View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Thesis and Dissertation Archive - Università di Pisa

Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis TD-06/06

Models for Cryptographic Protocol Analysis

Roberto Zunino

Supervisor
Pierpaolo Degano

Referees
Chris Hankin
Luca Viganò

Contents

1 Introduction **5**
1.1 Introduction 6

2 Perfect Encryption **13**
2.1 Introduction 14
2.2 A Simple Process Calculus: Syntax 17
2.3 A Simple Process Calculus: Semantics 18
2.3.1 The Adversary 19
2.3.2 The Transition System 20
2.4 Secrecy 24
2.4.1 A Probabilistic Secrecy Notion. 26
2.4.2 Comparing DY and $\mathsf{DY}_{\mathbb{P}}$ Adversaries 28
2.5 Authentication 33
2.5.1 A Probabilistic Authentication Notion 34
2.6 Non Constant–Cost Operations 35
2.7 Extensions 37
2.8 Conclusions 40

3 Tree Automata **41**
3.1 Introduction 42
3.2 Preliminaries 44
3.2.1 Matching 48
3.3 Semantics 49
3.3.1 Automaton Transformations 51
3.4 Algorithm 52
3.5 Relaxing the Left Linearity Assumption 54
3.6 Improving Precision 58
3.7 Intersecting Languages 61
3.8 Further Improvements 64
3.9 Implementation 65
3.9.1 Representing Transitions and $\simeq$ 65
3.9.2 More Expressive Rules 65
3.9.3 Heuristics 67

3.9.4 Other Optimizations 69
3.9.5 Test Cases 70
3.9.6 Further Information 81

4 Static Analysis 83
4.1 Introduction 84
4.2 Syntax 86
4.3 Dynamic Semantics 87
4.3.1 Diffie-Hellman Example 91
4.4 Static Semantics 93
4.4.1 Control Flow Analysis 93
4.4.2 Subject Reduction for CFA 97
4.4.3 A Refined CFA Analysis: CFA_{io} 99
4.4.4 Subject Reduction for CFA_{io} 102
4.4.5 Examples and Comparison 106
4.5 Implementation and Test Cases 111
4.5.1 Diffie-Hellman Example (continued) 111
4.5.2 Wide-Mouthed Frog 113
4.5.3 Session Keys via Asymmetric Key Pairs 115
4.5.4 Kerberos 117
4.6 A Bit of Compositionality 121
4.7 Future Work 122
4.8 Related Techniques and Tools 122
4.9 Conclusions 124

5 Conclusions 127

Chapter 1

Introduction

1.1 Introduction

A variety of models for the specification and the verification of security protocols has been proposed in the last years, focusing on the different problems that arise when studying distributed systems. While security issues are addressed in these models in many different ways, the techniques used can be classified into two broad classes.

The first class is the class of models from *computational complexity theory*. Computational models offer a very detailed, in-depth view of cryptosystems and protocols. Often, models are defined using low level notions: it is very common here to model messages as bit strings, and protocol participants as probabilistic Turing machines. In these models, security properties usually require systems to be resilient to a class of adversaries. Adversaries can run any algorithm (e.g. they are arbitrary Turing machines) and are subject only to some computational complexity bounds and some probabilistic constraint (typically, they must succeed in polynomial time with a non-negligible probability). While these models are very well suited to study virtually any aspect among the many facets of security, actually *proving* systems secure is a non trivial task. In fact, proofs in computational models tend to be complex and long. Therefore the process of verifying systems can be expensive, tedious and - most importantly - error prone.

The second class of models comes from *formal methods*. Formal models *abstract* the behaviour of distributed systems so that one can reason about its properties using high-level tools. Usually here, messages are modeled as terms, and participants are specified in process calculi. Often, the use of terms implicitly restricts the operations that can be performed on messages to a small set of selected primitives. Consequently, the adversary is also assumed to “play by the rules”: its behaviour must be specifiable through the calculus and its data can only be processed by the given primitives. In fact, formal security properties consider only the class of such adversaries. While this definition only covers a strict subset of the computational adversaries, verifying the robustness of systems against the formal adversaries is simpler than its computational counterpart. Indeed, this simplicity is a consequence of considering only a restricted class of adversaries. Often, security properties in formal models involve reachability over some transition system, disregarding complexity and probability. Moreover, many semi-automatic and automatic tools for carrying out security proofs have been developed and successfully used to check protocols. Because of the higher level abstraction, the process of formal verification is also less error prone with respect to writing a proof in a computational model.

Research has focused on establishing connections between the two approaches. Ideally, one would use formal methods and tools to prove security

properties as those definable in the computational models. Bridging the gap between the two worlds would unite the simplicity offered by formal models with the very precise view given by computational models. To this aim, work has been started in two opposite directions.

A first line of results proceeds in the direction from computational to formal models. Some selected "basic blocks" used in formal proofs are considered to establish a computational soundness property for these blocks. For instance, formal methods often use some equivalence relation between terms to model indistinguishability, i.e. the fact that no reasonable *formal* adversary can distinguish between them. Computational soundness for such a relation would ensure that any reasonable *computational* adversary can not distinguish between the encoding of formally-equivalent terms. Obviously, this is possible only if the encoding used satisfies some robustness property. So, this line of work also focuses on which assumptions one would require on the cryptosystem to ensure that the computational model behaves exactly as the formal model.

The other direction is the one from formal to computational models. Here, the focus is on making the models less abstract, by introducing in the model some notions from the computational world. For example, the adversaries in probabilistic formal models can be more faithful to the computational adversaries than the ones in deterministic formal models. Also, formal methods can take into account the *cost* associated to attacks, and therefore relate to the computational complexity. In essence, this line of work is about *weakening* the strong assumptions of current formal models about 1) the capabilities of the adversary and 2) the underlying cryptosystem. From another point of view, weakening the assumptions made in formal models also means to consider a more powerful adversary, which still "plays by the rules" of the model but has less restrictions on what it can do.

Looking at the many formal models proposed so far, we see that a large number of them share a common model of adversary. The assumptions behind this kind of adversary were firstly described by Dolev and Yao [32], so that it is commonly addressed as the Dolev-Yao adversary. Being so widely employed in formal models, this adversary is indeed a *de facto* standard.

