. Then τ is an instance of σ' and thus *isPriv*(σ'). $\square$

We prove that the compose-checks terminate by defining a weight for symbolic states and showing that every rule decreases the weight. Intuitively, after **Privacy split** there is one less pair of recipes to check, and after **Recipe split** we have applied some lazy intruder result, so using the previous lemma we have fewer FLICs where the unifier depends on intruder variables.

Theorem A.3.1 (Compose-check termination). *There does not exist any infinite sequence of symbolic states $(\mathcal{S}_i)_{i \geq 1}$ where for every i, there exists $\mathcal{C}_i$ such that $\mathcal{S}_i \rightarrowtail \mathcal{C}_i$ and $\mathcal{S}_{i+1} \in \mathcal{C}_i$.*

Proof. Let $\mathcal{S}$ be a symbolic state and $\mathcal{A}_1, \dots, \mathcal{A}_n$ be the FLICs in that state. We define the weight of $\mathcal{S}$ to be the pair (p, s), where

- p is the number of pairs recipes to check: $p = \#Pairs(\mathcal{S})$; and
- s is the sum, over the pairs of recipes, of the number of FLICs in which the unifier depends on intruder variables and there exists a solution to the constraints: $s = \sum_{(l,r) \in Pairs(\mathcal{S})} \#\{\mathcal{A}_i \mid \text{not } isPriv(\sigma_{i,(l,r)}) \text{ and } LI(\mathcal{A}_i, \sigma_{i,(l,r)}) \neq \emptyset\}$, where $\sigma_{i,(l,r)} = mgu(\mathcal{A}_i(l) \doteq \mathcal{A}_i(r))$ for $i \in \{1, \dots, n\}$ and $(l,r) \in Pairs(\mathcal{S})$.

The weights with the lexicographic order form a well-founded ordering. Every rule decreases the weight. Let $\mathcal{S}'$ be a symbolic state such that there exists $\mathcal{C}$ with $\mathcal{S} \rightarrowtail \mathcal{C}$ and $\mathcal{S}' \in \mathcal{C}$. First we consider that $\mathcal{S}'$ is produced by the rule **Privacy split**. One pair (l, r) is now checked and the FLICs are not changed, so p decreases.

Next we consider the case that $\mathcal{S}'$ is produced by the rule **Recipe split**. There exist $(l, r) \in Pairs(\mathcal{S})$ and $i \in \{1, \dots, n\}$ such that not *isPriv*(σ_i) and $LI(\mathcal{A}_i, \sigma_i) \neq \emptyset$, where $\sigma_i = mgu(\mathcal{A}_i(l) \doteq \mathcal{A}_i(r))$. The first subcase is that $\mathcal{S}'$ is produced by applying some choice of recipes $\rho \in LI(\mathcal{A}_i, \sigma_i)$. For every pair $(l', r') \in Pairs(\mathcal{S})$, there is at most one corresponding pair $(l', \rho(r')) \in Pairs(\mathcal{S}')$ so p may decrease (e.g., if some choice of recipes used to compute the pairs in $\mathcal{S}'$ is not an instance of ρ) but p cannot increase. By Lemma A.3.6, if the unifier only depends on privacy variables, this is still the case in $\mathcal{S}'$, and for the FLIC $\rho(\mathcal{A}_i)$, the unifier does not depend on intruder variables anymore, thus s decreases.

The second subcase is that $\mathcal{S}'$ is produced by excluding σ_i. Then the FLICs are not changed so p stays the same, but s decreases because now $\mathit{LI}(\mathcal{A}_i, \sigma_i) = \emptyset$, since σ_i is excluded.

There cannot be an infinite sequence of decreasing weights so the compose-checks terminate. $\square$

For correctness, we show that every rule partitions the set of ground states represented, so that all compose-checks preserve the semantics of symbolic states.

Theorem 3.3.1 (Compose-check correctness)**.** *Let $\mathcal{S}$ be a finished symbolic state, $(l, r) \in \mathit{Pairs}(\mathcal{S})$ and $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be the symbolic states such that $\mathcal{S} \rightarrowtail \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ w.r.t. (l, r). Then $[\![\mathcal{S}]\!] = \biguplus_{i=1}^{n} [\![\mathcal{S}_i]\!]$. Moreover, there does not exist any infinite sequence $(\mathcal{S}_i)_{i \geq 1}$ where for every i, there exists $\mathcal{C}_i$ such that $\mathcal{S}_i \rightarrowtail \mathcal{C}_i$ and $\mathcal{S}_{i+1} \in \mathcal{C}_i$.*

Proof. Let $\mathcal{P} = \{(0, \phi_1, \mathcal{A}_1, \_, \_, \_), \ldots, (0, \phi_n, \mathcal{A}_n, \_, \_, \_)\}$ be the possibilities in $\mathcal{S}$. First we consider the case that **Privacy split** is applicable. For every $i \in \{1, \ldots, n\}$, $\mathit{isPriv}(\sigma_i)$ or $\mathit{LI}(\mathcal{A}_i, \sigma_i) = \emptyset$, where $\sigma_i = \mathit{mgu}(\mathcal{A}_i(l) \doteq \mathcal{A}_i(r))$. We are partitioning the set of ground states based on the interpretations of privacy variables. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be the symbolic states produced by the first and second subcase of the rule, respectively. We start by showing that $[\![\mathcal{S}]\!] \subseteq [\![\mathcal{S}_1]\!] \uplus [\![\mathcal{S}_2]\!]$. Let $S = (\alpha, \beta_0, \gamma, \mathcal{P}') \in [\![\mathcal{S}]\!]$, ρ be the ground choice of recipes defining S and $\mathit{concr} = \gamma(\mathit{struct}_i)$ for some $i \in \{1, \ldots, n\}$ be the concrete frame in S, where $\mathit{struct}_j = \rho(\mathcal{A}_j)$ for $j \in \{1, \ldots, n\}$.

- If $\mathit{isPriv}(\sigma_i)$ and $\gamma \models \sigma_i$: Then we show that $S \in [\![\mathcal{S}_1]\!]$. Define
$$\beta' = \beta(S) \wedge \bigwedge_{j=1}^{n} \left(\phi_j \Rightarrow \begin{cases} \sigma_j & \text{if } \mathit{isPriv}(\sigma_j) \\ \mathsf{false} & \text{otherwise} \end{cases} \right) .$$
We need to show that $\beta(S) \equiv_\Sigma \beta'$. Let $\mathcal{I} \models_\Sigma \beta(S)$. There exists $j \in \{1, \ldots, n\}$ such that $\mathcal{I} \models_\Sigma \phi_j \wedge \mathit{concr} \sim \mathit{struct}_j$. Since $\gamma \models \sigma_i$ and $\mathit{concr} = \gamma(\rho(\mathcal{A}_i))$, $\mathit{concr}(l) = \mathit{concr}(r)$. Then $\mathcal{I}(\mathit{struct}_j)(l) = \mathcal{I}(\mathit{struct}_j)(r)$, so $\mathcal{I} \models \sigma_j$. Then $\mathcal{I} \models_\Sigma \phi_j \wedge \sigma_j \wedge \mathit{concr} \sim \mathit{struct}_j$, so $\mathcal{I} \models_\Sigma \beta'$. Conversely, for every $\mathcal{I} \models_\Sigma \beta'$, we have $\mathcal{I} \models_\Sigma \beta(S)$. Thus $\beta(S) \equiv_\Sigma \beta'$.

- Otherwise, we show that $S \in [\![\mathcal{S}_2]\!]$. Define
$$\beta' = \beta(S) \wedge \bigwedge_{j=1}^{n} \left(\phi_j \Rightarrow \begin{cases} \neg\sigma_j & \text{if } \mathit{isPriv}(\sigma_j) \\ \mathsf{true} & \text{otherwise} \end{cases} \right) .$$
We need to show that $\beta(S) \equiv_\Sigma \beta'$. Let $\mathcal{I} \models_\Sigma \beta(S)$. There exists $j \in \{1, \ldots, n\}$ such that $\mathcal{I} \models_\Sigma \phi_j \wedge \mathit{concr} \sim \mathit{struct}_j$. Since $\gamma \models \neg\sigma_i$ or $\mathit{LI}(\mathcal{A}_i, \sigma_i) = \emptyset$, $\mathit{concr}(l) \neq \mathit{concr}(r)$. Then $\mathcal{I}(\mathit{struct}_j)(l) \neq \mathcal{I}(\mathit{struct}_j)(r)$, so if $\mathit{isPriv}(\sigma_j)$ then $\mathcal{I} \models_\Sigma \phi_j \wedge \neg\sigma_j \wedge \mathit{concr} \sim \mathit{struct}_j$. Then $\mathcal{I} \models_\Sigma \beta'$. Conversely, for every $\mathcal{I} \models_\Sigma \beta'$, we have $\mathcal{I} \models_\Sigma \beta(S)$. Thus $\beta(S) \equiv_\Sigma \beta'$.

The cases are mutually exclusive, so $[\![\mathcal{S}]\!] \subseteq [\![\mathcal{S}_1]\!] \uplus [\![\mathcal{S}_2]\!]$. Similarly, we have $[\![\mathcal{S}_1]\!] \uplus [\![\mathcal{S}_2]\!] \subseteq [\![\mathcal{S}]\!]$.

Next we consider the case that **Recipe split** is applicable. There exists $i \in \{1, \ldots, n\}$ such that not $\mathit{isPriv}(\sigma_i)$ and $\mathit{LI}(\mathcal{A}_i, \sigma_i) = \{\rho_1, \ldots, \rho_k\}$, where $\sigma_i = \mathit{mgu}(\mathcal{A}_i(l) \doteq \mathcal{A}_i(r))$. We are partitioning the set of ground states based on the ground choices of recipes. Let $\mathcal{S}_j = \rho_j(\mathcal{S})$ for $j \in \{1, \ldots, k\}$, and $\mathcal{S}'$ be the symbolic state in which σ_i is excluded for $\mathcal{A}_i$. Let $S \in [\![\mathcal{S}]\!]$ and ρ be the corresponding ground choice of recipes. Then $S \in [\![\mathcal{S}_j]\!]$ if ρ is represented by ρ_j (note that the ρ_j are mutually exclusive); otherwise $S \in [\![\mathcal{S}']\!]$. Conversely, $[\![\mathcal{S}']\!] \uplus \biguplus_{j=1}^{n} [\![\mathcal{S}_j]\!] \subseteq [\![\mathcal{S}]\!]$.

The termination follows from Theorem A.3.1. $\square$

A.3.3 Normal symbolic states

We now prove our results for normal symbolic states. The lemma below says that given a ground state represented by a normal symbolic state (with some choice of recipes), the concrete frame in that state is statically equivalent to all other frames considered possible by the intruder (using the same choice of recipes). In other words, in a normal symbolic state, the intruder cannot distinguish the possibilities anymore. The proof assumes that there is an arbitrary witness against static equivalence and shows a contradiction.

Lemma A.3.7. *Let $\mathcal{S} = (\alpha_0, \beta_0, \_, \_)$ be a normal symbolic state, where the possibilities have conditions $\phi_1, \dots, \phi_n$ and FLICs $\mathcal{A}_1, \dots, \mathcal{A}_n$. Let $S \in [\![\mathcal{S}]\!]$, ρ_0 be the ground choice of recipes defining S and concr be the concrete frame in S. Let $\theta \models \alpha_0 \wedge \beta_0 \wedge \phi_i$ for some $i \in \{1, \dots, n\}$ and $concr' = \theta(\rho_0(\mathcal{A}_i))$. Then $concr \sim concr'$.*

Proof. Assume that the frames are not statically equivalent. This means there exists a witness, i.e., a pair of ground recipes (r_1, r_2) such that $concr(r_1) = concr(r_2)$ and $concr'(r_1) \neq concr'(r_2)$. We show that for each witness (r_1, r_2), either it contradicts that $\mathcal{S}$ is normal or there is a smaller witness according to the following well-founded ordering:

$$\begin{aligned}(r_1, r_2) < (r_1', r_2') \quad \text{iff} \quad & w(r_1) < w(r_1') \text{ and } w(r_2) \leq w(r_2') \\ \text{or } & w(r_1) \leq w(r_1') \text{ and } w(r_2) < w(r_2') \\ \text{or } & w(r_1) < w(r_2') \text{ and } w(r_2) \leq w(r_1') \\ \text{or } & w(r_1) \leq w(r_2') \text{ and } w(r_2) < w(r_1')\end{aligned}$$

where the weight $w(r)$ of recipe r is defined as the lexicographically ordered pair (s, h) where s is the size of $concr(r)$ and h is the number of the highest label in r, i.e., that occurs on the hth position in $concr$; and $h = 0$ if there are no labels in r.

We first handle the case that both r_1 and r_2 are composed. Then $r_1 = f(r_1^1, \dots, r_1^n)$ and $r_2 = f(r_2^1, \dots, r_2^n)$ for the same f (otherwise they cannot produce the same value in $concr$). Then at least one of the pairs (r_1^i, r_2^i) is already a witness that is smaller in the ordering.

Thus, in all remaining cases we have a pair (l, r) where l is a label and r is a ground recipe. Without loss of generality, we can assume that if r is also a label then l occurs after r in the frames. By definition of $[\![\mathcal{S}]\!]$, there exist $j \in \{1, \dots, n\}$, one FLIC $\mathcal{A}_j$ and one model $\gamma \models \alpha_0 \wedge \beta_0 \wedge \gamma_0 \wedge \phi_j$ such that $concr = \gamma(\rho_0(\mathcal{A}_j))$. Let R be a fresh recipe variable and $\mathcal{A} = \mathcal{A}_j.{+}R \mapsto \mathcal{A}_j(l)$. Let $\mathcal{I}$ be the interpretation such that $\mathcal{I}$ and γ agree on the privacy variables and for every R' such that $\mathcal{A}_j = \mathcal{A}_0.{+}R' \mapsto X.\mathcal{A}_0'$, $\mathcal{I}(X) = concr(\rho_0(R'))$. Let us extend ρ_0 with $\rho_0(R) = r$, where r is the ground recipe such that (l, r) is a witness. Then we have that ρ_0 constructs $\mathcal{I}(\mathcal{A})$. By Theorem 3.1.1, there exists $\rho \in LI(\mathcal{A}, \varepsilon)$ such that ρ represents ρ_0 w.r.t. $\mathcal{A}$ and $\mathcal{I}$. Let ρ_0' be the respective instance of ρ. Since $\mathcal{S}$ is normal, we know that $l \simeq \rho(R)$, i.e., we have checked that for every ground choice of recipes ρ', $(l, \rho'(\rho(R)))$ is not a witness.

Let us consider the case that $\rho(R) = R' \in rvars(\mathcal{A}_j)$, which can only happen if the repetition rule has been used, which in turn can only happen if $\mathcal{A}_j = \mathcal{A}_0.{+}R' \mapsto X.\mathcal{A}_0'.{-}l \mapsto X.\mathcal{A}_0''$, so l maps to a message that the intruder has sent earlier and that they received back from some agent. As mentioned above, since $\mathcal{S}$ is normal, the pair $(l, \rho_0(R'))$ is not a witness (the intruder can check that in all FLICs they received back at l whatever they sent at R'). Thus, the pair $(\rho_0(R'), r)$ must be a witness, and this is smaller than (l, r), because the size of the produced message is the same, but $\rho_0(R')$ can only use labels from $\mathcal{A}_0$ and has thus a lower weight than l.

