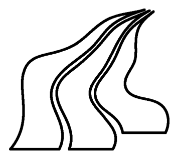
Type Checking Versus Flow Logics

- relations between static analysis methods for cryptographic protocols

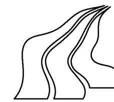
DAT5, Fall 2005.

Lars Hornbæk Jensen
Bjørn Mølgård Vester



Aalborg University
Department of Computer Science

Aalborg University
Department of Computer Science



TITLE:
Type Checking Versus Flow Logics
- relations between static analysis methods for cryptographic protocols

THEME:
Distributed Systems and Semantics

PROJECT PERIOD:
1/2/2005-12/6/2006

PROJECT GROUP:
d622a

GROUP MEMBERS:
Lars Hornbæk Jensen
Bjørn Mølgård Vester

SUPERVISOR:
Hans Hüttel

SYNOPSIS:

In this project we examine the relationship between control flow analysis and type checking. We develop a typed version of the LySA calculus using correspondence assertions, as well as an accompanying type system. We construct an encoding from processes with crypto-points in LySA to Typed LySA, and show that they have equivalent safety properties. Type inference is performed on the encoded process under a series of constraints. Lastly a new control flow analysis is created using correspondence assertions instead of crypto-points.

NUMBER OF COPIES: 6

NUMBER OF PAGES: 73

CONCLUDED: 12/6-2006

This report is a result of the work done in the Dat5 and Dat6 semester of the Cand.Scient study at Department of Computer Science, Aalborg University. This project was done in the research area of Distributed Systems and Semantics and constitutes the complete work towards a master thesis. In the first semester we examined the basics of a control flow analysis, chapter 1 and 2 is partly carried over from that work, and developed a first edition of TYPED LYSA with associated type system, encoding and type inference. In the second semester we completely revised the type system based on new ideas for the encoding which means any work on encoding and type inference is also original for this semester. Furthermore in the second semseter a new version of control flow analysis with correspondence assertions was developed.

We would like to thanks our supervisor hans Hüttel for supervising this project and the many helpful comments made throughout the project.

Lars Hornbæk Jensen

Bjørn Mølgård Vester

Summery

Previous studies has shown that security properties of communication protocols can be verified using syntax-based analysis techniques. In this thesis we focus on two of such methods and examine the relations between them.

Control flow analysis is a technique to compute the values which a given variable may be bound to in a dynamic setting. It is used to verify a security property called *dynamic authenticity* which expresses that no encryptions may be unexpectedly decrypted. To formalise this property, encryptions and decryptions occuring in a process is annotated with *crypto-points* as well as sets of crypto-points asserting the allowed destinations or origins of an encryption.

Type checking is a technique to verify a different security property called *robust safety*. This property is formalised by placing labeled begin and end events. The property holds if for every end event reachable in a reduction sequence, there is a preceding begin event with the same label. Type checking can verify this property by annotating encryption keys with types containing assertions about which begin events procedes the point of encryption. When decrypting a message with the same key, we know that the assertion on the key must hold.

In this thesis, we use the LySa calculus to formalise communication protocols. We develop a typed version of the LySa calculus using begin/end events, as well as creating an accompanying type system. We construct an encoding from processes with crypto-points in LySa to Typed LySa, and show that dynamic authenticity can be expressed as a robust safety property (but most likely not the other way around).

We then show that types in the encoded process can be inferred under a series of constraints, such that the robust safety property can be verified.

Lastly we create a new control flow analysis to verify the robust safety property using begin/end events. It is based on the original analysis, but uses ideas from the type system. We show that it is possible to verify a process which is verifiable by either the type system or, through encoding, the original analysis.

Resumé

Tidligere studier har vist at sikkerhedsegenskaber af kommunikationsprotekoller kan verificeres ved brug af syntaksbaserede analyseringsteknikker. I denne afhandling fokuserer vi på to af disse metoder, og undersøger relationen mellem dem.

Kontrol-flow-analyse er en teknik til at beregne de værdier som en given variable kan blive bundet til i et dynamisk miljø. Den bliver brugt til at verificerer en sikkerhedsegenskab kaldet *dynamisk autenticitet* som udtrykker at ingen krypteringer kan blive uventet dekrypteret. For at formalisere denne egenskab bliver krypteringer og dekrypteringer, som optræder i en proces, annoteret med *krypto-punkter* samt mængder af krypto-punkter der beskriver de tilladte destinationer eller oprindelser af en kryptering.

Typecheck er en teknik til at verificere en anderledes sikkerhedsegenskab kaldet *robust sikkerhed*. Denne egenskab er formaliseret ved at placere begynd- og slut-markeringer. Denne egenskab holder hvis der for hver slutmarkering der kan nås i en reduktionssekvens er en forudgående begyndmarkering. Typecheck kan verificere denne egenskab ved at annotere krypteringsnøgler med typer der indeholder postulater om hvilke begynd-markeringer der går forud for krypteringspunktet. Ved dekryptering af en besked med samme nøgle ved vi at postulaterne på nøglen må holde.

I denne afhandling bruger vi LySa-kalkylen til at formalisere kommunikationsprotekoller. Vi udvikler en typet udgave af LySa-kalkylen ved brug af begynd/slut-markeringer, samt et tilhørende typesystem. Vi konstruerer en indkodning fra processer med krypto-punkter i LySa til Typed LySa, og viser at dynamisk autenticitet kan udtrykkes som en robust sikkerhedsegenskab (men sandsynligvis ikke den anden vej).

Derefter viser vi at typer i den indkodede proces kan blive udledt under en mængde begrænsninger, sådan at den robuste sikkerhedsegenskab kan verificeres.

Til sidst udvikler vi en ny kontrol-flow-analyse til at verificere den robuste sikkerhedsegenskab ved brug af begynd/slut-markeringer. Den er baseret på den originale analyse, men bruger idéer fra typesystemet. Vi viser at det er muligt at verificere en proces som er verificerbar af enten typesystemet eller, gennem indkodning, den originale analyse.

Contents

1 Introduction **1**
1.1 Protocols 1
1.2 Analysis Methods 2
1.3 Goal 3
1.4 The Structure of the Report 3

2 Control Flow Analysis **5**
2.1 LySa 5
2.2 Assertions for Origin and Destination 6
2.3 Operational Semantics of LySa 6
2.4 Flow Logics 8
2.5 Modeling the Attacker 10
2.6 Properties of the Control Flow Analysis 12

3 The Type System **15**
3.1 Assertions for Correspondences 15
3.2 Types and Effects 16
3.3 Syntax for Typed LySa 16
3.4 Typing Examples 18
3.5 Typing Rules 19
3.6 Operational Semantics 23
3.7 Properties of the Type System 24
3.7.1 Elementary Properties 24
3.7.2 Subject Equivalence and Subject Reduction 30
3.7.3 Properties of the Opponent 34
3.7.4 Safety and Robust Safety 35

4 Encoding LySa to Typed LySa **37**
4.1 Encoding Algorithm 38
4.2 Properties of the Encoding 40

5 Type Inference on the Encoding **47**
5.1 Constraints 47
5.2 Auxiliary Relations 49

5.3 Constructing the Solution to a Type Check 50

6 Control Flow Analysis with Correspondences 55

6.1 Design 55

6.2 Analysis Rules 57

6.3 Properties of Analysis 59

6.4 Encodings 61

7 In Perspective 63

8 Conclusion 67

A Correspondence LySa 69

Chapter 1

Introduction

With an ever growing communications market, the security of communications protocols become an increasingly important issue. The use of cryptographic constructs alone does not guarantee the security of a protocol in itself, and the protocols need to be verified themselves. Several techniques for formal verification exist and can be used to examine protocols in a wide variety of ways.

The aim of this project is to examine the relationship between two such verification techniques, control flow analysis [BDNN01a] and type checking [GJ03].

1.1 Protocols

In a computer network, communication is governed by communications protocols between one or more entities in the network.

A protocol is often described somewhat informally using *protocol narrations*, which are simple sequences of message exchanges between different principals, and can be interpreted as the intended trace of the ideal execution of the protocol. An example of such a narration is a version of the *Wide Mouthed Frog* protocol [BAN90] for key-exchange between two principals with the use of a trusted server.

$$\begin{array}{lll} 1. & A \rightarrow S: & A, \{B, K\}_{K_A} \\ 2. & S \rightarrow B: & \{A, K\}_{K_B} \\ 3. & A \rightarrow B: & \{m_1, \ldots, m_k\}_K \end{array}$$

In stage 1 of the protocol, the principal A sends to the trusted server S a message saying that it is A and that it wants to communicate with B using key K. The server then informs B that A wishes to communicate using the key K which leads to stage 3 where A is able to send its message to B encrypted under key K.

This protocol narration for the Wide Mouthed Frog protocol is not very precise, and the security goals are left implicit. Since the narration only specifies the message exchanges, lots of details are not captured by this informal model. One example of lack of detail is when the server receives the message in stage one: a check on the first variable in the encrypted part needs to be done to figure out who the process A wishes to communicate with. To be able to validate security properties we then need a more formal model with more details. For that purpose we use process calculi. Several calculi has been proposed over the years such as the Spi calculus [AG97] or the Applied π calculus [AF01]. For this project we will use the LySa calculus which was developed in collaboration between University of Pisa and Technical University of Denmark [BBD+03]. It is a variant of the polyadic Spi calculus with pattern-matching, and differs mostly by the fact that channels are absent. The only cryptographic operations allowed are symmetric encryptions and decryptions[1].

1.2 Analysis Methods

It is difficult to analyse protocols for various reasons. One is that a precise notion of security is not easy to model. Another is that the security properties must be guaranteed to hold in infinitely many environments, which often leads to undecidability. Furthermore, any calculus for cryptographic protocols which are remotely interesting are as strong as Turing machines and therefore undecidable by model checking. The idea of a static analysis is to offer a set of decidable methods to perform an analysis for security properties, regardless of the actual data which flows through the protocol, or the environment in which it operates. The undecidability issue can be avoided by over-approximations and by "erring on the safe side". This means that a static analysis may fail to accept some secure protocols, but will never accept an insecure protocol[2]. In other words, if there are no violations in the static analysis, there is guaranteed not to be any violations at run-time.

Various approaches for modeling and analysing cryptographic protocols have been proposed over the years. These include equivalence testing [AG98] and methods based on logics of knowledge [BAN90]. In this report we will examine approaches based on control flow analysis and type checking.

Control flow analysis aims at computing the set of values which a given variable may be bound to in a dynamic setting. This information is then used statically to very the behavior of the program.

Type systems annotate a protocol with types and use these as a basis for

[1] LySa has later been extended to address some of the shortcomings – see LySa^{NS} [BNN04], LySa^{XP} [AN05], and LySa with asymmetric cryptography [BBD+04].

[2] We use the term *slack* of a method to denote the set of secure protocols which the method cannot verify.

checking security properties.

1.3 Goal

In this report we look at the two mentioned verification techniques, control flow analysis and type checking, and attempt to see if it is possible to make a type system such that whenever the control flow analysis tells us that a protocol is safe or unsafe, the type check will produce the same result. Furthermore we combine the ideas of control flow analysis and type checking by making a new control flow analysis using correspondence assertions instead of cryto-point annotations.

1.4 The Structure of the Report

The rest of the report is organized in the following chapters:

Chapter 2: The calculus LySa is presented including the verification technique control flow analysis.

Chapter 3: A typed version of LySa is constructed, which we call Typed LySa , and a type system for verifying security properties in this calculus is created.

Chapter 4: We construct an encoding algorithm from a process in LySa to Typed LySa wih preservation of security properties.

Chapter 5: Type inference is performed on an encoded process.

Chapter 6: A new control flow analysis is constructed based on correspondence assertions.

Chapter 7: Here we take an overall perspective of the methods introduced and developed in this report.

Chapter 8: In the last chapter we summarise and conclude on the work done in the report.

Chapter 2

Control Flow Analysis

Control flow analysis is a static technique for computing safe approximations to the set of values which may be bound to variables in a dynamic setting. It can be applied to a variety of calculi [BDNN01b], including the λ-calculus, CONCURRENT ML, and the π-calculus. In this chapter, we will use the LYSA calculus on which to perform the analysis. We use the analysis specified in [BBD+03].

2.1 LYSA

Instead of using channels, LYSA has all messages pass through the same global medium known as the *ether*, which all processes have access to. To control the intended destination of messages, they can be prefixed with sequences of terms representing the headers. Upon receiving a message, the principal can then check if it is intended for him through the use of pattern-matches in inputs and decryptions. The idea behind the pattern matches of input and decryption is that a pattern $(M_1, \ldots, M_j; x_{j+1}, \ldots, x_k)$ matches a tuple $(M'_1, \ldots, M'_k)$ if the first j values matches up pairwise, i.e. $M_1 = M'_1, \ldots, M_j = M'_j$. If that is the case, the remaining values $M'_{j+1}, \ldots, M'_k$ are bound to the variables $x_{j+1}, \ldots, x_k$. Encryptions are tuples of terms $M_1, \ldots, M_k$ using the symmetric key M_0. The syntax of LYSA is as follows, where $\mathcal{N}$ and $\mathcal{X}$ denote the set of names and variables, respectively:

Basic Syntax of LYSA:

$M ::=$	*terms*
$\quad n$	name $(n \in \mathcal{N})$
$\quad x$	variable $(x \in \mathcal{X})$
$\quad \{M_1, \ldots, M_k\}_{M_0}$	sym. encryption

$P ::=$	*processes*
0	nil
$\langle M_1, \ldots, N_k \rangle.P$	output
$(M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P$	input (with matching)
$P_1 \mid P_2$	parallel composition
$(\nu\, n)P$	restriction
$!P$	replication
decrypt M as $\{M_1, \ldots, M_j;$ $x_{j+1}, \ldots, x_k\}_{M_0}$ in P	sym. decryption (with matching)

2.2 Assertions for Origin and Destination

To describe the authentication property, each encryption and decryption is decorated with *crypto-points*, which labels the point of the cryptographic operation, and a list, which specifies the allowed origins and the destinations of the encrypted message. This allows the model to capture the security issues which may arise in a session-based analysis. Syntactically, an encryption in the LySa calculus is changed to the form:

$$\{M_1, \ldots, M_k\}^{\ell}_{M_0}[\text{dest } \mathcal{L}]$$

and similarly, decryptions are changed to the form:

$$\text{decrypt } M \text{ as } \{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k\}^{\ell}_{M_0}[\text{orig } \mathcal{L}] \text{ in } P$$

In these two processes, ℓ is the crypto-point where the cryptographic operation takes place and $\mathcal{L}$ is the set of crypto-point which specifies the intended crypto-points for encryption or decryption of the message. The set $\mathcal{C}$ (disjoint from the set of variable names and values) enumerates all crypto-points.

2.3 Operational Semantics of LySa

We define a structural equivalence of processes in the following way:

Structural Process Equivalence of LySa, $P \equiv Q$:

$P \equiv P$	(Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$	(Struct Symm)
$(P \equiv Q \land Q \equiv R) \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)

$P \mid 0 \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$!P \equiv P \mid !P$	(Struct Repl Par)
$P \equiv Q$ if P and Q are α-convertible	(Struct Alpha)
$(\nu\, n_1)(\nu\, n_2)P \equiv (\nu\, n_2)(\nu\, n_1)P$	(Struct Res)
$(\nu\, n)0 \equiv 0$	(Struct Res Nil)
$(\nu\, n)(P \mid Q) \equiv P \mid ((\nu\, n)Q)$ if $n \notin \mathrm{fn}(P)$	(Struct Extrusion)

As a notational convenience, we write $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$ for a term with all annotations removed. We also introduce a *faithful membership* E for matching when annotations are ignored. More formally:

$$V \;\mathrm{E}\; v \quad \text{iff} \quad \exists V' \in v : \lfloor\!\lfloor V \rfloor\!\rfloor = \lfloor\!\lfloor V' \rfloor\!\rfloor$$

We define a reduction relation $\rightarrow_{\mathcal{R}}$ using a function RM. We consider two variants of the semantics: one takes advantage of the annotations, and the other one discards them:

- The *reference monitor semantics*, written $P \rightarrow_{\mathrm{RM}} Q$, takes $\mathrm{RM}(\ell, \mathcal{L}', \ell', \mathcal{L}) = (\ell \in \mathcal{L}' \;\wedge\; \ell' \in \mathcal{L})$.
- The *standard semantics*, written $P \rightarrow Q$, takes RM to be universally true.

The reference monitor checks that the crypto-point of the encryption is acceptable at the decryption and the opposite, that the crypto-point of the decryption is acceptable by the encryption. We write $[x \mapsto M]$ for the capture-avoiding substitution of a variable x for the term M.

Operational Semantics $P \rightarrow_{\mathcal{R}} P'$:

(LySa Par)
$$\frac{P \rightarrow_{\mathcal{R}} P'}{P \mid Q \rightarrow_{\mathcal{R}} P' \mid Q}$$

(LySa Res)
$$\frac{P \rightarrow_{\mathcal{R}} P'}{(\nu\, n)P \rightarrow_{\mathcal{R}} (\nu\, n)P'}$$

(LySa Equiv)
$$\frac{P \equiv Q \;\wedge\; Q \rightarrow_{\mathcal{R}} Q' \;\wedge\; Q' \equiv P'}{P \rightarrow_{\mathcal{R}} P'}$$

(LySa IO)

$$\frac{\wedge_{i=1}^{j} \llbracket M_i \rrbracket = \llbracket M_i' \rrbracket}{\langle M_1, \ldots, M_k \rangle.P_1 \mid (M_1', \ldots, M_j'; x_{j+1}, \ldots, x_k).P_2 \rightarrow_{\mathcal{R}} P_1 \mid P_2[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]}$$

(LySa Decr)

$$\frac{\wedge_{i=0}^{j} \llbracket M_i \rrbracket = \llbracket M_i' \rrbracket \wedge \mathrm{RM}(\ell, \mathcal{L}', \ell', \mathcal{L})}{\text{decrypt } \{M_1, \ldots, M_k\}_{M_0}^{\ell}[\text{dest } \mathcal{L}] \text{ as } \{M_1', \ldots, M_j'; x_{j+1}, \ldots, x_k\}_{M_0'}^{\ell'}[\text{orig } \mathcal{L}'] \text{ in } P \rightarrow_{\mathcal{R}} P[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]}$$

2.4 Flow Logics

The control flow analysis of a protocol written in LySa is performed to obtain a safe approximation of values and behaviour arising dynamically. Since the combination of restriction and replication may introduce an infinite number of names, we assume that for each name or variable n there is a canonical representative $\lfloor n \rfloor$. This also serves to avoid ambiguity on names. Consider the following (un-annotated) process:

$$P \triangleq (\nu\, K)\langle \{M\}_K \rangle.(\nu\, K)\langle K \rangle$$

Without a way to make the Ks distinct, the analysis would not know if the key used in the encryption is the same one which is later published. Therefore, we assume that both names K has different canonical representatives[1].

Furthermore, α-renaming of bound names is disciplined such that two names are α-convertible only when they have the same canonical name. The function $\lfloor \cdot \rfloor$ is extended to handle terms: $\lfloor M \rfloor$ is the term where all names and variables are replaced by their canonical versions. With these abstract names, the approximation is represented as a triple (ρ, κ, ψ) that satisfies a set of judgements. These components define the following information:

- $\rho : \lfloor \mathcal{X} \rfloor \rightarrow \wp(\mathcal{V})$: maps each canonical variable to a set of canonical values that it may be bound to.
- $\kappa \subseteq \wp(\mathcal{V}^*)$ contains the possible message sequences which may flow on the network.

[1] In earlier work on the LySa calculus, this was instrumented as an explicit annotation on the form $(\nu\, n^{\chi})$ for restrictions and (x^{β}) for input variables. A *marker environment* would then be used to map free names and variables to their abstract representatives. In this report, as with most recent work on LySa, we will refrain from using a particular method of instrumenting the canonical names, as it adds an unnecessary layer of complexity to the analysis.

- $\psi \subseteq \mathcal{C} \times \mathcal{C}$ contains the possible violations of crypto-point annotations on the form (ℓ, ℓ'), specifying that something encrypted at ℓ was unexpectedly decrypted at ℓ', or something decrypted at ℓ' unexpectedly originated from ℓ.

The analysis consist of finding the least of the components (ρ, κ, ψ) which satisfies a set of judgements. We use the judgements as presented in [BBD+03]. The judgements for terms take the form:

$$\rho \models M : v$$

and expresses that v is an acceptable estimate of the set of values that M may evaluate to in the environment ρ.