We now briefly summarize the ideas behind the Dolev-Yao model. When a protocol is run by its participants, messages are exchanged over a public communication channel. The Dolev-Yao adversary can eavesdrop on this channel and intercept messages as they are sent through. In this way, the adversary is able to *learn* the messages. The adversary can then *operate* on the collected data and in this way add new data to its knowledge. Further, the adversary can take any piece of data it knows and *deliver* it to a protocol participant, pretending that is a message from someone else.

The actual operations the adversary is allowed to perform in the model

are doubtlessly a very significant part of the model itself. In a computational model we would restrict these to a class of algorithms, but in formal models data are represented as terms, and this heavily influences the operations that can be performed over it. Most commonly, in formal models one considers a limited, fixed number of such operations, only. These operations are often chosen so that they cover exactly the cryptographic primitives used in the protocols one wants to study. Common choices include encryption and signature primitives. So, both the protocol participants and the adversary use the same well-known tools to handle data and produce messages. This greatly simplifies the model, and allows for agile reasoning about the security of a protocol. However, choosing a fixed set of operations also means restricting the adversary to those alone. Indeed, the power of many tools for protocol verification derives from that they heavily rely on these restrictions.

One of the most common restrictions in formal models lies in that there is no way of decrypting a message other than using the right key. This is often modeled by having no operation to manipulate an encrypted term other than decryption. In other words, the underlying cryptosystem is assumed to be so robust that encryptions convey no information at all about their contents: this is the ideal, or *perfect* encryption assumption.

In Chapter 2, we relax this assumption by explicitly allowing the adversary to break encryptions through a special operation. Of course, this only happens only under suitable hypotheses, so that the adversary is not able to disrupt just any protocol. For this, we borrow from the computational model the idea to use probability and complexity to constrain the adversary power. In Chapter 2, we therefore define a probabilistic version of the Dolev-Yao adversary. We then consider two common security properties: the secrecy of a given message and non-injective authentication. Using these properties, we compare the enhanced Dolev-Yao adversary to the standard one.

In the subsequent chapters, we discuss another assumption that is also very common in formal models and tools for security. As we anticipated, during the design of the model one chooses the set of operations to consider, by which both the participants and the adversary can construct messages. Obviously, it is convenient to choose the cryptographic primitives that are the most widely used in the protocols presented in the literature. Indeed, it turns out that a very broad class of protocols *only* use the following primitives:

- encryptions (symmetric or asymmetric)
- digital signatures
- hashes
- pairs (lists)

These primitives are directly modeled with the terms below.

$$\mathsf{enc}(m,k) \text{ or } \mathsf{enc}(m,\mathsf{pub}(k))$$
$$\mathsf{sig}(m,\mathsf{pri}(k))$$
$$\mathsf{hash}(m)$$
$$\mathsf{cons}(m_1,m_2)$$

Above, m denotes a message, while k is a key. Terms $\mathsf{pri}(k)$ and $\mathsf{pub}(k)$ denote the private and public parts of the same asymmetric key k.

In formal models, the algebra given by the above primitives is usually assumed to be *free*. In other words, considering only symmetric encryption, we have that $\mathsf{enc}(m_1,k_1) = \mathsf{enc}(m_2,k_2)$ if and only if $m_1 = m_2$ and $k_1 = k_2$. This *freeness* property ensures that intended values and their term representation have a direct one-to-one correspondence.

Freeness enormously simplifies the definition of security properties. For instance, the secrecy of a message m can be defined in terms of m alone, instead of a class of equivalent forms. Automatic reasoning benefits from this simplification too: e.g. whenever we see two values with a different top-level constructor, we know these values are different, without needing to inspect them further. However, special care must be taken to avoid breaking this freeness property through other features in the model. Continuing the example above, one might be tempted to add a decryption operation $\mathsf{dec}(m,k)$ to the algebra, introducing the equality

$$\mathsf{dec}(\mathsf{enc}(m,k),k) = m$$

and thus making the algebra non-free. A common way to circumvent this problem is to provide decryption not as an operation on terms, but as a feature of the calculus that models the protocol participants. Most commonly, decryption is performed through pattern matching:

$$\textbf{decrypt}\ \ m\ \ \textbf{as}\ \ \mathsf{enc}(x,k)\ \ \textbf{in}\ \ \ldots \text{(use of } x \text{ and } k\text{)}$$

This technique can be used to provide destruction in the model (for pairs, etc.) and still keep terms free. However, the model obtained in this way is not completely equivalent to a model with real destructors. This is because using pattern matching instead of a proper decryption operation implicitly restricts the adversary to successfully run decryption on actual encrypted terms, only. If running decryption on other terms could be used to construct some kind of attacks, the formal model is not able to reason about these attacks. For this reason, a protocol might be regarded as safe in the model while, in practice, it is not.

Further, it is not always possible to simulate a non-free algebra using pattern matching as done above. Sometimes a cryptosystem is used which

might satisfy additional laws such as

$$\mathsf{enc}(\mathsf{dec}(m,k),k) = m \qquad \text{surjective encryption}$$
$$\mathsf{enc}(\mathsf{enc}(m,k_1),k_2) = \mathsf{enc}(\mathsf{enc}(m,k_2),k_1) \quad \text{commutative encryption}$$

Note that a protocol might actually *exploit* these laws, and fail to work if they do not hold. So, changing to a cryptosystem ensuring freeness might not be an option. Similarly, the protocol might use other cryptographic primitives that are intrinsically non-free such as XOR, or finite-field multiplication and exponentiation. In this case, we must cope with the many different term representations for the same value. Because of this, both the model and related verification tools must always consider the whole equivalence class of a term, and therefore work "up to" the laws that define the primitives.

In Chapter 3, we deal with the problem of constructing over-approximations for set of terms, when we consider them up to an equivalence relation. This equivalence is not fixed, but can be expressed through *any* term rewriting system. We stress that *any* rewriting system can be used to this purpose: for instance, we do not assume the existence of normal forms, confluence, or termination. Through rewriting rules, we can express the laws for many cryptographic primitives, without having to choose a particular fixed set of those. So, with this approach, we relax both the *free algebra* assumption, as well as the reliance on having a limited set of operations.