Next we consider the case that $\rho(R) \in dom(\mathcal{S})$. Since ρ represents ρ_0 with ρ_0', we have $\rho_0(R) \in dom(\mathcal{S})$ and either $\rho_0'(R) = \rho_0(R)$ or $\rho_0'(R) <_{\mathcal{A}} \rho_0(R)$. The subcase

$\rho_0'(R) = \rho_0(R) = l'$ is however impossible, because as mentioned above, $\mathcal{S}$ is normal so (l, l') is checked and cannot be a witness. For the subcase $\rho_0'(R) = l' <_{\mathcal{A}} \rho_0(R) = l''$, we have that (l, l'') is a witness and $l \simeq l'$, so (l', l'') must be a witness, and this is smaller because $l' <_{\mathcal{A}} l'' <_{\mathcal{A}} l$.

Finally we consider the case that $\rho(R)$ is a composed recipe. Since ρ represents ρ_0 with ρ_0', we have either $\rho_0'(R) = f(r_1', \ldots, r_n')$ and $\rho_0(R) = f(r_1, \ldots, r_n)$ such that $\mathcal{I}(\mathcal{A})(r_i') = \mathcal{I}(\mathcal{A})(r_i)$ for $i \in \{1, \ldots, n\}$ or $\rho_0(R) \in \mathit{dom}(\mathcal{S})$ and $\rho_0'(R) <_{\mathcal{A}} \rho_0(R)$. For the subcase $\rho_0'(R) = f(r_1', \ldots, r_n')$ and $\rho_0(R) = f(r_1, \ldots, r_n)$ such that $\mathcal{I}(\mathcal{A})(r_i') = \mathcal{I}(\mathcal{A})(r_i)$ for $i \in \{1, \ldots, n\}$, again since $\mathcal{S}$ is normal, $(l, f(r_1', \ldots, r_n'))$ has been checked and cannot be a witness. Thus, for $(l, f(r_1, \ldots, r_n))$ to be a witness, at least one of the pairs (r_i', r_i) has to be a witness. This is smaller than (l, r) since the recipes r_i', r_i produce proper subterms of the message $\mathit{concr}(l)$. For the subcase $\rho_0(R) \in \mathit{dom}(\mathcal{S})$ and $\rho_0'(R) <_{\mathcal{A}} \rho_0(R) = l'$, we have that (l, l') is a witness and $l \simeq \rho_0'(R)$, so $(\rho_0'(R), l')$ must be a witness, and this is smaller because $\rho_0'(R) <_{\mathcal{A}} l' <_{\mathcal{A}} l$.

Thus for every witness we can find a smaller witness, which is impossible along a well-founded ordering, and thus we can be sure that there are no witnesses. □

For the theorem below, we use the fact that in a normal symbolic state, the intruder cannot distinguish the possibilities (lemma above) to prove that all intruder deductions relevant for (α, β)-privacy are part of the formula β_0 in that state. Basically, the static equivalence encoded in the intruder knowledge β cannot lead to a violation of privacy, due to the indistinguishability, so it is enough to check consistency, i.e., (α, β_0) pairs.

Theorem 3.3.2. *Let $\mathcal{S}$ be a normal symbolic state. Then $\mathcal{S}$ satisfies privacy iff $\mathcal{S}$ is consistent.*

Proof. Let $\mathcal{P} = \{(0, \phi_1, \mathcal{A}_1, \_, \_, \_), \ldots, (0, \phi_n, \mathcal{A}_n, \_, \_, \_)\}$ be the possibilities in $\mathcal{S}$. First we assume that $\mathcal{S}$ satisfies privacy and show that $\mathcal{S}$ is consistent. Let $S = (\alpha, \_, \_, \_) \in [\![\mathcal{S}]\!]$ and $\mathcal{I} \models_{\Sigma_0} \alpha$. Since $\mathcal{S}$ satisfies privacy, $(\alpha, \beta(S))$-privacy holds so there exists $\mathcal{I}' \models_{\Sigma} \beta(S)$ such that $\mathcal{I}$ and $\mathcal{I}'$ agree on $\mathit{fv}(\alpha)$ and on the relations in Σ_0. Since $\beta(S) \models_{\Sigma} \beta_0$, $\mathcal{I}' \models_{\Sigma_0} \beta_0$. Therefore (α, β_0)-privacy holds. Thus $\mathcal{S}$ is consistent.

Next we assume that $\mathcal{S}$ is consistent and show that $\mathcal{S}$ satisfies privacy. Let $S = (\alpha, \beta_0, \_, \_) \in [\![\mathcal{S}]\!]$, ρ be the ground choice of recipes defining S and concr be the concrete frame in S. Let $\mathit{struct}_i = \rho(\mathcal{A}_i)$ for $i \in \{1, \ldots, n\}$ and $\mathcal{I} \models_{\Sigma_0} \alpha$. Since $\mathcal{S}$ is consistent, (α, β_0)-privacy holds, i.e., there exists $\mathcal{I}' \models_{\Sigma_0} \beta_0$ such that $\mathcal{I}$ and $\mathcal{I}'$ agree on $\mathit{fv}(\alpha)$ and the relations in Σ_0. Since $\alpha \wedge \beta_0 \models \bigvee_{i=1}^{n} \phi_i$, there exists $i \in \{1, \ldots, n\}$ such that $\mathcal{I}' \models \phi_i$. By Lemma A.3.7, $\mathit{concr} \sim \mathcal{I}'(\mathit{struct}_i)$ so $\mathcal{I}' \models_{\Sigma} \mathit{concr} \sim \mathit{struct}_i$. Therefore $\mathcal{I}' \models_{\Sigma} \beta(S)$, so $(\alpha, \beta(S))$-privacy holds, i.e., S satisfies privacy. This is true for every $S \in [\![\mathcal{S}]\!]$, thus $\mathcal{S}$ satisfies privacy. □

A.3.4 Algebraic properties

First, we prove that our congruence relation from Definition 3.4.1 is well-defined because our requirements imply that the supported term rewriting systems are convergent. Our destructor rules do not conflict with each other because every destructor occur in exactly one rule, and each rule yields a direct subterm of the message being decrypted.

Lemma 3.4.1. *Let E be a term rewriting system satisfying the requirements of Definition 3.4.1. Then E is convergent.*

Proof. For every rewrite rule, the right-hand side is a strict subterm of the left-hand side, therefore $\rightarrow_E$ is terminating. We now need to show that $\rightarrow_E$ is confluent. Let t, t_1, t_2 be terms such that $t \rightarrow_E t_1$ and $t \rightarrow_E t_2$. We show that we can join t_1 and t_2, i.e., there exists

t' such that $t_1 \to_E^* t'$ and $t_2 \to_E^* t'$. Let p_1 and p_2 be the positions of t where the rewrite rules $l_1 \to r_1$ and $l_2 \to r_2$ have been applied to yield t_1 and t_2, respectively. We proceed by case distinction:

1. $p_1 = p_2$: Then the two rewrite rules must be identical, because at that position there must be a destructor and each destructor can only occur in one rewrite rule. Thus $t_1 = t_2$.

2. p_1 and p_2 are disjoint, i.e., neither $p_1 \sqsubset p_2$ nor $p_2 \sqsubset p_1$ where $\sqsubset$ denotes the prefix relation: Then in t_1, the position p_2 is unchanged and vice-versa, so t_1 and t_2 can be joined.

3. $p_1 \sqsubset p_2$: Let $p_0 = p_2 - p_1$ be the relative position.

   - If $p_0 \notin Pos(l_1)$ or l_1 at position p_0 is a variable: Then the first rule is applicable both in t and t_2 at position p_1. The latter gives a term t' that is either identical to t_1 or the second rule can be still applied in t_1 to yield t'.
   - If $p_0 \in Pos(l_1)$ and l_1 at position p_0 is not a variable: Then it must be a destructor. Since l_1 only has a destructor at the root (recall that in every rewrite rule, all terms below the destructor, including the key terms, are destructor-free), $p_1 = p_2$ which is handled in Case 1 of this proof. This is however absurd, since p_0 can only have a destructor at the root , thus $p_1 = p_2$ which is a different case.

4. $p_2 \sqsubset p_1$: By symmetry with the previous case. □

For correctness of analysis, we show that the analysis strategy only adds shorthands to the FLICs and that it terminates (the strategy saturates the intruder knowledge by decrypting as far as possible).

Theorem 3.4.1 (Analysis correctness). *For a symbolic state $\mathcal{S}$, the analysis strategy produces in finitely many steps a set $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ of symbolic states that are analyzed. Further, for every ground state $S \in [\![\mathcal{S}]\!]$ there exists $S' \in [\![\mathcal{S}_i]\!]$, for some $i \in \{1, \ldots, n\}$, such that S and S' are equivalent except that the frames in S' may contain further shorthands; and vice versa, for every $S' \in [\![\mathcal{S}_i]\!]$ there exists $S \in [\![\mathcal{S}]\!]$ such that S' is equivalent to S except for shorthands.*

Proof. It is quite straightforward to observe that all states that we reach by analysis steps are equivalent modulo the augmentation with shorthands: the intruder learns only terms that could be obtained with access to destructors anyway, and none of the transactions puts a constraint on the intruder since in the worst case the decryption fails and the intruder just does not learn anything from it.

For termination, we define a measure (a, b) for symbolic states $\mathcal{S}$ as a lexicographical ordering of the following two well-founded components a and b:

- a is the total number of $\star$ marks and $+$ marks in the FLICs.
- b is the total number of $\star$ marks in the FLICs.

Consider going from a symbolic state $\mathcal{S}$ to $\mathcal{S}'$ with a destructor oracle transaction according to our strategy. We show that on the transition from $\mathcal{S}$ to $\mathcal{S}'$ the measure can only decrease. In an intermediate state of the symbolic execution, when we evaluate the try-catch, we split each possibility into two further cases (the one where the try succeeds, and where it fails), but from the snd steps only one possibility survives—the intruder observes from the outcome whether the destructor works or not. Thus the number of possibilities can only remain the same or decrease from $\mathcal{S}$ to $\mathcal{S}'$. (We have a decrease if in some FLICs the decryption works and in others not, because then each $\mathcal{S}'$ is reduced either to those that worked or those that did not.) Any instantiations of intruder variables that happen are neutral for the measure, because intruder variables in received messages are already marked $\checkmark$, and thus also the instantiation is marked $\checkmark$. The only changes in the measures are from updating the mark of the term under analysis and the marking of the newly received terms (i.e., the result of the analysis and the decryption key that is repeated by the oracle).

We now distinguish the two cases whether $\mathcal{S}'$ represents a successful decryption or failure (w.r.t. the destructor oracle that brings us from $\mathcal{S}$ to $\mathcal{S}'$).

In the first case, if the destructor fails, then in every FLIC where l maps to a term marked $\star$, we replace it with $+$ (others we leave alone). This does not change the a measure, but reduces the b measure by at least one (since there was at least one $\star$-marked term we have addressed).

In the second case, if the destructor is successful, let us consider decryption again. In every FLIC where the label l maps to $c(k', t_1, \dots, t_n)$ marked $\star$, recall that the strategy marks the newly received $l' \mapsto t_i$ with the same mark as the respective subterm t_i in l; in turn the term $c(k', t_1, \dots, t_n)$ with all its subterms gets marked $\checkmark$ (and similarly in a transparency rule). This reduces the a measure by at least one: even if $l' \mapsto t_i$ now contains several $\star$ or $+$ marks, these marks were counted in the previous marking of $l \mapsto c(k', t_1, \dots, t_n)$, which is now marked with $\checkmark$ for c and the subterms, so the mark $\star$ that c bore is not counted anymore. If there are any FLICs where l is mapped to a term marked $+$ or $\checkmark$, we do not necessarily have a reduction, but new l'-terms can only contain $\star$ and $+$ marks that are removed from l. Since there is always at least one $\star$-marked term in $\mathcal{S}$ to apply a destructor oracle, the a measure is strictly reduced from $\mathcal{S}$ to $\mathcal{S}'$.

The measure is well-founded and thus proves there is no infinite chain of analysis steps, and since the branching is also finite (because applying a transaction leads to finitely many successor states), it thus follows by Kőnig's lemma that for every state $\mathcal{S}$, we obtain a finite number of analyzed states $\mathcal{S}_1, \dots, \mathcal{S}_n$ with the destructor oracle strategy. □

The lemma below says that after analysis, every recipe using destructors is equivalent to a destructor-free recipe. This is the point of introducing shorthands where labels map to messages coming out of the analysis: the shorthands abbreviate destructor applications that the intruder would do if we gave them direct access to destructors.

Lemma A.3.8. *Let $\mathcal{S}$ be a normal analyzed state, $S \in [\![\mathcal{S}]\!]$ and r be any recipe over the domain of S. Then there is a destructor-free recipe r' such that $struct(r) \approx struct(r')$ in every frame $struct$ of S.*

Proof. Note that this proof works on a ground state S which does not contain intruder variables anymore (but still privacy variables). Thus, the FLICs are now frames that contain just incoming messages. We also formulate this only for decryption, transparency is in all cases very similar.

We have to show how to replace any subterm $r_d = d(r_1, r_2)$ of r with a destructor-free equivalent recipe. We can also w.l.o.g. assume that r_1 and r_2 are destructor-free (by starting with an inner-most occurrence of a destructor). Thus r_2 is either a label or a composed recipe:

1. Case $r_2 = c(r_1', \dots, r_n')$ for some public function c. If c is not a constructor corresponding to destructor d, then we can already replace r_d with $\mathtt{ff}$ and are done. Otherwise r_d means the intruder applies a destructor to a term they constructed themselves. We distinguish three subcases:

(a) If r_d does not yield $\mathtt{ff}$ in any frame, then the result of the destructor must be the ith subterm (for some $i \in \{1, \dots, n\}$) of r_2 in every frame, i.e., $\mathit{struct}(r_d) \approx \mathit{struct}(r'_i)$ for every FLIC *struct*, and we can thus replace r_d with r'_i.

(b) If r_d yields $\mathtt{ff}$ in all frames, i.e. $\mathit{struct}(r_d) \approx \mathtt{ff}$ in every frame *struct*, we can just replace r_d with $\mathtt{ff}$.

(c) If r_d yields $\mathtt{ff}$ in some frame struct_1 and does not yield $\mathtt{ff}$ in another frame struct_2, it means that comparing r_d with $\mathtt{ff}$ is an intruder experiment that distinguishes the frames. We show that this contradicts the fact that $\mathcal{S}$ is analyzed and normal. The only reason that struct_1 and struct_2 give different results is that the encryption and decryption key do not match in struct_1 but do match in struct_2. Recall that in a decryption rule with decryption key k and encryption key k', we require that either $k = k'$ or $k \approx f(k')$ or $k' \approx f(k)$ for some public function f. If $k = k'$, then comparing r_1 with r'_1 is an experiment that distinguishes the frames, which contradicts that $\mathcal{S}$ is normal. Otherwise, we only prove the case $k \approx f(k')$, the other case is analogous. In struct_2, r_1 and r'_1 correspond to k and k', respectively. Thus, comparing r_1 with $f(r'_1)$ is also an experiment that distinguishes the frames. If f is a constructor, this directly contradicts that $\mathcal{S}$ is normal. If f is a destructor, we now show that this has already been analyzed, i.e., there must be a label l' that is a shorthand for $f(r'_1)$ and thus this contradicts that $\mathcal{S}$ is normal (because then the intruder has already compared r_1 with l'). If r'_1 is a label, then directly the analysis rule $f(r'_1)$ must have been applied; if $r'_1 = c(r''_1, \dots, r''_n)$ and since f is unary, c is transparent, i.e., it is directly equivalent to one of r''_i. Thus the experiment to compare r_1 with r''_i already distinguishes the frames and that must have been done already since $\mathcal{S}$ is normal and these recipes are destructor-free. Thus, in all cases this contradicts that $\mathcal{S}$ is normal.