Control Flow Analysis of Terms:

(CFA Name)

$$\frac{\lfloor n \rfloor \in v}{\rho \models n : v}$$

(CFA Variable)

$$\frac{\rho(\lfloor x \rfloor) \subseteq v}{\rho \models x : v}$$

(CFA Encryption)

$$\frac{\begin{array}{c} \wedge_{i=0}^{k} \rho \models M_i : v_i \wedge \\ \forall V_0, V_1, \ldots, V_k : \wedge_{i=0}^{k} V_i \in v_i \Rightarrow \{V_1, \ldots, V_k\}_{V_0}^{l}[\text{dest } \mathcal{L}] \in v \end{array}}{\rho \models \{M_1, \ldots, M_k\}_{M_0}^{l}[\text{dest } \mathcal{L}] : v}$$

The judgements for processes take the form:

$$(\rho, \kappa) \models P : \psi$$

Control Flow Analysis of Processes:

(CFA Par)

$$\frac{(\rho, \kappa) \models P_1 : \psi \ \wedge \ (\rho, \kappa) \models P_2 : \psi}{(\rho, \kappa) \models P_1 \mid P_2 : \psi}$$

(CFA Res)

$$\frac{(\rho, \kappa) \models P : \psi}{(\rho, \kappa) \models (\nu\, n)P : \psi}$$

(CFA Repl)

$$\frac{(\rho, \kappa) \models P : \psi}{(\rho, \kappa) \models !P : \psi}$$

(CFA Nil)

$$\frac{}{(\rho, \kappa) \models 0 : \psi}$$

(CFA Output)

$$\frac{\begin{array}{c}\wedge_{i=1}^{k}\ \rho \models M_i : v_i \ \wedge \\ \forall V_1,\ldots,V_k\ \wedge_{i=1}^{k}\ V_i \in v_i \Rightarrow \langle V_1,\ldots,V_k\rangle \in \kappa \ \wedge \\ (\rho,\kappa) \models P : \psi\end{array}}{(\rho,\kappa) \models \langle M_1,\ldots,M_k\rangle.P : \psi}$$

(CFA Input)

$$\frac{\begin{array}{c}\wedge_{i=1}^{j}\ \rho \models M_i : v_i \ \wedge \\ \forall \langle V_1,\ldots,V_k\rangle \in \kappa : \wedge_{i=1}^{j}\ V_i \ \mathrm{E}\ v_i \Rightarrow \wedge_{i=j+1}^{k}\ V_i \in \rho(\lfloor x_i \rfloor) \ \wedge \\ (\rho,\kappa) \models P : \psi\end{array}}{(\rho,\kappa) \models (M_1,\ldots,M_j; x_{j+1},\ldots,x_k).P : \psi}$$

(CFA Decrypt)

$$\frac{\begin{array}{c}\rho \models M : v \ \wedge\ \wedge_{i=0}^{j}\ \rho \models M_i : v_i \ \wedge \\ \forall \{V_1,\ldots,V_k\}_{V_0}^{\ell}[\mathrm{dest}\ \mathcal{L}] \in v : \wedge_{i=0}^{j}\ V_i \ \mathrm{E}\ v_i \Rightarrow \wedge_{i=j+1}^{k}\ V_i \in \rho(\lfloor x_i \rfloor) \ \wedge \\ (\neg \mathrm{RM}(\ell,\mathcal{L}',\ell',\mathcal{L}) \Rightarrow (\ell,\ell') \in \psi) \ \wedge \\ (\rho,\kappa) \models P : \psi\end{array}}{(\rho,\kappa) \models \mathrm{decrypt}\ M\ \mathrm{as}\ \{M_1,\ldots,M_j; x_{j+1},\ldots,x_k\}_{E_0}^{l'}[\mathrm{orig}\ \mathcal{L}']\ \mathrm{in}\ P : \psi}$$

2.5 Modeling the Attacker

Protocols may be executed in an unsafe environment which renders them vulnerable to attacks from other parties. The protocol P running in such an environment is modeled by

$$P \mid P_{\bullet},$$

where $P_{\bullet}$ represents an arbitrary attacker with the capabilities presented by Dolev and Yao [DY81]:

1. Obtain any message passing through the ether.
2. Decrypt messages if he knows the key (we assume perfect cryptography).
3. Construct new encryptions from the values he knows.
4. Send messages of values he knows.
5. Generate new values.

In this paper, we do not consider *deletion attacks*, where an attacker deletes messages from the protocol without the principals noticing it. This will normally lead to a halt of the protocol, where an error handling procedure takes over in order to recover.

This arbitrary attacker is modeled by a formula $\mathcal{F}^{\text{DY}}$, which captures all the Dolev-Yao capabilities and makes the formula as strong as any attacker. This means that if (ρ, κ, ψ) satisfies $\mathcal{F}^{\text{DY}}$, then $(\rho, \kappa, \psi) \models P_\bullet$ for all attackers $P_\bullet$. To characterise the attacker, we make a set of assumptions which has no influence on the semantics of LySA: we say that a process P is of type $(\mathcal{N}_{\text{f}}, \mathcal{A}_\kappa, \mathcal{A}_{\text{Enc}})$ whenever:

1. It is closed (i.e. it has no free variables).
2. All free names are in $\mathcal{N}_{\text{f}}$.
3. All the arities used for sending or receiving are in $\mathcal{N}_\kappa$.
4. All the arities used for encryption or decryption are in $\mathcal{N}_{\text{Enc}}$.

The attacker cannot forge annotations as these express the intentions of the protocol, and being able to do so would lead to failure of validation for any protocol. However, the syntax of LySA requires annotations and therefore only the trivial [dest $\mathcal{C}$] and [orig $\mathcal{C}$], and only the crypto-point $\ell_\bullet$ (not occurring in P) are used.

Another aspect of an attacker is that we must be able to control the use of canonical names and variables in it. For a process P executing in parallel with the attacker, P must be examined to find the finite set $\mathcal{N}_c$ of all canonical names, and the finite set $\mathcal{X}_c$ for all canonical variables. We then construct the new canonical name $n_\bullet$ and the new canonical variable $z_\bullet$ such that $n_\bullet \notin \mathcal{N}_c$ and $z_\bullet \notin \mathcal{X}_c$. Given an attacker Q of type $(\mathcal{N}_{\text{f}}, \mathcal{A}_\kappa, \mathcal{A}_{\text{Enc}})$, the semantically equivalent process $\overline{Q}$ is constructed as follows:

- All restrictions $(\nu\, n)P$ are α-converted (in the classical sense) into $(\nu\, n')P$, where n' has the canonical representative $n_\bullet$.
- All variables x_i in subprocesses $(M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P$ and decrypt M as $\{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k\}^{\ell}_{M_0}$[orig $\mathcal{L}$] in P are α-converted (in the classical sense) to variables x'_i with the canonical representative $z_\bullet$.

The canonical variable $n_\bullet$ then represents knowledge known from the beginning by the attacker, while $z_\bullet$ represents all knowledge gained.

The formula $\mathcal{F}^{\text{DY}}$ is defined as the conjunction of the following five components corresponding to the Dolev-Yao conditions mentioned in the beginning of this section:

The Dolev-Yao Attacker:

(1) $\wedge_{k \in \mathcal{A}_\kappa} \forall \langle V_1, \ldots, V_k \rangle \in \kappa : \wedge_{i=1}^{k} V_i \in \rho(z_\bullet)$

(2) $\wedge_{k \in \mathcal{A}_{\mathrm{Enc}}} \forall \{V_1, \ldots, V_k\}_{V_0}^{\ell}[\mathrm{dest}\ \mathcal{L}] \in \rho(z_\bullet) :$
$V_0\ \mathrm{E}\ \rho(z_\bullet) \Rightarrow (\wedge_{i=1}^{k} V_i \in \rho(z_\bullet)\ \wedge\ (\neg \mathrm{RM}(\ell, \mathcal{C}, \ell_\bullet, \mathcal{L}) \Rightarrow (\ell, \ell_\bullet) \in \psi))$

(3) $\wedge_{k \in \mathcal{A}_{\mathrm{Enc}}} \forall V_1, \ldots, V_k : \wedge_{i=0}^{k} V_i \in \rho(z_\bullet) \Rightarrow \{V_1, \ldots, V_k\}_{V_0}^{\ell_\bullet}[\mathrm{dest}\ \mathcal{C}] \in \rho(z_\bullet)$

(4) $\wedge_{k \in \mathcal{A}_\kappa} \forall V_1, \ldots, V_k : \wedge_{i=1}^{k} V_i \in \rho(z_\bullet) \Rightarrow \langle V_1, \ldots, V_k \rangle \in \kappa$

(5) $\{n_\bullet\} \cup \lfloor \mathcal{N}_{\mathrm{f}} \rfloor \subseteq \rho(z_\bullet)$

Once the analysis has found the smallest (ρ, κ, ψ) which satisfies the protocol executed in parallel with the attacker, authentication is guaranteed if $\psi = \emptyset$ and the secrecy of messages is guaranteed if they do not occur in $\rho(z_\bullet)$.

2.6 Properties of the Control Flow Analysis

The following properties of the control flow analysis are proved in [BBD+03].

Theorem 2.1 states that if an analysis (ρ, κ, ψ) is an acceptable estimate for a process P, then that analysis is also an acceptable estimate for any process that P may evolve into.

Theorem 2.1 (Subject Reduction). *If $P \rightarrow_{\mathcal{R}} Q$ and $(\rho, \kappa) \models P : \psi$ then also $(\rho, \kappa) \vdash Q : \psi$. This holds for both the standard semantics as well as the reference monitor semantics.*

Definition 2.2 (Reference Monitor).
We say that the reference monitor *cannot abort* a process P whenever there exist no Q, Q' such that $P \rightarrow^* Q \rightarrow Q'$ and $P \rightarrow^*_{\mathrm{RM}} Q \not\rightarrow_{\mathrm{RM}}$.

Theorem 2.3 tells us that the analysis correctly predicts when we do not need the reference monitor.

Theorem 2.3 (Static Check for the Reference Monitor).
If $(\rho, \kappa) \models P : \emptyset$ then the reference monitor cannot abort P.

Definition 2.4 (Static Authentication).
We say that a process P guarantees *static authentication* if there exists (ρ, κ) such that $(\rho, \kappa) \models P : \emptyset$ and $(\rho, \kappa, \emptyset)$ satisfies $\mathcal{F}^{\mathrm{DY}}$.

Definition 2.5 (Dynamic Authentication).
We say that a process P guarantees *dynamic authentication* with respect to the annotations in P iff the reference monitor cannot abort $P \mid Q$ regardless of the choice of attacker Q.

Soundness for $\mathcal{F}^{\mathrm{DY}}$ is established. This means that $\mathcal{F}^{\mathrm{DY}}$ captures the capabilities of any attacker.

Theorem 2.6 (Soundness of the Dolev-Yao Condition). *If* (ρ, κ, ψ) *satisfies* $\mathcal{F}^{\mathrm{DY}}$ *of type* $(\mathcal{N}_f, \mathcal{A}_\kappa, \mathcal{A}_{Enc})$*, then* $(\rho, \kappa) \models Q : \psi$ *for all attackers* Q *of type* $(\mathcal{N}_f, \mathcal{A}_\kappa, \mathcal{A}_{Enc})$*.*

Theorem 2.7 establishes the connection between the Dolev-Yao conditions and that of *hardest attacker* from [NNH02]. It also shows that there actually exists an attacker as strong as $\mathcal{F}^{\mathrm{DY}}$.

Theorem 2.7 (Completeness of the Dolev-Yao Condition). *There exists an attacker* Q_{hard} *of type* $(\mathcal{N}_f, \mathcal{A}_\kappa, \mathcal{A}_{Enc})$ *such that the formula* $(\rho, \kappa) \models Q_{hard} : \psi$ *is equivalent to the formula* $\mathcal{F}^{\mathrm{DY}}$ *of type* $(\mathcal{N}_f, \mathcal{A}_\kappa, \mathcal{A}_{Enc})$*.*

The main result is then that of Theorem 2.8.

Theorem 2.8 (Authentication). *If* P *guarantees static authentication then* P *guarantees dynamic authentication.*

Chapter 3

The Type System

Type systems are used to control information flow in a process. A natural extension is to enrich types with effects and place assertions in the process which further describe intensional aspects of the dynamic behaviour. Abadi [Aba99] proposes the idea of checking security properties of cryptographic protocols by type checking. Using this idea, Gordon and Jeffrey [GJ04a] developed a type and effect systems to prove *correspondence*[1], which is an authentication property pioneered by Woo and Lam [WL93]. In this system, the assertions in a process are guaranteed to hold if the process is well-typed.

In this chapter, we will develop a type system in LySa based on Gordon and Jeffrey's idea[2].

3.1 Assertions for Correspondences

The correspondence property address authentication concerns by annotating sequences of message exchanges with events on the form "begin($\vec{M}$)" or "end($\vec{M}$)", where $\vec{M}$ is a label which typically indicates the names of the principals involved (we use $\vec{M}$ as a possibly empty sequence ranging over M). These events mark the progress of the protocol and are also known as injective agreements. While they have no effect at runtime, the intention is to guarantee that for every end($\vec{M}$) event, there is a distinct preceding begin($\vec{M}$) event[3]. This assures that the authenticating principal is "talking" to the principal it has in mind.

[1] Correspondences has later appeared in other forms – see [Gol03] by Gollman for an overview.

[2] Gordon and Jeffrey's type system has, among other things, been expanded with pattern-matching [HJ04], asymmetric cryptography [GJ04b] and to handle authorization policies [FGM05].

[3] Note that this does not put a requirement of liveness on the system, so a begin($\vec{M}$) event does not necessarily have a succeeding end($\vec{M}$) event.

These annotations are one-to-one in the sense that there is exactly one distinct begin($\vec{M}$) event for every end($\vec{M}$) event. To assert that this is not violated, some form of nonces or timestamps are required. Since LySa does not support nonces, the conditions for a correspondence need to be relaxed in the following way: let a one-to-many correspondence be the condition that there exists a preceding, but not necessarily distinct, begin-event for every end-event [GJ02]. This is also known as a non-injective agreement. To implement this, we will only consider begin-events on the form begin!($\vec{M}$), which can be used to end an arbitrary number of end($\vec{M}$) events. For example, consider the following correspondence sequences:

begin!(a), end(a)	safe
begin!(a), end(a), end(a)	safe
begin!(a), begin!(b), end(a), end(b)	safe
end(a)	unsafe
end(a), begin!(a)	unsafe

The formal definition of safety can be found in Section 3.7 (Properties of the Type System).

3.2 Types and Effects

In our type system, we must deal with types for cryptographic keys, plaintexts and cipher-texts. Since any term can be used as a key, we unify the framework for types by defining a type as a sequence of *embedded types* and a set of *latent effects*. If we cannot trust the key, or it represent data with can be published, we give it type Un, short for untrusted.

If the type is the annotation of a key, it has a sequence of embedded types which specifies the type sequence it may be used to encrypt. This is necessary since a decryption must give types to the decrypted names, which would otherwise be unknown.

Lastly, if the type is the annotation of a key, it has a set of latent effects on the form !begun($\vec{M}$). Any decryptions treat these effects as a credit towards subsequent end-events. This is sound since we require all encryptions to treat the effects as a debit which must be accounted for by previous begin-events. This way, effects can be "transferred" between processes since all principals in the protocol must agree to this. This also imply that the key must never be obtained by an attacker, since he cannot be expected to follow these rules.

3.3 Syntax for Typed LySa

The syntax of processes P is similar to LySa, except we introduce one-to-many correspondence assertions, and new names are annotated with types:

Syntax of TYPED LYSA:

$M ::=$	*terms*
$\quad n$	name ($n \in \mathcal{N}$)
$\quad x$	variable ($x \in \mathcal{X}$)
$\quad \{M_1, \ldots, M_k\}_{M_0}$	symmetric encryption
$P ::=$	*processes*
$\quad 0$	nil
$\quad \langle M_1, \ldots, M_k \rangle.P$	output
$\quad (M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P$	input
$\quad P_1 \mid P_2$	parallel composition
$\quad (\nu\, n : T)P$	restriction
$\quad !P$	replication
$\quad$ decrypt M as $\{M_1, \ldots, M_j;$ $x_{j+1}, \ldots, x_k\}_{M_0}$ in P	symmetric decryption
$\quad$ begin!$(M_1, \ldots, M_k).P$	begin-event
$\quad$ end$(M_1, \ldots, M_k).P$	end-event

The syntax of types is as follows:

Syntax of Types:

$B ::=$	*begun-assertions*
$\quad$!begun$(M_1, \ldots, M_k)$	begun-many assertion
$T ::=$	*types*
$\quad$ Key$(x_1 : T_1, \ldots, x_k : T_k)[\vec{B}]$	symmetric key (scope of x_i is $\vec{B}$)
$\quad$ Un	Public data

We will use the notation $\vec{x} : T$ for $x_1 : T, \ldots, x_k : T$, and $\vec{x} : \vec{T}$ for $x_1 : T_1, \ldots, x_k : T_k$. In order to keep track of the types and effects of a process, we introduce an assertion environment E, which contains the effects of the process and the types of the free names in the process. The syntax of environments is as follows:

Syntax of Environments:

$A ::=$		*assertions*
	B	begun assertion
	$n : T$	type assertion on name
	$x : T$	type assertion on variable
$E ::=$		*environments*
	$A_0, \ldots, A_k$	assertions

We use the notation $\emptyset$ for the empty environment, and the notation E_1, A, E_2 for an environment where both E_1 and E_2 may be empty.

3.4 Typing Examples

We will consider two examples which illustrates how to specify two security properties.

Example 3.1 (Typing an Example for Aliveness).
Consider the following (safe) process:

$$\begin{aligned} P_A(K) &\triangleq (\nu\, m : T_m)\ \text{begin!}(A, B).\langle\{m\}_K\rangle \\ P_B(K) &\triangleq (;x).\text{decrypt } x \text{ as } \{;m_x\}_K \text{ in end}(A, B) \\ P_{sys} &\triangleq (\nu\, K : T_K)(P_A(K) \mid P_B(K)) \end{aligned}$$

Here, P_A and P_B share a key K which is used to send an encrypted name m from P_A to P_B. Therefore, the process is annotated with the correspondence (A, B) which must begin in P_A and ended in P_B. This insures "aliveness" in the sense that when P_B asserts the end-event, it asserts that P_A must have been alive at some point in the past. To verify this, we define the following types to be used in the process:

$$\begin{aligned} T_m &\triangleq \text{Un} \\ T_K &\triangleq \text{Key}(y : T_m)[!\text{begun}(A, B)] \end{aligned}$$

We annotate the type T_K of key K with the latent effect $!\text{begun}(A, B)$ to type check that the correspondence is indeed valid. Since K is used to encrypt the message m of type T_m, the embedded type of T_K is T_m. Lastly, the name m is not used as a key, so we we give it type Un. We we wanted to insure the secrecy of m, we could give it type Key(Un)[ ] instead as the type system does not allow keys to be published.

This process is safe since P_B cannot assert $\text{end}(A, B)$ without P_A asserting a $\text{begin!}(A, B)$ event, as the key K is kept secret among the two.

◆

Example 3.2 (Typing an Example for Non-injective Agreements). Correspondences can be used to verify a stronger security property than aliveness. Consider P_A and P_B from the previous example. If we wanted not only to specify that P_A must have been alive at some point when P_B executes the end event, but the two principals must also "agree" on the encrypted value sent from P_A and received at P_B, we could specify this property in the following way:

$$\begin{aligned}
P_A(K) &\triangleq (\nu\, m : T_m)\ \text{begin!}(m, A, B).\langle\{m\}_K\rangle \\
P_B(K) &\triangleq (;x).\text{decrypt } x \text{ as } \{;m_x\}_K \text{ in end}(m_x, A, B) \\
P_{sys} &\triangleq (\nu\, K : T_K)(P_A(K) \mid P_B(K))
\end{aligned}$$

Since x is eventually substituted with m in the reduction sequence, P_B can be sure that the received message does indeed originate from P_A. This is what is known as a non-injective agreement. To verify this, we define the following types to be used in the process:

$$\begin{aligned}
T_m &\triangleq \text{Un} \\
T_K &\triangleq \text{Key}(y : T_m)[!\text{begun}(y, A, B)]
\end{aligned}$$

When type checking the encryption, the variable y will be bound to the name m such that the latent effect $!\text{begun}(m, A, B)$ must hold. In the decryption, the variable y will be bound to the variable m_x, and the latent effect $!\text{begun}(m_x, A, B)$ is carried over to work as a credit towards the $\text{end}(m_x, A, B)$ event. This is sound as m_x can only be substituted with m because the key K is kept secret among the two.

◆

3.5 Typing Rules

In the following we define our type system. We have four judgements in our type system, which are inductively defined by the rules presented on the following pages:

Judgements:

$E \vdash \diamond$	E is a good environment.
$E \vdash \vec{A}$	Assertions $\vec{A}$ are valid in E.
$E \vdash P$	Process P is well-typed in E.

The domain of environments is defined using the following rules:

Domain of Environments:

$$
\begin{aligned}
\mathrm{dom}(\emptyset) &\triangleq \emptyset \\
\mathrm{dom}(\vec{A}, A) &\triangleq \mathrm{dom}(\vec{A}) \cup \mathrm{dom}(A) \\
\mathrm{dom}(n : T) &\triangleq \{n\} \\
\mathrm{dom}(x : T) &\triangleq \{x\} \\
\mathrm{dom}(!\mathrm{begun}(M_1, \ldots, M_k)) &\triangleq \emptyset
\end{aligned}
$$

Free names are defined using the following rules (we capture both free names and free variables):

Free Names of Terms, Types and Processes:

$$
\begin{aligned}
\mathrm{fn}(n) &\triangleq \{n\} \\
\mathrm{fn}(x) &\triangleq \{x\} \\
\mathrm{fn}(\{M_1, \ldots, M_k\}_{M_0}) &\triangleq \mathrm{fn}(M_0) \cup \cdots \cup \mathrm{fn}(M_k) \\
\mathrm{fn}(M : T) &\triangleq \mathrm{fn}(M) \cup \mathrm{fn}(T) \\
\mathrm{fn}(\mathrm{Un}) &\triangleq \emptyset \\
\mathrm{fn}(\mathrm{Key}(x_1 : T_1, \ldots, x_k : T_k)[\vec{B}]) &\triangleq \begin{cases} (\mathrm{fn}(\mathrm{T_1}) \cup \cdots \cup \mathrm{fn}(\mathrm{T_k}) \ \cup \ \mathrm{fn}(\vec{B})) \\ -\{x_1, \ldots, x_k\} \end{cases} \\
\mathrm{fn}(A_1, \ldots, A_k) &\triangleq \mathrm{fn}(A_1) \cup \cdots \cup \mathrm{fn}(A_k) \\
\mathrm{fn}(!\mathrm{begun}(M_1, \ldots, M_k)) &\triangleq \mathrm{fn}(M_1) \cup \cdots \cup \mathrm{fn}(M_k) \\
\mathrm{fn}(\diamond) &\triangleq \emptyset \\
\mathrm{fn}(0) &\triangleq \emptyset \\
\mathrm{fn}(\langle M_1, \ldots, M_k \rangle.P) &\triangleq \mathrm{fn}(M_1) \cup \cdots \cup \mathrm{fn}(M_k) \cup \mathrm{fn}(P) \\
\mathrm{fn}((M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P) &\triangleq \begin{cases} \mathrm{fn}(M_1) \cup \cdots \cup \mathrm{fn}(M_j) \cup \\ (\mathrm{fn}(P) - \{x_{j+1}, \ldots, x_k\}) \end{cases} \\
\mathrm{fn}(P_1 \mid P_2) &\triangleq \mathrm{fn}(P_1) \cup \mathrm{fn}(P_2) \\
\mathrm{fn}((\nu\, n : T)P) &\triangleq \mathrm{fn}(T) \cup (\mathrm{fn}(P) - \{n\}) \\
\mathrm{fn}(!P) &\triangleq \mathrm{fn}(P) \\
\left.\begin{aligned} \mathrm{fn}(\mathrm{decrypt}\ M\ \mathrm{as}\ \{M_1, \ldots, M_j; \\ x_{j+1}, \ldots, x_k\}_{M_0}\ \mathrm{in}\ P) \end{aligned}\right\} &\triangleq \begin{cases} \mathrm{fn}(M) \cup \ \mathrm{fn}(M_0) \cup \cdots \cup \ \mathrm{fn}(M_j) \cup \\ (\mathrm{fn}(P) - \{x_{j+1}, \ldots, x_k\}) \end{cases} \\
\mathrm{fn}(\mathrm{begin!}(M_1, \ldots, M_k).P) &\triangleq \mathrm{fn}(M_1) \cup \cdots \ \cup \mathrm{fn}(M_k) \cup \mathrm{fn}(P) \\
\mathrm{fn}(\mathrm{end}(M_1, \ldots, M_k).P) &\triangleq \mathrm{fn}(M_1) \cup \cdots \ \cup \mathrm{fn}(M_k) \cup \mathrm{fn}(P)
\end{aligned}
$$

An environment E is good if every name appears at most once in the domain of E, and there is no free variables:

Good Environments:

(Env Begun)
$$\frac{E \vdash \diamond \quad (\mathrm{fn}(M_1) \cup \cdots \cup \mathrm{fn}(M_k)) \subseteq \mathrm{dom}(E)}{E, !\mathrm{begun}(M_1, \ldots, M_k) \vdash \diamond}$$

(Env Empty)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env Name)
$$\frac{E \vdash \diamond \quad \mathrm{fn}(T) \subseteq \mathrm{dom}(E) \quad n \notin \mathrm{dom}(E)}{E, n : T \vdash \diamond}$$

(Env Variable)
$$\frac{E \vdash \diamond \quad \mathrm{fn}(T) \subseteq \mathrm{dom}(E) \quad x \notin \mathrm{dom}(E)}{E, x : T \vdash \diamond}$$

An encryption can be made by either a trusted or an untrusted key. When using a trusted key, the rule (Encrypt) makes sure that all the types of terms in the encryption matches the embedded types of the key. In addition, any assertions in the key type is also required to hold under substitution in the embedded assertion. When using a public key, the rule (Encrypt Un) makes sure that everything encrypted is public. This last rule is necessary to make an opponent well-typed, since he is not allowed by the type system to use secret keys.