Here, we approximate terms up to rewriting through non deterministic finite tree automata. We define a sequence of algorithms to compute sound approximations, and compare them. Also, we implemented these algorithms, so to validate their capability to deal with usual non-free cryptographic primitives. We also discuss here some details of our implementation that turned out to be relevant for the precision of the computed approximation.

Exploiting the above techniques for dealing with terms up to equivalence, we finally face the problem of protocol verification in Chapter 4. Here, we define a process calculus for the specification of protocols. We keep the issues of defining the semantics of the calculus separated from those related to the primitives used. We still allow these primitives to be defined through any rewriting system, without losing generality from our previous results. Then, we consider the static analysis of protocols.

The first static analysis we present borrows from the *control flow analysis* [57] approach. Roughly, this analysis statically computes a finite representation for a superset of the messages exchanged through the network at run-time. So, terms not within this superset are known not to be used in communications. For instance, this can be used to prove the secrecy of a selected piece of data.

With small changes, we adapt this analysis to our calculus. Further, by leveraging the approximation of Chapter 3, our adaptation is able to

cope with terms up to rewriting. This enables us to statically reason about security properties of protocols, given

- the specification of the protocol, written as a process in our calculus, and
- the specification of the employed primitives, defined through a rewriting system.

In the same chapter we also enhance the analysis to better exploit the approximation techniques of Chapter 3. Moreover, we refine the analysis so that it produces a more precise results for some constructs of the calculus. We compare this refined analysis with the previous one through examples.

We developed a tool implementing the analyses, so we can test the effectiveness of our techniques on concrete test cases. To this purpose, we consider some real-world security protocols. One of the most relevant cases we consider is a protocol based on the Diffie-Hellman key exchange. The protocol involves exponentials over a finite field. We model multiplication, inversion and exponentiation using rewriting rules, together with other common cryptographic operations, and allow the adversary to use those. Here, we are using a number of non-free primitives, satisfying many equations and, for that reason, usually difficult to handle. Yet, we use our tool to prove the *forward secrecy* of a message M, ensuring that

- the adversary is not able to learn M by interacting with the protocol participants, and
- if, after the protocol has been run, certain secret keys are disclosed, this still does not enable the adversary to obtain M.

This forward secrecy property is particularly interesting to us, since it includes some notion of *temporal dependency*, or causality, albeit in a limited form. This is because the second point above considers what happens *after* the keys are revealed.

Another interesting protocol we consider involving time is a variant of Kerberos. Here, the protocol explicitly uses *timestamps*. The security property we focus on ensures that the disclosure of older secrets, i.e. those with old timestamps, does not compromise the newer sessions of the protocol. We are actually able to prove this result exploiting our tool.

In the conclusions of this thesis, we summarize the most significative accomplished goals. Further, we discuss about the nice interactions between the results of different chapters, linking them. Finally, we express our opinions and hopes about future research.

Our Recent Works

Our first work on the *perfect encryption* assumption appeared in [73]. There, we weaken this assumption in a model considering a simple secrecy property. That work was further extended in [74], where we also dealt with authentication and more adversary rules.

We wrote in [71] about our first ideas about analyzing security protocols without assuming a fixed set of cryptographic primitives. There, however, we relied on the *freeness* of the algebra of terms, limiting the choice of the primitives. Our efforts to relax this hypothesis lead us to write [72] and explore the opportunities offered by tree automata. Exploiting those foundations, we derived a static analysis for security protocols involving non-free operations [75]. We provided an implementation of this analysis [61].

Chapter 2

The Perfect Encryption Assumption

Abstract

We consider secrecy and authentication in a simple process calculus with cryptographic primitives. The standard Dolev–Yao adversary is enhanced so that it can guess the key required to decrypt an intercepted message. We borrow from the computational complexity approach the assumptions that guessing succeeds with a given negligible probability and that the resources available to adversaries are polynomially bounded. Under these hypotheses we prove that the standard Dolev–Yao adversary is as powerful as the enhanced one.

2.1 Introduction

A lot of recent work concerns the analysis of security protocols. While the problem has been approached in many different ways, the techniques used can be roughly classified into two main classes. On the one side, we have results coming from *computational complexity theory* which offers a detailed, in–depth view of cryptosystems and protocols and deals with probability and algorithms. On the other side, *formal methods* provide abstractions that allow for mechanical proofs of cryptographic protocol properties, but often require stronger assumptions, among which is the *perfect* or *unbreakable* encryption assumption.

Often, formal methods work on protocols specified in some process calculi and assume the existence of the so–called Dolev–Yao adversary (DY for short). Besides fully controlling the network, it manipulates messages but it cannot break encryptions. We study here ways to weaken the perfect encryption property, taking care of some computational aspects.

We use a process calculus to model key–exchange protocols and we enhance the standard DY adversary by explicitly allowing it to break encryptions through a *guessing* operation. We call the enhanced adversary $\mathsf{DY}_{\mathbb{P}}$. Roughly speaking, once intercepted a message, the adversary can deduce the key to decrypt it with a given probability.

Computational reasoning predicts that guessing is hard, provided that

- the cryptosystem is robust,
- the adversary has only a reasonably bounded computational power (e.g. the adversary is in probabilistic polynomial time, $\mathcal{PP}$–Time), and
- the keys are long enough.

Many other factors affect the probability of guessing, but for the sake of simplicity, often one considers only those above. Also, the computational approach leaves the actual probability distribution implicit. So the proof

that a protocol is secure considers a suitable class of probability distributions, rather than a specific one. Here, we shall make the same assumptions.

We compare the traditional Dolev–Yao model in which guessing is not permitted against our enhanced one. The comparison is made easy by the fact that, if we forbid the use of the guessing operation, the $\mathsf{DY}_{\mathbb{P}}$ model collapses onto DY. Moreover, since we only deal with discrete probability distributions, removing the guessing operation is equivalent to assuming the guessing probability to be null everywhere.

Our study is carried on two security notions: secrecy and authentication. We give two definitions of *secrecy* for a protocol. Both are given in Sect. 2.4 and consider only the secrecy of a selected piece of data. One definition is the traditional one and it can be summarized as follows: the adversary cannot learn a given secret by interacting with the protocol and by constructing/destructing the intercepted data with the only limitation that encryptions are unbreakable. The other definition, instead, is adapted from the computational complexity approach. Roughly, the *probability* of learning a certain secret is a function of the key length, assuming the adversary is in $\mathcal{PP}$–Time and breaking the cryptosystem is a problem not in $\mathcal{PP}$–Time. Then, one studies the asymptotic behaviour of this function. Obviously, our first secrecy definition is based on the DY model, and our second one is designed for the $\mathsf{DY}_{\mathbb{P}}$ model.