2. Case $r_2 = l$ for a label l. We distinguish two subcases:

(a) Case l maps to a term t in at least one of the frames such that t was at some point marked $\star$, i.e., t is a term for which a destructor exists and the respective destructor rule has been tried for l by the analysis strategy. (The other cases being that the t in every frame is marked $\checkmark$, because it has no destructor or originated from the intruder.) The state resulting from the application of the respective destructor oracle has the property that the destructor either succeeded in all frames or failed in all frames. In the case of failure, we can simply replace r_d by $\mathtt{ff}$ and are done. In the case of success, there are labels holding the result of the destructor, say, l_1 for decryption result and l_2 repeating the decryption key if it is a decryption rule. (For the case of transparency the proof is similar.) One may wonder if comparing r_1 with l_2 could distinguish the frames. This would contradict that $\mathcal{S}$ is normal because r_1 and l_2 have no destructors. Thus, $\mathit{struct}(r_1) \approx \mathit{struct}(l_2)$ in every frame *struct*, and thus $\mathit{struct}(r_d) \approx \mathit{struct}(l_1)$ and we can replace r_d with l_1.

(b) Case l maps in all frames to terms that have been marked $\checkmark$ throughout. If they are all terms that have no destructor, then we can of course directly replace r_d with $\mathtt{ff}$. Otherwise, in at least one frame *struct*, l maps to a term $c(s_1, \dots, s_m)$ which was composed by the intruder, i.e., there are destructor-free recipes $r'_1, \dots, r'_m$ that produce s_i in *struct*, thus $\mathit{struct}(l) = \mathit{struct}(c(r'_1, \dots, r'_m))$. As these recipes are all destructor-free, this is an experiment that must work in all frames (otherwise $\mathcal{S}$ is not normal). Thus, we can first replace $r_d =$

$d(r_1, c(r'_1, \ldots, r'_m))$ which then can be reduced to a destructor-free recipe following Case 1 of this proof. □

Definition 3.3.3 for normal symbolic states only considers pairs of recipes that we find with our compose-checks (i.e., without destructors). After analysis, using the lemmas above we show that the intruder still cannot distinguish any possibilities even with recipes that may contain destructors. In fact, all steps that distinguish the possibilities must have been done either with compose-checks after executing a destructor oracle, or during the execution of a destructor oracle (because the intruder knows whether decryption succeeded).

Lemma A.3.9. *Let $\mathcal{S}$ be an analyzed state and normal. Then it is also normal w.r.t. arbitrary recipes.*

Proof. Suppose $\mathcal{S}$ is analyzed and normal w.r.t. destructor-free recipes, and let $S \in [\![\mathcal{S}]\!]$. Suppose there are recipes r_1 and r_2 with destructors such that comparing r_1 and r_2 is an experiment that distinguishes *concr* from a $struct_i$ in S, then by Lemma A.3.8, there exist equivalent destructor-free r'_1 and r'_2 that thus also distinguish *concr* and $struct_i$ and thus S (thus $\mathcal{S}$) is not normal w.r.t. destructor-free recipes. □

Finally, we conclude our results for the decision procedure with the proof that our representation with symbolic states is correct even in the presence of destructors. The argument is that whatever the intruder can learn by applying destructors themselves, they can learn the same with destructor oracles.

Theorem 3.4.2 (Procedure correctness). *Given a protocol specification for (α, β)-privacy, a bound on the number of transitions and an algebraic theory allowed by Definition 3.4.1, our decision procedure is sound, complete and terminating.*

Proof. This is essentially lifting Proposition A.2.1 to the case where the intruder has access to destructors (except private extractors, of course). A problem is however that the states that our lifting produces include shorthands, i.e., the terms obtained from the destructor oracles. The construction ensures that such shorthands are indeed just shorthands in the sense that each corresponds to a recipe with destructor (that gives the same term in each FLIC as the shorthand). We can thus regard a state with shorthands as an equivalent representation of the state without shorthands.

Let now $\mathcal{S}$ be a symbolic state that is analyzed and normal w.r.t. destructor-free recipes. By Lemma A.3.9, it is also normal w.r.t. arbitrary recipes. In the model where destructors are private, by Proposition A.2.1, we have for transaction P that $[\![\mathcal{S}_{\Longrightarrow_P}]\!] = [\![\mathcal{S}]\!]_{\longrightarrow_P}$, i.e., what is reachable on the symbolic level is equivalent to what is reachable on the ground level using P. We now show how to arrive at the same result for the case where the intruder can access destructors (except private extractors). Consider first the recipes for messages that the intruder may send during this transaction. These recipes can only use labels that already occur in $\mathcal{S}$—whatever messages the process sends out in response is not available to the intruder when sending. Given a ground state $S \in [\![\mathcal{S}]\!]$ and some recipes with destructors that the intruder sends during this transition, they are equivalent to destructor-free recipes due to Lemma A.3.8. Thus, $[\![\mathcal{S}]\!]_{\longrightarrow_P}$ is the same when allowing destructors in recipes the intruder sends for the messages that P receives.

Observe that the symbolic states $\mathcal{S}_{\Longrightarrow_P}$ that are reached from $\mathcal{S}$ with P are not yet analyzed and only normalized w.r.t. destructor-free experiments. By applying the destructor oracle strategy to every symbolic state in $\mathcal{S}_{\Longrightarrow_P}$, we obtain finitely many analyzed states $\mathcal{S}_1, \ldots, \mathcal{S}_n$ such that $[\![\mathcal{S}_{\Longrightarrow_P}]\!] = \bigcup_{i=1}^{n} [\![\mathcal{S}_i]\!]$ by Theorem 3.4.1. By Lemma A.3.9 these symbolic states in $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are also normal w.r.t. recipes with destructors.

Thus, starting at a normal analyzed symbolic state $\mathcal{S}$ and given a transaction P, our procedure computes a finite set of normal analyzed symbolic states that represent exactly those states that can be reached on the ground level with P from any state represented by $\mathcal{S}$. Thus, by repeatedly applying this procedure, we obtain a correct finite representation of all states reachable from $\mathcal{S}$ after a given number of transactions. □

A.4 Typing

We start by showing that Definition 4.1.5 maintains the invariant that a given recipe produces messages of the same type in every possibility.

Lemma 4.1.1. *Let S be a reachable state in a protocol satisfying Definition 4.1.5, $struct_1, \dots, struct_n$ be the frames in S and r be a recipe over the domain of the $struct_i$. Then $\Gamma(struct_1(r)) = \cdots = \Gamma(struct_n(r))$.*

Proof. Initially, the property holds because the intruder has not observed any message yet. Then whenever a transaction is receiving, the message is determined by the intruder and thus, if the intruder before the transaction knows the type of every message in their knowledge, then they know the types of the messages the transaction receives. They also know the type of every other variable in the transaction, because privacy variables are chosen from homogeneous domains, the type of messages in the memory cells never changes (only the content can), and the result of a destructor application has the type of a subterm of the input (if it does not fail anyway). Since destructor applications occur before any cell read or conditional statement and behave as 0 in case of failures, if any destructor fails then the entire transaction behaves as 0, i.e., it terminates immediately. Thus the intruder can determine the type of every message sent in a given execution path. Moreover, the intruder can observe how many messages the process sends and rule out all those execution paths that are not compatible with that. By Definition 4.1.5, the remaining execution paths, being not statically distinguishable, must have the same type for corresponding messages for any given input messages from the intruder. Thus, the intruder may not know which of the remaining execution paths is the case, but they still know which types the respective messages have, so also after the transaction the intruder knows the type of every message in their knowledge. □

We continue with the proof that using pattern matching is correct w.r.t. destructor applications. This proof refers to the semantics for ground states, while our later proofs will work directly on symbolic states.

Lemma 4.1.2. *A protocol satisfying Definition 4.1.5 and its transformation to use pattern matching according to Definition 4.1.7 yield the same set of reachable ground states (up to logical equivalence of the contained formulas α and β).*

Proof. The ground semantics in Chapter 2 of $\mathsf{try}\ X := d(k,t)\ \mathsf{in}\ P\ \mathsf{catch}\ Q$ is syntactic sugar for a conditional $\mathsf{if}\ \phi\ \mathsf{then}\ P[X \mapsto d(k,t)]\ \mathsf{else\ catch}\ Q$ where $Q = 0$ for the typing result and ϕ is a formula expressing that the destructor application is successful. By construction, if ϕ is true, then $d(k,t)$ yields the respective subterm of t.

Since in this work, $Q = 0$ and try is only allowed before cell reads and conditional statements (i.e., before branching), then a sequence of try's can be written as a single if condition ϕ (the conjunction of the conditions of the individual try's) and that can again be split $\phi = \phi_1 \land \phi_2$ into conditions ϕ_1 on the structure (that the transformed process handles as a pattern) and a condition on the values ϕ_2.

In the transformed specification, if the pattern is satisfied, then the pattern variables are bound to the corresponding subterms of t as the destructor terms $d(k,t)$ mentioned above. This also leads to the same possibilities in both models: in the original process, each possibility splits into two, namely whether ϕ is satisfied or not. In the transformed specification, if ϕ_1 holds there is also a split into two on whether ϕ_2 holds or not. Otherwise, if $\neg\phi_1$ holds, there is no split (we arrive at 0 for sure). Now in each model, the intruder knows the typing of the messages and thus whether ϕ_1 holds. Thus, if ϕ_1 holds, the intruder in the original model can simplify the conditions ϕ and $\neg\phi$ to ϕ_2 and $\neg\phi_2$, respectively, yielding exactly the same conditions as in the new model. Conversely, if $\neg\phi_1$ holds, then the intruder in the original model can rule out the ϕ case and then the process just goes to 0, exactly as in the new model. $\square$

A.4.1 Well-typedness of the constraint solving

We now prove that the lazy intruder rules only return well-typed solutions when used on a type-flaw resistant protocol. First, we consider a single rule application. The main rule that matters is **Unification** and in that case we are sure that the unifier is well-typed by the type-flaw resistance assumption.

Lemma A.4.1. *Let Spec be a type-flaw resistant protocol, $\mathcal{A}$ be a FLIC such that $terms(\mathcal{A}) \subseteq SMP(patterns(Spec))$, ρ be a choice of recipes such that $dom(\rho) \cap rvars(\mathcal{A}) = \emptyset$, σ be a well-typed substitution such that $dom(\sigma) \cap vars(\mathcal{A}) = \emptyset$, and $(\rho', \mathcal{A}', \sigma')$ be such that $(\rho, \mathcal{A}, \sigma) \rightsquigarrow (\rho', \mathcal{A}', \sigma')$. Then σ' is well-typed.*

Proof. To show that the constraint solving always makes well-typed instantiations of intruder and privacy variables, we proceed by distinguishing which lazy intruder rule has been applied.

Unification: Then $\mathcal{A} = \mathcal{A}_1.{-l} \mapsto s.\mathcal{A}_2.{+R} \mapsto t.\mathcal{A}_3$, $\rho' = [R \mapsto l]\rho$ and $\sigma' = mgu(\sigma \wedge s \doteq t)$. We have that $s, t \in SMP(patterns(Spec)) \setminus \mathcal{V}$. Since *Spec* is type-flaw resistant and s and t are unifiable, $\Gamma(s) = \Gamma(t)$. Thus, σ' is well-typed.

Guessing: Then $\mathcal{A} = \mathcal{A}_1.{+R} \mapsto x.\mathcal{A}_2$, $\rho' = [R \mapsto c]\rho$ and $\sigma' = mgu(\sigma \wedge x \doteq c)$, for some $c \in dom(x)$. The guess c is a constant in the domain of the privacy variable x so $\Gamma(x) = \Gamma(c)$. Thus, σ' is well-typed.

Composition or **Repetition**: Then $\sigma' = \sigma$, i.e., no intruder or privacy variables are instantiated. Thus, σ' is well-typed.

Note that since σ' is well-typed and $SMP(patterns(Spec))$ is closed under well-typed instantiations, then $terms(\mathcal{A}') \subseteq SMP(patterns(Spec))$. $\square$

The previous lemma considers a single transition from the lazy intruder rules. We then have that every lazy intruder result is well-typed because every transition preserves well-typedness.

Theorem 4.2.1 (Lazy intruder well-typedness). *Let Spec be a type-flaw resistant protocol, $\mathcal{A}$ be a simple FLIC such that $terms(\mathcal{A}) \subseteq SMP(patterns(Spec))$ and let σ be a well-typed substitution. Then every $\rho \in LI(\mathcal{A}, \sigma)$ is well-typed w.r.t. $\mathcal{A}$.*

Proof. By induction and using Lemma A.4.1, for every σ' such that $(\varepsilon, \sigma(\mathcal{A}), \sigma) \rightsquigarrow^* (\rho, \mathcal{A}', \sigma')$ and $\mathcal{A}'$ is simple, we have that σ' is well-typed. For every $+R \mapsto X \in \mathcal{A}$, we have $\sigma'(\mathcal{A}'(\rho(R))) = \sigma'(X)$, so $\Gamma(\mathcal{A}'(\rho(R))) = \Gamma(X)$. Moreover, $\rho(\mathcal{A})(\rho(R))$ and $\mathcal{A}'(\rho(R))$ are unifiable, because they only differ in the variables introduced by applying ρ to $\mathcal{A}$. Since there are infinitely many variables of each type, then without loss of generality the fresh intruder variables introduced by ρ can be taken of the appropriate types such that $\Gamma(X) = \Gamma(\rho(\mathcal{A})(\rho(R)))$. Thus, ρ is well-typed w.r.t. $\mathcal{A}$. $\square$

A.4.2 Well-typedness of the state transitions

As an intermediate result, we show that, given a set of FLICs with the same domain and constraints, solving the constraints to send a message pattern in one FLIC is equivalent to solving the constraints in any other FLIC. We start by reasoning about a single rule application. The interesting case is again **Unification**. Here we use the fact that every pattern in a message received is linear, and thus if a label maps in one FLIC to a message that unifies with the pattern, then any other message of the same type (including what the label maps to in other FLICs) also unifies.

Lemma A.4.2. *Let Spec be a type-flaw resistant protocol and $\mathcal{A}_1, \dots, \mathcal{A}_n$ be FLICs such that:*

- *$dom(\mathcal{A}_1) = \cdots = dom(\mathcal{A}_n)$ and for every label, the messages in the different FLICs have the same type.*
- *The messages sent in each FLIC are equal.*
- *For every $i \in \{1, \dots, n\}$, $terms(\mathcal{A}_i) \subseteq SMP(patterns(Spec))$ and for every $+R \mapsto t \in \mathcal{A}_i$, t is linear, does not contain constants and the intruder variables in $fv(t)$ do not occur in any other message sent.*

Then a lazy intruder rule is applicable in $\mathcal{A}_1$ iff that rule is applicable in every $\mathcal{A}_i$.

Proof. Let us consider the first non-simple constraints, say it is to send a message t. First, we assume that **Unification** is applicable in $\mathcal{A}_1$. Then it means that t can be unified with another message observed earlier, i.e., there is a label l that maps to a message unifying with t. Since t contains only fresh variables and no constants, then t can be unified with any message of the same type. Since the label l maps to messages of the same type in every FLIC, then l is a solution in every FLIC so **Unification** is applicable in the same way in every FLIC. Note that the variables in t that are substituted do not make any other constraints non-simple, since these variables do not occur in any other message sent.

Second, we assume that **Composition** is applicable. Then it means that t is composed with a public function at the top-level. The intruder can produce t with a composed recipe, using the same function at the top-level and subrecipes for the arguments, and this is applicable in every FLIC.

Since every message to send is linear and contains only fresh intruder variables, the rules of **Guessing** and **Repetition** are not applicable. Moreover, after one rule application, the updated FLICs still form a set of FLICs with identical messages to send. This means that **Guessing** and **Repetition** will never be applicable when solving the constraints. □

We extend the lemma above from one lazy intruder rule application to lazy intruder results (i.e., the solutions returned after solving all the constraints) and obtain well-typedness for the receive transitions.