Good Assertions:

(And)
$$\frac{E \vdash \vec{A} \quad E \vdash A}{E \vdash \vec{A}, A}$$

(Id)
$$\frac{E \vdash \diamond \quad A \in E}{E \vdash A}$$

(Empty)
$$\frac{}{E \vdash \emptyset}$$

(Encrypt)
$$\frac{E \vdash M_0 : \mathrm{Key}(x_1 : T_1, \ldots, x_n : T_n)[\vec{B}] \quad E \vdash M_1 : T_1, \ldots, M_k : T_k \qquad E \vdash \vec{B}[x_1 \mapsto M_1, \ldots, x_k \mapsto M_k]}{E \vdash \{M_1, \ldots, M_k\}_{M_0} : \mathrm{Un}}$$

(Encrypt Un)
$$\frac{E \vdash M_0 : \mathrm{Un}, \ldots, M_k : \mathrm{Un}}{E \vdash \{M_1, \ldots, M_k\}_{M_0} : \mathrm{Un}}$$

Processes are well-typed given the rules below. The rules for parallel processes, restriction, replication and nil are standard. The rule (Proc Begin) adds a begin assertion to the environment, while the rule (Proc End) states that a matching begin assertion must be valid in the environment. The rule for output, (Proc Output), checks that all terms are of the special type Un, while the input rule, (Proc Input), assign Un to all binding variables. Note, that there is no need to type the pattern-matching part of the input, since anything on the ether has type Un.

In addition, there are four decrypt rules. The decrypt rule (Proc Decrypt) is used for secret keys, and assigns the embedded types from the key to all the binding variables and adds any assertions from the key to the environment as well. For keys of type Un, the rule (Proc Decrypt Un) is used, which assigns the special type Un to all binding variables.

Well-Typed Processes:

(Proc Par)

$$\frac{E \vdash P_1 \quad E \vdash P_2}{E \vdash P_1 \mid P_2}$$

(Proc Res)

$$\frac{E, n : T \vdash P}{E \vdash (\nu\, n : T)P}$$

(Proc Repl)

$$\frac{E \vdash P}{E \vdash !P}$$

(Proc Nil)

$$\frac{}{E \vdash 0}$$

(Proc Begin)

$$\frac{\text{E, !begun}(M_1, \ldots, M_k) \vdash P}{E \vdash \text{begin!}(M_1, \ldots, M_k).P}$$

(Proc End)

$$\frac{E \vdash !\text{begun}(M_1, \ldots, M_k) \quad E \vdash P}{E \vdash \text{end}(M_1, \ldots, M_k).P}$$

(Proc Output)

$$\frac{E \vdash M_1 : \text{Un}, \ldots, M_k : \text{Un} \quad E \vdash P}{E \vdash \langle M_1, \ldots, M_k \rangle.P}$$

(Proc Input)

$$\frac{\text{E, } x_{j+1} : \text{Un}, \ldots, x_k : \text{Un} \vdash P}{E \vdash (M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P}$$

(Proc Decrypt)

$$\frac{\begin{array}{c} E \vdash M : \text{Un} \quad E \vdash M_0 : \text{Key}(x'_1 : T_1, \ldots, x'_k : T_k)[\vec{B}] \\ E, x_{j+1} : T_{j+1}, \ldots, x_k : T_k, \\ \vec{B}[x'_1 \mapsto M_1, \ldots, x'_j \mapsto M_j, x'_{j+1} \mapsto x_{j+1}, \ldots, x'_k \mapsto x_k] \vdash P \end{array}}{E \vdash \text{decrypt } M \text{ as } \{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k\}_{M_0} \text{ in } P}$$

(Proc Decrypt Un)

$$\frac{E \vdash M : \text{Un} \quad E \vdash M_0 : \text{Un} \quad E, x_{j+1} : \text{Un}, \dots, x_k : \text{Un} \vdash P}{E \vdash \text{decrypt } M \text{ as } \{M_1, \dots, M_j; x_{j+1}, \dots, x_k\}_{M_0} \text{ in } P}$$

3.6 Operational Semantics

We define the structural equivalence of processes similarly to LySA with the change that restrictions are annotated with types, and a premiss is added to LySA's (Struct Res).

Structural Process Equivalence of LySA, $P \equiv Q$:

$P \equiv P$	(Typed Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$	(Typed Struct Symm)
$(P \equiv Q \land Q \equiv R) \Rightarrow P \equiv R$	(Typed Struct Trans)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Typed Struct Par)
$P \mid 0 \equiv P$	(Typed Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Typed Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Typed Struct Par Assoc)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Typed Struct Repl)
$!P \equiv P \mid !P$	(Typed Struct Repl Par)
$P \equiv Q$ if P and Q are α-convertible	(Typed Struct Alpha)
$(\nu\, n_1 : T_1)(\nu\, n_2 : T_2)P \equiv (\nu\, n_2 : T_2)(\nu\, n_1 T_1)P$	
if $n_1 \neq n_2 \land n_1 \notin \text{fn}(T_2) \land n_2 \notin \text{fn}(T_1)$	(Typed Struct Res)
$(\nu\, n : T)0 \equiv 0$	(Typed Struct Res Nil)
$(\nu\, n : T)(P \mid Q) \equiv P \mid ((\nu\, n : T)Q)$	
if $n \notin \text{fn}(P)$	(Typed Struct Extrusion)

We define the reduction relation somewhat similarly to the standard semantics of LySA, except that annotations are different and rules for begin and end events are added:

Operational Semantics $P \to P'$:

(Typed Par)

$$\frac{P \to P'}{P \mid Q \to P' \mid Q}$$

(Typed Res)

$$\frac{P \to P'}{(\nu\, n : T)P \to (\nu\, n : T)P'}$$

(Typed Equiv)
$$\frac{P \equiv Q \wedge Q \rightarrow Q' \wedge Q' \equiv P'}{P \rightarrow P'}$$

(Typed IO)
$$\frac{}{\langle M_1, \ldots, M_k \rangle.P_1 \mid (M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P_2 \rightarrow P_1 \mid P_2[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]}$$

(Typed Decr)
$$\frac{}{\text{decrypt } \{M_1, \ldots, M_k\}_{M_0} \text{ as } \{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k\}_{M_0} \text{ in } P \rightarrow P[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]}$$

(Typed Begin)
$$\frac{P \rightarrow P'}{\text{begin!}(\vec{M}).P \rightarrow \text{begin!}(\vec{M}).P'}$$

(Typed End)
$$\frac{}{\text{end}(\vec{M}).P \rightarrow P}$$

3.7 Properties of the Type System

In this section, we will prove the main properties of the type system. Some of the proofs are inspired by [FGM05].

3.7.1 Elementary Properties

Definition 3.3 (Inert Processes).
We say that a process is *inert* if it only consists of restrictions and begin events. We use the notation γ for inert processes.

Definition 3.4 (Generic Judgement).
We use the meta-syntax $\mathcal{J}$ as a notion for any right-side of a judgement, ie.

$$\mathcal{J} ::= \diamond \mid \vec{A} \mid P$$

Definition 3.5 (Well-formed Typing Judgements).
A typing judgement $E \vdash \mathcal{J}$ is well-formed if and only if all occurrences of binding names and variables are pairwise different and different from those in $\text{dom}(E)$.

Any typing judgement $E \vdash \mathcal{J}$ can be rewritten using α-conversion such that it is well-formed. For simplicity, will assume that typing judgements are always well-formed.

Lemma 3.6 (Good Sub-Environments). *If* $E_1, E_2 \vdash \diamond$ *then* $E_1 \vdash \diamond$

Proof. Proof proceeds by induction on the derivation of $E_1, E_2 \vdash \diamond$. Consider the last rule used:

Case (Env Name): If $E_1, E_2', n : T \vdash \diamond$, then $E_1, E_2' \vdash \diamond$.

Case (Env Variable): If $E_1, E_2', x : T \vdash \diamond$, then $E_1, E_2' \vdash \diamond$.

Case (Env Begun): If E_1, E_2', !begun($\vec{M}$) $\vdash \diamond$, then $E_1, E_2' \vdash \diamond$.

□

Lemma 3.7 (Weakening).

(a) If $E_1, E_2 \vdash \mathcal{J}$ *and* $fn(T) \subseteq dom(E_1)$ *and* $M \notin dom(E_1, E_2)$ *and* $M \in \mathcal{N} \cup \mathcal{X}$ *then* $E_1, M : T, E_2 \vdash \mathcal{J}$.

(b) If $E_1, E_2 \vdash \mathcal{J}$ *and* $fn(B) \subseteq dom(E_1)$, *then* $E_1, B, E_2 \vdash \mathcal{J}$.

Proof. We split the proof of each point depending on $\mathcal{J}$:

(a) (1) If $E_1, E_2 \vdash \diamond$ and $\mathrm{fn}(T) \subseteq \mathrm{dom}(E_1)$ and $M \notin \mathrm{dom}(E_1, E_2)$ and $M \in \mathcal{N} \cup \mathcal{X}$ then $E_1, M : T, E_2 \vdash \diamond$.
By Lemma 3.6 (Good Sub-Environments) and either (Env Name) or (Env Variable), $E_1, M : T \vdash \diamond$.
Proof proceeds by induction on the depth of the derivation of $E_1, M : T, E_2 \vdash \diamond$.
Suppose $E_2 = E_2', N : U$.
By hypothesis, $\mathrm{fn}(U) \subseteq \mathrm{dom}(E_1, E_2')$ and $N \notin \mathrm{dom}(E_1, E_2')$.
Since $M \notin \mathrm{dom}(E_1, E_2)$ then $M \neq N$.
Therefore, $\mathrm{fn}(U) \subseteq \mathrm{dom}(E_1, M : T, E_2')$ and $N \notin \mathrm{dom}(E_1, M : T, E_2')$.
By inductive hypothesis, $E_1, M : T, E_2' \vdash \diamond$.
The result then follows by either (Env Name) or (Env Variable).
Suppose instead $E_2 = E_2'$, !begun$(M_1, \ldots, M_k)$.
Using the same procedure, the result then follows by (Env Begun).

(2) If $E_1, E_2 \vdash A$ and $\mathrm{fn}(T) \subseteq \mathrm{dom}(E_1)$ and $M \notin \mathrm{dom}(E_1, E_2)$ and $M \in \mathcal{N} \cup \mathcal{X}$ then $E_1, M : T, E_2 \vdash A$.
By Point (1), $E_1, M : T, E_2 \vdash \diamond$.
Consider the rule used:

Case (Id): By hypothesis of rule, $A \in \mathrm{dom}(E_1, E_2)$.
Then also $A \in \mathrm{dom}(E_1, M : T, E_2)$.

Case (Encrypt): Follows by induction on each premiss.

Case (Encrypt Un): Follows by induction on each premiss.

Case (And): Follows by induction on each assertion.

Case (Empty): Follows immediately from the rule.

(3) If $E_1, E_2 \vdash P$ and $\text{fn}(T) \subseteq \text{dom}(E_1)$ and $M \notin \text{dom}(E_1, E_2)$ and $M \in \mathcal{N} \cup \mathcal{X}$ then $E_1, M : T, E_2 \vdash P$.
Proof proceeds by induction on the depth of the derivation of $E_1, E_2 \vdash P$.
Consider the rule used:

Case (Proc Nil) Follows Immediately from the rule.

Case (Proc Output): Suppose $P = \langle M_1, \ldots, M_k \rangle.P'$.
By induction hypothesis, $E_1, M : T, E_2 \vdash P'$.
By hypothesis, $E_1, E_2 \vdash M_1 : \text{Un}, \ldots, M_k : \text{Un}$.
Then by Point (2), $E_1, M : T, E_2 \vdash M_1 : \text{Un}, \ldots, M_k : \text{Un}$.

Case (Proc Input): Suppose $P = (M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P'$.
If $M \in \{x_{j+1}, \ldots, x_k\}$, then by Definition 3.5 (Well-formed Typing Judgements), the conflicting x is assumed to be α-converted to a name which does not occur in $\text{dom}(E_1, M_T, E_2)$.
Then by applying (Env Variable) repeatedly, we get $E_1, M : T, E_2, x_{j+1} : \text{Un}, \ldots, x_k : \text{Un} \vdash \diamond$.
By inductive hypothesis, $E_1, M : T, E_2, x_{j+1} : \text{Un}, \ldots, x_k : \text{Un} \vdash P$.

Case (Proc Par): Suppose $P = P_1 \mid P_2$.
By inductive hypothesis, $E_1, M : T, E_2 \vdash P_1$ and $E_1, M : T, E_2 \vdash P_2$.

Case (Proc Res): Suppose $P = (\nu\, n : T)P$.
If $M = n$, then by Definition 3.5 (Well-formed Typing Judgements), n is assumed to be α-converted to a name which does not occur in $\text{dom}(E_1, M : T, E_2)$.
Then by (Env Name), $E_1, M : T, E_2, n : T \vdash \diamond$.
By inductive hypothesis, $E_1, M : T, E_2, n : T \vdash P$.

Case (Proc Repl): Suppose $P = !P$.
By inductive hypothesis, $E_1, M : T, E_2 \vdash P$.

Case (Proc Decrypt): Suppose $P = \text{decrypt } N$ as $\{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k\}_{M_0}$ in P'.
By hypothesis and point (2), we get $E_1, M : T, E_2 \vdash N : \text{Un}$.
Also by hypothesis and point (2), we get $E_1, M : T, E_2 \vdash M_0 : \text{Key}(y_1 : T_1, \ldots, y_k : T_k)[\vec{B}]$.
If $M \in \{x_{j+1}, \ldots, x_k\}$, then by Definition 3.5 (Well-formed Typing Judgements), the conflicting x is assumed to be α-converted to a name which does not occur in $\text{dom}(E_1, M : T, E_2)$.
Then by applying (Env Variable) and (Env Begin) repeatedly,

we get $E_1, M : T, E_2, x_{j+1} : T_{j+1}, \ldots, x_k : T_k, \vec{B}[y_1 \mapsto M_1, \ldots, y_k \mapsto M_k] \vdash \diamond$.
By inductive hypothesis, $E_1, M : T, E_2, x_{j+1} : T_{j+1}, \ldots, x_k : T_k, \vec{B}[y_1 \mapsto M_1, \ldots, y_k \mapsto M_k] \vdash P$.

(b) Proofs are similar to point (a).

□

Lemma 3.8 (Substitution). *If* $E_1, x : T, E_2 \vdash \mathcal{J}$ *and* $E_1 \vdash M : T$ *then* $E_1, E_2[x \mapsto M] \vdash \mathcal{J}[x \mapsto M]$.

Proof. We split the proof depending on $\mathcal{J}$:

(a) If $E_1, x : T, E_2 \vdash \diamond$ and $E_1 \vdash M : T$ then $E_1, E_2[x \mapsto M] \vdash \diamond$. Proof proceeds by induction on the depth of the derivation of $E_1, x : T, E_2 \vdash \diamond$. Consider the last rule used:

Case (Env Name): Suppose $E, y : U \vdash \diamond$.
By hypothesis, $E \vdash \diamond$, $\mathrm{fn}(U) \subseteq \mathrm{dom}(E)$ and $y \notin \mathrm{dom}(E)$.
Suppose $E = E_1, x : T, E_2'$.
By inductive hypothesis, $E_1, E_2'[x \mapsto M] \vdash \diamond$.
By hypothesis of the lemma, $E_1 \vdash M : T$.
By hypothesis, $\mathrm{fn}(U) \subseteq \mathrm{dom}(E_1, x : T, E_2')$.
Applying the substitution, then $\mathrm{fn}(U[x \mapsto M]) \subseteq \mathrm{dom}((E_1, E_2')[x \mapsto M])$.
Since $x \notin \mathrm{dom}(E_1)$, we have $\mathrm{fn}(U[x \mapsto M]) \subseteq \mathrm{dom}(E_1, E_2'[x \mapsto M])$.
Since $x \neq y$, we have $(y : U)[x \mapsto M] = y : U[x \mapsto M]$.
By (Env Name), $E_1, E_2'[x \mapsto M], y : U[x \mapsto M] \vdash \diamond$.

Case (Env Variable): Similar to the Case (Env Name).

Case (Env Begun): Suppose E, $!\mathrm{begun}(\vec{M}) \vdash \diamond$.
By hypothesis, $E \vdash \diamond$, $\mathrm{fn}(\mathrm{begin}!(\vec{M}) \subseteq \mathrm{dom}(E)$.
Suppose $E = E_1, x : T, E_2'$.
By inductive hypothesis, $E_1, E_2'[x \mapsto M] \vdash \diamond$.
By hypothesis of the lemma, $E_1 \vdash M : T$.
By hypothesis, $\mathrm{fn}(\mathrm{begin}!(\vec{M})) \subseteq \mathrm{dom}(E_1, x : T, E_2')$.
Applying the substitution, then $\mathrm{fn}(\mathrm{begin}!(\vec{M})[x \mapsto M]) \subseteq \mathrm{dom}((E_1, E_2')[x \mapsto M])$.
By (Env Begin), $E_1, E_2'[x \mapsto M]$, $\mathrm{begin}!(\vec{M})[x \mapsto M] \vdash \diamond$.

(b) If $E_1, x : T, E_2 \vdash A$ and $E_1 \vdash M : T$ then $E_1, E_2[x \mapsto M] \vdash A[x \mapsto M]$.
By point (a), $E_1, E_2[x \mapsto M] \vdash \diamond$.
Consider the rule used:

Case (Id) by Type Assertion: Suppose $A = y : U$.
By hypothesis, $E_1, x : T, E_2 \vdash \diamond$ and $y : U \in \mathrm{dom}(E_1, x : T, E_2)$.
We distinguish two cases. Suppose $y \neq x$.
Then $y : U \in (E_1, E_2)$.
If $y : U \in E_1$, then $x \notin \mathrm{fn}(U)$ and $y : U[x \mapsto M] \in (E_1, E_2[x \mapsto M])$.
If $y : U \in E_2$, then $y : U[x \mapsto M] \in (E_1, E_2[x \mapsto M])$.
By (Id), $E_1, E_2[x \mapsto M] \vdash y : U[x \mapsto M]$.
Suppose instead $y = x$.
By hypothesis of the lemma, $E_1 \vdash M : T$.
By Lemma 3.7 (Weakening), $E_1, E_2[x \mapsto M] \vdash M : T$.
Since $y = x$, then $T = U$.
By substitution, $(y : T)[y \mapsto M] = M : T$ and we conclude.

Case (Id) by Begun Assertion: Suppose $E_1, x : T, E_2 \vdash \mathrm{begin!}(\vec{M})$.
By hypothesis, $!\mathrm{begun}(\vec{M}) \in (E_1, x : T, E_2)$.
Then $!\mathrm{begun}(\vec{M}) \in (E_1, E_2)$.
Suppose $!\mathrm{begun}(\vec{M}) \in E_1$.
Then $x \notin \vec{M}$ so $!\mathrm{begun}(\vec{M}) = !\mathrm{begun}(\vec{M})[x \mapsto M]$.
Suppose instead $!\mathrm{begun}(\vec{M}) \in E_2$.
Then $!\mathrm{begun}(\vec{M})[x \mapsto M] \in E_2[x \mapsto M]$.
In both cases, we get $!\mathrm{begun}(\vec{M})[x \mapsto M] \in (E_1, E_2[x \mapsto M])$.

Case (Encrypt): Follows by induction on each premiss.

Case (Encrypt Un): Follows by induction on each premiss.

Case (And): Follows by induction on each assertion.

Case (Empty): Follows immediately from the rule.