Similarly, we give two definitions of *authentication* for a protocol in Sect. 2.5. The first one is a formal definition adapted from [62]. This property requires that a responder successfully completes a run only if an initiator started it. The other notion is its $\mathsf{DY}_{\mathbb{P}}$ counterpart: the probability the adversary has to disrupt authentication is represented as a function of the key length. Again, we then study its asymptotic behaviour.

Our main result shows that, for both secrecy (Sect. 2.4) and authentication (Sect. 2.5), the two security models DY and $\mathsf{DY}_{\mathbb{P}}$ are equivalent. On the one hand, this result is not surprising: increasing the length of the keys and still keeping the adversary polynomially bounded (with respect to the key length) results in a virtually perfect encryption. On the other hand, the traditional DY model is considerably simpler and has an accuracy comparable with that of the $\mathsf{DY}_{\mathbb{P}}$ model, under the standard hypothesis that guessing is hard.

For the sake of presentation, here we consider a core calculus, with the essential features, only. Sometimes, we make some assumptions to keep our model as simple as possible, even if they are not strictly needed and could be relaxed (see e.g. Sect. 2.6). More advanced primitives can be easily added along the lines discussed in Sect. 2.7. In the same section, we also sketch some further possible extensions to our $\mathsf{DY}_{\mathbb{P}}$ model.

Related Work. Several authors recently started to fill the abstraction gap between the computational approach and the formal methods approach to the analysis of cryptographic protocols. Abadi and Rogaway [6] propose an

equivalence on terms of a formal language that reflects the computational notion of indistinguishability of the bit–strings representing the terms. A finer equivalence is proposed by Troina, Aldini and Gorrieri in [68]. Abadi and Jürjens [5] extend the equivalence of [6] to a process calculus and study indistinguishability in presence of a passive adversary. Micciancio and Warinschi further extend the equivalences of [6] and [5] by considering a broader class of cryptosystems; their logic is complete: two terms are equivalent if and only if their corresponding bit–strings are indistinguishable.

Our proposal differs from the above in that we assume a full active DY adversary. Also, we extend the adversary model with a computationally–oriented feature, instead of giving (to parts of) it a computational account.

Our approach is similar to that of Mitchell, Ramanathan, Scedrov and Teague [60] who use a probabilistic calculus and an equivalence based on probabilistic bisimulation. This equivalence is then used to establish indistinguishability of process behaviour, under common cryptographic assumptions.

Herzog in [43] shows that sufficiently strong cryptography can limit a computational adversary, to make it as powerful as the DY formal adversary.

Pfitzmann, Backes and Waidner [9, 8] compare the computational adversary and the formal one exploiting the notion of simulatability. The protocol principals are modeled as Turing machines and all the cryptographic operations are performed by an oracle which is used as a cryptographic library. The oracle implements the library operations by manipulating either terms or bit–strings, offering the same interface to principals. Therefore, the oracle can interact with a computational or with a formal adversary. Simulatability ensures that every computational attack can be simulated formally, thus proving the connection between the two adversary models.

Coming from a formal side, we use a process algebra to specify protocols and existing techniques to analyze them, while [9, 8] pay more attention to computationally–based descriptions of the principals and of the cryptosystem.

Process algebras are also used by Laud [47]. There, the formal semantics of processes is connected to the computational models of Pfitzmann, Backes and Waidner. A type system is then used to prove protocols secure in the computational setting. So, this work provides formal tools (types) to reason in a computational model.

Moreover, in [10, 11] Baudet considers a probabilistic extension to DY, with an approach rather similar to ours. Also here, time complexity is taken into account, so that adversary need to run in $\mathcal{PP}$-time. Further, the extended adversary is shown to be as powerful as the DY one.

Finally, we mention a paper by Lowe [48] that studies intruders who can guess secrets. The focus of his paper, quite different from ours, is on the feasibility of verifying whether the guess succeeded.

2.2 A Simple Process Calculus: Syntax

We introduce a simple process calculus that we use to model key–exchange protocols. This calculus is basically a variant of the π–calculus [52] enriched with cryptographic primitives, much like the Spi–calculus [4]. Since we want to keep our calculus as simple as possible, we only use symmetric encryption. A brief account of asymmetric encryption is in Sect. 2.7.

Let $\mathcal{N}$ be a denumerable set of *names* and let $\mathcal{V}$ be a denumerable set of *variables*. We use the letters n, m, o to range over $\mathcal{N}$ and the letters x, y, z to range over $\mathcal{V}$. Under these assumptions, we define *terms* in the following way. Terms represent the messages that are sent over the network when the protocol is run. Their meaning is standard.

$$
\begin{array}{rlll}
L, M, N ::= & 0 \mid 1 & & \text{bit} \\
\mid & x & & \text{variable} \\
\mid & n & & \text{name} \\
\mid & (M, N) & & \text{pair} \\
\mid & \{M\}_N & & \text{symmetric encryption}
\end{array}
$$

We now define *processes* in the following way:

$$
\begin{array}{rlll}
P, Q, R ::= & \mathsf{nil} & & \text{null process} \\
\mid & (x).P & & \text{input} \\
\mid & \langle M \rangle.P & & \text{output} \\
\mid & (P \mid Q) & & \text{composition} \\
\mid & !\, P & & \text{replication} \\
\mid & (\mathsf{new}\ n)P & & \text{declaration} \\
\mid & \mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q & & \text{conditional} \\
\mid & \mathsf{split}\ M\ \mathsf{as}\ (x, y)\ \mathsf{in}\ P & & \text{split} \\
\mid & \mathsf{decrypt}\ M\ \mathsf{as}\ \{x\}_N\ \mathsf{in}\ P & & \text{symmetric decryption}
\end{array}
$$

The above syntax is quite standard, and readers familiar with other process calculi will find it straightforward. A simple intuitive description follows.