Lemma A.4.3. *Given a type-flaw resistant protocol, the transitions for receiving message patterns always perform well-typed instantiations.*

Proof. First, we consider the case that the intruder makes some choice of recipes computed with the lazy intruder. The transition is:

$$\begin{aligned}&(\alpha_0, \beta_0, \{(\mathsf{rcv}(t).P_i, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \mathit{Checked})\\ &\Rightarrow \rho((\alpha_0, \beta_0, \{(P_i, \phi_i, \mathcal{A}_i.{+R} \mapsto X, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \dots, n\}\}, \mathit{Checked}))\end{aligned}$$

where $t \notin \mathcal{V}_{intruder}$, R is a fresh recipe variable, X is a fresh intruder variable, $\rho \in LI(\mathcal{A}_1.{+R} \mapsto X, [X \mapsto t])$. By Theorem 4.2.1, ρ is well-typed w.r.t. $\mathcal{A}_1.{+R} \mapsto X$, and thus also w.r.t. $\mathcal{A}_1$. By induction using Lemma A.4.2, the lazy intruder gives the same results in every FLIC. Therefore ρ is actually well-typed w.r.t. $\mathcal{A}_i.{+R} \mapsto X$, and thus $\mathcal{A}_i$, for every $i \in \{1, \dots, n\}$.

Next, we consider the case that the intruder sends a message that does not match the pattern. Then there is no instantiation of variables. □

Next we show that our rules for pattern matching in symbolic states are a correct representation of pattern matching in ground states. This is an update of the correctness result in Proposition A.2.1, when adding the new pattern-matching construct.

Lemma 4.2.1. *Given a type-flaw resistant specification, then the set of reachable states in the symbolic semantics represents exactly the reachable states of the ground semantics.*

Proof. We use the fact that this is already proved for all previous constructs in Appendix A.2 and just show it for the newly added rules for receiving with pattern matching (found in Definitions 4.1.6 and 4.2.2).

Given a symbolic state $\mathcal{S}$ with possibilities $\{(\mathsf{rcv}(t).P_i, \phi_i, \mathcal{A}_i, \mathcal{X}_i, \alpha_i, \delta_i) \mid i \in \{1, \dots, n\}\}$ such that the pattern-matching receive rules are applicable, i.e., $t \notin \mathcal{V}_{intruder}$ is a linear pattern with fresh variables and no constants.

For the positive case (i.e., satisfying the pattern) for a type-flaw resistant protocol, it follows from Lemma A.4.3 that all $\mathcal{A}_i$ have the same set ρ of solutions for producing t, i.e., for $\mathcal{A}_i.{+R} \mapsto t$.

Completeness (all reachable ground states are represented on the symbolic level): The new ground rule for pattern-matching receive (Definition 4.1.6) is applicable for an arbitrary recipe r in every state ground S represented by $\mathcal{S}$. If r produces an instance of t, then by the correctness of the lazy intruder, the choice of r is represented by one of the ρ's that solve $\mathcal{A}_i.{+R} \mapsto t$. The resulting ground state S' is thus covered by the symbolic state that uses the positive rule with ρ.

If r does not produce an instance of t, then we go directly to 0 process in all possibilities, and this is covered by the second rule of the symbolic level, since in this case the pattern cannot be a variable.

Soundness (only reachable ground states are represented on the symbolic level): If ρ is a solution in $\mathcal{A}_i.{+R} \mapsto t$, then all ground states represented by $\rho(\mathcal{S})$ allow for applying the pattern rule with an instance r of $\rho(R)$. Moreover, if the second rule is applicable, then there is a ground recipe r that produces a message which is not an instance of t, thus the transition that makes all processes 0 is also possible on the ground level. □

Before proving, for type-flaw resistant protocols, the equivalence between analysis transitions and destructor oracles, we show an intermediate result. Whenever a label maps to a composed term, then in every FLIC the label maps to a composed term with the same top-level function. This will be useful to make sure that if a destructor oracle can be applied, then also the transition for analysis can be applied.

Lemma A.4.4. *Let $\mathcal{S}$ be a normal symbolic state with FLICs $\mathcal{A}_1, \dots, \mathcal{A}_n$ and $l \in dom(\mathcal{S})$ such that $-l \mapsto c(t_1^1, \dots, t_1^m) \in \mathcal{A}_1$, where $m > 0$. Then for every $i \in \{1, \dots, n\}$, we have $-l \mapsto c(t_i^1, \dots, t_i^m) \in \mathcal{A}_i$ for some terms t_i^j.*

Proof. Assume that in some FLIC $\mathcal{A}_j$ $(j \neq 1)$ we have $-l \mapsto X$, where $X \in \mathcal{V}_{intruder}$. Since $\mathcal{S}$ is normal, the experiment (l, R) must have been done already, i.e., $(l, R) \in$ *Checked*, where ${+R} \mapsto X \in \mathcal{A}_j$. We will show that this contradicts the assumption that $\mathcal{S}$ is well-formed. (All states in the symbolic execution are well-formed by construction.) Let

$\sigma_i = mgu(\mathcal{A}_i(l) \doteq \mathcal{A}_i(R))$ for i in $\{1,\dots,n\}$. Since $(l,R) \in \mathit{Checked}$, either for every $i \in \{1,\dots,n\}$, $\mathit{isPriv}(\sigma_i)$ and $\alpha_0 \wedge \beta_0 \wedge \phi_i \models \sigma_i$, or for every $i \in \{1,\dots,n\}$, $LI(\mathcal{A}_i,\sigma_i) = \emptyset$ or ($\mathit{isPriv}(\sigma_i)$ and $\alpha_0 \wedge \beta_0 \wedge \phi_i \models \neg\sigma_i$).

However, we have not $\mathit{isPriv}(\sigma_1)$. We also have that $\sigma_j = \varepsilon$, so $LI(\mathcal{A}_j,\sigma_j) = \{\varepsilon\} \neq \emptyset$ and $\alpha_0 \wedge \beta_0 \wedge \phi_j \not\models \neg\sigma_j$. This contradicts well-formedness, so we conclude that in every FLIC, the label does not map to an intruder variable. Recall that, since the messages sent in different branches have the same types, every label maps to messages of the same type. Moreover, since $m > 0$, it cannot be the type of a privacy variable. Therefore the message mapped by l has the same constructor in every FLIC. □

The lemma below says that the analysis transitions of Definition 4.2.3 are equivalent to destructor oracles. The main difference is that destructor oracles are applied to more general patterns based on the destructor rules, while the analysis transitions only consider instances of these oracles with messages in the FLICs. However, we can use the lemma above together with Requirement 6 in Definition 4.1.5 to make sure that if a message can be analyzed in one FLIC, then all corresponding messages in the other FLICs can also be analyzed.

Lemma A.4.5. *Given a type-flaw resistant protocol,* $\Longrightarrow = \Longrightarrow^{\bullet}$.

Proof. Let $\mathcal{S}, \mathcal{S}'$ be reachable symbolic states in a type-flaw resistant protocol. For executing regular transactions, the same transitions are possible in both relations. The only thing we have to show is that destructor oracles and analysis transitions are equivalent. Let $\mathcal{A}_1,\dots,\mathcal{A}_n$ be the FLICs in the state $\mathcal{S}$.

First, we assume that $\mathcal{S} \Longrightarrow \mathcal{S}'$, where some destructor oracle is executed. Then it means that $\mathcal{S}$ is normal and there exist a label $l \in \mathit{dom}(\mathcal{S})$ and a public destructor $d \in \Sigma_{pub}$ such that l can be analyzed with d, i.e., $-l \mapsto c(k', t_1, \dots, t_m)$ in some FLIC, where c occurs in the rewrite rule for d. By Lemma A.4.4, l also maps in all the other FLICs to composed messages with the same constructor c, i.e., for every $i \in \{1,\dots,n\}$, we have $-l \mapsto c(k_i', t_i^1, \dots, t_i^m) \in \mathcal{A}_i$. By Definition 4.1.5, for every $i \in \{1,\dots,n\}$, we have $d(k_i, c(k_i', t_i^1, \dots, t_i^m)) \to t_i^j$ as an instance of the rewrite rule for d, for some term k_i and some $j \in \{1,\dots,m\}$. Executing the destructor oracle specialized with label l means that for each FLIC $\mathcal{A}_i$, the process is now $\mathsf{rcv}(Y).\mathsf{if}\ Y \doteq k_i\ \mathsf{then}\ \mathsf{snd}(t_i^j).\mathsf{snd}(k_i)$, which is exactly what we get from the corresponding analysis transition. Thus $\mathcal{S} \Longrightarrow^{\bullet} \mathcal{S}'$.

Second, we assume that $\mathcal{S} \Longrightarrow^{\bullet} \mathcal{S}'$, where some analysis transition is taken. Then it means that $\mathcal{S}$ is normal and there exist a label $l \in \mathit{dom}(\mathcal{S})$ and a public destructor $d \in \Sigma_{pub}$ such that l can be analyzed with d, i.e., $-l \mapsto c(k', t_1, \dots, t_n)$ in some FLIC, where c occurs in the rewrite rule for d. The corresponding destructor oracle transaction can be executed, leading to the same state. Thus $\mathcal{S} \Longrightarrow \mathcal{S}'$. □

Finally, we conclude with our main theorem for the typing result by combining the intermediate results: every part of the procedure only does well-typed instantiations, so it is enough only consider well-typed traces.

Theorem 4.2.2 (Typing result). *Given a type-flaw resistant protocol, it is correct to restrict the intruder model to well-typed recipes/messages for verifying privacy.*

Proof. The only way that variables are instantiated during the transitions is by applying some lazy intruder result. For every transition, we ensure that all messages in the FLICs are in the set of sub-message patterns of the protocol. By Definition 4.1.4, all the constraints occurring during the symbolic execution are well-typed, and thus by Theorem 4.2.1, the lazy intruder only performs well-typed instantiations. In a reachable state, all constraints are simple, i.e., all the messages sent are pairwise distinct intruder variables. Since the intruder can produce an unbounded number of messages of each type, then they can instantiate the intruder variables in a well-typed way. □

A.5 Compositionality

A.5.1 Compositionality on the frame level

We start with our proofs for the compositionality on the frame level. In Definition 3.4.4, we introduced a notion of *shorthand* where a label is equivalent to some recipe. This was used in the context of our analysis strategy for the decision procedure of Chapter 3, where we add shorthands to frames as labels mapping to terms resulting from decryption. In this section, we redefine a notion of shorthands where a label is not mapping directly to a term, but rather is assigned a recipe. The reason is that for the proofs below, we need to keep track of which concrete recipes were used in the shorthands, and not only which terms are produced.

Definition A.5.1. *A* shorthand $i : l \leftarrow r$ *consists of a label l associated with a recipe r and marked with i. We extend the notion of recipes so that they can use labels from shorthands: a recipe containing label l from a shorthand $l \leftarrow r$ produces the same message as the recipe where l is replaced with r.*

Let F be a ground frame. A frame F' is an extension *of F* with shorthands *iff $dom(F) \subseteq dom(F')$ and for every label $l \in dom(F') \setminus dom(F)$ such that $F' = F_1.i : l \leftarrow r.F_2$, we have $F_1(r) \in GSMP_i$ (this requires that every label in r is in $dom(F_1)$) and r is i-homogeneous.*

We extend the notion of comparable frames as well: two frames are comparable iff they have the same domain, the same shorthands and for every label, the frames agree on its mark.

The notion of shorthands does not impact the static equivalence between frames.

Lemma A.5.1. *Let F_1, F_2 be ground frames and let F_1', F_2' be extensions of the respective frames with the same shorthands. Then $F_1 \sim F_2$ iff $F_1' \sim F_2'$.*

Proof. Case $F_1 \not\sim F_2$: Then there exists a witness (r_1, r_2) that distinguishes the frames, which is then also a witness that distinguishes F_1' and F_2'.

Case $F_1' \not\sim F_2'$: Then there exists a witness (r_1, r_2) that distinguishes the frames. Any label l that comes from a shorthand $l \leftarrow r$ can be replaced with r while preserving the fact that the pair of recipes if a witness. Then there exists a witness that distinguishes F_1 and F_2. □

Given two leakage-free frames, a destructor-free recipe producing a message from one protocol can either be made homogeneous or we may find a homogeneous witness against static equivalence. The proof argument is that shared terms are either public or secret, so we can replace non-homogeneous labels with equivalent homogeneous recipes (with a public term seen as a recipe or with a recipe for a declassified secret), unless we find a witness along the way.

Lemma A.5.2. *Let F_1, F_2 be leakage-free comparable frames, $i \in \{1, 2, \star\}$ and r be a destructor-free recipe such that $F_1(r), F_2(r) \in GSMP_i$. Then at least one of the following is true:*

1. *There exist a destructor-free i-homogeneous recipe r' and frames F_1', F_2' extensions of F_1, F_2 with the same shorthands such that $F_1(r) \approx F_1'(r')$ and $F_2(r) \approx F_2'(r')$.*

2. *There exist $j \in \{1,2\}$ and a j-homogeneous witness against static equivalence of F_1 and F_2.*

Proof. Assume that $i = 1$ (the cases that $i = 2$ or $i = \star$ are handled similarly). We define the weight of recipe r as $w(r) = \#\{l \mid l \text{ occurs in } r \text{ and is marked with } 2\}$. The weights form a well-founded ordering. If $w(r) = 0$, then we already have a destructor-free 1-homogeneous recipe. Otherwise, let l be a label in r marked with 2, $t_1 = F_1(l)$ and $t_2 = F_2(l)$. Since l is marked with 2, $t_1, t_2 \in \mathit{GSMP}_2$. Since r is destructor-free, t_1 is a subterm of $F_1(r)$ and thus $t_1 \in \mathcal{T}_{pub} \cup \mathit{Secrets}$.

- Case $t_1 \in \mathcal{T}_{pub}$: If $t_1 = t_2$, then we continue with the recipe $r[l \mapsto t_1]$ and the weight decreases by 1. Otherwise, (l, t_1) is a witness and thus Item 2 is true, since this pair of recipes is 2-homogeneous.
- Case $t_1 \in \mathit{Secrets}$: Since F_1 is leakage-free and $F_1|_2(l) = t_1$, we have $t_1 \notin \mathit{Secrets} \setminus \mathit{declassified}(F_1)$ and thus $t_1 \in \mathit{declassified}(F_1)$. By definition of $\mathit{declassified}(F_1)$, there exists a recipe r' such that $F_1(r') \approx t_1$ and all labels in r' are marked with $\star$. If (l, r') is a witness, then Item 2 is true since this pair of recipes is 2-homogeneous. Otherwise, we add the shorthand $\star : l' \leftarrow r'$ in both frames. Then we have $F_1(l') \approx t_1$ and $F_2(l') \approx t_2$, so we continue with the recipe $r[l \mapsto l']$ and the weight decreases by 1.

By repeating these steps, we can successively replace all labels in the recipe r with equivalent labels (possibly using shorthands) so that the recipe becomes 1-homogeneous, or we may find a 2-homogeneous witness during some step. □

Given a leakage-free frame, a destructor-free recipe producing a message from one protocol can be made homogeneous. This is a special case of the previous lemma.

Lemma A.5.3. *Let F be a leakage-free frame, $i \in \{1, 2, \star\}$ and r be a destructor-free recipe such that $F(r) \in \mathit{GSMP}_i$. Then there exists an i-homogeneous recipe r' such that $F(r) \approx F(r')$.*

Proof. By Lemma A.5.2, when considering twice the same frame, there exist an extension F' of F with shorthands and a destructor-free i-homogeneous recipe r'' such that $F(r) \approx F'(r'')$. Then we obtain the recipe r' from r'' by replacing every label from a shorthand with the recipe it is assigned to (and removing the shorthands preserves i-homogeneity, since the recipe in a shorthand is homogeneous w.r.t. the mark of the shorthand). □

The lemma above can be extended to arbitrary recipes. We show this by considering an inner-most destructor and replacing it with either $\mathtt{ff}$ (if the decryption fails) or a simpler recipe (if the decryption succeeds); simpler in the sense that either it is directly homogeneous or it contains one less destructor.