(c) If $E_1, x : T, E_2 \vdash P$ and $E_1 \vdash M : T$ then $E_1, E_2[x \mapsto M] \vdash P[x \mapsto M]$.
By point (a), $E_1, E_2[x \mapsto M] \vdash \diamond$.
Proof by induction on the depth of the derivation of $E_1, x : T, E_2 \vdash P$ using point (b).

□

Lemma 3.9 (Strengthening).

(a) If $E_1, x : T, E_2 \vdash \mathcal{J}$ *and* $x \notin \mathrm{fn}(\mathcal{J}) \cup \mathrm{fn}(E_2)$ *then* $E_1, E_2 \vdash \mathcal{J}$.

(b) If $E_1, B, E_2 \vdash \mathcal{J}$ *and* $E_1 \vdash B$ *then* $E_1, E_2 \vdash \mathcal{J}$.

Proof. We split the proof of each point depending on $\mathcal{J}$:

(a) (1) If $E_1, x : T, E_2 \vdash \diamond$ and $x \notin \mathrm{fn}(\mathcal{J}) \cup \mathrm{fn}(E_2)$ then $E_1, E_2 \vdash \diamond$.
By induction on the depth of the derivation of $E_1, x : T, E_2 \vdash \diamond$.
Consider the last rule used.

Case (Env Name): Suppose $E_2 = E_2', y : U$.
By inductive hypothesis, $E_1, E_2' \vdash \diamond$.
By hypothesis, $y \notin \text{dom}(E_1, x : T, E_2')$ and therefore $y \notin \text{dom}(E_1, E_2')$.
By hypothesis, $\text{fn}(U) \subseteq \text{dom}(E_1, x : T, E_2')$.
Since, by hypothesis of the lemma, $x \notin \text{fn}(U)$, then $\text{fn}(U) \subseteq \text{dom}(E_1, E_2')$.
By (Env Name), $E_1, E_2', y : U \vdash \diamond$.

Case (Env Variable): Similar to the Case (Env Name).

Case (Env Begun): Suppose $E_2 = E_2'$, begun!$(\vec{M}$.
By inductive hypothesis, $E_1, E_2' \vdash \diamond$.
By hypothesis, $\text{fn}(\text{begun!}(\vec{M})) \subseteq \text{dom}(E_1, x : T, E_2')$.
Since, by hypothesis of the lemma, $x \notin \text{fn}(\text{begun!}(\vec{M})$, then $\text{fn}(\text{begun!}(\vec{M}) \subseteq \text{dom}(E_1, E_2')$.

(2) If $E_1, x : T, E_2 \vdash A$ and $x \notin \text{fn}(\mathcal{J}) \cup \text{fn}(E_2)$ then $E_1, E_2 \vdash A$.
By Point (1), $E_1, E_2 \vdash \diamond$.
Consider the rule used:

Case (Id) Since, by hypothesis of the lemma, $x \notin \text{fn}(A)$, it is the case that $A \in E_1, E_2$.

Case (Encrypt): Follows from induction on each premiss.

Case (Encrypt Un): Follows from induction on each premiss.

Case (And): Follows from induction on each assertion.

Case (End): Follows immediately from the rule.

(3) If $E_1, x : T, E_2 \vdash P$ and $x \notin \text{fn}(P) \cup \text{fn}(E_2)$ then $E_1, E_2 \vdash P$. Proof by induction on the depth of the derivation of $E_1, x : T, E_2 \vdash P$ using point (2).

(b) If $E_1, B, E_2 \vdash \mathcal{J}$ and $E_1 \vdash B$ then $E_1, E_2 \vdash \diamond$.
Proof by induction using the fact that if $E_1 \vdash B$ then $B \in E_1$, which means B is duplicated.
All rules relying on $B \in E_1, B, E_2$ then holds because $B \in E_1, E_2$.

□

Lemma 3.10 (Exchange). *If* $E_1, E_2, E_3, E_4 \vdash \mathcal{J}$ *and* $dom(E_2) \cap fn(E_3) = \emptyset$ *and* $fn(E_2) \cap dom(E_3) = \emptyset$ *then* $E_1, E_3, E_2, E_4 \vdash \mathcal{J}$.

Proof. We split the proof depending on $\mathcal{J}$:

(a) By induction on the depth of the derivation of $E_1, E_2, E_3, E_4 \vdash \diamond$. Consider the last rule:

Case (Env Empty): Trivial.

Case (Env Name): Suppose $E, x : T \vdash \diamond$.
By hypothesis, $E \vdash \diamond$, $\text{fn}(T) \subseteq \text{dom}(E)$ and $x \notin \text{dom}(E)$.
If $E = E_1, E_2, E_3, E_4$, where $E_4 = E_4', x : T$, we conclude by applying the inductive hypothesis.
If $E = E_1, E_2, E_3$ and $E_3 = E_3', x : T$, by inductive hypothesis, $E_1, E_3', E_2 \vdash \diamond$.
By Lemma 3.6 (Good Sub-Environments), $E_1, E_3' \vdash \diamond$.
By (Env Name), $E_1, E_3 \vdash \diamond$.
By repeatedly applying Lemma 3.7 (Weakening), $E_1, E_3, E_2 \vdash \diamond$.
The case for $E = E_1, E_2$ is trivial.

Case (Env Variable): Similar to Case (Env Name).

Case (Env Begun): Similar to Case (Env Name).

(b) By induction on the depth of the derivation of $E_1, E, E', E_2 \vdash A$, using point (a).

(c) By induction on the depth of the derivation of $E_1, E, E', E_2 \vdash P$, using point (b).

□

3.7.2 Subject Equivalence and Subject Reduction

Lemma 3.11 (Subject Equivalence). *If $E \vdash P$ and $P \equiv P'$ then $E \vdash P'$.*

Proof. By induction on the derivation of $P \equiv P'$ we show:

(a) If $E \vdash P$ then $E \vdash P'$

(b) If $E \vdash P'$ then $E \vdash P$

Case (Typed Struct Refl): Suppose $P \equiv P$.
Both (a) and (b) are immediate.

Case (Typed Struct Symm): Suppose $P \equiv Q$.
By hypothesis, $Q \equiv P$.
Both (a) and (b) follow immediately by applying the inductive hypothesis (a) and (b).

Case (Typed Struct Trans): Suppose $P \equiv R$.
By hypothesis, $P \equiv Q$ and $Q \equiv R$.
Both cases follow from transitivity of implication and the inductive hypothesis.

Case (Typed Struct Par): Suppose $P \mid Q \equiv P' \mid Q$.
By hypothesis of the lemma, $P \equiv P'$.
By hypothesis of (a), $E \vdash P \mid Q$.

By (Proc Par), $E \vdash P$ and $E \vdash Q$.
By inductive hypothesis, $E \vdash P'$.
By (Proc Par), $E \vdash P' \mid Q$.
The proof for (b) is symmetric.

Case (Typed Struct Par Zero): Suppose $P \mid 0 \equiv P$.
By hypothesis of (a), $E \vdash P \mid 0$.
By (Proc Par), $E \vdash P$.
The proof for (b) is similar.

Case (Typed Struct Par Comm): Suppose $P \mid Q \equiv Q \mid P$.
By hypothesis of (a), $E \vdash P \mid Q$.
By (Proc Par), $E \vdash P$ and $E \vdash Q$.
Also by (Proc Par), $E \vdash Q \mid P$.
The proof for (b) is symmetric.

Case (Typed Struct Par Assoc): Suppose $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$.
By hypothesis of (a), $E \vdash (P \mid Q) \mid R$.
By (Proc Par), $E \vdash P \mid Q$ and $E \vdash R$.
By (Proc Par), $E \vdash P$ and $E \vdash Q$).
By (Proc Par), $E \vdash Q \mid Q$.
By (Proc Par), $E \vdash (Q \mid Q) \mid R$.
The proof for (b) is similar.

Case (Typed Struct Repl): Suppose $!P \equiv !P'$.
By hypothesis, $P \equiv P'$.
By hypothesis of (a), $E \vdash !P$.
By (Proc Repl), $E \vdash !P$.
By inductive hypothesis, $E \vdash P'$.
By (Proc Repl), $E \vdash !P'$.
The proof for (b) is symmetric.

Case (Typed Struct Repl Par): Suppose $!(P \mid Q) \equiv !P \mid !Q$.
By hypothesis of (a), $E \vdash !P \mid Q$.
By (Proc Repl), $E \vdash P \mid Q$.
By (Proc Par), $E \vdash P$ and $E \vdash Q$.
By (Proc Repl), $E \vdash !P$ and $E \vdash !Q$.
By (Proc Par), $E \vdash !P \mid !Q$.
The proof for (b) is similar.

Case (Typed Struct Alpha): By Definition 3.5 (Well-formed Typing Judgements), if (Typed Struct Alpha) converts a bound name to a name which is already bound elsewhere, then by Definitiondef:well-formed-jud (Well-formed Typing Judgements), we assume it is converted again to a name which does not occur elsewhere.

Case (Typed Struct Res): Suppose $(\nu\, n_1 : T_1).(\nu\, n_2 : T_2).P \equiv (\nu\, n_2 : T_2).(\nu\, n_1 : T_1).P$.
By (Proc Res), $E, n_1 : T_1 \vdash (\nu\, n_2 : T_2).P$.
By (Proc Res), $E, n_1 : T_1, n_2 : T_2 \vdash P$.
Since, by hypothesis, $n_1 \neq n_2, n_1 \notin \mathrm{fn}(T_2), n_2 \notin \mathrm{fn}(T_1)$, then by Lemma 3.10 (Exchange), $E, n_2 : T_2, n_1 : T_1 \vdash P$.
By two applications of (Proc Res), $E \vdash (\nu\, n_2 : T_2).(\nu\, n_1 : T_1).P$.
The proof for (b) is symmetric.

Case (Typed Struct Res Nil): Suppose $(\nu\, n : T).0 \equiv 0$.
By hypothesis, $E \vdash (\nu\, n : T)0$.
By (Proc Nil), $E \vdash 0$.
The proof for (b) is similar.

Case (Typed Struct Extrusion): Suppose $(\nu\, n : T).(P \mid Q) \equiv P \mid (\nu\, n : T).Q$.
By hypothesis of (a), $E \vdash (\nu\, n : T).(P \mid Q)$.
By (Proc Res), $E, n : T \vdash P \mid Q$.
By (Proc Par), $E, n : T \vdash P$ and $E, n : T \vdash Q$.
By (Proc Res), $E, n : T \vdash Q$.
Since, by hypothesis, $n \notin \mathrm{fn}(P)$, then by Lemma 3.9 (Strengthening) part (a), $E \vdash P$.
By (Proc Par), $E \vdash P \mid (\nu\, n : T).Q$.
The proof for (b) is similar, using Lemma 3.7 (Weakening) part (a) instead of Lemma 3.9 (Strenghening).

□

Theorem 3.12 (Subject Reduction). *If $E \vdash P$ and $P \to P'$ then $E \vdash P'$.*

Proof. The proof proceeds by induction on the derivation of $P \to P'$.

Case (Typed Par): Follows from (Proc Par) and the inductive hypothesis.

Case (Typed Res): Suppose $(\nu\, n : T).P \to (\nu\, n : T).P'$.
By hypothesis, $P \to P'$.
By hypothesis of the lemma, $E \vdash (\nu\, n : T).P$.
By inductive hypothesis, $E, n : T \vdash P'$.

Case (Typed Equiv): This follows from Lemma 3.11 (Subject Equivalence) and the inductive hypothesis.

Case (Typed LySa IO): Let

$P = (\langle M_1, \ldots, M_k \rangle.P_1) \mid ((M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P_2)$ and
$P' = P_1 \mid P_2[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]$

and assume that $E \vdash P$ and $P \rightarrow P'$ due to (TYPED LYSA IO). We get

$$\begin{aligned} E \vdash P \quad \text{iff} \quad & E \vdash \langle M_1, \ldots, M_k \rangle.P_1 \wedge \\ & E \vdash (M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P_2 \\ \text{iff} \quad & E \vdash M_1 : \text{Un} , \ldots, M_k : \text{Un} \wedge \\ & E \vdash P_1 \wedge \\ & E, x_{j+1} : \text{Un}, \ldots, x_k : \text{Un} \vdash P_2 \end{aligned}$$

By applying Lemma 3.8 (Substitution) repeatedly on $(E, x_{j+1} : \text{Un}, \ldots, x_k : \text{Un} \vdash P_2)$, we may conclude that $E \vdash P_2[x_{j+1} \mapsto M_{j+1}, \ldots x_k \mapsto M_k]$. Using (Proc Par), we get $E \vdash P_1 \mid P_2[x_{j+1} \mapsto M_{j+1}, \ldots x_k \mapsto M_k]$, i.e. that $E \vdash P'$ as wanted.

Case (TYPED LYSA Decr): Let

$$\begin{aligned} P = \ & (\text{decrypt } \{M_1, \ldots, M_k\}_{M_0} \text{ as } \{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k\}_{M_0} \\ & \text{in } P'') \text{ and} \\ P' = \ & P''[x_{j+1} \mapsto M_{j+1}, \ldots x_k \mapsto M_k] \end{aligned}$$

and assume that $E \vdash P$ and $P \rightarrow P'$ due to (Redn Dec). The key can be either trusted or untrusted. We treat both possibilities separately:

Subcase (Proc Decrypt) We get

$$\begin{aligned} E \vdash P \quad \text{iff} \quad & E \vdash \{M_1, \ldots, M_k\}_{M_0} : \text{Un} \\ & E \vdash M_0 : \text{Key}(x'_1 : T_1, \ldots, x'_k : T_k)[\vec{B}] \wedge \\ & E, x_{j+1} : T_{j+1}, \ldots, x_k : T_k, \vec{B}[x'_1 \mapsto M_1, \ldots, \\ & \quad x'_j \mapsto M_j, x'_{j+1} \mapsto x_{j+1}, \ldots, x'_k \mapsto x_k] \vdash P'' \\ \text{iff} \quad & E \vdash M_0 : \text{Key}(x'_1 : T_1, \ldots, x'_k : T_k)[\vec{B}] \wedge \\ & E \vdash M_1 : T_1 \wedge \cdots \wedge E \vdash M_k : T_k \wedge \\ & E \vdash \vec{B}[x_1 \mapsto M_1, \ldots, x_k \mapsto M_k] \\ & E, x_{j+1} : T_{j+1}, \ldots, x_k : T_k, \vec{B}[x'_1 \mapsto M_1, \ldots, \\ & \quad x'_j \mapsto M_j, x'_{j+1} \mapsto x_{j+1}, \ldots, x'_k \mapsto x_k] \vdash P'' \end{aligned}$$

By applying Lemma 3.9 (Strengthening) part (b) and Lemma 3.8 (Substitution) repeatedly on $E, x_{j+1} : T_{j+1}, \ldots, x_k : T_k, \vec{B}[x_1 \mapsto M_1, \ldots, x_k \mapsto M_k] \vdash P''$, we get $E \vdash P''[x_{j+1} \mapsto M_{j+1}, \ldots x_k \mapsto M_k]$, i.e. that $E \vdash P'$ as wanted.

Subcase (Proc Decrypt Un) We get

$$\begin{aligned} E \vdash P \quad \text{iff} \quad & E \vdash \{M_1, \ldots, M_k\}_{M_0} : \text{Un} \wedge \\ & E \vdash M_0 : \text{Un} \wedge \\ & E, x_{j+1} : \text{Un}, \ldots, x_k : \text{Un} \vdash P \\ \text{iff} \quad & E \vdash M_0 : \text{Un} \wedge \cdots \wedge E \vdash M_k : \text{Un} \wedge \\ & E, x_{j+1} : \text{Un}, \ldots, x_k : \text{Un} \vdash P \end{aligned}$$

By applying Lemma 3.8 (Substitution) repeatedly on E, x_{j+1} : $\text{Un}, \ldots, x_k : \text{Un} \vdash P''$, we get $E \vdash P''[x_{j+1} \mapsto M_{j+1}, \ldots x_k \mapsto M_k]$, i.e. that $E \vdash P'$ as wanted.

Case (Typed Begin): Follows from (Proc Begin) and the inductive hypothesis.

Case (Typed End): Follows directly from (Proc End).

□

3.7.3 Properties of the Opponent

In TYPED LYSA, we do not consider completely arbitrary attacker processes, but restrict ourselves to *opponent* processes[4] which satisfy the following two conditions:

1. Opponents cannot assert events. If they could, no process would be robustly safe because of the attacker "end(n)".
2. Opponents cannot have have access to private names. Therefore, all types occurring in the process must by Un.

Definition 3.13 (Formulation of Opponent).
A process P is *assertion-free* iff it contains no begin- or end-assertions. A process P is *untyped* iff the only type occurring in P is type Un. An *opponent* O is an assertion-free untyped process.

Lemma 3.14 (Opponent Typability of Terms). *For any M, if fn*$(M) = \vec{M}$ *then* $\vec{M} : \text{Un} \vdash M : \text{Un}$.

Proof. By structural induction on M:

Case $M = n$: By Rules (Env Name) and (Env Empty), $x : \text{Un} \vdash \diamond$. By (Id), $x : \text{Un} \vdash x : \text{Un}$.

Case $M = x$: Same as Case $M = n$.

Case $M = \{M_1, \ldots, M_k\}_{M_0}$: Let $\vec{M_i} = \text{fn}(M_i)$. By inductive hopothesis, $\vec{M_0} : \text{Un} \vdash M_0 : \text{Un}, \ldots, \vec{M_k} : \text{Un} \vdash M_k : \text{Un}$. Let $\vec{M} = \vec{M_1}, \ldots, \vec{M_k}$. By Lemma 3.7 (Weakening), $\vec{M} \vdash M_i$. By (Encrypt Un), $\vec{M_0} : \text{Un}, \ldots \vec{M_k}$: $\text{Un} \vdash \{M_1, \ldots, M_k\}_{M_0}$.

□

Lemma 3.15 (Opponent Typability). *If a process O is an opponent (i.e. an assertion-free untyped process), and fn*$(O) \subseteq \vec{M}$, *then* $\vec{M} : \text{Un} \vdash O$.

[4] This is similar to what was done in [GJ04a].

Proof. By induction on the structure of O. Suppose O is an opponent, and $\mathrm{fn}(O) \subseteq \vec{M}$:

Case $O = 0$: By (Proc Nil), $(\vec{M} : \mathrm{Un} \vdash O)$.

Case $O = \langle M_1, \ldots, M_k \rangle.P$: By Lemma 3.14 (Opponent Typability of Terms), $\vec{M} : \mathrm{Un} \vdash M_1 : \mathrm{Un}, \ldots, M_k : \mathrm{Un}$.
By inductive hypothesis, $\vec{M} : \mathrm{Un} \vdash P$.
By (Proc Output), $\vec{M} \vdash O$.

Case $O = (M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P$: Since $\{x_{j+1}, \ldots, x_k\} \not\subseteq \vec{M}$, then $\vec{M} : \mathrm{Un}, x_{j+1} : \mathrm{Un}, \ldots, x_k : \mathrm{Un} \vdash P$ by inductive hypothesis.
By (Proc Input), $\vec{M} : \mathrm{Un}, x_{j+1} : \mathrm{Un}, \ldots, x_k : \mathrm{Un} \vdash O$.

Case $O = P_1 \mid P_2$: Let $\vec{M_1} = \mathrm{fn}(P_1)$ and $\vec{M_2} = \mathrm{fn}(P_2)$.
By induction hypothesis, $(\vec{M_1} : \mathrm{Un} \vdash P_1)$ and $(\vec{M_2} : \mathrm{Un} \vdash P_2)$.
By Lemma 3.7 (Weakening) and (Proc Par), $(\vec{M} : \mathrm{Un} \vdash O)$.

Case $O = (\nu\, n : T)P$: Since $n \notin \vec{M}$, then $(\vec{M} : \mathrm{Un}, n : T \vdash P)$ by induction hypothesis.
By (Proc Res), $(\vec{M} : \mathrm{Un}, n : T \vdash P)$.

Case $O = !P$: By induction hypothesis, $(\vec{M} : \mathrm{Un} \vdash P)$.
By (Proc Repl), $(\vec{M} : \mathrm{Un} \vdash O)$.

Case $O =$ decrypt M as $\{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k\}_{M_0}$ in P: Since $\{x_{j+1}, \ldots, x_k\} \not\subseteq \vec{M}$, then $\vec{M} : \mathrm{Un}, x_{j+1} : \mathrm{Un}, \ldots, x_k : \mathrm{Un} \vdash P$ by inductive hypothesis.
By Lemma 3.14 (Opponent Typability of Terms) and Lemma 3.7 (Weakening), $\vec{M} : \mathrm{Un} \vdash M_0$.
By (Proc Decrypt Un), $\vec{M} : \mathrm{Un}, x_{j+1} : \mathrm{Un}, \ldots, x_k : \mathrm{Un} \vdash O$.

□

3.7.4 Safety and Robust Safety

A process is *safe* (ie. satisfies non-injective agreement) if and only if for every run of the process, and for every M, there is a begin-event labelled M preceding every end-event labelled M. We formalise this in the following definition:

Definition 3.16 (Safety and Robust Safety).
A process P has an *authenticity error* iff it is of the form $(\gamma.\mathrm{end}(\vec{M}).P'')$, and $\mathrm{begin}!(\vec{M}) \notin \gamma$. (In P, the restrictions are renamed such that they bind pairwise different names, and names different from free names.) A process P is *safe* iff $P \to^* P'$ implies that P' does not have an authenticity error. A process P is *robustly safe* iff $(P \mid O)$ is safe for every opponent process O.