The nil process does not perform any operation. The input process $(x).P$ reads a value from the network, assigns it to the variable x and then behaves as P. The output process $\langle M \rangle.P$ sends a value over the network and then behaves as P. The parallel composition of two processes is denoted by $(P \mid Q)$; P and Q can evolve independently. The replication $!\, P$ behaves as the parallel composition of an unlimited number of P processes. We write $(\mathsf{new}\ n)P$ for the creation of new names (i.e., new keys, fresh nonces, etc.). The conditional tests whether two terms are equal. Split and decryption are used to destruct pair and encryption terms. Note that the decryption requires the (secret) key to succeed.

The most important features of this calculus are the following:

- All the input and output operations use the same *global* public channel, unlike some other calculi such as the π–calculus or the Spi–calculus, where input and output occur on given channels.

  While this makes it impossible, for instance, to specify the destination for a message (which is very inconvenient from a programmer's point of view), it models the standard Dolev–Yao assumption which states that the adversary is able to reroute messages. In a sense, the global channel acts as a black–board on/from which processes write/read messages.

- There are no private channels in our calculus. Although private channels could be used to model *secure links*, we do not include them in our calculus in order to keep it simple.

We stress that the above features do not have any significant impact on the results we present in this chapter, while they allow for a simple presentation. Indeed, we could instead use a richer calculus such as Spi and establish the same results, at the expense of more cumbersome definitions and proofs.

A variable is said to be *bound* if it occurs under an input prefix, split, or decryption; otherwise it is said to be *free*. Similarly, a name is *bound* if it occurs under a declaration (new n), otherwise it is *free*. We write $\mathsf{fv}(P), \mathsf{bv}(P), \mathsf{fn}(P), \mathsf{bn}(P)$ respectively for (the sets of) the free variables, the bound variables, the free names, and the bound names of the process P.

2.3 A Simple Process Calculus: Semantics

We define an *adversary-aware* semantics of our process calculus. This means that our semantics explicitly assumes the presence of an adversary and models the interaction between it and the system which executes a certain protocol (i.e. a process P).

We first make two assumptions about what the adversary can or cannot do:

- Like the Dolev–Yao adversary, ours can intercept and learn messages as they are sent over the network. The adversary can then send over the network terms built from the known messages by using the following *operations*: pairing, splitting of pairs, encryption, and decryption. This also implies the adversary is able to reroute, discard, and replace messages, possibly pretending that they are from someone else.

- Unlike the Dolev–Yao adversary, ours can also *guess* a key which has been used to encrypt some message. This special operation, however, only succeeds with a given probability.

2.3.1 The Adversary

We introduce the following *entailment rules*, that define the operations of the adversary. As usual, each rule can be read as: "the adversary can build the term of the right hand side of the arrow provided it knows the terms on the left hand side".

$$\begin{array}{llll} \mathsf{DYCons} & M, N & \stackrel{1}{\longmapsto}_{\mathsf{DY}} & (M, N) \\ \mathsf{DYFst} & (M, N) & \stackrel{1}{\longmapsto}_{\mathsf{DY}} & M \\ \mathsf{DYSnd} & (M, N) & \stackrel{1}{\longmapsto}_{\mathsf{DY}} & N \end{array} \qquad \begin{array}{llll} \mathsf{DYEnc} & M, n & \stackrel{1}{\longmapsto}_{\mathsf{DY}} & \{M\}_n \\ \mathsf{DYDec} & \{M\}_n, n & \stackrel{1}{\longmapsto}_{\mathsf{DY}} & M \end{array}$$

$$\mathsf{DYGuess} \quad \{M\}_n \quad \stackrel{p}{\longmapsto}_{\mathsf{DY}} \quad n$$

Each arrow has a label, expressing the probability of the operation to succeed. The first five rules are the traditional Dolev–Yao rules and get probability 1 as they never fail. Rule DYCons allows the adversary to construct a pair (M, N) from its components. Dually, the adversary can also destruct a pair through rules DYFst and DYSnd, thus obtaining its constituents M and N. Encryptions have similar rules. The construction of an encryption $\{M\}_n$ is handled by rule DYEnc, which of course require that the adversary knows both the message M to encrypt and the secret key n (we shall discuss afterwards why here keys are names, only). Decryption is done through DYDec: given an encrypted message and its related key, the adversary can recover the cleartext message M.

The last rule, DYGuess, extends the Dolev–Yao model of adversary. This rule allows the adversary to guess the secret key n with probability p. As anticipated in the Introduction of Sect. 2.1, we shall not assign specific values to the *success probability* p for DYGuess, just as it happens within the computational complexity approach. Indeed, the guessing probability depends on many factors, such as which cryptosystem was used for the encryption, the length of the key used, whether the adversary knows other messages encrypted with the same key, whether the adversary knows the plaintext, the amount of computational resources (e.g. time) spent by the adversary, and so on. Even if all these factors are taken into account, there is no handy formula for the guessing probability. In order to keep our system simple, we make some further assumptions:

- We constrain encryptions to use only names as keys: all the encryptions thus must have the form $\{M\}_n$. This can be enforced by a simple *type system*. Furthermore, we assume that all names are encoded as bit strings of the same length η.[1]

[1] In the computational complexity approach η is the *security parameter* that affects the length of keys. For the sake of simplicity, here we simply identify it with the key length.

- We assume that the guessing probability depends only on η and on the number of operations $\mathcal{T}$ previously performed by the adversary: we write it as $\mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T})$. In particular, it does not depend on the result (success or failure) of previous DYGuess operations. Most importantly, here we stipulate that the adversary consumes one unit of computational resources in order to perform the guess. (This last assumption is not crucial: see Sect. 2.7)

Intuitively, the probability $\mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T})$ decreases as soon as longer keys are used, i.e. as η increases. Similarly, when the adversary performs some operation, it might learn some information that can help it to guess a key afterwards. This means that the probability $\mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T})$ increases as soon as $\mathcal{T}$ increases. The way $\mathbb{P}_{\mathsf{guess}}$ depends on η and $\mathcal{T}$ is related to the strength of the cryptosystem.