Lemma 5.4.1. *Let F be a leakage-free frame, $i \in \{1, 2, \star\}$ and r be a recipe such that $F(r) \in \mathit{GSMP}_i$. Then there exists an i-homogeneous recipe r' such that $F(r) \approx F(r')$.*

Proof. We define the weight of recipe r as the number of destructors that occur in r. The weights form a well-founded ordering. If $w(r) = 0$, i.e., r is destructor-free, then the lemma holds by Lemma A.5.3. Otherwise, we consider an inner-most destructor in r. We have $r = r_0[d(r_k, r_t)]$ where r_k and r_t are destructor-free and $r_0[\cdot]$ is a recipe context. (A recipe context is a recipe with a hole, and the hole is filled when the context is applied to a recipe.) We have the rewrite rule $d(k, c(k', X_1, \ldots, X_n)) \to X_j$, where $j \in \{1, \ldots, n\}$. If $F(d(r_k, r_t)) \approx \mathtt{ff}$, then we continue with the recipe $r_0[\mathtt{ff}]$ and the weight decreases by 1. Otherwise:

- Case $r_t = c(r'_k, r'_1, \dots, r'_n)$: The decryption succeeds, so we continue with the recipe $r_0[r'_j]$ and the weight decreases by 1.
- Case $r_t = l \in dom(F)$: The decryption succeeds. By definition of *GSMP*, we have $F(r_k), F(r_t) \in GSMP_i$. By Lemma A.5.3, there exist i-homogeneous recipes r'_k and r'_t such that $F(r_k) \approx F(r'_k)$ and $F(r_t) \approx F(r'_t)$. Then we add the shorthand $i : l' \leftarrow d(r'_k, r'_t)$. Note that $d(r'_k, r'_t)$ is i-homogeneous, so the shorthand is well-defined. Then we continue with the recipe $r_0[l']$ and the weight decreases by 1.

By repeating these steps, we can successively replace all destructor applications in the recipe with equivalent labels (possibly using shorthands) so that we can get an extension F' of F with shorthands and a destructor-free recipe r_1 such that $F(r) \approx F'(r_1)$. By Lemma A.5.3, there exists a recipe r_2 such that $F'(r_1) \approx F'(r_2)$ and r_2 is i-homogeneous. Then we obtain the recipe r' from r_2 by replacing every label from a shorthand with the recipe it is assigned to. □

Given two leakage-free frames, we obtain a homogeneous witness against static equivalence by reducing an arbitrary witness step by step towards a homogeneous one. The replacement of subrecipes is similar to the previous lemmas: either we use public terms as recipes or we use recipes for declassified secrets.

Lemma A.5.4. *Let F_1, F_2 be leakage-free comparable frames. If there exists a destructor-free witness against static equivalence of F_1 and F_2, then there exist $i \in \{1, 2\}$ and an i-homogeneous witness.*

Proof. For recipe r_1, r_2, let $labels(r_1, r_2)$ denote the set of labels occurring in r_1 or r_2 that are *not* marked with $\star$. Given a pair of recipes (r_1, r_2), we define the weight of the pair as:

$$w_{F_1,F_2}(r_1, r_2) = \begin{cases} (\#labels(r_1, r_2), size(F_1(r_1))) \\ \quad \text{if } F_1(r_1) \approx F_1(r_2) \text{ and } F_2(r_1) \not\approx F_2(r_2) \\ (\#labels(r_1, r_2), size(F_2(r_1))) \\ \quad \text{if } F_1(r_1) \not\approx F_1(r_2) \text{ and } F_2(r_1) \approx F_2(r_2) \end{cases}$$

and $w_{F_1,F_2}(r_1, r_2)$ is undefined otherwise (i.e., the weight is only defined for witnesses). The weights with the lexicographic order form a well-founded ordering.

Assume that there exists a destructor-free witness (r_1, r_2) against static equivalence of F_1 and F_2. If both recipes are composed, then they must have the same constructor at the top-level, because they produce the same term in exactly one frame: $r_1 = c(r_1^1, \dots, r_1^n)$ and $r_2 = c(r_2^1, \dots, r_2^n)$. Then at least one of the (r_1^i, r_2^i) must already be a witness. We can thus move to a smaller witness: if $labels(r_1^i, r_2^i) \subsetneq labels(r_1, r_2)$, then the first component of the weight decreases; otherwise, the second component of the weight decreases (the recipes produce a strict subterm of the original message). This is well-founded: at some point, at least one of the recipes is no longer composed and thus a label. Therefore there exists a witness (l, r) such that l is a label and r is destructor-free. Assume that:

- $F_1(l) \approx F_1(r)$ and $F_2(l) \not\approx F_2(r)$.
- l is marked with 1 and r contains a label l' marked with 2.

Note that these assumptions are without loss of generality, because we can just rename the frames and the case that l is marked with 2 or $\star$ is handled in a similar way as below. Since l is marked with 1, $F_1(l), F_2(l) \in GSMP_1$. Let $t = F_1(l')$. Since l' is marked with 2, $t \in GSMP_2$. Since r is destructor-free, t is a subterm of $F_1(l)$ and thus $t \in \mathcal{T}_{pub} \cup Secrets$.

- Case $t \in \mathcal{T}_{pub}$: If (l', t) is a witness, then it is a 2-homogeneous one and the lemma holds. Otherwise, we continue with the witness $(l, r[l' \mapsto t])$. This witness is smaller because we are removing a label not marked with $\star$, so the first component of the weight decreases.

- Case $t \in \mathit{Secrets}$: Since F_1 is leakage-free and $F_1|_2(l') = t$, we have $t \notin \mathit{Secrets} \setminus \mathit{declassified}(F_1)$ and thus $t \in \mathit{declassified}(F_1)$. By definition of $\mathit{declassified}(F_1)$, there exists a recipe r' such that $F_1(r') \approx t$ and all labels in r' are marked with $\star$. If (l', r') is a witness, then it is a 2-homogeneous one and the lemma holds. Otherwise, we add the shorthand $\star : l'' \leftarrow r'$ in both frames. Then we have, $F_1(l') \approx F_1(l'')$ and $F_2(l') \approx F_2(l'')$, so continue with the witness $(l, r[l' \mapsto l''])$. This witness is smaller because we are removing a label not marked with $\star$ and the label we introduce is a shorthand marked with $\star$, so the first component of the weight decreases.

By repeating these steps, we can successively replace all labels in the recipe r with equivalent labels (possibly using shorthands) so that the witness becomes 1-homogeneous, or we may find a 2-homogeneous witness during some step. $\square$

The lemma above can be extended to arbitrary recipes. Again, the proof looks at an inner-most destructor and we consider the different cases of the witness: it could be that the encryption/decryption key terms already distinguish the frames, or we can distinguish with another recipe to produce the key, or we can replace the destructor with the subterm it produces (when decryption succeeds in both frames).

Lemma A.5.5. *Let F_1, F_2 be leakage-free comparable frames. If there exists a witness against static equivalence of F_1 and F_2, then there exist $i \in \{1, 2\}$ and an i-homogeneous witness.*

Proof. Assume that there exists a witness (r_1, r_2) against static equivalence of F_1 and F_2. Given a pair of recipes (r_1, r_2), we define the weight of the pair as:

$$w_{F_1,F_2}(r_1, r_2) = \begin{cases} (nd, \mathit{size}(F_1(r_1))) \\ \quad \text{if } F_1(r_1) \approx F_1(r_2) \text{ and } F_2(r_1) \not\approx F_2(r_2) \\ (nd, \mathit{size}(F_2(r_1))) \\ \quad \text{if } F_1(r_1) \not\approx F_1(r_2) \text{ and } F_2(r_1) \approx F_2(r_2) \end{cases}$$

where nd is the number of destructors in r_1 and r_2, and $w_{F_1,F_2}(r_1, r_2)$ is undefined otherwise (i.e., the weight is only defined for witnesses). The weights with the lexicographic order form a well-founded ordering.

If the witness is destructor-free, then the lemma holds by Lemma A.5.4. Otherwise, we assume w.l.o.g. that r_1 is not destructor-free. We consider an inner-most destructor in r_1. We have $r_1 = r[d(r_k, r_t)]$ where r_k and r_t are destructor-free. We have the rewrite rule $d(k, c(k', X_1, \ldots, X_n)) \to X_j$, where $j \in \{1, \ldots, n\}$, and either $k = k'$ or there exists some public function f such that $k \approx f(k')$ or $k' \approx f(k)$. If $F_1(d(r_k, r_t)) \approx \mathtt{ff}$ and $F_2(d(r_k, r_t)) \approx \mathtt{ff}$, then $(r[\mathtt{ff}], r_2)$ is a witness and the first component of the weight decreases. Otherwise:

- Case $r_t = c(r'_k, r'_1, \ldots, r'_n)$:
  - Case $k = k'$: If (r_k, r'_k) is a witness, then by Lemma A.5.4 there exist $i \in \{1, 2\}$ and an i-homogeneous witness so the lemma holds. Otherwise, the decryption succeeds in both frames so we continue with the witness $(r[r'_j], r_2)$ and the first component of the weight decreases.

  - Case $k \approx f(k')$: If $(r_k, f(r'_k))$ is a witness, then we continue with that simpler witness (the first component of the weight may only decrease, and even if it stays the same, this witness produces a strict subterm of the original witness, so the second component of the weight decreases). Otherwise, the decryption succeeds in both frames so we continue with the witness $(r[r'_j], r_2)$ and the second component of the weight decreases.
  - Case $k' \approx f(k)$: Similar to the previous case.

- Case $r_t = l \in dom(F_1)$: Let i be the mark of l, or let $i = 1$ if l is marked with $\star$. By Lemma A.5.2, we may find $j \in \{1, 2\}$ and a j-homogeneous witness so that the lemma holds, or there exist a destructor-free i-homogeneous recipe r'_k and frames F'_1, F'_2 extensions of F_1, F_2 with the same shorthands such that $F_1(r_k) \approx F'_1(r'_k)$ and $F_2(r_k) \approx F'_2(r'_k)$. If $(d(r_k, l), \mathtt{ff})$ is a witness, then $(d(r'_k, l), \mathtt{ff})$ is also a witness and since it is i-homogeneous, the lemma holds. Otherwise, the decryption succeeds in both frames and we add the shorthand $i : l' \leftarrow d(r'_k, l)$. Then we continue with the witness $(r[l'], r_2)$ and the second component of the weight decreases.

By repeating these steps, we can successively replace all destructor applications in the witness with equivalent labels (possibly using shorthands) so that the witness becomes homogeneous, or we may find a homogeneous witness during some step. □

Our theorem for frames is simply the contraposition of the lemma above.

Theorem 5.4.1. *Let F_1, F_2 be leakage-free comparable frames. If for every $i \in \{1, 2\}$, $F_1|_i \sim F_2|_i$, then $F_1 \sim F_2$.*

Proof. We proceed by contraposition. Assume that $F_1 \not\sim F_2$. By Lemma A.5.5, there exist $i \in \{1, 2\}$ and an i-homogeneous witness (r_1, r_2) against static equivalence of F_1 and F_2. Then (r_1, r_2) is a witness against static equivalence of $F_1|_i$ and $F_2|_i$. □

A.5.2 Compositionality on the state level

We now define an equivalence relation between states and show some intermediate results about equivalent states. In particular, if two states are equivalent, then their reachable states are also equivalent.

Definition A.5.2. *Let $S = (\alpha, \gamma, \mathcal{P}, \_, \mathit{flag})$ and $S' = (\alpha', \gamma', \mathcal{P}', \_, \mathit{flag}')$, where the respective sets of possibilities are $\mathcal{P} = \{(R_1, \phi_1, \mathit{struct}_1, \delta_1), \ldots, (R_n, \phi_n, \mathit{struct}_n, \delta_n)\}$ and $\mathcal{P}' = \{(R'_1, \phi'_1, \mathit{struct}'_1, \delta'_1), \ldots, (R'_n, \phi'_n, \mathit{struct}'_n, \delta'_n)\}$.*
S and S' are equivalent *iff*

- *$\alpha \equiv \alpha'$;*
- *$\gamma \equiv \gamma'$;*
- *$\mathit{flag} = \mathit{flag}'$; and*
- *For every $i \in \{1, \ldots, n\}$, $\beta(S) \wedge \phi_i \equiv_\Sigma \beta(S') \wedge \phi'_i$ and there exists a substitution σ_i such that $\sigma_i(R_i) = \sigma_i(R'_i)$ (with the same thread IDs), $\beta(S) \wedge \phi_i \models_\Sigma \sigma_i$, $\sigma_i(\mathit{struct}_i) = \sigma_i(\mathit{struct}'_i)$ and $\sigma_i(\delta_i) = \sigma_i(\delta'_i)$.*

Lemma A.5.6. *The relation in Definition A.5.2 is an equivalence relation.*

Proof. Straightforward, it suffices to expand the definitions. □

Two equivalent states represent the same intruder knowledge.

Lemma A.5.7. *Let S, S' be equivalent states. Then $\beta(S) \equiv_\Sigma \beta(S')$.*

Proof. We only show $\beta(S) \models_\Sigma \beta(S')$, the other direction follows by symmetry. Let $\mathcal{I} \models_\Sigma \beta(S)$. Then there exists a possibility with condition ϕ in S such that $\mathcal{I} \models_\Sigma \phi$. Since S and S' are equivalent, there exists a possibility with condition ϕ' in S' such that $\beta(S) \wedge \phi \equiv_\Sigma \beta(S') \wedge \phi'$. Then $\mathcal{I} \models_\Sigma \beta(S') \wedge \phi'$ and thus $\mathcal{I} \models_\Sigma \beta(S')$. □

Equivalence between states is preserved by the symbolic execution.

Lemma A.5.8. *Let S_1, S_1' be equivalent states. Then for every state S_2 such that $S_1 \to S_2$, there exists a state S_2' such that $S_1' \to S_2'$ and S_2 and S_2' are equivalent.*

Proof. By definition of equivalence, S_1 and S_1' have the same payload α, truth formula γ and assertion flag *flag*. Moreover, for every possibility $(R_i, \phi_i, \mathit{struct}_i, \delta_i)$ in S_1, there is a corresponding possibility $(R_i', \phi_i', \mathit{struct}_i', \delta_i')$ in S_1' and a substitution σ_i such that $\sigma_i(R_i) = \sigma_i(R_i')$ (with the same thread IDs), $\beta(S_1) \wedge \phi_i \models_\Sigma \sigma_i$, $\sigma_i(\mathit{struct}_i) = \sigma_i(\mathit{struct}_i')$ and $\sigma_i(\delta_i) = \sigma_i(\delta_i')$. Let S_2 be a state such that $S_1 \to S_2$. We proceed by case distinction of the rule applied in the transition $S_1 \to S_2$. Note that for every step in a process in S_1, there is a corresponding step in a process in S_1', where only the instantiations of privacy variables may differ. Thus there exists a state S_2' such that $S_1' \to S_2'$. We show that every transition preserves the equivalence.