Theorem 3.17 (Safety). *If* $(\vec{M} : \vec{T} \vdash P)$ *then* P *is safe.*

Proof. Suppose that $\vec{M} : \vec{T} \vdash P$ and $P \rightarrow^* \gamma.\text{end}(\vec{M}).P''$.
By Theorem 3.12 (Subject Reduction), we have $\vec{M} : \vec{T} \vdash \gamma.\text{end}(\vec{M}).P''$.
By (Proc End), we have $E \vdash !\text{begun}(\vec{M})$.
By (Id), $!\text{begun}(\vec{M}) \in E$.
Since γ is inert, this can only follow from applying Rule (Proc Begin) which means $\text{begin}!(\vec{M}) \in P'$.
By Definition 3.16 (Safety and Robust Safety), P is safe. □

Theorem 3.18 (Robust Safety). *If* $(\vec{M} : \text{Un} \vdash P)$*, then* P *is robustly safe.*

Proof. Suppose $(\vec{M} : \text{Un} \vdash P)$.
Let O be an opponent process such that $\text{fn}(O) - \vec{M} = \vec{M}'$.
Let $E = \vec{M} : \text{Un}, \vec{M} : \text{Un}$.
By Lemma 3.15 (Opponent Typability), we have $E \vdash O$.
By Lemma 3.7 (Weakening), $E \vdash P$.
By (Proc Par), $E \vdash P \mid O$.
By Theorem 3.17 (Safety), $P \mid O$ is safe. □

Chapter 4

Encoding LySa to Typed LySa

In this chapter we will construct an encoding from a process in LySa to Typed LySa, and show that message authenticity properties using crypto-point annotations can be formulated as non-injective agreements using correspondence assertions. We will use o and d as crypto-points.

In the following we present an encoding algorithm $[\![\cdot]\!] : \text{LySa} \rightarrow \text{Typed LySa}$. The encoding is performed using the following informal rules:

- Each crypto-point, be it an annotation of an encryption or decryption, is represented by public names. Therefore, there is no need for them to be restricted.

- Each encryption has a crypto-point o and a set of destinations $\{d_1, \ldots, d_k\}$. The encoded encryption must be preceded by the assertions $\text{begin!}(o, d_1). \cdots .\text{begin!}(o, d_k)$, which are placed in the beginning of the encoded process. In addition, the crypto-point o is sent as part of the encrypted message. That is, an encrypted message $\{m_1, \ldots, m_n\}_K$ at crypto-point o is changed to $\{m_1, \ldots, m_n, o\}_K$. To be able to use crypto-points as names, we will extend the set of names such that $n \in \mathcal{N} \cup \mathcal{C}$.

- Each encoded decryption with crypto-point d must be followed by the single annotation $\text{end}(o, d)$, where o is the variable representing the crypto-point from the encoding, which is part of the encoded message as shown in the previous point.

- Since types are required to check that the correspondences hold, each restriction of a new name must be annotated with a type. The choice of types does not effect the authenticity property, so the encoding simply annotates names with type Un. In Chapter 5 (Type Inference on the Encoding), we will introduce an algorithm which can infer types using the information from the control flow analysis such that if the

analysis is verifiable, we can type check the encoded process given certain constraints.

4.1 Encoding Algorithm

The encoding algorithm $[\![\cdot]\!]$ is a two-pass implementation of the previously introduced informal rules.

The first pass over the LySa process uses an auxiliary relation $\textsc{ac}(P)$ (short for allowed correspondences) to capture the correspondences which are allowed to begin (e.g. those generated from encryptions in the process). These begin events are placed in the beginning of the process. Note that the relation AC only uses on the information on destinations from the encryptions, and ignores the origin sets on decryptions. This is to make the presentation of the encoding easier. In the next section, we will introduce a definition of ***well-formed crypto-sets*** that specifies when we can safely ignore the origin sets.

The second pass uses $(\!|\cdot|\!)$ to perform the actual encoding. This involves removing the annotations specific to LySa (e.g. crypto-points and origin/destination sets), annotate restrictions with types, and finally place end events after each decryption.

Encoding Function $[\![\cdot]\!]$:

$$[\![P]\!] \triangleq \text{begin!}(o_1, d_1). \cdots .\text{begin!}(o_n, d_n).(\!|P|\!)$$
$$\textbf{where } \textsc{ac}(P) = \{(o_1, d_1), \ldots, (o_n, d_n)\}$$

Auxiliary Relation $\textsc{ac}(P)$:

$$\begin{array}{rcl}
\textsc{ac}(n) & \triangleq & \emptyset \\
\textsc{ac}(x) & \triangleq & \emptyset \\
\textsc{ac}(\{M_1, \ldots, M_k\}^{o}_{M_0}[\text{dest } D]) & \triangleq & (o \times D) \cup \textsc{ac}(M_1) \cup \cdots \cup \textsc{ac}(M_k) \\
\textsc{ac}(0) & \triangleq & \emptyset \\
\textsc{ac}(\langle M_1, \ldots, M_k\rangle.P) & \triangleq & \textsc{ac}(M_1) \cup \cdots \cup \textsc{ac}(M_k) \cup \textsc{ac}(P) \\
\textsc{ac}((M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P) & \triangleq & \textsc{ac}(P) \\
\textsc{ac}(P \,|\, Q) & \triangleq & \textsc{ac}(P) \cup \textsc{ac}(Q) \\
\textsc{ac}((\nu\, n)P) & \triangleq & \textsc{ac}(P) \\
\textsc{ac}(!P) & \triangleq & \textsc{ac}(P) \\
\left.\begin{array}{r}\textsc{ac}(\text{decrypt } M \text{ as } \{M_1, \ldots, M_j; \\ x_{j+1}, \ldots, x_k\}^{d}_{M_0}[\text{orig } O] \text{ in } P)\end{array}\right\} & \triangleq & \textsc{ac}(M) \cup \textsc{ac}(P)
\end{array}$$

Encoding Algorithm $(\!|\cdot|\!)$:

$$
\begin{aligned}
(\!|n|\!) &\triangleq n \\
(\!|x|\!) &\triangleq x \\
\left.\begin{array}{r}(\!|\{M_1,\ldots,M_k\}^{o}_{M_0} \\ [\text{dest } \{d_0,\ldots,d_n\}]|\!)\end{array}\right\} &\triangleq \{(\!|M_1|\!),\ldots,(\!|M_k|\!),o\}_{(\!|M_0|\!)} \\
(\!|0|\!) &\triangleq 0 \\
(\!|\langle M_1,\ldots,M_k\rangle.P|\!) &\triangleq \langle (\!|M_1|\!),\ldots,(\!|M_k|\!)\rangle \\
(\!|(M_1,\ldots,M_j;x_{j+1},\ldots,x_k).P|\!) &\triangleq ((\!|M_1|\!),\ldots,(\!|M_j|\!);x_{j+1},\ldots,x_k).(\!|P|\!) \\
(\!|P \mid Q|\!) &\triangleq (\!|P|\!) \mid (\!|Q|\!) \\
(\!|(\nu\, n)P|\!) &\triangleq (\nu\, n : \text{Un})(\!|P|\!) \\
(\!|!P|\!) &\triangleq !(\!|P|\!) \\
\left.\begin{array}{r}(\!|\text{decrypt } M \text{ as } \{M_1,\ldots,M_j; \\ x_{j+1},\ldots,x_k\}^{d}_{M_0} \\ [\text{orig } \{o_1,\ldots,o_n\}] \text{ in } P|\!)\end{array}\right\} &\triangleq \left\{\begin{array}{l}\text{decrypt } (\!|M|\!) \text{ as } \{(\!|M_1|\!),\ldots,(\!|M_j|\!); \\ x_{j+1},\ldots,x_k,y\}_{(\!|M_0|\!)} \text{ in end}(y,d).(\!|P|\!)\end{array}\right.
\end{aligned}
$$

Example 4.1 (Example of Process Encoding).
As an example of process encoding from LySa to Typed LySa consider the following process P_{Sys}:

$$
\begin{aligned}
P_A(K) &\triangleq (\nu\, m)\langle \{m\}^{o_A}_K[\text{dest } \{d_B\}]\rangle \\
P_B(K) &\triangleq (;s).\text{decrypt } s \text{ as } \{;m_s\}^{d_B}_K[\text{orig } \{o_A\}] \text{ in } 0 \\
P_C(K) &\triangleq (;t).\text{decrypt } t \text{ as } \{;m_t\}^{d_C}_K[\text{orig } \{o_D\}] \text{ in } 0 \\
P_{Sys}(K) &\triangleq (\nu\, K)(P_A(K) \mid P_B(K) \mid P_C(K))
\end{aligned}
$$

This process has three principals P_A, P_B and P_C. Here, P_A outputs an encrypted message meant for P_B and P_C. The principal P_B is willing to recieve the message, however, P_C is not, it will only accept messages to come from an unused crypto-point.

Now for the encoding of this process we need to use the auxiliary relation on all encryptions, in this case, we only have one:

$$\text{AC}(\{m\}^{o_A}_K[\text{dest } \{d_B, d_C\}]) = \{(o_A, d_B), (o_A, d_C)\}.$$

The encoded process is then as follows:

$$
\begin{aligned}
(\!|P_A(K)|\!) &= (\nu\, m : \mathrm{Un}).\langle\{m, o_A\}_K\rangle \\
(\!|P_B(K)|\!) &= (;s).\mathrm{decrypt}\ s\ \mathrm{as}\ \{;m_s, x\}_K\ \mathrm{in}\ \mathrm{end}(x, d_B) \\
(\!|P_C(K)|\!) &= (;t).\mathrm{decrypt}\ t\ \mathrm{as}\ \{;m_t, x\}_K\ \mathrm{in}\ \mathrm{end}(x, d_C) \\
[\![P_{Sys}(K)]\!] &= \mathrm{begin!}(o_A, d_B).(\nu\, K : \mathrm{Un})((\!|P_A(K)|\!) \mid (\!|P_B(K)|\!) \mid (\!|P_C(K)|\!))
\end{aligned}
$$

◆

Because of the changes the encoding makes to encryptions, appending the crypto-point, this has some consequences on pattern matching. Pattern matches on encryptions in an encoded process has a dependency on the crypto-point which is now part of the encrypted message, while pattern matches done on encryptions in LySa is ignoring all crypto-point annotations. That is, in LySa the two encryptions $\{m\}_k^o$ and $\{m\}_k^d$ would match, while encoded versions $\{m, o\}_k$ and $\{m, d\}_k$ would not. This could be solved by introducing that pattern matches on encryptions would ignore the last term, however we choose not to do that since that would make Typed LySa too specialised for just this translation purpose and quite strange to use in other cases. Instead we introduce the following constraint:

Constraint 4.2 (Pattern Matches). *Only names may be used in pattern matches.*

4.2 Properties of the Encoding

For the rest of this section, we use Q for a process in LySa, and P for a process in Typed LySa.

Definition 4.3 (Well-formed Crypto-sets).
We say that a LySa process has well-formed crypto-sets, if for any encryption with crypto-point o and destination set $\{d_1, \dots, d_k\}$, it is the case that all decryptions with crypto-point $d_1, \dots, d_k$ has o in their origin set.

Theorem 4.4. *Any process can be rewritten such that it has well-formed crypto-sets without affecting the security property.*

Proof. Let Q be a process without well-formed crypto-sets. By Definition 4.3 (Well-formed Crypto-sets), not all encryptions with crypto-point o and destination set $\{d_1, \dots, d_k\}$ satisfy the requirement that all decryptions with crypto-point $d_1, \dots, d_k$ must have o in their origin set. By hypothesis, assume a decryption with crypto-point d does not satisfy this requirement. If Q can be reduced to a decryption of o at d, it will be the case that $\mathrm{RM}(o, O, d, \{d_1, \dots, d_k\}) = \textsc{false}$ because $d \notin \{d_1, \dots, d_k\}$. We rewrite the destination set of o such that d no longer occurs in $\{d_1, \dots, d_k\}$. Then $\mathrm{RM}(o, O, d, \{d_1, \dots, d_k\}) = \textsc{false}$ because $d \notin \{d_1, \dots, d_k\}$ and $o \notin O$.

Therefore, RM does not change at this point in a reduction sequence by this rewrite. Since crypto-points are unique, the rewrite does not affect RM in any other point, and the security property is thus preserved.

This procedure is repeated until all encryptions violating the requirement has been modified. Since there is a finite number of encryptions in Q, this is guaranteed to terminate. □

Lemma 4.5 (Substitution in Encodings).
$(\!|Q|\!)[x_1 \mapsto (\!|M_1|\!), \ldots, x_k \mapsto (\!|M_k|\!)] = (\!|Q[x_1 \mapsto M_1, \ldots, x_k \mapsto M_k]|\!)$.

Proof. By induction on the structure of $(\!|Q|\!)$. □

Lemma 4.6 (Equivalences in Encodings).

(a) If $Q \equiv Q'$ *then* $(\!|Q|\!) \equiv (\!|Q'|\!)$.

(b) If $(\!|Q|\!) \equiv P$ *then* $Q \equiv Q'$, *where* $(\!|Q'|\!) = P$.

Proof. We split the proof on each point depending on the equivalence rule. Since all but two of the structural equivalence rules in LySA are identical to the ones in Typed LySA, we will only examine those two rules:

(a) **(Struct Res):** Let $Q = (\nu\, n_1)(\nu\, n_2).Q''$.
By the encoding, $(\!|Q|\!) = (\nu\, n_1 : \mathrm{Un})(\nu\, n_2 : \mathrm{Un}).(\!|Q''|\!)$.
Let $Q' = (\nu\, n_2)(\nu\, n_1).Q''$.
By the encoding, $(\!|Q'|\!) = (\nu\, n_2 : \mathrm{Un})(\nu\, n_1 : \mathrm{Un}).(\!|Q''|\!)$.
By hypothesis of the lemma, $Q \equiv Q'$.
Since $\mathrm{fn}(\mathrm{Un}) = \emptyset$, then $n_1 \notin \mathrm{fn}(\mathrm{Un}) \wedge n_2 \notin \mathrm{fn}(\mathrm{Un})$ is satisfied.
Suppose the side condition $n_1 \neq n_2$ is satisfied.
The result then follows by (Typed Struct Res).
Suppose instead that $n_1 = n_2$.
Then $Q = Q'$ and by (Typed Struct Refl), $(\!|Q|\!) \equiv (\!|Q'|\!)$.

(Struct Extrusion): The only difference between (Struct Extrusion) and (Typed Struct Extrusion) is the definition of free names. In LySA , fn is defined in the same way as in LySA, but with the inclusion of free names from types as well. Since all restrictions are of type Un[1], and all begin and end events uses crypto-points which are not restricted, the extruded name will not be in the set of free names in the encoding if it is not in the set of free names in the unencoded process. The result then follows from (Typed Struct Extrusion).

(b) **(Typed Struct Res):** The result follow immediately from (Struct Res).

[1] In Chapter 5 (Type Inference on the Encoding), we will infer types different than Un, but these types will not include free names other than names for crypto-points.

(Typed Struct Extrusion): By the encoding algorithm and the definition of free names, $\text{fn}(P) \subseteq \text{fn}((\!|P|\!))$. Therefore, if the side condition of (Typed Struct Extrusion) on $(\!|P|\!)$ is satisfied, the side condition of (Struct Extusion) on P will be satisfied as well.

□

In the following, we put forward a way of simulating a LySa process in Typed LySa and vice versa. We call these simulations for soundness and completeness of the encoding, respectively. Any reduction in LySa can be matched with a corresponding reduction in Typed LySa. However, Typed LySa can perform the reduction (Typed End) which does not have a corresponding reduction in LySa. Therefore, we will introduce a new semantics for Typed LySa that only differs from the regular semantics in the fact that a decryption followed by an end event is reduced in the same rule.

Definition 4.7 (Reductions in Encodings).
Let $\rightarrow_{Enc}$ consist of the rules (Typed IO), (Typed Res), (Typed Equiv), (Typed Par) as well as the new rule (Typed Decr End) defined in the following way:

Reductions in Encodings $\rightarrow_{Enc}$:

(Typed Decr End)

$$\frac{}{\text{decrypt } (\{M_1, \ldots, M_k, \ell\}_{M_0} \text{ as } \{M_1, \ldots, M_k; x_{j+1}, \ldots, x_k, y\}_{M_0} \text{ in } \text{end}(y, \ell').P \rightarrow_{Enc} P[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k, y \mapsto \ell]}$$

Lemma 4.8 (Reductions in Encodings).

(a) If $P \rightarrow_{Enc} P'$, then $P \rightarrow^* P'$.

(b) Let $P = \gamma$.decrypt $\{M_1, \ldots, M_k, \ell\}$ as $\{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k, y\}_{M_0}$ in end$(y, \ell').Q'''$. If $(\!|Q|\!) \rightarrow^* P$, then $(\!|Q|\!) \rightarrow^*_{Enc} P$.

Proof. Proof proceeds by induction on the depth of the derivation. We split the proof on each point:

(a) Case (Typed IO), (Typed Res), (Typed Equiv), (Typed Par) are immediate. Case (Typed Decr End) follows by applying first (Typed Decr) followed by (Typed End).

(b) Case (Typed IO), (Typed Res), (Typed Equiv) and (Typed Par) are immediate. Case (Typed Begin) is not applicable as $(\!|\cdot|\!)$ does not produce begin events. Case (Typed Decr) and (Typed End) must always be

applied together because of the encoding. Therefore, (Typed Decr End) is applicable as well.

□

Lemma 4.9 (Soundness of Encoding). *Let Q be a LySA process, which satisfies Constraint 4.2 (Pattern Matches). If $Q \to Q'$, then $(\!|Q|\!) \to_{Enc} (\!|Q'|\!)$.*

Proof. Let Q and Q' be LySA processes and assume that $Q \to Q'$. Consider the LySA reduction used:

Case (LySA Par): Let $Q = Q_1 \mid Q_2$.
By the encoding, $(\!|Q|\!) = (\!|Q_1|\!) \mid (\!|Q_2|\!)$.
Let $Q' = Q_1' \mid Q_2$.
By the encoding, $(\!|Q'|\!) = (\!|Q_1'|\!) \mid (\!|Q_2|\!)$.
Assume $Q \to Q'$ by (LySA Par) because $Q_1 \to Q_1'$.
By inductive hypothesis, $(\!|Q_1|\!) \to_{Enc} (\!|Q_1'|\!)$.
By (Typed Par), $(\!|Q|\!) \to_{Enc} (\!|Q'|\!)$.

Case (LySA Res): Let $Q = (\nu\, n)Q'$.
By the encoding, $(\!|Q|\!) = (\nu\, n : \mathrm{Un})(\!|Q'|\!)$.
Let $Q' = (\nu\, n)Q''$.
By the encoding, $(\!|Q'|\!) = (\nu n : \mathrm{Un})(\!|Q''|\!)$.
Assume $Q \to Q'$ by (LySA Res) because $Q' \to Q''$.
By inductive hypothesis, $(\!|Q'|\!) \to_{Enc} (\!|Q''|\!)$.
By (Typed Res), $(\!|Q|\!) \to_{Enc} (\!|Q'|\!)$.

Case (LySA Equiv): This is a result of Lemma 4.6 (Equivalences in Encodings) part (a) and the inductive hypothesis.

Case (LySA IO): Let $Q = \langle M_1, \ldots, M_k\rangle.Q_1 \mid (M_1', \ldots, M_j'; x_{j+1}, \ldots, x_k).Q_2$.
By the encoding, $(\!|Q|\!) = \langle (\!|M_1|\!), \ldots, (\!|M_k|\!)\rangle.(\!|Q_1|\!) \mid ((\!|M_1'|\!), \ldots, (\!|M_j'|\!); x_{j+1}, \ldots, x_k).(\!|Q_2|\!)$.
Let $Q' = Q_1 \mid Q_2[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]$.
By the encoding, $(\!|Q'|\!) = (\!|Q_1|\!) \mid (\!|Q_2[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]|\!)$.
Assume $Q \to Q'$ by (LySA Par) because $\wedge_{i=1}^{j} \lfloor\!\lfloor M_i \rfloor\!\rfloor = \lfloor\!\lfloor M_i' \rfloor\!\rfloor$.
By Constraint 4.2 (Pattern Matches), $\wedge_{i=1}^{j} M_i = M_i'$.
By Lemma 4.5 (Substitution in Encodings),
$(\!|Q'|\!) = (\!|Q_1|\!) \mid (\!|Q_2|\!)[x_{j+1} \mapsto (\!|M_{j+1}|\!), \ldots, x_k \mapsto (\!|M_k|\!)]$.
By (Typed IO), $(\!|Q|\!) \to_{Enc} (\!|Q'|\!)$.

Case (LySA Decr): Let $Q =$ decrypt $\{M_1, \ldots, M_k\}_{M_0}^{\ell}[\text{dest } \mathcal{L}])$ as $\{M_1', \ldots, M_k'; x_{j+1}, \ldots, x_k\}_{M_0'}^{\ell'}[\text{orig } \mathcal{L}']$ in Q''
By the encoding, $(\!|Q|\!) =$ decrypt $\{(\!|M_1|\!), \ldots, (\!|M_k|\!), \ell\}_{(\!|M_0|\!)}$ as $\{(\!|M_1'|\!), \ldots, (\!|M_k'|\!); x_{j+1}, \ldots, x_k, y\}_{(\!|M_0'|\!)}$ in $\mathrm{end}(y, \ell').(\!|Q''|\!)$.
Let $Q' = Q''[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]$.

By the encoding, $(\!|Q'|\!) = (\!|Q''[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]|\!)$.
Assume $Q \to Q'$ by (LySa Decr) because $\wedge_{i=1}^{j} [\![M_i]\!] = [\![M'_i]\!]$.
By Constraint 4.2 (Pattern Matches), $\wedge_{i=1}^{j} M_i = M'_i$.
By Lemma 4.5 (Substitution in Encodings),
$(\!|Q'|\!) = (\!|Q''|\!)[x_{j+1} \mapsto (\!|M_{j+1}|\!), \ldots, x_k \mapsto (\!|M_k|\!)]$.
By (Typed Decr End), $(\!|Q|\!) \to_{Enc} (\!|Q'|\!)$.

□

Lemma 4.10 (Completeness of Encoding). *Let Q be a LySa process. If $(\!|Q|\!) \to_{Enc} P$, then $Q \to Q'$, where $(\!|Q'|\!) = P$.*

Proof. Consider the rule used:

Case (Typed Par): Let $Q = Q_1 \mid Q_2$.
By the encoding, $(\!|Q|\!) = (\!|Q_1|\!) \mid (\!|Q_2|\!)$.
Let $P = P' \mid (\!|Q_2|\!)$.
Assume $(\!|Q|\!) \to_{Enc} P$ by (Typed Par) because $(\!|Q_1|\!) \to_{Enc} P'$.
By inductive hypothesis, there is a Q'_1 such that $Q_1 \to Q'_1$ and $(\!|Q'_1|\!) = P'$.
Let $Q' = Q'_1 \mid Q_2$.
By (LySa Par), $Q \to Q'$.
By the encoding, $(\!|Q'|\!) = (\!|Q'_1|\!) \mid (\!|Q_2|\!)$.
By hypothesis, $(\!|Q'|\!) = P' \mid (\!|Q_2|\!)$.