As discussed in the related work, in the computational complexity approach there are different ways to regard a cryptosystem as robust. In all of them, it turns out that the function $\mathbb{P}_{\mathsf{guess}}$ "quickly" approaches zero as soon as η increases, provided that $\mathcal{T}$ is reasonably bounded (see Def. 3). Somehow arbitrarily, we shall only require this kind of asymptotic assumption on $\mathbb{P}_{\mathsf{guess}}$ in Sects. 2.4 and 2.5, stipulating that the longer the keys, the stronger the encryptions. As a matter of fact, our assumption on $\mathbb{P}_{\mathsf{guess}}$ is not sufficient to imply robustness in a computational sense, yet natural and convenient in our formal framework. We shall come back on this issue in Sect. 2.7, where we shall also consider an alternative view: guessing the contents of encryptions without breaking keys.

We do not put further assumptions on $\mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T})$. In particular, we take in no account the properties a cryptosystem must have to ensure that $\mathbb{P}_{\mathsf{guess}}$ satisfies the requirement above. Instead, we shall consider $\mathbb{P}_{\mathsf{guess}}$ a *free parameter*, and we shall prove our results for a suitable class of probability functions.

2.3.2 The Transition System

In order to define the semantics of our calculus, we use a *labeled transition system* (LTS). Its states are composed of:

- the adversary's knowledge $\mathcal{K}$, which represents the set of terms that the adversary has learnt or built so far;
- the number of operations $\mathcal{T}$ the adversary performed during the previous steps of the computation;
- a process P, which represents the current state of the execution of the protocol;
- an optional pair (p, N), used to model the fact that the adversary is about to learn the term N with probability p.

We write $(\mathcal{K}, \mathcal{T}, P)$ (or, if the optional part is present, $(\mathcal{K}, \mathcal{T}, P)^{p,N}$) to represent a generic state of the LTS.

We now give the intuition underlying our transitions. A first kind of transition models a "standard" action of a process P:

$$(\mathcal{K}, \mathcal{T}, P) \xrightarrow{\mu} (\mathcal{K}', \mathcal{T}, P')$$

Initially, the adversary has knowledge $\mathcal{K}$. Then, P performs an action and becomes P'. If the action is the output of a message M, the new knowledge is $\mathcal{K}' = \mathcal{K} \cup \{M\}$, otherwise $\mathcal{K}' = \mathcal{K}$.

Another kind of transition models an action of the adversary:

$$(\mathcal{K}, \mathcal{T}, P) \xrightarrow{\spadesuit} (\mathcal{K}, \mathcal{T}, P)^{p,N}$$

This transition represents the choice of some operation *op* the adversary is going to apply to the terms it knows. The superscript p is the probability to get the result N ($p \neq 1$ only when the operation is a *guess*). From the target state of a $\xrightarrow{\spadesuit}$ transition, two transitions exit. One transition represents the success of operation *op* and has the form

$$(\mathcal{K}, \mathcal{T}, P)^{p,N} \xrightarrow[p]{\spadesuit_{\mathsf{s}}} (\mathcal{K} \cup \{N\}, \mathcal{T} + 1, P)$$

The other transition instead represents the failure of *op*:

$$(\mathcal{K}, \mathcal{T}, P)^{p,N} \xrightarrow[1-p]{\spadesuit_{\mathsf{f}}} (\mathcal{K}, \mathcal{T} + 1, P)$$

Both these transitions are labeled with the corresponding probability and increase the number $\mathcal{T}$ of operations performed by the adversary.

Before giving the definitions of the transitions $\xrightarrow{\mu}$, $\xrightarrow{\spadesuit}$ and $\xrightarrow{\spadesuit_{\mathsf{s/f}}}$, we make some assumptions in order to simplify the definition of our system. To this purpose, we consider processes up to the *structural congruence* relation $\equiv$, defined as the minimum congruence on the set of processes $\mathcal{P}$ including α–conversion (renaming of bound names), and such that $(\mathcal{P}/_{\equiv}, |, \mathsf{nil})$ is an abelian monoid.

While on the one hand this allows for a simpler semantics, on the other hand this makes it harder to express properties on processes. For example, consider the following processes

$$\begin{array}{lll} R_1 = & (\mathsf{new}\ n\ )P & \mid\ (\mathsf{new}\ m)Q \\ R_2 = & (\mathsf{new}\ m)P\{m/n\} & \mid\ (\mathsf{new}\ n\ )Q\{n/m\} \end{array}$$

where, as usual, $\{m/n\}$ denotes substitution. The above processes are equivalent according to $\equiv$. Suppose that we want to state "R_1 never performs the output of n", meaning that (some instance of) the name n that occurs

in $(\mathsf{new}\ n)P$ is never sent through the network. In order to formalize this property, we need to cope with the fact that α–conversion can perform any arbitrary renaming, e.g. transforming R_1 into R_2, thus causing confusion among names. This may lead to quite a complex formalization.

We want to avoid this kind of confusion. To this aim, we partition the set of names $\mathcal{N}$ into equivalence classes, and we assume that the α–conversion of a name is only performed *within* its equivalence class. We write $n \sim_\alpha m$ when n and m belong to the same equivalence class. We also extend homomorphically the relation $\sim_\alpha$ to terms.

We still assume processes up to $\equiv$, where we use the just introduced α-conversion, rather than the usual one, to avoid confusion. E.g., provided that $n \not\sim_\alpha m$ in the example above, R_1 can not be renamed into R_2 and thus $R_1 \not\equiv R_2$. In general, we shall require $\not\sim_\alpha$ to hold for all the names in a process that we want to keep distinct. This requirement can be formally stated as follows.

Definition 1 *A process P is* canonical *iff*

- $\forall n, m.$ *n and m occur in different declarations in P* $\implies n \not\sim_\alpha m$
- $\forall n, m \in \mathsf{fn}(P) \cup \mathsf{bn}(P).\ n \neq m \implies n \not\sim_\alpha m$

If a process P is canonical, we can simply state properties referring to names that occur in P (up to $\sim_\alpha$). In the above example, assuming R_1 is canonical, we can state "R_1 never performs the output of n" as "whenever R_1 outputs a term M, then $M \not\sim_\alpha n$". Note that this statement precisely reflects the intended meaning of the property.

Further, note that we only consider bound *names* in the definition above, disregarding bound *variables*. This is because bound names correspond to the generation of new values, and it is natural to express properties that involve such generated values. In contrast, bound variables are used for inputs, splits, and decryptions. So, variables are bound to pre-existing terms, only, and we never need to refer to them for the properties we consider. For this reasons, we allow arbitrary α-conversion between variables.