- **Choice**: The truth formula is extended in the same way in S_2 and S_2'.
- **Receive**: For the message received $\mathsf{rcv}(t)$, the intruder uses a recipe r, then the variables bound in the linear term t are substituted in the rest of the process according to what the recipe r produces in the respective frame. The messages $\mathit{struct}_i(r)$ and $\mathit{struct}_i'(r)$ may be different, therefore the bound variables may be substituted differently in the rest of the process, but we have that $\sigma_i(\mathit{struct}_i(r)) = \sigma_i(\mathit{struct}_i'(r))$. Thus the messages occurring in the rest of the process are still related by the substitutions σ_i in S_2, S_2'.
- **Let**: For a step $\mathsf{let}\ X = t$ in S_1, there is a corresponding step $\mathsf{let}\ X = t'$ in S_1' and a substitution σ_i such that $\sigma_i(t) = \sigma_i(t')$. The variable X may be substituted differently but the messages in the rest of the process are still related by the σ_i in S_2, S_2'.
- **Cell read**: For the possibility with memory δ_i that is doing a cell read in S_1, there is a corresponding possibility with memory δ_i' that is also doing a cell read and a substitution σ_i such that $\sigma_i(\delta_i) = \sigma_i(\delta_i')$. The cell read introduces conditional statements with the memory updates and the variable bound to the cell read is substituted in each branch according to the respective memory update. The values read from δ_i and δ_i' may be different, therefore the bound variable may be substituted differently in the rest of the process, but the messages occurring in the rest of the process are still related by the σ_i in S_2, S_2'.
- **Cell write**: For a step $\mathsf{cell}(s) := t$ in S_1, there is a corresponding step $\mathsf{cell}(s') := t'$ in S_1' and a substitution σ_i such that $\sigma_i(s) = \sigma_i(s')$ and $\sigma_i(t) = \sigma_i(t')$. The sequences of memory updates are still related by the σ_i in S_2, S_2'.
- **Conditional**: For a statement branching on condition ψ in the possibility with condition ϕ_i in S_1, there is a corresponding branching on condition ψ' in a possibility

with condition ϕ'_i in S'_1 and a substitution σ_i such that $\psi \wedge \sigma_i \equiv_\Sigma \psi' \wedge \sigma_i$ and $\beta(S_1) \wedge \phi_i \equiv_\Sigma \beta(S'_1) \wedge \phi'_i$. The possibility can be split in two in each state. Then we have that $\gamma \models \phi_i \wedge \psi$ iff $\gamma \models \phi'_i \wedge \psi'$, and $\gamma \models \psi_i \wedge \neg\psi$ iff $\gamma \models \phi'_i \wedge \neg\psi'$.

- **Release**: For a step $\star\ \psi$ in the possibility with condition ϕ_i in S_1 such that $\gamma \models \phi_i$, there is the same step $\star\ \psi$ in a possibility with condition ϕ'_i in S'_1 such that $\gamma \models \phi'_i$. Thus the new payload in S_2 and S'_2 is the same. Note that we have the same formula ψ in both cases because the variables in ψ can only be privacy variables chosen in previous transactions following Definition 5.3.4. If we did not make this restriction, then the formulas released would be related by the σ_i but the payload might be different and thus we would not obtain equivalent states.

- **Assert**: For a step $\mathsf{assert}(\psi)$ in the possibility with condition ϕ_i in S_1 such that $\gamma \models \phi_i$, there is a corresponding step $\mathsf{assert}(\psi')$ in a possibility with condition ϕ'_i in S'_1 such that $\gamma \models \phi'_i$ and a substitution σ_i such that $\psi \wedge \sigma_i \equiv_\Sigma \psi' \wedge \sigma_i$. Moreover, $\gamma \models \sigma_i$ because $\beta(S_1) \wedge \phi \models_\Sigma \sigma_i$, γ is consistent with $\beta(S_1)$ and $\gamma \models \phi$. Thus $\gamma \models \psi$ iff $\gamma \models \psi'$ and the assertion flag is the same in S_2 and S'_2.

- **Stop** or **Milestone**: For every possibility that starts with stop in S_1, there is a corresponding possibility that also starts with stop in S'_1 (and similarly for 0 instead of stop). Then corresponding possibilities are discarded in the same way in the transitions to S_2, S'_2.

- **Send**: For every step $\mathsf{snd}(t)$ in S_1, there is a corresponding step $\mathsf{snd}(t')$ in S' and a substitution σ_i such that $\sigma_i(t) = \sigma_i(t')$. Thus the frames in S_2 and S'_2 are still related by the σ_i in S_2, S'_2. Moreover, for every possibility in S_1 that is discarded (because it starts with stop or 0) in the transition to S_2, there is a possibility in S'_1 that is also discarded in the transition to S'_2.

- **Eliminate**: For a possibility with condition ϕ_i in S_1 such that $\beta(S_1) \models_\Sigma \neg\phi_i$, there is a possibility with condition ϕ'_i in S'_1 such that $\beta(S'_1) \models_\Sigma \neg\phi'_i$.

- **Next**: The transition to the next transaction in the sequence can be done for both S_2 and S'_2. □

Finally, we show our results for the compositionality on the state level. Given a state S, we denote with *concr*(S) the concrete frame in that state. We first show that we can assume homogeneous recipes w.l.o.g. The proof argument relies on the result for a single frame that for every recipe producing a message from one protocol, there exists a homogeneous recipe producing the same message. Technically, changing to homogeneous recipes does not lead to the same exact states but to equivalent states.

Lemma A.5.9. *Let $(P_1; \dots; P_n, \gamma, \rho)$ be an attack trace and $S_0, \dots, S_n$ be the milestones such that for every $j \in \{1, \dots, n\}$, executing P_j, starting from S_{j-1}, leads to S_j (following the truth γ and using the recipes in ρ).*

Then there exists ρ' such that $(P_1; \dots; P_n, \gamma, \rho')$ is an attack trace leading to milestones $S'_0, \dots, S'_n$ where S_j and S'_j are equivalent (for $j \in \{0, \dots, n\}$) and for every step $i : \mathsf{rcv}(t)$ during the execution of the transactions, where $i \in \{1, 2, \star\}$, the recipe given by ρ' to produce t is i-homogeneous.

Proof. Let $j \in \{1, \dots, n\}$ and $concr = concr(S_{j-1})$. Consider a step $i : \mathsf{rcv}(t)$ $(i \in \{1, 2, \star\})$ in the transaction P_j, let r be the recipe given by ρ for this message received and $t' = concr(r)$. We have $t' \in GSMP_i$, because we consider only well-typed instantiations and by definition $GSMP_i$ contains all the well-typed instances of t. By Lemma 5.4.1, there exists an i-homogeneous recipe r' such that $\mathit{concr}(r') \approx t'$. Thus we can consider ρ' that is the same as ρ except that it uses r' instead of r for this message received.

We know that the intruder is able to use an i-homogeneous recipe to produce the same message. However, we now need to argue that using r' instead of r is correct w.r.t. the symbolic execution. Indeed, the two recipes produce the same message in *concr*, i.e., the concrete messages are equal, but changing recipes can make a difference for the instantiation of privacy variables. Let $(\_, \gamma, \mathcal{P}, \_, \_) = S_{j-1}$ where $\mathcal{P} = \{\underline{(0; R_1, \phi_1, \mathit{struct}_1, \delta_1)}, \ldots, (0; R_m, \phi_m, \mathit{struct}_m, \delta_m)\}$. The underlined possibility is what really is the case, i.e., $\gamma \models \phi_1$ and $\mathit{concr} = \gamma(\mathit{struct}_1)$. For every $k \in \{1, \ldots, m\}$, the intruder knows that $\beta(S) \wedge \phi_k \models_\Sigma \mathit{concr} \sim \mathit{struct}_k$. Since r and r' produce the same in *concr*, the intruder also knows that $\beta(S) \wedge \phi_k \models \mathit{struct}_k(r) \doteq \mathit{struct}_k(r')$. Let $\sigma_k = \mathit{mgu}(\mathit{struct}_k(r) \doteq \mathit{struct}_k(r'))$. The state S'_{j-1} obtained from S_{j-1} by applying the $\sigma_1, \ldots, \sigma_m$ to the respective possibilities is equivalent to S_{j-1}. Thus by Lemma A.5.8, the states reached when using r' instead of r leads are equivalent.

This argument holds for every message received in the transaction P_j and for every transaction in the trace. □

Every possibility that the intruder has not ruled out corresponds to some concrete execution.

Lemma A.5.10. *Let $S = (\alpha, \gamma, \mathcal{P}, \rho, \_)$ be a state reached with trace $(P_1; \ldots; P_n, \gamma, \rho)$. Then for every possibility $(\_, \phi, \mathit{struct}, \_) \in \mathcal{P}$ and interpretation $\gamma' \models \beta(S) \wedge \phi$, we have that $(P_1; \ldots; P_n, \gamma', \rho)$ is a trace and it leads to a state S' such that $\mathit{concr}(S') = \gamma'(\mathit{struct})$.*

Proof. We consider the transitions, in the symbolic execution, that depend on the truth formula.

- Release: The formulas released in the possibility underlined by γ' are added to the payload α' of S' while the releases in other possibilities are ignored. Thus the payload in S and S' may be different, but the possibilities contain the same frames in both states.

- Stop, Send or Milestone: Since the possibility with ϕ, struct remains in S, i.e., it was not ruled out by the intruder, its process was stopping, sending or reaching a milestone at the same time as the possibility underlined by γ, so the same transitions can be taken when considering the truth γ'.

- Assert: The assertions in the possibility underlined by γ' are checked while the assertions in other possibilities are ignored. Thus the assertion flag in S and S' may be different, but the possibilities contain the same frames in both states.

- Eliminate: The intruder knowledge in the intermediate states to reach S and S' may be different, so the eliminated possibilities may be different, but the intruder did not rule out the possibility with ϕ, struct in S, so also in S' there is a possibility with the same frame.

The other transitions in the symbolic execution do not depend on the truth formula, thus when executing the transactions following truth γ', there is a possibility with frame *struct* considered by the intruder and it is actually the underlined one, so $\mathit{concr}(S') = \gamma'(\mathit{struct})$. Note that by well-formedness of states, we have $\alpha' \models \bigvee_{\gamma_0 \in \Gamma_0} \gamma_0$, where α' is the payload in S'. Thus even if γ and γ' do not agree on the relations in Σ_0, there exists some $\gamma_0 \in \Gamma_0$ such that $\gamma' \models \gamma_0$ and thus $(P_1; \ldots; P_n, \gamma', \rho)$ is a trace and S' is a reachable state. □

For our main compositionality result, we start with an arbitrary attack trace on the composed protocol and show that there exists an attack trace on one of the components.

Theorem 5.4.2. *If for every $i \in \{1, 2\}$, $Spec|_i$ has no attack, then $Spec$ has no attack.*

Proof. We proceed by contraposition. We assume that the composed protocol *Spec* has an attack and we show that one of the projections to a component, i.e., $Spec|_1$ or $Spec|_2$, also has an attack. Let $(P_1; \ldots; P_n, \gamma, \rho)$ be an attack trace. The symbolic execution of the transactions, following recipes in ρ and truth formula γ, defines the milestones $S_0, \ldots, S_n$ such that for every $j \in \{1, \ldots, n\}$, executing P_j, starting from S_{j-1}, leads to S_j. Recall that by definition of attack traces, S_n is an attack state but all the milestones $S_0, \ldots, S_{n-1}$ do not have any attack.

By Lemma A.5.9, we can assume w.l.o.g. that in the attack trace, only homogeneous recipes are used. Now, consider the symbolic execution of a conditional statement $\mathsf{if}\ \phi\ \mathsf{then}\ P\ \mathsf{else}\ Q$ occurring in one of the $P_1, \ldots, P_{n-1}$ such that:

- The conditional statement is marked with a protocol-specific index, i.e., 1 or 2 but not $\star$.
- The underlined possibility, i.e., what really happened concretely, went into branch Q.

Since the branching is protocol-specific, we know that $Q = \mathsf{stop}$ (following Definition 5.3.4), and the execution of stop is observable by the intruder. Moreover, a transaction cannot have any effects (e.g., writing to memory, sending a message or releasing a formula) before reaching stop because this step is in the center part of processes. Since reaching this stop step did not result in a privacy violation (as we are considering transactions before the last one), we can simplify the trace by removing the execution of this transaction and the successive ones that the stop would have filtered out. Thus, w.l.o.g. we can assume that whenever there is a protocol-specific branching occurring in the attack trace before the last transaction, the underlined possibility went into the first branch.

We know that the last transaction is the one that introduces an attack. Due to procedure call expansion, this transaction may contain steps from both protocols, i.e., some steps marked with 1 and other steps marked with 2: it may happen that the left part of the process has been specified by, say, protocol 1, then during the procedure call expansion a center process specified by protocol 2 was inserted. However, given a transaction, we can always uniquely identify the protocol that specifies the center process in that transaction (because procedure calls can only happen in the left part of the process). Let i be the index of the protocol that specifies the center process in the final transaction P_n. We show the existence of an attack by looking at the execution of the transactions $P_1|_i; \ldots; P_n|_i$.

Let $j \in \{1, \ldots, n\}$. We go over the different steps that can occur in P_j, and argue that for the projection $P_j|_i$, the steps are either present or have been soundly abstracted.

- Non-deterministic choice, release or nil process: The step remains in $P_j|_i$ because it is always present in every projection.
- Receive: If the message received is marked with i or $\star$, then it remains in $P_j|_i$ and we know that the intruder uses an i-homogeneous recipe. Otherwise, since the role containing $P_j|_i$ is closed, the variables bound in that message are not used in the projection so the step can be skipped.
- Let statement or cell read: If the step is marked with i or $\star$, then it remains in $P_j|_i$. Otherwise, since the role containing $P_j|_i$ is closed, the variable bound by this step is not used in the projection so the step can be skipped.

- Cell write: If the cell write is marked with i or $\star$, then it remains in $P_j|_i$. Otherwise, the memory cell does not occur in $\mathit{Spec}|_i$ so the step can be skipped.

- Conditional statement: If the branching is marked with i or $\star$, then it remains in $P_j|_i$. Otherwise, we know that the underlined possibility went into the first branch (as justified above), so executing $P_j|_i$ (where this branching does not occur, if $j < n$) preserves the attack.

- Assertion: If the assertion is marked with i or $\star$, then it remains in $P_j|_i$. Otherwise, the assertion was not present in the underlined possibility or was true (since the transactions before the last one do not lead to any attack) so the step can be skipped.

- Stop: This step is never reached in the transactions before the last one (as justified above). For the last transaction P_n, the stop remains in $P_n|_i$ because it is always present in every projection.

- Send: If the message sent is marked with i or $\star$, then it remains in $P_j|_i$ and the message is added to the frames with the same mark. Otherwise, the step can be skipped since in the projected transactions, the intruder only uses labels marked with i or $\star$.

Let ρ' be the same as ρ except that we remove the recipes corresponding to receive steps skipped when projecting to i.

We now consider different cases of attacks:

- If the flag in S_n is set to true: Then executing the trace $(P_1|_i; \dots; P_n|_i, \gamma, \rho')$ leads to a state with the same assertion that does not hold. Thus $\mathit{Spec}|_i$ has an attack.

- If the flag in S_n is set to false and S_n is not leakage-free: Then there exist $t \in \mathit{Secrets} \setminus \mathit{declassified}(\mathit{concr}(S_n))$, $i' \in \{1, 2\}$ and r such that $\mathit{concr}(S_n)|_{i'}(r) \approx t$. Since it is the last transaction that leads to the attack, we have $i' = i$. Then executing the trace $(P_1|_i; \dots; P_n|_i, \gamma, \rho')$ leads to a state leaking the same secret t. Thus $\mathit{Spec}|_i$ is not leakage-free.