Case (Typed Res): Let $Q = (\nu\, n)Q''$.
By the encoding, $(\!|Q|\!) = (\nu\, n : \mathrm{Un})(\!|Q''|\!)$.
Let $P = (\nu\, n : \mathrm{Un})P'$.
Assume $(\!|Q|\!) \to_{Enc} P$ by (Typed Res) because $(\!|Q''|\!) \to_{Enc} P'$.
By inductive hypothesis, there is a Q''' such that $Q'' \to Q'''$ and $(\!|Q'''|\!) = P'$.
Let $Q' = (\nu\, n)Q'''$.
By (LySa Res), $Q \to Q'$.
By the encoding, $(\!|Q'|\!) = (\nu\, n : \mathrm{Un})(\!|Q'''|\!)$.
By hypothesis, $(\!|Q'|\!) = (\nu\, n : \mathrm{Un})P'$.

Case (Typed Equiv): This is a result of Lemma 4.6 (Equivalences in Encodings) part (b) and the inductive hypothesis.

Case (Typed IO): Let $Q = \langle M_1, \ldots, M_k\rangle.Q_1 \mid (M'_1, \ldots, M'_j; x_{j+1}, \ldots, x_k).Q_2$.
By the encoding, $(\!|Q|\!) = \langle (\!|M_1|\!), \ldots, (\!|M_k|\!)\rangle.(\!|Q_1|\!) \mid ((\!|M_1|\!), \ldots, (\!|M_j|\!); x_{j+1}, \ldots, x_k).(\!|Q_2|\!)$.
Let $P = (\!|Q_1|\!) \mid (\!|Q_2|\!)[x_{j+1} \mapsto (\!|M_{j+1}|\!), \ldots, x_k \mapsto (\!|M_k|\!)]$.
Assume $(\!|Q|\!) \to_{Enc} P$ by (Typed IO).
Let $Q' = Q_1 \mid Q_2[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]$.
By (LySa IO), $Q \to Q'$.

By the encoding, $(\!|Q'|\!) = (\!|Q_1|\!) \mid (\!|Q_2[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]|\!)$.
By Lemma 4.5 (Substitution in Encodings),
$(\!|Q'|\!) = (\!|Q_1|\!) \mid (\!|Q_2|\!)[x_{j+1} \mapsto (\!|M_{j+1}|\!), \ldots, x_k \mapsto (\!|M_k|\!)]$.

Case (Typed Decr End): Let $Q =$ decrypt $\{M_1, \ldots, M_k\}^{\ell}_{M_0}[\text{dest } \mathcal{L}])$ as $\{M'_1, \ldots, M'_k; x_{j+1}, \ldots, x_k\}^{\ell'}_{M'_0}[\text{orig } \mathcal{L}']$ in Q''.
By the encoding, $(\!|Q|\!) =$ decrypt $\{(\!|M_1|\!), \ldots, (\!|M_k|\!)\}_{(\!|M_0|\!)}$ as $\{(\!|M_1|\!), \ldots, (\!|M_k|\!);\ x_{j+1}, \ldots, x_k\}_{(\!|M'_0|\!)}$ in $\text{end}(\ell, \ell').(\!|Q''|\!)$.
Let $P = (\!|Q''|\!)[x_{j+1} \mapsto (\!|M_{j+1}|\!), \ldots, x_k \mapsto (\!|M_k|\!)]$.
Assume $(\!|Q|\!) \rightarrow_{Enc} P$ by (Typed Decr End).
Let $Q' = Q''[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]$.
By (LySA Decr), $Q \rightarrow Q'$.
By the encoding and Lemma 4.5 (Substitution in Encodings),
$(\!|Q'|\!) = (\!|Q''|\!)[x_{j+1} \mapsto (\!|M_{j+1}|\!), \ldots, x_k \mapsto (\!|M_k|\!)]$.

□

Theorem 4.11 (Security Preservation). *A process Q with well-formed crypto-sets and satisfying Constraint 4.2 (Pattern Matches) guarantees dynamic authenticity if and only if $[\![Q]\!]$ is safe.*

Proof. We split the proof in two parts:

Case (Q does not guarantee dynamic authenticity):
Let $Q' = \gamma.$decrypt $\{M_1, \ldots, M_k\}^{\ell}_{M_0}[\text{dest } \mathcal{L}]$ as $\{M'_1, \ldots, M'_j; x_{j+1}, \ldots, x_k\}^{\ell'}_{M_0}[\text{orig } \mathcal{L}']$ in Q'''.
By hypothesis, $Q \rightarrow^* Q'$ and $Q' \not\rightarrow_{\text{RM}}$ because $\text{RM}(\ell, \mathcal{L}', \ell', \mathcal{L}) =$ FALSE.
By hypothesis, $\wedge^{j}_{i=1} \lfloor\!\lfloor M_i \rfloor\!\rfloor = \lfloor\!\lfloor M'_i \rfloor\!\rfloor$.
By Constraint 4.2 (Pattern Matches), $\wedge^{j}_{i=1} M_i = M'_i$.
By the encoding, $[\![Q]\!] = \gamma_{\text{AC}}.(\!|Q|\!)$.
By repeatedly applying Lemma 4.9 (Soundness of Encoding), $(\!|Q|\!) \rightarrow^*_{Enc} (\!|Q'|\!)$.
By repeatedly applying (Typed Begin), $\gamma_{\text{AC}}.(\!|Q|\!) \rightarrow^*_{Enc} \gamma_{\text{AC}}.(\!|Q'|\!)$.
By Lemma 4.8 (Reductions in Encodings) part (a), $\gamma_{\text{AC}}.(\!|Q|\!) \rightarrow^* \gamma_{\text{AC}}.(\!|Q'|\!)$.
By the encoding, $(\!|Q'|\!) = (\!|\gamma|\!).$decrypt $\{(\!|M_1|\!), \ldots, (\!|M_k|\!), \ell\}_{(\!|M_0|\!)}$ as $\{(\!|M'_1|\!), \ldots, (\!|M'_j|\!); x_{j+1}, \ldots, x_k, y\}_{(\!|M_0|\!)}$ in $\text{end}(y, \ell').(\!|Q'''|\!)$.
Let $\sigma = [x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k, y \mapsto \ell]$.
By (Typed Decr), (Typed Begin) and (Typed Res) where applicable, $\gamma_{\text{AC}}.(\!|Q'|\!) \rightarrow \gamma_{\text{AC}}.(\!|\gamma|\!).\text{end}(y, \ell')\sigma.(\!|Q'''|\!)\sigma$.
Since crypto-points are unique, $\text{end}(y, \ell')\sigma = \text{end}(\ell, \ell')$.
Because the encoding asserts $\text{end}(\ell, \ell')$, we must examine if the encoding ever produces a matching $\text{begin!}(\ell, \ell')$ assertion.
Since γ is an inert LySA process, $(\!|\gamma|\!)$ only consists of restrictions.

Since crypto-points are unique, we need only examine $\textsc{ac}(\{M_1, \ldots, M_k\}^{\ell}_{M_0}[\text{dest } \mathcal{L}])$ to check if (ℓ, ℓ') is contained.
By hypothesis, $\text{RM}(\ell, \mathcal{L}', \ell', \mathcal{L}) = \textsc{false}$, which means $\ell \notin \mathcal{L}' \vee \ell' \notin \mathcal{L}$.
Since Q has well-formed crypto-sets, it is the case that $\ell' \notin \mathcal{L}$.
Therefore, AC will never contain the pair (ℓ, ℓ').

Case ($[\![Q]\!]$ is not safe): Let $[\![Q]\!] = \gamma_{\textsc{ac}}.(\!|Q|\!)$.
Let $P = \gamma.\text{decrypt } \{M_1, \ldots, M_k, \ell\}_{M_0}$ as $\{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k, y\}_{M_0}$ in $\text{end}(y, \ell').(\!|Q'''|\!)$.
Let $\sigma = [x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k, y \mapsto \ell]$.
By hypothesis, $\gamma_{\textsc{ac}}.\gamma.\text{end}(y, \ell')\sigma.(\!|Q'''|\!)\sigma$ is an authenticity error.
End events are only produced directly following a decryption, so by hypothesis, assume $(\!|Q|\!) \rightarrow^* P$.
By Lemma 4.8 (Reductions in Encodings) part (b), $(\!|Q|\!) \rightarrow^*_{Enc} P$.
By repeatedly applying Lemma 4.10 (Completeness of Encoding), $Q \rightarrow^* Q'$, where $(\!|Q'|\!) = P$.
To prove that Q does not guarantee dynamic authenticity, cf. Definition 2.5 (Dynamic Authentication), we will prove that the reference monitor aborts on Q' because $\text{RM}(\ell, \mathcal{L}', \ell', \mathcal{L}) = \textsc{false}$.
By hypothesis, $\text{begin!}(\ell, \ell') \notin \gamma_{\textsc{ac}}.\gamma$.
Then, $\textsc{ac}(\{M_1, \ldots, M_k\}^{\ell}_{M_0}[\text{dest } \mathcal{L}])$ will not contain the pair (ℓ, ℓ').
Therefore, $\ell \notin \mathcal{L}' \vee \ell' \notin \mathcal{L}$, and consequently $\text{RM}(\ell, \mathcal{L}', \ell', \mathcal{L}) = \textsc{false}$.

□

Chapter 5

Type Inference on the Encoding

Given a LySA process annotated with crypto-points and the encoding algorithm, the question arises whether or not we can match the control flow analysis on the encoded process with a type check. That is, given a solution to a control flow analysis, does there exist a solution to the type check? A solution to a type check in this context is an annotation of types such that the encoded process becomes well-typed under Theorem 3.18 (Robust Safety).

5.1 Constraints

We cannot create a solution to an arbitrary encoded process, since our type system introduces certain issues because of the use of embedded types. This means that we have to restrict the number of processes to which we can find a solution, to those which satisfies the constraints introduced in this section. The issue that gives rise to these first three constraints comes from the use of embedded types, which imposes strict requirements on the messages which are allowed to be encrypted under each key.

Constraint 5.1. *Messages must have the same number of elements when encrypted with the same key.*

Messages violating Constraint 5.1 can in most cases be rewritten by "padding" the message sequence with public names not used anywhere else in the process. These names can then be caught by pattern-matching in the decrypt-process.

Constraint 5.2. *If a key K_1 is used to encrypt another key K_2 in place i in the message, then key K_1 must never be used to encrypt anything else than K_2 in place i.*

Since we have no notion of subtyping in the type system, we need constraint 5.2 to ensure that the terms which are encrypted in a given position using the same key, are of the same type.

Constraint 5.3. *If a key K_1 is used to encrypt a key K_2 either directly or through layered encryptions, then K_2 must never be used to encrypt K_1 directly or through layered encryptions.*

To illustrate way Constraint 5.3 is needed, consider the two encryptions $\{K_1\}_{K_2}$ and $\{K_2\}_{K_1}$. If we try to type the keys, we see the problem. Typing K_1, we get $\text{Key}(x_1 : ?)[]$. We cannot determine the type at the ? spot before we examine the type for K_2, which is then $\text{Key}(x_1 : \text{Key}(x_1 : ?)[])[]$ and we see the infinite loop of embedded types.

Constraint 5.4. *Only names are used as keys.*

Constraint 5.4 is needed since encryptions are always of type Un, and we cannot add latent effect to this type. Therefore, we will not be able to "transfer" the effects from the point of encryption to the point of decryption.

Constraint 5.5. *All names must be able to flow to all variables.*

The final constraint has nothing to do with the embedded types, but instead related to the core quality of control flow analysis. Namely that it knows which values flow where. Consider the following LySA process:

$$
\begin{array}{rcl}
P_A(K) & \triangleq & (\nu\, m)\langle\{\{m\}_K^{o'_A}[\text{dest } \{d_X\}]\}_K^{o_A}[\text{dest } \{d_B\}]\rangle \\
P_B(K) & \triangleq & (;s).\text{decrypt } s \text{ as } \{;m_s\}_K^{d_B}[\text{orig } \{o_A\}] \text{ in } 0 \\
P_{Sys}(K) & \triangleq & (\nu\, K)(P_A(K) \,|\, P_B(K))
\end{array}
$$

It is clear that the "outer" encryption poses no security risk, but the inner encryption if ever decrypted would make the process insecure. Fortunately, in this case, it is never decrypted, so the process is safe, we know this in CFA, but we do not have that knowledge in our type system. Whenever something is encrypted with a key we have to assume that any decryptions performed by this key could be decrypting any of the messages encrypted. Therefore, if an encryption never reaches a matching decryption, we cannot guarantee the same safety result with type checking. Because of this, we use Constraint 5.5 to ensure that this does not happen.

Lastly the constraint from the previous chapter, 4.2, regarding that only names may be used in pattern matches is still assumed to hold on any processes on which the type inference is attempted. Otherwise, the encoding will not be possible in the first place.

5.2 Auxiliary Relations

Before we present the algorithm for type inference under our list of constraints, we need two auxiliary relations to help with the main tasks.

The relation $\mathcal{K}_{enc} : \lfloor\mathcal{N}\rfloor \rightarrow \wp(\mathcal{V}^*)$ gathers what each key is used to encrypt. We need this information to create the embedded types.

The second auxiliary relation $\mathcal{K}_{dest} : \lfloor\mathcal{N}\rfloor \rightarrow \wp(\mathcal{C})$ is used to gather information to create the embedded assertions. For all keys, we gather a list of which destinations they are used to decrypt messages at.

We use the following notation for updating a one-to-many relation:

$$f[n \overset{+}{\mapsto} r] \triangleq \begin{cases} f[n \mapsto r], & \text{if } n \notin \text{Dom}(f); \\ f[n \mapsto (r \cup f(n))], & \text{if } n \in \text{Dom}(f). \end{cases}$$

Since a key or message can be a variable, we use ρ to find what names may be bound to it. As a convenience, assume that $\rho(\lfloor n \rfloor) = \lfloor n \rfloor$ for all $n \in \mathcal{N}$. Also assume that $\rho(\lfloor M \rfloor) = M$ if M is an encryption. We will present the auxiliary relations in the form of transition relations working on an encoded process. The transition format is $\mathcal{K}_{enc}, \mathcal{K}_{dest}, P^M \rightarrow \mathcal{K}_{enc}, \mathcal{K}_{dest}$, where P^M is either a term M or a process P

Creating Auxiliary Relations $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ (Given ρ):

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ Name)

$$\frac{}{\mathcal{K}_{enc}, \mathcal{K}_{dest}, n \rightarrow \mathcal{K}_{enc}, \mathcal{K}_{dest}}$$

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ Variable)

$$\frac{}{\mathcal{K}_{enc}, \mathcal{K}_{dest}, x \rightarrow \mathcal{K}_{enc}, \mathcal{K}_{dest}}$$

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ Enc)

$$\frac{\mathcal{K}_{enc}, \mathcal{K}_{dest}, M_1 \rightarrow \mathcal{K}^1_{enc}, \mathcal{K}^1_{dest} \quad \cdots \quad \mathcal{K}^{k-1}_{enc}, \mathcal{K}^{k-1}_{dest}, M_k \rightarrow \mathcal{K}^k_{enc}, \mathcal{K}^k_{dest}}{\begin{array}{c}\mathcal{K}_{enc}, \mathcal{K}_{dest}, \{M_1, \ldots, M_k\}_{M_0} \rightarrow \\ \mathcal{K}^k_{enc}[v_1 \overset{+}{\mapsto} (\overline{v_1} \times \cdots \times \overline{v_k}), \ldots, v_i \overset{+}{\mapsto} (\overline{v_1} \times \cdots \times \overline{v_k})], \mathcal{K}^k_{dest}\end{array}}$$

where $\rho(\lfloor M_0 \rfloor) = \{v_1, \ldots, v_i\} \wedge \rho(\lfloor M_1 \rfloor) = \overline{v_1} \quad \cdots \quad \rho(\lfloor M_k \rfloor) = \overline{v_k}$

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ Out)

$$\frac{\begin{array}{c}\mathcal{K}_{enc}, \mathcal{K}_{dest}, M_1 \rightarrow \mathcal{K}^1_{enc}, \mathcal{K}^1_{dest} \quad \cdots \quad \mathcal{K}^{k-1}_{enc}, \mathcal{K}^{k-1}_{dest}, M_k \rightarrow \mathcal{K}^k_{enc}, \mathcal{K}^k_{dest} \\ \mathcal{K}^k_{enc}, \mathcal{K}^k_{dest}, P \rightarrow \mathcal{K}'_{enc}, \mathcal{K}'_{dest}\end{array}}{\mathcal{K}_{enc}, \mathcal{K}_{dest}, \langle M_1, \ldots, M_k \rangle.P \rightarrow \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}$$

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ In)

$$\frac{\mathcal{K}_{enc}, \mathcal{K}_{dest}, P \rightarrow \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}{\mathcal{K}_{enc}, \mathcal{K}_{dest}, (M_1, \ldots, M_j; x_1, \ldots, x_k).P \rightarrow \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}$$

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ Par)
$$\frac{\mathcal{K}_{enc}, \mathcal{K}_{dest}, P_1 \to \mathcal{K}''_{enc}, \mathcal{K}''_{dest} \quad \mathcal{K}''_{enc}, \mathcal{K}''_{dest}, P_2 \to \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}{\mathcal{K}_{enc}, \mathcal{K}_{dest}, (P_1 \mid P_2) \to \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}$$

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ Res)
$$\frac{\mathcal{K}_{enc}, \mathcal{K}_{dest}, P \to \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}{\mathcal{K}_{enc}, \mathcal{K}_{dest}, (\nu\, n : \mathrm{Un})P \to \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}$$

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ Repl)
$$\frac{\mathcal{K}_{enc}, \mathcal{K}_{dest}, P \to \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}{\mathcal{K}_{enc}, \mathcal{K}_{dest}, !P \to \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}$$

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ Begin)
$$\frac{\mathcal{K}_{enc}, \mathcal{K}_{dest}, P \to \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}{\mathcal{K}_{enc}, \mathcal{K}_{dest}, \text{ begin!}(\vec{M}).P \to \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}$$

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ Nil)
$$\frac{}{\mathcal{K}_{enc}, \mathcal{K}_{dest}, 0 \to \mathcal{K}_{enc}, \mathcal{K}_{dest}}$$

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ End)
$$\frac{\mathcal{K}_{enc}, \mathcal{K}_{dest}, P \to \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}{\mathcal{K}_{enc}, \mathcal{K}_{dest}, \text{ End}(\vec{M}).P \to \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}$$

(Rel $\mathcal{K}_{enc}, \mathcal{K}_{dest}$ Decr)
$$\frac{\mathcal{K}_{enc}, \mathcal{K}_{dest}, M \to \mathcal{K}''_{enc}, \mathcal{K}''_{dest} \quad \mathcal{K}''_{enc}, \mathcal{K}''_{dest}, P \to \mathcal{K}'_{enc}, \mathcal{K}'_{dest}}{\begin{array}{c}\mathcal{K}_{enc}, \mathcal{K}_{dest}, \text{ decrypt } M \text{ as } \{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k\}_{M_0} \text{ in end}(x_k, d).P \to \\ \mathcal{K}'_{enc}, \mathcal{K}'_{dest}[v_1 \overset{+}{\mapsto} d, \ldots, v_i \overset{+}{\mapsto} d] \\ \text{where } \rho(\lfloor M_0 \rfloor) = \{v_1, \ldots, v_i\}\end{array}}$$

5.3 Constructing the Solution to a Type Check

The last relation $\Gamma : \mathcal{N} \to T$ uses the information gathered by the auxiliary relations to create types for any names introduced in a TYPED LYSA process. While creating types for keys we can not always finish these since we need to know the types of what will be encrypted with the key. Therefore, if we are missing the type of a certain name n to finish the type of a key, we add instead a placeholder H_n for that name. This placeholder is then replaced everywhere in the relation whenever we find the actual type of n. This update is done as follows:

$$f[n \overset{*}{\mapsto} r] = g, \quad \text{where} \quad g(x) = \begin{cases} r, & \text{if } x = n; \\ f(x)[r/H_n], & \text{otherwise.} \end{cases}$$

The two primary rules for the type inference algorithm are Sol Res Non-key and Sol Res Key. The first rule is for when the name is not used as

a key which we know from relation $\mathcal{K}_{enc}$ and simply updates this name to be of type Un. The second rule is for when the name is used as a key and looks at what names this key is used to encrypt at which places. Because of Constraint 5.2 we can safely assume that everything encrypted in a spot are of the same type. This means we just look for if we know the type of one of the names encrypted in a certain spot. If we know it, we insert it, otherwise we insert the placeholder. In the end we get a relation from names to their types which we can then insert in our encoded process instead of the type Un. We present the rules for creating Γ with transition rules again and the transition format is $\Gamma, P \rightarrow \Gamma$ where P is an encoded LySA process.