We will study secrecy (Sect. 2.4) and authentication (Sect. 2.5) only for canonical processes, as this simplifies the formalization of those properties.

Process Actions ($\xrightarrow{\mu}$).

We now give the rules for $\xrightarrow{\mu}$, representing one action performed by P. The transition label μ can be either a name or the special symbol τ. We write $P\{M/x\}$ for the substitution of the term M at every free occurrence of x in

P, possibly α–converting bound names if necessary.

$$\mu\mathsf{In}\quad \frac{M \in \mathcal{K}}{(\mathcal{K},\mathcal{T},(x).P) \xrightarrow{\tau} (\mathcal{K},\mathcal{T},P\{M/x\})}$$

$$\mu\mathsf{Out}\quad \frac{}{(\mathcal{K},\mathcal{T},\langle M\rangle.P) \xrightarrow{\tau} (\mathcal{K}\cup\{M\},\mathcal{T},P)}$$

$$\mu\mathsf{Decl}\quad \frac{n \text{ does not occur in any term of } \mathcal{K}}{(\mathcal{K},\mathcal{T},(\mathsf{new}\ n)P) \xrightarrow{n} (\mathcal{K},\mathcal{T},P)}$$

$$\mu\mathsf{Rep}\quad \frac{}{(\mathcal{K},\mathcal{T},!\ P) \xrightarrow{\tau} (\mathcal{K},\mathcal{T},P \mid !\ P)}$$

$$\mu\mathsf{Par}\quad \frac{(\mathcal{K},\mathcal{T},P) \xrightarrow{\mu} (\mathcal{K}',\mathcal{T},P') \qquad \mu=\tau \vee \mu \notin \mathsf{fn}(Q)}{(\mathcal{K},\mathcal{T},P|Q) \xrightarrow{\mu} (\mathcal{K}',\mathcal{T},P'|Q)}$$

$$\mu\mathsf{Then}\quad \frac{}{(\mathcal{K},\mathcal{T},\mathsf{if}\ M=M\ \mathsf{then}\ P\ \mathsf{else}\ Q) \xrightarrow{\tau} (\mathcal{K},\mathcal{T},P)}$$

$$\mu\mathsf{Else}\quad \frac{M \neq N}{(\mathcal{K},\mathcal{T},\mathsf{if}\ M=N\ \mathsf{then}\ P\ \mathsf{else}\ Q) \xrightarrow{\tau} (\mathcal{K},\mathcal{T},Q)}$$

$$\mu\mathsf{Dec}\quad \frac{}{(\mathcal{K},\mathcal{T},\mathsf{decrypt}\ \{M\}_n\ \mathsf{as}\ \{x\}_n\ \mathsf{in}\ P) \xrightarrow{\tau} (\mathcal{K},\mathcal{T},P\{M/x\})}$$

$$\mu\mathsf{Split}\quad \frac{}{(\mathcal{K},\mathcal{T},\mathsf{split}\ (M,N)\ \mathsf{as}\ (x,y)\ \mathsf{in}\ P) \xrightarrow{\tau} (\mathcal{K},\mathcal{T},P\{M/x\}\{N/y\})}$$

These rules are rather standard. The μIn rule requires $M \in \mathcal{K}$, thus modeling the adversary sending a known term to the process. Rule μOut instead makes the adversary learn the term M. Rule μDecl and μPar ensure fresh names are generated each time a $(\mathsf{new}\ n)$ is executed.

Moreover, we note that if $(\mathcal{K},\mathcal{T},P)$ is closed (i.e., $\mathcal{K}$ only contains ground terms and P has no free variables) all the states reachable from it are also closed. Additionally, in the rules above the terms M and N are ground, provided that the source state is closed. Finally, a canonical process may evolve to a non-canonical one through μRep:

$$(\mathcal{K},\mathcal{T},!\ (\mathsf{new}\ n)P) \xrightarrow{\tau} (\mathcal{K},\mathcal{T},(\mathsf{new}\ n')P \mid !\ (\mathsf{new}\ n)P)$$

This is indeed a wanted effect: if n has to be kept secret, all all its instances n' must be too (with $n \sim_\alpha n'$); similarly when we study authentication.

Adversary Actions.

We can now define the $\xrightarrow{\spadesuit}$ and $\xrightarrow{\spadesuit_{\mathsf{s/f}}}$ transitions as follows.

$$\spadesuit\mathsf{Decision}_1\quad \frac{M \in \mathcal{K} \qquad M \xmapsto{p}_{\mathsf{DY}} N}{(\mathcal{K},\mathcal{T},P) \xrightarrow{\spadesuit} (\mathcal{K},\mathcal{T},P)^{p,N}}$$

$$\spadesuit\mathsf{Decision}_2 \quad \frac{L, M \in \mathcal{K} \qquad L, M \overset{p}{\longmapsto}_{\mathsf{DY}} N}{(\mathcal{K}, \mathcal{T}, P) \xrightarrow{\spadesuit} (\mathcal{K}, \mathcal{T}, P)^{p,N}}$$

$$\spadesuit\mathsf{Success} \quad \frac{}{(\mathcal{K}, \mathcal{T}, P)^{p,N} \xrightarrow[p]{\spadesuit_\mathsf{s}} (\mathcal{K} \cup \{N\}, \mathcal{T}+1, P)}$$

$$\spadesuit\mathsf{Failure} \quad \frac{}{(\mathcal{K}, \mathcal{T}, P)^{p,N} \xrightarrow[1-p]{\spadesuit_\mathsf{f}} (\mathcal{K}, \mathcal{T}+1, P)}$$

The $\xrightarrow{\spadesuit}$ transitions model the choice of:

- which entailment rule the adversary is going to apply, and
- which arguments it is applied to.

The target state records the term N the adversary is trying to obtain and the probability p the adversary has to actually get it. When the adversary chooses a DYGuess operation, the probability p is $\mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T})$ where $\mathcal{T}$ is taken from the source state of the $\xrightarrow{\spadesuit}$ transition. Once the choice is performed, the ♠Success and ♠Failure rules model the actual application of the entailment rule. In case of success, the term N is added to the adversary's knowledge. Note that the transition $\xrightarrow[p]{\spadesuit_\mathsf{s}}$ (resp. $\xrightarrow[1-p]{\spadesuit_\mathsf{f}}$) records the success (resp. failure) probability, inherited by the entailment rule applied. Finally, note that $\mathcal{T}$ is incremented by either transitions of this kind.