- Otherwise, i.e., the flag in S_n is set to false, S_n is leakage-free and does not satisfy privacy: Let $(\alpha, \gamma, \mathcal{P}, \rho, \mathsf{false}) = S_n$ where the set of possibilities is $\mathcal{P} = \{(\_, \phi_1, \mathit{struct}_1, \_), \dots, (\_, \phi_m, \mathit{struct}_m, \_)\}$. Then we have that $(\alpha, \beta(S_n))$-privacy does not hold, so there exists a model $\mathcal{I}$ such that $\mathcal{I} \models \alpha$ and $\mathcal{I} \not\models_\Sigma \beta(S_n)$. Executing the trace $(P_1|_i; \dots; P_n|_i, \gamma, \rho')$ leads to a state $S' = (\alpha, \gamma, \mathcal{P}', \rho', \mathsf{false})$.
  - If there exists $j \in \{1, \dots, m\}$ such that $\mathcal{I} \models_\Sigma \phi_j$: Then $\mathcal{I} \not\models_\Sigma \mathit{concr}(S_n) \sim \mathit{struct}_j$, so $\mathit{concr}(S_n) \not\sim \mathcal{I}(\mathit{struct}_j)$. Since S_n is leakage-free, we know that concr is leakage-free. However, to apply our results on frames we need to have both frames leakage-free.
    * If $\mathcal{I}(\mathit{struct}_j)$ is leakage-free: Then by Theorem 5.4.1, there exists $i' \in \{1, 2\}$ such that $\mathit{concr}(S_n)|_{i'} \not\sim \mathcal{I}(\mathit{struct}_j)|_{i'}$. Since it is the last transaction that leads to the attack, we have $i' = i$. Moreover, there exists a possibility $(\_, \phi'_j, \mathit{struct}'_j, \_) \in \mathcal{P}'$ such that $\phi_j \models_\Sigma \phi'_j$ and $\mathit{struct}_j|_i = \mathit{struct}'_j$. Note also that $\mathit{concr}(S') = \mathit{concr}(S_n)|_i$. Then $\mathcal{I} \models_\Sigma \alpha \land \phi'$ and $\mathit{concr}(S') \not\sim \mathcal{I}(\mathit{struct}')$, so S' does not satisfy privacy. Thus $\mathit{Spec}|_i$ does not satisfy privacy.
    * Otherwise: Then by Lemma A.5.10, there is a reachable state where the concrete frame is $\mathcal{I}(\mathit{struct}_j)$, i.e., there is a reachable state that leaks a

secret and we can show (as done in a previous case) that one component of the protocol has an attack.

- Otherwise, i.e., $\mathcal{I} \not\models_\Sigma \bigvee_{j=1}^m \phi_j$: The same branching occurs both in the full trace $(P_1; \ldots ; P_n, \gamma, \rho)$ and in the projected trace $(P_1|_i; \ldots ; P_n|_i, \gamma, \rho')$, except for protocol-specific branching that has been abstracted, but in this case the intruder observed in the full trace which branch was taken (as justified earlier). Moreover, if there are branches in the transactions $P_1, \ldots, P_{n-1}$ that are distinguishable before the projection but not after (e.g., due to one branch sending a protocol-specific message), then the intruder can eliminate one of the branches in the projected trace because observing which branch was taken did not violate privacy, so even if the condition depended on privacy variables, then it was allowed by the payload. Thus $\beta(S') \models_\Sigma \bigvee_{j=1}^m \phi_j$. Then $\mathcal{I} \models \alpha$ and $\mathcal{I} \not\models_\Sigma \beta(S')$, so S' does not satisfy privacy and thus $\mathit{Spec}|_i$ does not satisfy privacy. $\square$

Appendix B

Models and details for case studies

We give here the (α, β)-privacy specification of the protocols for the case studies. Below, we present the models with type annotations and after our changes to achieve type-flaw resistance, except for OSK since it is not supported by our typing result. Note that in some of the models below, we write steps of right processes before steps of center processes. This is syntactic sugar to avoid repetition, for instance if we write snd(t).if ϕ then P else Q it means that snd(t) happens in both branches.

Note that in the models given in this chapter, for convenience we make use of syntactic sugar that is not supported by the tool (yet) and we include type annotations (which are also not supported yet). The original models in the untyped model are provided together with the sources of the `noname` tool [39].

B.1 Running example

We extend the running example presented in this thesis to include the appropriate releases allowing the server to reply to a dishonest agent, and we also add formats to achieve type-flaw resistance.

```
Sigma0: public a/0 b/0 i/0 s/0 yes/0 no/0
Sigma: public f1/2 f2/2 df11/1 df12/1 df21/1 df22/1
Types: a:agent b:agent i:agent s:agent yes:decision no:decision
Algebra: df11(f1(X,Y))->X
         df12(f1(X,Y))->Y
         df21(f2(X,Y))->X
         df22(f2(X,Y))->Y

Transaction ReceivePrivateKey:
send inv(pk(i))

Transaction Server:
* x in {a,b,i}.
* y in {yes,no}.
receive M:crypt(pk(agent),f1(nonce,nonce),nonce).
try N:=dcrypt(inv(pk(s)),M) in
try N1:=df11(N) in
try N2:=df12(N) in
new R.
if y=yes then
  send crypt(pk(x),f2(yes,N1),R).
  if x=i then
    * x=i and y=yes
  else
    * not x=i
```

```
else
  send crypt(pk(x),f2(no,N2),R).
  if x=i then
    * x=i and y=no
  else
    * not x=i
```

Then we have that this protocol satisfies Definition 4.1.5. There is no destructor application to remove in the transaction sending the private key of the dishonest agent. However, for the server transaction, we apply Definition 4.1.7 to get the following transaction with pattern matching.

```
Transaction ServerPat:
* x in {a,b,i}.
* y in {yes,no}.
receive crypt(pk(A:agent),f1(N1:nonce,N2:nonce),R':nonce).
if inv(pk(A))=inv(pk(s)) then
  new R:nonce.
  if y=yes then
    send crypt(pk(x),f2(yes,N1),R).
    if x=i then
      * x=i and y=yes
    else
      * not x=i
  else
    send crypt(pk(x),f2(no,N2),R).
    if x=i then
      * x=i and y=no
    else
      * not x=i
```

Thus, we have the following message patterns:

$$\begin{aligned}M = \{&\mathsf{inv}(\mathsf{pk}(\mathsf{i})), x, \mathsf{a}, \mathsf{b}, \mathsf{i}, y, \mathsf{yes}, \mathsf{no}, \mathsf{crypt}(\mathsf{pk}(A), \mathsf{f}_1(N_1, N_2), R'), \mathsf{inv}(\mathsf{pk}(A)), \mathsf{inv}(\mathsf{pk}(\mathsf{s})), R,\\ &\mathsf{crypt}(\mathsf{pk}(x), \mathsf{f}_2(\mathsf{yes}, N_1), R), \mathsf{crypt}(\mathsf{pk}(x), \mathsf{f}_2(\mathsf{no}, N_2), R)\}\end{aligned}$$

with the following types for variables and constants:

$$\begin{aligned}&\Gamma(\mathsf{i}) = \Gamma(x) = \Gamma(\mathsf{a}) = \Gamma(\mathsf{b}) = \Gamma(A) = \mathsf{agent}\ ,\\ &\Gamma(y) = \Gamma(\mathsf{yes}) = \Gamma(\mathsf{no}) = \mathsf{decision}\ ;\\ &\Gamma(N_1) = \Gamma(N_2) = \Gamma(R') = \Gamma(R) = \mathsf{nonce}\ .\end{aligned}$$

The set M is type-flaw resistant, and thus Runex is type-flaw resistant.

B.2 Basic Hash

```
Sigma0: public t1/0 t2/0 t3/0
Sigma: public h/2 ok/0
       private sk/1 extract/1
Types: t1:tag t2:tag t3:tag ok:reply
Algebra: extract(h(sk(X),Y))->X

Transaction Tag:
* x in {t1,t2,t3}.
new N:nonce.
send pair(N,h(sk(x),N))

Transaction Reader:
```

```
receive M:pair(nonce,h(sk(tag),nonce)).
try N:=proj1(M) in
try H:=proj2(M) in
try X:=extract(H) in
if H=h(sk(X),N) then
  send ok
```

Then we have that this protocol satisfies Definition 4.1.5. There is no destructor application to remove in the tag transaction. However, for the reader, we apply Definition 4.1.7 to get the following transaction with pattern matching:

```
Transaction ReaderPat:
receive pair(N:nonce,h(sk(T:tag),N':nonce)).
if N=N' then
  send ok
```

Thus, we have the following message patterns:

$$M = \{x, \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3, N, \mathsf{pair}(N, \mathsf{h}(\mathsf{sk}(x), N)), \mathsf{pair}(N', \mathsf{h}(\mathsf{sk}(X), N'')), N', N'', \mathsf{ok}\}$$

with the following types for variables and constants:

$$\begin{aligned}&\Gamma(x) = \Gamma(\mathsf{t}_1) = \Gamma(\mathsf{t}_2) = \Gamma(\mathsf{t}_3) = \Gamma(X) = \mathsf{tag}\ ,\\ &\Gamma(N) = \Gamma(N') = \Gamma(N'') = \mathsf{nonce}\ ,\\ &\Gamma(\mathsf{ok}) = \mathsf{reply}\ .\end{aligned}$$

The set M is type-flaw resistant, and thus Basic Hash is type-flaw resistant.

B.3 OSK

This protocol is not supported by our typing result. We only give here the untyped models of the two variants where, respectively, no desynchronization and one desynchronization step is tolerated.

```
Sigma0: public t1/0 t2/0 t3/0
Sigma:  public g/2 h/1 ok/0
        private initr/1
        extract/1
Algebra: extract(g(Tag,Key))->Tag
Cells: t[X]:=initr(X)
       r[X]:=initr(X)

Transaction Tag:
* x in {t1,t2,t3}.
State:=t[x].
t[x]:=h(State).
send g(x,State)

Transaction Reader:
receive X.
try Tag:=extract(X) in
State:=r[Tag].
if X=g(Tag,State) then # Accept only if keys match
  r[Tag]:=h(State).
  send ok
else nil # No desynchronization allowed
```

When tolerating one desynchronization step, only the reader transaction changes: instead of going to 0 when the keys do not match, the reader checks if applying one time the hash function makes the key match.

```
Transaction Reader:
receive X.
try Tag:=extract(X) in
State:=r[Tag].
if X=g(Tag,State) then # Accept if the keys match
  r[Tag]:=h(State).
  send ok
else if X=g(Tag,h(State)) then # Allowing one desynchronization step
  r[Tag]:=h(h(State))
  send ok
else nil # All other cases fail
```

B.4 BAC

The model found in [41] contains, in the response transaction, a non-empty catch branch, which is not supported by the typing result. Therefore we change the model by replacing the symmetric decryption with private extractors. Note that the original model is slightly different from the model below: in case the message M received by the tag is not of the form $\mathsf{scrypt}(\mathsf{sk}(\cdot), \cdot, \cdot)$, the original model returns a format error, while the model here does not send anything in case M does not have the right form. However, in the original model, even if the intruder sends a message that does not have the right form message, the tag will respond with a message of type reply. Thus, the intruder knows the types of the messages in their knowledge. Therefore, the intruder also knows, before sending, whether a message matches the pattern, so they would not learn anything by sending a message that is not an encryption of the right form. This is why we consider our changes reasonable.

```
Sigma0: public t1/0 t2/0
Sigma: public ok/0 formatErr/0 fixedR/0
       private sk/1 fresh/0 spent/0 session/2 sfst/1 ssnd/1
               recipient/1 content/1
Types: t1:tag t2:tag ok:reply nonceErr:reply formatErr:reply
       fixedR:nonce fresh:state spent:state
Algebra: sfst(session(X,Y))->X
         ssnd(session(X,Y))->Y
         recipient(scrypt(sk(X),M,R))->X
         content(scrypt(sk(X),M,R))->M
Cells: noncestate[N:nonce]:=fresh

Transaction Challenge:
* x in {t1,t2}.
new N:nonce.
send session(x,N).
send N.
send scrypt(sk(x),N,fixedR)

Transaction Response:
receive Session:session(tag,nonce).
receive M:scrypt(sk(tag),nonce,nonce).
try X:=sfst(Session) in
try N:=ssnd(Session) in
try Y:=recipient(M) in
try NN:=content(M) in
if Y=X then
  State:=noncestate[N].
  if N=NN and State=fresh then
    noncestate[N]:=spent.
    send ok
  else send formatErr
else send formatErr
```

Then we have that this protocol satisfies Definition 4.1.5. There is no destructor application to remove in the challenge transaction. However, for the response transaction, we apply Definition 4.1.7 to get the following transaction with pattern matching:

```
Transaction ResponsePat:
receive session(X:tag,N:nonce).
receive scrypt(sk(Y:tag),NN:nonce,R:nonce).
if Y=X then
  State:=noncestate[N].
  if N=NN and State=fresh then
    noncestate[N]:=spent.
    send ok
  else send formatErr
else send formatErr
```

Thus we have the following message patterns:

$$\begin{aligned}M = \{&x, \mathsf{t}_1, \mathsf{t}_2, N, \mathsf{session}(x, N), \mathsf{scrypt}(\mathsf{sk}(x), N, \mathsf{fixedR}), \mathsf{session}(X, N'),\\ &\mathsf{scrypt}(\mathsf{sk}(Y), NN, R), Y, X, \mathit{State}, N', NN, \mathsf{fresh}, \mathsf{spent}, \mathsf{ok}, \mathsf{formatErr}\}\end{aligned}$$

with the following types for variables and constants:

$$\begin{aligned}&\Gamma(x) = \Gamma(\mathsf{t}_1) = \Gamma(\mathsf{t}_2) = \Gamma(X) = \Gamma(Y) = \mathsf{tag}\ ,\\ &\Gamma(N) = \Gamma(N') = \Gamma(\mathsf{fixedR}) = \Gamma(NN) = \Gamma(R) = \mathsf{nonce}\ ,\\ &\Gamma(\mathit{State}) = \Gamma(\mathsf{fresh}) = \Gamma(\mathsf{spent}) = \mathsf{state}\ ,\\ &\Gamma(\mathsf{ok}) = \Gamma(\mathsf{formatErr}) = \mathsf{reply}\ .\end{aligned}$$

The set M is type-flaw resistant, and thus BAC is type-flaw resistant.