Construction of Γ (given $\mathcal{K}_{enc}, \mathcal{K}_{dest}$):

(Sol Repl)
$$\frac{\Gamma, P \rightarrow \Gamma'}{\Gamma, !P \rightarrow \Gamma'}$$

(Sol Par)
$$\frac{\Gamma, P_1 \rightarrow \Gamma'' \quad \Gamma'', P_2 \rightarrow \Gamma'}{\Gamma, (P_1 \mid P_2) \rightarrow \Gamma'}$$

(Sol Out)
$$\frac{\Gamma, P \rightarrow \Gamma'}{\Gamma, \langle M_1, \ldots, M_k \rangle.P \rightarrow \Gamma'}$$

(Sol In)
$$\frac{\Gamma, P \rightarrow \Gamma'}{\Gamma, (M_1, \ldots, M_j, x_{j+1}, \ldots, x_k).P \rightarrow \Gamma'}$$

(Sol Begin)
$$\frac{\Gamma, P \rightarrow \Gamma'}{\Gamma, \text{begin!}(\vec{M}).P \rightarrow \Gamma'}$$

(Sol End)
$$\frac{\Gamma, P \rightarrow \Gamma'}{\Gamma, \text{end}(\vec{M}).P \rightarrow \Gamma'}$$

(Sol Nil)
$$\frac{}{\Gamma, 0 \rightarrow \Gamma}$$

(Sol Decr)
$$\frac{\Gamma, P \rightarrow \Gamma'}{\Gamma, \text{decrypt } M \text{ as } \{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k\}_{M_0} \text{ in } P \rightarrow \Gamma'}$$

(Sol Res Non-key)
$$\frac{\lfloor n \rfloor \notin \text{Dom}(\mathcal{K}_{enc}) \quad \Gamma[\lfloor n \rfloor \overset{*}{\mapsto} \text{Un}], P \rightarrow \Gamma'}{\Gamma, (\nu\, n : \text{Un})P \rightarrow \Gamma'}$$

(Sol Res Key)

$$\frac{\lfloor n \rfloor \in \text{Dom}(\mathcal{K}_{enc}) \quad \Gamma[\lfloor n \rfloor \overset{*}{\mapsto} \text{Key}(x_1 : T'_1, \ldots, x_k : T'_k, x_{k+1} : \text{Un})[\vec{B}]], P \to \Gamma'}{\Gamma, (\nu\, n : \text{Un})P \to \Gamma'}$$

$$\begin{gathered}\text{where } \mathcal{K}_{dest}(n) = \{d_1, \ldots, d_n\} \\ \text{and } \vec{B} = \text{!begun}(x_{k+1}, d_1), \ldots, \text{!begun}(x_{k+1}, d_n) \\ \text{and } \mathcal{K}_{enc}(\lfloor n \rfloor) = \{(v_{1,1}, \ldots, v_{1,k}), \ldots, (v_{j,1}, \ldots, v_{j,k})\} \\ \text{and } \forall i \in [1..k] : (v_{i,1} \in \text{dom}(\Gamma) \Rightarrow T'_i = \Gamma(v_{i,1})) \wedge \\ ((v_{i,1} \notin \text{dom}(\Gamma) \ \wedge \ v_{i,1} \in \mathcal{N}) \Rightarrow T'_i = H_{v_{i,1}}) \wedge \\ ((v_{i,1} \notin \text{dom}(\Gamma) \ \wedge \ v_{i,1} \notin \mathcal{N}) \Rightarrow T'_i = \text{Un})\end{gathered}$$

Definition 5.6 (Type Substitution).
Let Q be a Typed LySa process, then Q_Γ is the same process, except any restrictions $(\nu\, n : T)$ are changed such that $T = \Gamma(\lfloor n \rfloor)$.

Conjecture 5.7 (Type Inference on the Encoding). *Given a* LySa *process P that conforms to Constraints 5.1 to 5.5, and has a analysis $(\rho, \kappa, \emptyset)$, such that $(\rho, \kappa) \models P : \emptyset$ and $(\rho, \kappa, \emptyset)$ satisfies $\mathcal{F}^{DY}$, then $\vec{n} : \text{Un} \vdash [\![P]\!]_\Gamma$, where $\vec{n} = \text{fn}([\![P]\!]_\Gamma)$.*

Example 5.8 (Infering Types).
Consider the following encoded process:

$$\begin{aligned} (\!|P_A(K)|\!) &= (\nu\, m : \text{Un}).\langle\{m, o_A\}_K\rangle \\ (\!|P_B(K)|\!) &= (; s).\text{decrypt } s \text{ as } \{; m_s, x\}_K \text{ in end}(x, d_B) \\ [\![P_{Sys}(K)]\!] &= \text{begin!}(o_A, d_B).(\nu\, K : \text{Un})((\!|P_A(K)|\!) \mid (\!|P_B(K)|\!)) \end{aligned}$$

with

ρ :

$\lfloor s \rfloor$	$\{\lfloor m \rfloor\}_{\lfloor K \rfloor}$
$\lfloor m_s \rfloor$	$\lfloor m \rfloor$

First step is to create the auxiliary relations $\mathcal{K}_{enc}$ and $\mathcal{K}_{dest}$. We se that there is only one encryption and only one key is ever used for that, so we get the following:

$\mathcal{K}_{enc}$:

$\lfloor K \rfloor$	$\{\lfloor m \rfloor\}$

and

$$\mathcal{K}_{dest}: \quad \begin{array}{c|c} \lfloor K \rfloor & \{d_B\} \end{array}$$

We are now in a position to generate the types. As we pass the restriction for K we have $\lfloor K \rfloor \in \text{dom}(\mathcal{K}_{enc})$ so we use the rule (Sol Res Key). We have $\mathcal{K}_{dest}(\lfloor K \rfloor) = \{d_B\}$ so the embedded assertion $\vec{B} = !\text{begun}(x_2, d_B)$. In creating the embedded types, we have $\mathcal{K}_{enc} = \{\lfloor m \rfloor\}$. For $\lfloor m \rfloor$ we have that it is not in dom(Γ) but it is a name, that means its merely an unfinished type so we use the placeholder H_m. Together this gives $\Gamma(\lfloor K \rfloor) = \text{Key}(x_1 : H_{\lfloor m \rfloor}, x_2 : \text{Un})[!\text{begun}(x_2, d_B)]$. For the restriction of m we see that this is never used as a key and therefore update $\Gamma(\lfloor m \rfloor) = \text{Un}$ and at the same time update all occurrences of the placeholder $H_{\lfloor m \rfloor}$ in any values of Γ which gives us $\Gamma(\lfloor K \rfloor) = \text{Key}(x_1 : \text{Un}, x_2 : \text{Un})[!\text{begun}(x_2, d_B)]$.

Using these inferred types we can then update the encoded process to:

$$\begin{array}{rcl} (\!|P_A(K)|\!)_\Gamma & = & (\nu\, m : \text{Un}).\langle \{m, o_A\}_K \rangle \\ (\!|P_B(K)|\!)_\Gamma & = & (;s).\text{decrypt } s \text{ as } \{;m_s, x\}_K \text{ in end}(x, d_B) \\ [\![P_{Sys}(K)]\!]_\Gamma & = & \text{begin!}(o_A, d_B).(\nu\, K : \text{Key}(x_1 : \text{Un}, x_2 : \text{Un})[!\text{begun}(x_2, d_B)]) \\ & & ((\!|P_A(K)|\!)_\Gamma \mid (\!|P_B(K)|\!)_\Gamma) \end{array}$$

◆

Chapter 6

Control Flow Analysis with Correspondences

In the light of the results from the previous chapters, we propose a new control flow analysis based on the formalisation by Nielson and Nielson et. al., but using the ideas from Gordon and Jeffrey's type systems. This new method is capable of verifying the correspondence property through the use of begin/end annotations, but is stronger in the sense that it can verify many of the processes rejected by our type system, while still capable of verifying the same processes which are verifiable by our type system. Even more interesting, this new analysis method is also able to verify all processes which can be verified by the original control flow analysis. We will refer to this method as *CFAC* for *Control Flow Analysis with Correspondences*.

6.1 Design

In the original control flow analysis, the error component ψ is *verbose* in the sense that it is a global component (it is the same at each step of the analysis). In CFAC, this component is *succinct* (it may be different at different steps of the analysis) such that begin events may be collected in much the same way as in the type system.

The main idea behind this new method is to annotate encryptions found in ρ and κ with latent effects in much the same way as latent effects works in our type system. One of the advantages to this approach, compared to annotating the keys of encryptions as in the type system, is that we can perform a much finer grained "transfer" of effects. When verifying an encryption, the annotated latent effects must be included in the (local) ψ component. When decrypting, we know from the ρ component which encryptions may flow to this point and be successfully decrypted. By examining the annotations on these encryptions, we have information on which effects held at the time of encryption. The maximum set of effects which we can be sure would hold

at this point would therefore be the intersection of all latent effects from the annotations. As an example, consider the following protocol narration:

Protocol Narration with Begin/End Events:

(1) A : begin!(n)

(2) $A \rightarrow$: $\{n\}_K$

(3) $\rightarrow B$: x

(4) B: decrypt x as $\{x_n\}_K$

(5) B: end(x_n)

Since begin(n) is expected to hold at the time of encryption, we will annotated the encryption with the latent effect !begun($\lfloor n \rfloor$) in the analysis. This analysis (ρ, κ, ψ), where $\psi = \emptyset$, is defined as in the following table:

ρ:	$\lfloor x \rfloor$	$\mapsto$	$\{\lfloor n \rfloor\}_{\lfloor K \rfloor}$[!begun($\lfloor n \rfloor$)]
	$\lfloor x_n \rfloor$	$\mapsto$	$\lfloor n \rfloor$
κ:	$\{\lfloor n \rfloor\}_{\lfloor K \rfloor}$[!begun($\lfloor n \rfloor$)]		

The steps needed to verify this analysis, corresponding to the steps in the protocol narration, would be as follows:

(1) Place the effect !begun($\lfloor n \rfloor$) in the local ψ component.

(2) Output as regular LySA, but annotate encryptions with the local ψ component. Since the local ψ component consists of a !begun($\lfloor n \rfloor$)] effect, then verify that $\langle\{\lfloor n \rfloor\}_{\lfloor K \rfloor}$[!begun($\lfloor n \rfloor$)]$\rangle$ is in κ.

(3) Input as regular LySA.

(4) Find the intersection of the sets of effect annotated on the terms which may be decrypted at this point. Since only $\{\lfloor n \rfloor\}_{\lfloor K \rfloor}$[!begun($\lfloor n \rfloor$)] may flow to $\lfloor x \rfloor$, place the effect !begun($\lfloor n \rfloor$) in the local ψ component.

(5) Find the set of effects which must hold at this end event. Since only $\lfloor n \rfloor$ may flow to $\lfloor x_n \rfloor$, verify that !begun($\lfloor n \rfloor$) is in the local ψ component.

This approach is sufficient to verify the encoding of a LySA process Q if the original control flow analysis is able to verify Q. However, it is not able to verify processes using variables in begin events. Consider the event begin!(x), where x is a variable bound at some earlier point. Assume the two values n, m may flow to x. This, however, does not mean we may put both !begun(n) and !begun(m) in the local ψ component since only *one* of

these events will take place, and not both. Therefore, we will extend the analysis in much the same way as types were made dependent in the type system (eg. by substitutions of variables in the set of latent effects). This solves this problem of handling variables in begin events, at the cost of being a bit more complex to reason about.

The approach we have chosen to handle variables in begin events is to introduce *flow-positions* of the form $\$i$, where i is a positive integer. It is used in annotations to reason about a term at location i in an encrypted message sequence. Instead of just annotating encryptions with the local ψ component, we will perform the substitution σ_{out} on ψ such that all variables at position i being encrypted gets substituted with $\$i$. Consider the process

$$\text{begin}(x).\langle\{x\}_K\rangle$$

Since $\psi = !\text{begin}(\lfloor x \rfloor)$ at the point of encryption, the annotation on the encryptions would be $\psi\sigma_{out} = \text{begin}(\$1)$. Assuming only two names n and m may flow to x, then we would include the two values $\langle\{n\}_K[!\text{begun}(\$1)]\rangle$ and $\langle\{m\}_K[!\text{begun}(\$1)]\rangle$ in κ. Now consider a decryption

$$\text{decrypt } y \text{ as } \{z\}_K \text{ in end}(z).$$

and assume only the earlier mentioned encryption may flow to y. By examining the intersection of annotations, we know that !begun($1) holds. Again we perform a substitution $\sigma_{in} = [\$0 \mapsto \lfloor M_0 \rfloor, \ldots, \$k \mapsto \lfloor M_k \rfloor]$, where M_i corresponds to the variables or pattern-matches in the decryption. In this case, $\sigma_{in} = [\$0 \mapsto \lfloor z \rfloor]$ such that we may place $!\text{begun}(\lfloor z \rfloor)$ in the local ψ component. This procedure is more or less the same as in the type system, where variables in dependent types are substituted for the actual values.

6.2 Analysis Rules

The syntax and semantics are similar to TYPED LYSA with the removal of type annotations. For completeness, they can be found in Appendix A (CORRESPONDENCE LYSA). The security property is defined as in TYPED LYSA; see Definition 3.16 (Safety and Robust Safety).

We extend $\lfloor\cdot\rfloor$ to work on encryptions, i.e.

$$\lfloor\{M_1, \ldots, M_k\}_{M_0}\rfloor \triangleq \{\lfloor M_1\rfloor, \ldots, \lfloor M_k\rfloor\}_{\lfloor M_0\rfloor}$$

We will use the syntax ψ for a set of begun assertions. To avoid ambiguity on the judgements, we will use the symbol $\models$ as the judgements of the CFAC. This leads to the definition of a set of modifications and additions to the original CFA rules, which can be found on the next page. Since these rules are a bit more complex than the one for the type system, we will explain each one in detail.

The rule (CFAC Begin) verifies a begin event only using names, followed by a process P. The only thing which is needed is to update the local ψ component to include this new begun effect.

The rule (CFAC End) verifies an end event of terms $(M_1, \ldots, M_k)$ followed by a process P. There are two clauses which will conclude the validity of an end event. Either the corresponding begun effect must be in ψ, or for each sequence of values that these terms may evaluate to, the corresponding begun effects must be included in the local ψ component. This allows us to verify the process begin!(a).begin!$(b) \ldots$ end(x) if only a and b may flow to x, while still being able to verify processes using flow-positions.

The rule (CFAC Encryption) verifies that an encryption of terms $(M_1, \ldots, M_k)$ with key M_0 may evaluate to a set of encryptions v. For each sequence of values that the terms may evaluate to, the corresponding encryption, annotated with the local ψ component under the substitution for flow-positions, must be in v.

The rule (CFAC Decrypt) performs the actions of the original (CFA Decrypt) rule, but also verifies that a set of begun effects ψ' may be included with ψ when verifying the rest of the process. This ψ' component is the intersection of the sets of annotations on each value that may result in a decryption. For instance, if $\{V_1, \ldots, V_k\}_{V_0}[\psi'']$ may be decrypted at this point, we require that $\psi' \in \psi''$. As in the rule for encryption, we perform a substitution for the use of flow-positions.

Control Flow Analysis with Correspondences of Terms:

(CFAC Name)

$$\frac{\lfloor n \rfloor \in v}{(\rho, \psi) \models n : v}$$

(CFAC Variable)

$$\frac{\rho(\lfloor x \rfloor) \subseteq v}{(\rho, \psi) \models x : v}$$

(CFAC Encryption)

$$\frac{\begin{array}{c}\wedge_{i=0}^{k} \, (\rho, \psi) \models M_i : v_i \wedge \\ \forall V_0, V_1, \ldots, V_k : \wedge_{i=0}^{k} V_i \in v_i \Rightarrow \{V_1, \ldots, V_k\}_{V_0}[\psi\sigma] \in v\end{array}}{(\rho, \psi) \models \{M_1, \ldots, M_k\}_{M_0} : v}$$

where $\wedge_{i=0}^{k}(M_i \in \mathcal{X} \Rightarrow (\lfloor M_i \rfloor \mapsto \$i) \in \sigma)$.

Control Flow Analysis with Correspondences of Processes:

(CFAC Nil)

$$\frac{}{(\rho, \kappa) \models 0 : \psi}$$

(CFAC Res)

$$\frac{(\rho, \kappa) \models P : \psi}{(\rho, \kappa) \models (\nu\, n)P : \psi}$$

(CFAC Repl)

$$\frac{(\rho, \kappa) \models P : \psi}{(\rho, \kappa) \models \; !P : \psi}$$

(CFAC Par)

$$\frac{(\rho,\kappa) \models P_1 : \psi \;\wedge\; (\rho,\kappa) \models P_2 : \psi}{(\rho,\kappa) \models P_1 \mid P_2 : \psi}$$

(CFAC Begin)

$$\frac{(\rho,\kappa) \models P : \psi \cup \text{!begun}(\lfloor M_1 \rfloor, \ldots, \lfloor M_k \rfloor)}{(\rho,\kappa) \models \text{ begin!}(M_1, \ldots, M_k).P : \psi}$$

(CFAC End)

$$\frac{\begin{array}{c}(\rho,\kappa) \models P : \psi \wedge (\text{!begun}(\lfloor M_1 \rfloor, \ldots, \lfloor M_k \rfloor) \in \psi \vee \\ (\wedge_{i=1}^{k}\, (\rho,\psi) \models M_i : v_i \;\wedge\; \forall\, V_1, \ldots, V_k : \wedge_{i=0}^{k} V_i \in v_i \Rightarrow \text{!begun}(V_1, \ldots, V_k) \in \psi))\end{array}}{(\rho,\kappa) \models \text{ end}(M_1, \ldots, M_k).P : \psi}$$

(CFAC Output)

$$\frac{\begin{array}{c}\wedge_{i=1}^{k}\, (\rho,\psi) \models M_i : v_i \;\wedge\; (\rho,\kappa) \models P : \psi \wedge \\ \forall V_1, \ldots, V_k : \wedge_{i=1}^{k}\, V_i \in v_i \Rightarrow \langle V_1, \ldots, V_k \rangle \in \kappa\end{array}}{(\rho,\kappa) \models \langle M_1, \ldots, M_k \rangle.P : \psi}$$

(CFAC Input)

$$\frac{\begin{array}{c}\wedge_{i=1}^{j}\, (\rho,\psi) \models M_i : v_i \;\wedge\; (\rho,\kappa) \models P : \psi \wedge \\ \forall \langle V_1, \ldots, V_k \rangle \in \kappa : \wedge_{i=1}^{j} V_i \mathrel{\text{E}} v_i \Rightarrow \wedge_{i=j+1}^{k} V_i \in \rho(\lfloor x_i \rfloor)\end{array}}{(\rho,\kappa) \models (M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P : \psi}$$

(CFAC Decrypt)

$$\frac{\begin{array}{c}(\rho,\psi) \models M : v \;\; \wedge_{i=0}^{j}\, (\rho,\psi) \models M_i : v_i \;\wedge\; (\rho,\kappa) \models P : \psi \cup \psi' \wedge \\ \forall \{V_1, \ldots, V_k\}_{V_0}[\psi''] \in v :\; \wedge_{i=0}^{j} V_i \mathrel{\text{E}} v_i \Rightarrow (\wedge_{i=j+1}^{k}\, V_i \in \rho(\lfloor x_i \rfloor) \;\wedge\; \psi' \subseteq \psi''\sigma)\end{array}}{(\rho,\kappa) \models \text{ decrypt } M \text{ as } \{M_1, \ldots, M_j; x_{j+1}, \ldots, x_k\}_{M_0} \text{ in } P : \psi}$$

where $\sigma = [\$0 \mapsto \lfloor M_0 \rfloor, \ldots, \$j \mapsto \lfloor M_j \rfloor, \$(j+1) \mapsto \lfloor x_{j+1} \rfloor, \ldots, \$k \mapsto \lfloor x_k \rfloor]$

6.3 Properties of Analysis

Conjecture 6.1 (CFAC Subject Reduction). *If* $(\rho,\kappa) \models P : \psi$ *and* $P \rightarrow P'$ *then* $(\rho,\kappa) \models P' : \psi$.

Rational. As a subject reduction result holds for the original control flow analysis, cf. 2.1 (Subject Reduction), we expect it to hold on the modified analysis as well. ☐

The attacker needs to be reformulated since the annotations of CORRESPONDENCE LYSA are different from regular LYSA. To avoid ambiguity on the name, we will refer to the reformulated attacker as the formulae $\mathcal{G}^{DY}$.

Definition 6.2 (Remodelling the Attacker).
The formulae $\mathcal{G}^{DY}$ is defined as a conjunction of the following:

1. $\wedge_{k \in \mathcal{A}_\kappa} \forall \langle V_1, \ldots, V_k \rangle \in \kappa : \wedge_{i=1}^{k} V_i \in \rho(z_\bullet)$
2. $\wedge_{k \in \mathcal{A}_{\text{Enc}}} \forall \{V_1, \ldots, V_k\}_{V_0}[\psi] \in \rho(z_\bullet) : V_0 \text{ E } \rho(z_\bullet) \Rightarrow \wedge_{i=1}^{k} V_i \in \rho(z_\bullet)$
3. $\wedge_{k \in \mathcal{A}_{\text{Enc}}} \forall V_1, \ldots, V_k : \wedge_{i=0}^{k} V_i \in \rho(z_\bullet) \Rightarrow \{V_1, \ldots, V_k\}_{V_0}[\psi] \in \rho(z_\bullet)$
4. $\wedge_{k \in \mathcal{A}_\kappa} \forall V_1, \ldots, V_k : \wedge_{i=1}^{k} V_i \in \rho(z_\bullet) \Rightarrow \langle V_1, \ldots, V_k \rangle \in \kappa$
5. $\{n_\bullet\} \cup \lfloor \mathcal{N}_{\text{f}} \rfloor \subseteq \rho(z_\bullet)$

Conjecture 6.3 (CFAC Soundness of the Dolev-Yao Condition). *If (ρ, κ, ψ) satisfies $\mathcal{G}^{\text{DY}}$ of type $(\mathcal{N}_f, \mathcal{A}_\kappa, \mathcal{A}_{Enc})$, then $(\rho, \kappa) \models Q : \psi$ for all attackers Q of type $(\mathcal{N}_f, \mathcal{A}_\kappa, \mathcal{A}_{Enc})$.*

Rational. The proof for this conjecture should be identical to the one for Theorem 2.6 (Soundness of the Dolev-Yao Condition) from the original control flow analysis. □

Conjecture 6.4 (CFAC Completeness of the Dolev-Yao Condition). *There exists an attacker Q_{hard} of type $(\mathcal{N}_f, \mathcal{A}_\kappa, \mathcal{A}_{Enc})$ such that the formula $(\rho, \kappa) \models Q_{hard} : \psi$ is equivalent to the formula $\mathcal{G}^{\text{DY}}$ of type $(\mathcal{N}_f, \mathcal{A}_\kappa, \mathcal{A}_{Enc})$.*

Rational. The proof for this conjecture should be identical to the one for Theorem 2.7 (Completeness of the Dolev-Yao Condition) from the original control flow analysis. □

Conjecture 6.5 (Existence of the Least Solution). *There is always a least choice of (ρ, κ, ψ) for any given* Correspondence LySa *process such that $(\rho, \kappa \models P : \psi$ and (ρ, κ, ψ) satisfies $\mathcal{G}^{\text{DY}}$.*

Rational. Nielson, Nielson and Seidl [NNS01] have showed that the control flow analysis as specified in Chapter 2 (Control Flow Analysis) can be calculated despite the fact that the components ρ, κ, ψ are interpreted over an infinite universe. The procedure is as follows. Firstly, the succinct specification of the analysis is transformed into a verbose specification. Secondly, the specification is transformed to use a finite universe by encoding terms as production rules in a tree grammar. An actual implementation has been made using the *succinct solver* [NNS02]. We expect this procedure is applicable to the control flow analysis with correspondences as presented in this chapter. □

Theorem 6.6 (CFAC Safety). *If there exists (ρ, κ) such that $(\rho, \kappa) \models P : \emptyset$, then P is safe*[1].

[1] This is under the assumption that Conjecture 6.1 (CFAC Subject Reduction) holds.

Proof. Suppose that $(\rho, \kappa) \models P : \emptyset$ and $P \rightarrow^* \gamma.\text{end}(\vec{M}).P''$. By Conjecture 6.1 (CFAC Subject Reduction), we have $(\rho, \kappa) \models \gamma.\text{end}(\vec{M}).P'' : \emptyset$ where $(\rho, \kappa) \models \text{end}(\vec{M}).P'' : \psi$. By (CFAC End), and because $\vec{M}$ will not contain variables at this point in the reduction, we get $!\text{begun}(\lfloor M_1 \rfloor, \ldots, \lfloor M_k \rfloor) \in \psi$. Since γ is inert, this can only be the result of verifying begin events in γ. Because γ precedes the end event, then by Definition 3.16 (Safety and Robust Safety), P is safe. □

Theorem 6.7 (CFAC Robust Safety). *If there exists (ρ, κ) such that $(\rho, \kappa) \models P : \emptyset$ and $(\rho, \kappa, \emptyset)$ satisfies $\mathcal{G}^{DY}$, then P is robustly safe*[2]*.*

Proof. If $(\rho, \kappa) \models P : \emptyset$ satisfies G^{DY} then by Conjecture 6.3 (CFAC Soundness of the Dolev-Yao Condition), it must also be the case that $(\rho, \kappa) \models P_\bullet : \emptyset$ for any attacker $P_\bullet$. By (CFAC Par), we get $(\rho, \kappa) \models P \mid P_\bullet$ for any $P_\bullet$. By Theorem 6.6 (CFAC Safety) and Definition 3.16 (Safety and Robust Safety) P is robustly safe. □

6.4 Encodings

We define an encoding $[\![\cdot]\!]_l$ from LySa to Correspondence LySa. It works like the encoding defined in Chapter 4 (Encoding LySa as Typed LySa), but does not write type annotations. We also define an encoding $[\![\cdot]\!]_t$ from Typed LySa to Correspondence LySa. The only change the encoding need to perform is to remove type annotations on restricted names.

Conjecture 6.8 (CFA is a Subset of CFAC). *If $\rho, \kappa \models Q : \emptyset$, Q satisfies Constraint 4.2 (Pattern Matches) and $(\rho, \kappa, \emptyset)$ satisfies $\mathcal{F}^{DY}$, then there is a (ρ', κ') such that $(\rho', \kappa') \models [\![Q]\!]_l : \emptyset$ and $(\rho', \kappa', \emptyset)$ satisfies $\mathcal{G}^{DY}$.*

Rational. Since the encoding places all begin events as a prefix to the process, there will be no need to "transfer" the effects by annotating encryptions. An analysis $(\rho', \kappa', \emptyset)$ would then be defined as $(\rho, \kappa, \emptyset)$ with the replacement of crypto-point annotations with annotations on encryptions coming from ac. □

Conjecture 6.9 (Type Checking is a Subset of CFAC). *If $\vec{n} : \text{Un} \vdash P$, then there is a (ρ, κ) such that $\rho, \kappa \models [\![P]\!]_t : \emptyset$ and $(\rho, \kappa, \emptyset)$ satisfies $\mathcal{G}^{DY}$.*

Rational. By Conjecture 6.5 (Existence of the Least Solution), there is a least choice of (ρ, κ, ψ) such that $(\rho, \kappa) \models [\![P]\!]_t : \psi$ and (ρ, κ, ψ) satisfies $\mathcal{G}^{DY}$. For the conjecture to hold, we need to prove that $\psi = \emptyset$, which can only be done if we knew how the three components were created. This boils down to examining if all the effects which can be "transfered" from one point in the process to another in the type system (using latent effects on the key

[2] This is under the assumption that Conjecture 6.1 (CFAC Subject Reduction) holds.

annotations) is also able to be transfered in the control flow analysis. The rational for this is that if we can verify the typing rule $E \vdash \{M_1, \ldots, M_k\}_{M_0}$ and $E \vdash M_0 : \text{Key}(\vec{x} : \vec{T})[\vec{B}]$, then we know that $\vec{B}$ holds at this point in the process. At the same point, the control flow analysis will therefore verify that $\{M_1, \ldots, M_k\}_{M_0}[\vec{B} \ \cup \ \vec{B}']$ is in the corresponding tuple in κ. When decrypting a term M with key K, all values which may flow to M will therefore be annotated with a set of begun assertions such that $\vec{B}$ is contained in the intersection of these assertions, and will be placed in the local ψ component just as it is placed in E when type checking. $\square$

Chapter 7

In Perspective

In this report we have introduces two kinds of security properties: dynamic authentication using crypto-points, and correspondences using begin and end events. These properties can be verified using control flow analysis and type checking. In this chapter, we will put these methods in perspective and give our take on the practical feasibility of using them.

In Figure 7.1 on the next page, an illustration can be found of the relation between the two security properties and the two methods to verify them. The left and right rectangle in the figure represents the set of processes annotated with crypto-points and begin/end events, respectively.

The left rectangle consists of three shapes. The processes represented by the green shape in the top can be typed in our type system through the encoding. It is likely that a type system could be created that operates directly on crypto-points, but this would probably not make the method capable of verifying additional processes than our current type system. The turquoise shape in the middle represents processes which can be analysed through the control flow analysis introduced in Chapter 2 (Control Flow Analysis). This method is also capable of verifying anything which can be type checked. The red shape in the bottom represents those processes which cannot be verified in either of the methods.

The right rectangle also consists of the same three fills with the same relation between each other. As we proved in Chapter 4 (Encoding LySA as Typed LySA), crypto-points can be translated into begin/end events (but most likely not the other way). We have illustrated this by letting the right rectangle contain a copy of the left rectangle with dashed lines. The shapes in the dashed rectangle are the same as in the left rectangle to signify that the encoding process cannot make an unverifyable process verifyable and vice versa.

In the following, we will present two canonical examples of secure processes annotated with crypto-points and begin/end events, respectively, which cannot be verified.

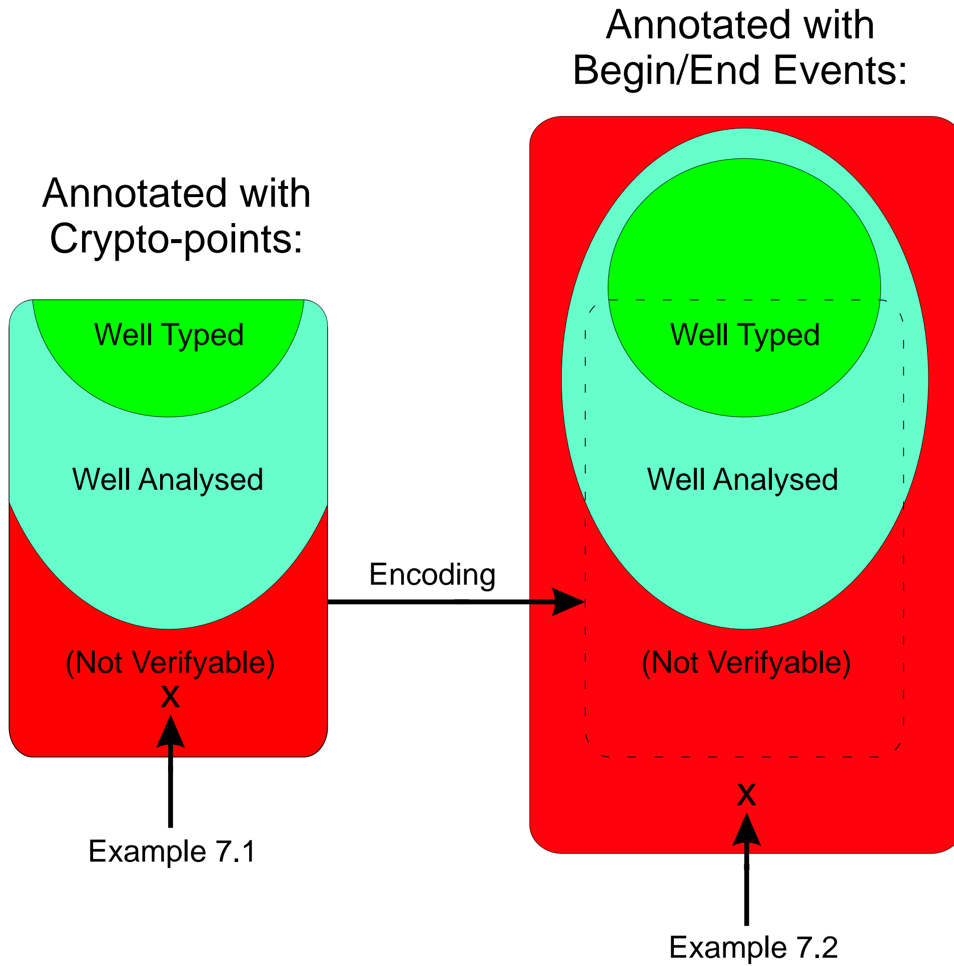


Figure 7.1: World View of Verifiable Processes.

Example 7.1 (Non-verifiable Crypto-point Annotation).
This next example shows a process annotated with crypto-points, which is not verifiable even though it guarantees dynamic authenticity. In the process, P_A outputs an encryption with crypto-point o_A and waits for an input. The principal P_B waits for an input to decrypt, which may only originate from o_A. After a successful decryption, P_B outputs a new encryption with crypto-point o_B using the same key. The key is secret so an attacker will not be able to forge encryptions.

$$\begin{aligned}
P_A(K) &\triangleq (\nu\, m)\langle \{m\}_K^{o_A}[\text{dest } \{d_A, d_B\}]\rangle. \\
&\quad (;y).\text{decrypt } y \text{ as } \{; m_y\}_K^{d_A}[\text{orig}\{o_A, o_B\}] \text{ in } 0 \\
P_B(K) &\triangleq (;x).\text{decrypt } x \text{ as } \{; m_x\}_K^{d_B}[\text{orig}\{o_A\}] \text{ in} \\
&\quad (\nu\, n)\langle \{n\}_K^{o_B}[\text{dest } \{d_A\}]\rangle \\
P_{sys} &\triangleq (\nu\, K)(P_A(K) \mid P_B(K))
\end{aligned}$$

We cannot find an acceptable analysis to the original control flow analysis because the analysis has no concept of linearity in possible reduction sequences. Since both encryptions with crypto-points o_A and o_B may flow on the ether (i.e. is known to an attacker), they must both be included in the κ component. The analysis rules will then believe the encryption with crypto-point o_B may arrive at d_B, even tough it is not possible since it is only created later in any reduction sequence.

Let us examine why it is neither possible to type check it nor verify it in the new control flow analysis. Encoding P_{sys}, we get:

$$\begin{aligned}
(\!| P_A(K) |\!) &\triangleq (\nu\, m : \text{Un})\langle \{m, o_A\}_K\rangle. \\
&\quad (;y).\text{decrypt } y \text{ as } \{; m_y, o_y\}_K \text{ in end}(o_y, d_A) \\
(\!| P_B(K) |\!) &\triangleq (;x).\text{decrypt } x \text{ as } \{; m_x, o_x\}_K \text{ in} \\
&\quad \text{end}(o_x, d_B).(\nu\, n : \text{Un})\langle \{n, o_B\}_K\rangle \\
[\![P_{sys}]\!] &\triangleq \text{begin!}(o_A, d_A).\text{begin!}(o_A, d_B).\text{begin!}(o_B, d_A). \\
&\quad (\nu\, K : \text{Un})((\!| P_A(K) |\!) \mid (\!| P_B(K) |\!))
\end{aligned}$$

We cannot find a solution to the type system because we need to find a dependent type for K on the form Key$(z_1 : \text{Un}, z_2 : \text{Un})[!\text{begun}(z_2, d_A), !\text{begun}(z_2, d_B)]$. The two begun effects are needed to count as a credit towards the two end events. However, when encrypting the message in P_B, the dependent variables in the latent effects get substituted with actual values such that the assertions $!\text{begun}(o_B, d_A)$ and $!\text{begun}(o_B, d_B)$ must hold. As $!\text{begun}(o_B, d_B)$ does *not* hold, it is not possible to find a type for K to make the process well-typed.

We cannot find an acceptable analysis to the new control flow analysis basically because of the same reason the original control flow analysis fails. The κ component in the least solution is

κ:	$\{m, o_A\}_K[!\text{begun}(o_A, d_A), !\text{begun}(o_A, d_B), !\text{begun}(o_B, d_A)]$
	$\{n, o_B\}_K[!\text{begun}(o_A, d_A), !\text{begun}(o_A, d_B), !\text{begun}(o_B, d_A)]$

In the case of any encoded process, all begin events precedes the entire process such that they will hold on the point of decryption. There is therefore no need to examine the sets of latent effects at the point of decryption. When verifying the end process, the analysis rules will find that both o_A and o_B may be bound to o_x. This means that both $!\text{begun}(o_A, d_B)$ and $!\text{begun}(o_B, d_B)$

is required to hold, even though only the former holds. Therefore, the ρ component cannot be empty.

◆

Example 7.2 (Non-verifiable Correspondence Annotation).
Consider the following process:

$$\begin{aligned} P_A(K) &\triangleq \text{begin!}(K).\langle K \rangle \\ P_B(K) &\triangleq (K;).\text{end}(K) \\ P_{sys} &\triangleq (\nu\, K)(P_A(K) \mid P_B(K)) \end{aligned}$$

First of all, this process has no immediate equivalent crypto-point annotation, as it does not make use of encryptions. This is way it is depicted outside the dashed rectangle in Figure 7.1.

In this process, P_A and P_B shares a secret name K. Imagine that there is no longer a need for it to be secret, so P_A publishes it for anyone to see. The process is robustly safe as an attacker cannot know the name K before it is published by P_A, and before that happens, P_A has issued a begin!(K) event.

We cannot find a solution to the type system because we need to "transfer" the effect !begun(K) from P_A to P_B. As there are no encryptions, and hence no keys, this cannot be done. Of the same reason, it is not possible to find an acceptable analysis with $\psi = \emptyset$.

◆

Chapter 8

Conclusion

The goal of this project was to examine the relationship between control flow analysis and type checking. This relationship was examined by looking at whether or not it is possible to duplicate a verification from one technique in the other. We chose the control flow analysis and the LySa calculus as presented in [BBD+03], where an authentication property was specified using crypto-point annotations. We developed a typed version of LySa using correspondence assertions, and constructed a type system for verifying a correspondence property. An encoding from LySa processes to Typed LySa processes was constructed, and it was proved that the encoding preserves the safety properties of the original process. Furthermore, it was shown how to construct a set of type definitions for an encoded process, such that it would make the security properties verifiable in the type system if the control flow analysis of the original LySa version had no crypto-point violations given certain constraints on the LySa process.

Lastly, we created a new control flow analysis based on the original analysis, but utilising begin/end annotations instead of crypto-points. We showed that this new analysis is capable of verifying any process which is verifyable by either the type system or, through encoding, the original analysis.

Appendix A

Correspondence LySa

Syntax of Correspondence LySa:

$M ::=$	*terms*
n	name ($n \in \mathcal{N}$)
x	variable ($x \in \mathcal{X}$)
$\{M_1, \ldots, M_k\}_{M_0}$	symmetric encryption
$P ::=$	*processes*
0	nil
$\langle M_1, \ldots, M_k \rangle.P$	output
$(M_1, \ldots, M_j; x_{j+1}, \ldots, x_k).P$	input
$P_1 \mid P_2$	parallel composition
$(\nu\, n)P$	restriction
$!P$	replication
decrypt M as $\{M_1, \ldots, M_j;$ $x_{j+1}, \ldots, x_k\}_{M_0}$ in P	symmetric decryption
begin!$(M_1, \ldots, M_k).P$	begin-event
end$(M_1, \ldots, M_k).P$	end-event

Structural Process Equivalence of Correspondence LySa, $P \equiv Q$:

$P \equiv P$	(CFAC Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$	(CFAC Struct Symm)
$(P \equiv Q \wedge Q \equiv R) \Rightarrow P \equiv R$	(CFAC Struct Trans)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(CFAC Struct Par)
$P \mid 0 \equiv P$	(CFAC Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(CFAC Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(CFAC Struct Par Assoc)
$P \equiv Q \Rightarrow !P \equiv !Q$	(CFAC Struct Repl)
$!P \equiv P \mid !P$	(CFAC Struct Repl Par)
$P \equiv Q$ if P and Q are α-convertible	(CFAC Struct Alpha)
$(\nu\, n_1)(\nu\, n_2)P \equiv (\nu\, n_2)(\nu\, n_1)P$	(CFAC Struct Res)

$(\nu\, n)0 \equiv 0$		(CFAC Struct Res Nil)
$(\nu\, n)(P \,\vert\, Q) \equiv P \,\vert\, ((\nu\, n)Q)$	if $n \notin \text{fn}(P)$	(CFAC Struct Extrusion)

Operational Semantics of CORRESPONDENCE LYSA, $P \to P'$:

(CFAC Par)
$$\frac{P \to P'}{P \,|\, Q \to P' \,|\, Q}$$

(CFAC Res)
$$\frac{P \to P'}{(\nu\, n)P \to (\nu\, n)P'}$$

(CFAC Equiv)
$$\frac{P \equiv Q \wedge Q \to Q' \wedge Q' \equiv P'}{P \to P'}$$

(CFAC IO)
$$\frac{\wedge_{i=1}^{j} [\![M_i]\!] = [\![M_i']\!]}{\langle M_1, \ldots, M_k\rangle.P_1 \mid (M_1', \ldots, M_j'; x_{j+1}, \ldots, x_k).P_2 \to_{\mathcal{R}} P_1 \mid P_2[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]}$$

(CFAC Decr)
$$\frac{\wedge_{i=0}^{j} [\![M_i]\!] = [\![M_i']\!]}{\text{decrypt } \{M_1, \ldots, M_k\}_{M_0}[\vec{B}] \text{ as } \{M_1', \ldots, M_j'; x_{j+1}, \ldots, x_k\}_{M_0} \text{ in } P \to P[x_{j+1} \mapsto M_{j+1}, \ldots, x_k \mapsto M_k]}$$

(CFAC Begin)
$$\frac{P \to P'}{\text{begin!}(\vec{M}).P \to \text{begin!}(\vec{M}).P'}$$

(CFAC End)
$$\frac{}{\text{end}(\vec{M}).P \to P}$$

References

[Aba99] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[AF01] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–115. ACM Press, 2001.

[AG97] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[AG98] M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nordic J. of Computing*, 5(4):267–303, 1998.

[AN05] E. H. Andersen and C. R. Nielsen. Static validation of voting protocols. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2005.

[BAN90] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[BBD+03] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Nielson. Automatic validation of protocol narration. In *proceedings of 16th IEEE Computer Security Foundations Workshop (CSFW 16)*, pages 126–140, 2003.

[BBD+04] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Control flow analysis can find new flaws too. In *Proceedings of Workshop on Issues in the Theory of Security (WITS 04)*, 2004.

[BDNN01a] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Security analysis using flow logics. In *Current Trends in Theoretical*

Computer Science, pages 525–542. World Scientific Publishing Co., Inc., 2001.

[BDNN01b] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for the π-calculus with application to security. *INFCTRL: Information and Computation (formerly Information and Control)*, 168, 2001.

[BNN04] M. Buchholtz, H. Riis Nielson, and F. Nielson. A calculus for control flow analysis of security protocols. *International Journal of Information Security*, 2(3-4):145–167, 2004.

[DY81] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the 22th IEEE Symposium on Foundations of Computer Science*, pages 350–357, 1981.

[FGM05] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. Technical Report MSR-TR-2005-01, Microsoft Research, 2005.

[GJ02] A. D. Gordon and A. S. A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Proc. Int. Software Security Symp.*, volume 2609 of *Lecture Notes in Computer Science*, pages 263–282. Springer-Verlag, 2002.

[GJ03] A. D. Gordon and A. S. A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300(1-3):379–409, 2003.

[GJ04a] A. D. Gordon and A. S. A. Jeffrey. Authenticity by typing for security protocols. *J. Computer Security*, 11(4):451–519, 2004.

[GJ04b] A. D. Gordon and A. S. A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *J. Computer Security*, 12(3/4):435–484, 2004.

[Gol03] D. Gollmann. Authentication by correspondence. In *IEEE Selected Areas in Communications*, volume 21, issue 1, pages 88–95, 2003.

[HJ04] C. Haack and A. S. A. Jeffrey. Pattern-matching spi-calculus. In *Proc. IFIP WG 1.7 Workshop on Formal Aspects in Security and Trust*, volume 173, pages 55–70. Kluwer Academic Press, 2004.

[NNH02] F. Nielson, H. Nielson, and R. Hansen. Validating firewalls using flow logics. *Theoretical Computer Science*, 283(2):318–418, 2002.

[NNS01] F. Nielson, H. Nielson, and H. Seidl. Cryptographic analysis in cubic time, 2001.

[NNS02] F. Nielson, H. Nielson, and H. Seidl. A succinct solver for ALFP. *Nordic Journal of Computing*, 9:335–372, 2002.

[WL93] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *RSP: IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.