B.5 Private Authentication

Like for BAC, the model found in [41] contains, in the responder transaction, a non-empty catch branch, which is not supported by the typing result. Moreover, the reply sent by the responder is either an encryption or a fresh nonce as decoy. In general, the intruder does not know which is the case so when they observe that such a reply is sent, they do not know a priori its type. Therefore we change the model: first by replacing the asymmetric decryption with private destructors, and second by replacing the fresh nonce as decoy with a fresh encryption. In this formulation, the protocol is still not type-flaw resistant because a reply from the responder can be confused with the message sent by initiator, even though these have different types. Thus, our final change is replacing the pairing function with distinct formats.

```
Sigma0: public a/0 b/0 i/0
Sigma: public f1/2 f2/1 df11/1 df12/1 df2/1
       private recipient/1 sender/1
Types: a:agent b:agent i:agent
Algebra: recipient(crypt(pk(B),f1(A,NA),R))->B
         sender(crypt(pk(B),f1(A,NA),R))->A
         df11(f1(X,Y))->X
         df12(f1(X,Y))->Y
         df2(f2(X))->X
```

```

Transaction ReceivePrivateKey:
send inv(pk(i))

Transaction Initiator:
* xA in {a,b}.
* xB in {a,b,i}.
new NA:nonce,R:nonce.
send crypt(pk(xB),f1(xA,NA),R).
if xB=i then
  * xA=gamma(xA) and xB=gamma(xB)
else
  * xB in {a,b}

Transaction Responder:
* xB in {a,b}.
receive M:crypt(pk(agent),f1(agent,nonce),nonce).
try C:=recipient(M) in
try A:=sender(M) in
new NB:nonce,R:nonce.
if C=xB and A in {a,b,i} then
  send crypt(pk(A),f2(NB),R).
  if A=i then
    * xB=gamma(xB)
else
  new AA:agent.
  send crypt(pk(AA),f2(NB),R).
  if A in {a,b,i} and C in {a,b} then
    * not (C=xB and A=i)
```

Then we have that this protocol satisfies Definition 4.1.5. There is no destructor application to remove in the initiator transaction. However, for the responder transaction, we apply Definition 4.1.7 to get the following version with pattern matching:

```
Transaction ResponderPat:
* xB in {a,b}.
receive crypt(pk(C:agent),f1(A:agent,NA':nonce)),R':nonce).
new NB:nonce,R:nonce.
if C=xB then
  send crypt(pk(A),f2(NB),R).
  if A=i then
    * xB=gamma(xB)
else
  new AA:agent.
  send crypt(pk(AA),f2(NB),R).
  if A in {a,b,i} and C in {a,b} then
    * not (C=xB and A=i)
```

Thus we have the following message patterns:

$$M = \{\mathsf{inv}(\mathsf{pk}(\mathsf{i})), xA, \mathsf{a}, \mathsf{b}, xB, \mathsf{i}, NA, R, \mathsf{crypt}(\mathsf{pk}(xB), \mathsf{f}_1(xA, NA), R), xB', NB, R'', C, A, AA,$$
$$\mathsf{crypt}(\mathsf{pk}(C), \mathsf{f}_1(A, NA'), R'), \mathsf{crypt}(\mathsf{pk}(A), \mathsf{f}_2(NB), R''), \mathsf{crypt}(\mathsf{pk}(AA), \mathsf{f}_2(NB), R'')\}$$

with the following types for variables and constants:

$$\Gamma(\mathsf{i}) = \Gamma(xA) = \Gamma(\mathsf{a}) = \Gamma(\mathsf{b}) = \Gamma(xB) = \Gamma(xB') = \Gamma(C) = \Gamma(A) = \Gamma(AA) = \mathsf{agent}\ ,$$
$$\Gamma(NA) = \Gamma(R) = \Gamma(NA') = \Gamma(R') = \Gamma(NB) = \Gamma(R'') = \mathsf{nonce}\ .$$

The set M is type-flaw resistant, and thus Private Authentication (AF0 variant) is type-flaw resistant.

As is done in [41], we can extend AF0 to include a relation *talk*, where an agent sends a decoy when they do not want to talk to the claimed sender.

```
Sigma0: public a/0 b/0 i/0
        rel talk/2
Sigma: public f1/2 f2/1 df11/1 df12/1 df2/1
       private recipient/1 sender/1
Types: a:agent b:agent i:agent
gamma0: talk: (a,b),(a,i),(b,a)
Algebra: recipient(crypt(pk(B),f1(A,NA),R))->B
         sender(crypt(pk(B),f1(A,NA),R))->A
         df11(f1(X,Y))->X
         df12(f1(X,Y))->Y
         df2(f2(X))->X

Transaction ReceivePrivateKey:
send inv(pk(i))

Transaction Initiator:
* xA in {a,b}.
* xB in {a,b,i}.
if talk(xA,xB) then
  new NA:nonce,R:nonce.
  send crypt(pk(xB),f1(xA,NA),R).
  * talk(xA,xB).
  if xB=i then
    * xA=gamma(xA) and xB=gamma(xB)
  else
    * xB in {a,b}
else
  * not talk(xA,xB)

Transaction Responder:
* xB in {a,b}.
receive M:crypt(pk(agent),f1(agent,nonce),nonce).
try C:=recipient(M) in
try A:=sender(M) in
new NB:nonce,AA:agent,R:nonce.
if C=xB then
  if A=i then
    if talk(xB,A) then
      send crypt(pk(A),f2(NB),R).
      * talk(xB,A) and xB=gamma(xB)
    else
      send crypt(pk(AA),f2(NB),R).
      * not talk(xB,A)
  else
    if A in {a,b} then
      if talk(xB,A) then
        send crypt(pk(A),f2(NB),R)
      else
        send crypt(pk(AA),f2(NB),R)
  else
    send crypt(pk(AA),f2(NB),R)
else
  send crypt(pk(AA),f2(NB),R).
  if A in {a,b,i} and C in {a,b} then
    * not (C=xB and A=i and talk(xB,A))

Transaction ResponderPat:
* xB in {a,b}.
receive crypt(pk(C:agent),f1(A:agent,NA':nonce),R':nonce).
new NB:nonce,AA:agent,R:nonce.
if C=xB then
```

```
  if A=i then
    if talk(xB,A) then
      send crypt(pk(A),f2(NB),R).
      * talk(xB,A) and xB=gamma(xB)
    else
      send crypt(pk(AA),f2(NB),R).
      * not talk(xB,A)
  else
    if A in {a,b} then
      if talk(xB,A) then
        send crypt(pk(A),f2(NB),R)
      else
        send crypt(pk(AA),f2(NB),R)
    else
      send crypt(pk(AA),f2(NB),R)
else
  send crypt(pk(AA),f2(NB),R).
  if A in {a,b,i} and C in {a,b} then
    * not (C=xB and A=i and talk(xB,A))
```

B.6 Results

Table B.1 gives an overview of the results of our tool. In the column "Type-flaw resistant", we report the execution time of the `noname` tool for the models after our reasonable adaptations to achieve type-flaw resistance. The column "Ratio" compares the time before and after those changes. In all cases, we only considered the variants of the protocols that do not have any privacy violation (at least until the bounds verified). Note that to make the running example type-flaw resistant, we have not only added formats but we also introduced a second nonce, so the encrypted message received by the server has to go through more checks, which is why in that example the type-flaw resistant version takes more time to verify. On the other examples we have that either the protocol is already type-flaw resistant and thus the verification time does not change, or the type-flaw resistant version takes the same or less time to verify.

Table B.1: Evaluation of the tool

Protocol	Bound	Result	Untyped	Type-flaw resistant	Ratio
Runex	2	ϟ	0.22s		
Runex (fix attempt)	2	ϟ	0.43s		
Runex (fixed)	2	✔	0.43s	0.53s	0.8
Basic Hash	4	✔	1.96s	1.96s	1
Basic Hash (compromised tag)	2	ϟ	0.14s		
OSK (no desynchronization)	3	ϟ	0.37s		
OSK (1 desynchronization step)	4	ϟ	3.12s		
BAC (different error messages)	3	ϟ	0.18s		
BAC (same error message)	4	✔	1.12s	1.12s	1
BAC (parallel)	4	✔	1.32s		
BAC (sequential)	4	✔	1.51s		
AF0	2	ϟ	2.07s		
AF0 (fixed)	2	✔	5.97s	4.08s	1.46
AF0 (fixed)	3	✔	4min33s	2min38s	1.73
AF	2	✔	10.31s	7.84s	1.32
AF	3	✔	13min19s	8min22s	1.59

✔ = No violation, ϟ = Violation
Machine used: laptop with i7-4720HQ @ 2.60GHz, 8GB RAM, GHC 9.10.1, cvc5 1.1.2

Appendix C

Simplified TLS for composition

In our running example of Fig. 5.1, the lookup procedure is sending requests to a fixed, trusted key server. As a further example, we model below the establishment of a fresh key between some agent and a server, taking inspiration from TLS 1.2 [68].

C.1 Fixed, trusted server

In Fig. C.1, we model a simple version we call TLS0, where the server is always the same trusted server s, and we also assume that agents already know the public key of that server so we do not model certificates. In the models below, the wildcard _ is an unused variable of type nonce. This protocol could for instance be used inside *lookup*, where the agent that makes the request is calling *tls_client* to get a fresh key instead of a long-term shared secret, and inside *Server*, where the server calls *tls_server* to also get the key used for encrypting the later messages.

Procedure $\mathit{tls\_client}(C : \mathsf{agent})$:
$\nu N_C : \mathsf{nonce}, \mathit{PMS} : \mathsf{pms}, R : \mathsf{nonce}.$
$\mathsf{snd}(N_C).$
$\mathsf{snd}(\mathsf{ox}(\mathit{PMS})).$
$\mathsf{snd}(\mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{g}_1(\mathit{PMS}, N_C), R))$
;
$\mathsf{rcv}(N_S : \mathsf{nonce}).$
$\mathsf{rcv}(\mathsf{scrypt}(\mathsf{kdf}(\mathit{PMS}, N_C, N_S), \mathsf{g}_2(N_S), \_)).$
$\mathsf{return}(\mathsf{kdf}(\mathit{PMS}, N_C, N_S))$

Procedure $\mathit{tls\_server}()$:
$\mathsf{rcv}(N_C : \mathsf{nonce}).$
$\mathsf{rcv}(\mathsf{ox}(\mathit{PMS} : \mathsf{pms})).$
$\mathsf{rcv}(\mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{g}_1(\mathit{PMS}, N_C), \_)).$
$\nu N_S : \mathsf{nonce}, R : \mathsf{nonce}.$
$\mathsf{snd}(N_S).$
$\mathsf{snd}(\mathsf{scrypt}(\mathsf{kdf}(\mathit{PMS}, N_C, N_S), \mathsf{g}_2(N_S), R)).$
$\mathsf{return}(\mathsf{kdf}(\mathit{PMS}, N_C, N_S))$

Role $\mathit{Dishonest\_client}$:
$\nu \mathit{PMS} : \mathsf{pms}.$
$\mathsf{snd}(\mathsf{ox}(\mathit{PMS})).$
$\mathsf{snd}(\mathit{PMS})$

Figure C.1: Specification of the TLS0 protocol

Note that in the client procedure, we highlighted the reception of server random N_S, because we need to bind it before returning the value for the fresh key. In the projection, the intruder could freely choose any nonce, e.g., some value that the server does not know, and this does not matter because in the projection the server does not actually need to be executed: as in the previous model of the lookup, the correct public key is directly returned without actual communication with the server.

In this model, we use a private function ox in order to abstract the communication of the pre-master secret PMS: the client sends the abstract message $\mathsf{ox}(PMS)$ separately from the protocol-specific message $\mathsf{crypt}(\mathsf{pk}(\mathsf{s}), \mathsf{g}_1(PMS, N_C), R)$. Then in the *tls_server* procedure, the server receives the abstract message containing some PMS and expects a encrypted message containing the same value. With our additional transaction for dishonest clients, the intruder can simulate the behavior of honest clients: a fresh PMS is generated and the abstract message $\mathsf{ox}(PMS)$ is sent. Since the function is private, PMS is still a secret, for a dishonest client we also have to declassify PMS to give the intruder access to it.[1] This allows us to not highlight the protocol-specific encryption, and keep the projections well-formed (PMS is bound in the projections even though the protocol-specific messages are removed).

The modeling "trick" we do with ox to bind the occurrence of PMS could also be used in the modeling of the initiator and responder roles of NSL: In Fig. 5.1, we highlighted a protocol-specific message sent by the initiator, so that the responder can receive it, binding variables standing for the agent names and a nonce. We could instead introduce a private function $\mathsf{session}$ used in the abstract interface and completely remove the protocol-specific messages from the projections.

Role *Initiator* :
$\star\ x_A \in \mathsf{Honest}.$
$\star\ x_B \in \mathsf{Agent}.$
$PKB := lookup(x_A, x_B)$
;
$\nu N_A : \mathsf{nonce}, R : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{session}(x_A, x_B, N_A)).$
$\mathsf{snd}(\mathsf{crypt}(PKB, \mathsf{f}_1(N_A, x_A), R)).$
$\dots$

Role *Responder* :
$\mathsf{rcv}(\mathsf{session}(A : \mathsf{agent}, B : \mathsf{agent}, N_A : \mathsf{nonce})).$
$\mathsf{rcv}(\mathsf{crypt}(\mathsf{pk}(B), \mathsf{f}_1(N_A, A), \_)).$
$\dots$

Note that this does make a change in the protocol, since the abstract messages with ox and $\mathsf{session}$ are purely for modeling purposes and would never occur in concrete executions of the protocol. However, we can show that this is sound because if there is an attack in the protocol using these abstract messages, then there is also an attack in the original protocol without them.

C.2 Server as a parameter

We further develop in Fig. C.2 the model inspired by TLS to make the server identity a parameter. For the lookup procedure, the argument would be replaced with the concrete name of the key server. Here we do not assume that the client knows the public key of the server, but we assume that they know the public key of a trusted certificate authority so that they can check the certificate sent by the server.

[1] In *tls_client*, we do not check in the process that the client C is honest and thus that declassification of PMS is not needed, because this can be ensured statically by checking the calls to the procedure.

```
Procedure $tls\_client(C : \mathsf{agent}, S : \mathsf{agent})$ :
$\nu N_C : \mathsf{nonce}.$
$\mathsf{snd}(N_C)$
;
$\mathsf{rcv}(\mathsf{g}_0(N_C, \mathsf{sign}(\mathsf{inv}(\mathsf{pk}(\mathsf{ca})), \mathsf{cert}(S, PKS : \mathsf{pk}(\mathsf{agent}))))).$
$\nu PMS : \mathsf{pms}, R : \mathsf{nonce}.$
$\mathsf{snd}(\mathsf{ox}(PMS)).$
$\mathsf{snd}(\mathsf{crypt}(PKS, \mathsf{g}_1(PMS, N_C), R))$
;
$\mathsf{rcv}(N_S : \mathsf{nonce}).$
$\mathsf{rcv}(\mathsf{scrypt}(\mathsf{kdf}(PMS, N_C, N_S), \mathsf{g}_2(N_S), \_)).$
$\mathsf{return}(\mathsf{kdf}(PMS, N_C, N_S))$

Procedure $tls\_server(S : \mathsf{agent})$ :
$\mathsf{rcv}(N_C : \mathsf{nonce}).$
$\mathsf{snd}(\mathsf{g}_0(N_C, \mathsf{sign}(\mathsf{inv}(\mathsf{pk}(\mathsf{ca})), \mathsf{cert}(S, \mathsf{pk}(S))))).$
;
$\mathsf{rcv}(\mathsf{ox}(PMS : \mathsf{pms})).$
$\mathsf{rcv}(\mathsf{crypt}(\mathsf{pk}(S), \mathsf{g}_1(PMS, N_C), \_)).$
$\nu N_S : \mathsf{nonce}, R : \mathsf{nonce}.$
$\mathsf{snd}(N_S).$
$\mathsf{snd}(\mathsf{scrypt}(\mathsf{kdf}(PMS, N_C, N_S), \mathsf{g}_2(N_S), R)).$
$\mathsf{return}(\mathsf{kdf}(PMS, N_C, N_S))$

Role $Dishonest\_client$ :
$\nu PMS : \mathsf{pms}.$
$\mathsf{snd}(\mathsf{ox}(PMS)).$
$\mathsf{snd}(PMS)$
```

Figure C.2: Specification of the simplified TLS protocol

Errata

Corrections made compared to the submitted version of this thesis:

Page	Error	Correction
v	de pågældende privathedsegenskaben	den pågældende privathedsegenskab
vi	udveskle	udveksle
8	fresh r, omit r	fresh R, omit R
9	Well-formed of	Well-formedness of
21	$R' \mapsto X$, $R \mapsto X$, $R = R'$	$R_1 \mapsto X$, $R_2 \mapsto X$, $R_2 = R_1$
25	**Destructor** (1)	**Destructor** (1.1)
27	**Destructor** (2), **Destructor** (3)	**Destructor** (1.2), **Destructor** (2)
33	rule $(d(c(X_1, \ldots, X_n)) \to X_i) \in E$	rules $(d_i(c(X_1, \ldots, X_n)) \to X_i) \in E$
66	$x_B \in \mathsf{Honest}$	$\star\ x_B \in \mathsf{Honest}$
66	$x_A \doteq \gamma(x_A) \land x_B \doteq \gamma(x_B)$	$\star\ x_A \doteq \gamma(x_A) \land x_B \doteq \gamma(x_B)$
68	sequence of has	sequence has
70	starts initially	starts
79	because of	because
97	show with	with