2.4 Secrecy

Having established our adversary–aware semantics, we can evaluate the strength of a given protocol. We first focus on checking whether a protocol guarantees *secrecy*. This property can be easily expressed in our model because the states of our LTS explicitly expose the knowledge $\mathcal{K}$ of the adversary. This allows us to check whether the adversary knows a term M by simply looking whether M belongs in $\mathcal{K}$. Since names in M can be α–converted, we actually check whether $M \sim_\alpha N \in \mathcal{K}$ for some N (written $M \sqsubseteq \mathcal{K}$ for short).

We define the probability that, given an initial knowledge $\mathcal{K}$, the adversary will learn a given term M by interacting with a process P for a *limited* period. In order to accomplish this, we put a bound t on the number of transitions of the computation at hand. This bound limits the sum of

- the number of interactions between the adversary and the process P (i.e., of $\xrightarrow{\mu}$ transitions), and
- the number of operations the adversary can perform (i.e., of $\xrightarrow{\spadesuit}\xrightarrow{\spadesuit_{\mathsf{s/f}}}$ pairs of transitions).

We could instead use two different bounds for each kind of transition; however, this would not affect the results we are going to present. To keep our presentation as concise as possible, we use thus the single parameter t.

We write σ for the generic state $(\mathcal{K}, \mathcal{T}, P)$, and $\sigma^{p,N}$ for $(\mathcal{K}, \mathcal{T}, P)^{p,N}$; also, we write the probability the adversary has to learn the term M as

$$\mathbb{S}^{t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma)$$

For brevity, we often omit the parameters η and $\mathbb{P}_{\mathsf{guess}}$, insisting that they are free. In order to define $\mathbb{S}^{t}_{M}(\sigma)$, we make the following assumptions. The first is technical, while the second one is typical of a "worst case" analysis.

- σ is closed and $\mathcal{K}$ is nonempty (typically, $\mathcal{K}$ contains the bits $0, 1$);
- among the transitions outgoing from a state and labeled by μ or $\spadesuit$, we consider only the one leading to the best state for the adversary. This is fairly standard in formal models: a protocol is regarded as *unsafe* if some execution trace leads to the disclosure of a secret.

Definition 2 *The probability $\mathbb{S}^{t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma)$ is defined by induction on t by the following equations.* [2]

$$\begin{aligned}
\mathbb{S}^{t}_{M}(\sigma) &= 1 \qquad \text{if } M \sqsubseteq \mathcal{K} && (2.1)\\
\mathbb{S}^{0}_{M}(\sigma) &= 0 && (2.2)\\
\mathbb{S}^{t}_{M}(\sigma) &= \max\Big(\Big\{\mathbb{S}^{t-1}_{M}(\sigma')\Big|\ \sigma \xrightarrow{\mu} \sigma'\Big\} \cup \\
&\qquad \Big\{\mathbb{S}^{t}_{M}(\sigma^{p,N})\Big|\ \sigma \xrightarrow{\spadesuit} \sigma^{p,N}\Big\}\Big) && (2.3)\\
\mathbb{S}^{t}_{M}(\sigma^{p,N}) &= \sum \Big\{\Big| q * \mathbb{S}^{t-1}_{M}(\sigma') \ \Big|\ \sigma^{p,N} \xrightarrow[q]{\spadesuit_{\mathsf{s/f}}} \sigma' \Big|\Big\} && (2.4)
\end{aligned}$$

Equation (2.1) checks whether M was disclosed in a given $\sigma = (\mathcal{K}, \mathcal{T}, P)$. Equation (2.2) instead checks whether M was not disclosed and time ran out (i.e., $t = 0$). Equation (2.3) chooses the transition which is the best for the adversary; the chosen transition may be a process action $\xrightarrow{\mu}$ or an adversary action $\xrightarrow{\spadesuit}$ corresponding to some operation *op*. Equation (2.4) considers both the cases when the operation *op* succeeds $\left(\sigma^{p,N} \xrightarrow[p]{\spadesuit_{\mathsf{s}}} \sigma'\right)$ and fails $\left(\sigma^{p,N} \xrightarrow[1-p]{\spadesuit_{\mathsf{f}}} \sigma''\right)$. Then, the probability of discovering M from $\sigma^{p,N}$ is the sum of the probability of discovering M from σ' weighted by p, and from σ'', weighted by $1 - p$. Equation (2.4) could also be written as

$$\begin{aligned}
\mathbb{S}^{t}_{M}(\mathcal{K}, \mathcal{T}, P)^{p,N} &= p \cdot \mathbb{S}^{t-1}_{M}(\mathcal{K} \cup \{N\}, \mathcal{T}+1, P) + \\
&\quad (1-p) \cdot \mathbb{S}^{t-1}_{M}(\mathcal{K}, \mathcal{T}+1, P)
\end{aligned}$$

[2] We assume that, whenever two equations overlap, the topmost one applies.

The right hand sides of the equations in Definition 2 do not depend on the choice of names, i.e. they are not affected by our constrained version of α–conversion. Indeed, we use in (2.1) the relation $\sqsubseteq$ that gets rid of α–renaming; the other cases are trivial (case (2.2)) or follow immediately by induction. Also, from any state σ, only finitely many transitions $\xrightarrow{\mu}$ and $\xrightarrow{\spadesuit}$ exit (up to α–conversion). Therefore, in (2.3), max is applied to a finite set. Moreover that set is nonempty because there is always at least the $\xrightarrow{\spadesuit}$ transition corresponding to the DYCons of some term belonging to the nonempty knowledge $\mathcal{K}$. Thus $\mathbb{S}^{\,t}_{M}(\sigma)$ is well–defined.

2.4.1 A Probabilistic Secrecy Notion.

The probability $\mathbb{S}^{\,t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma)$ measures the strength of a protocol P as a function of the parameters $t, \mathcal{K}, \mathcal{T}, \eta$ and $\mathbb{P}_{\mathsf{guess}}$. We now aim for a definition of *safe* protocol which abstracts from those parameters. We recall that a function is said *negligible* if it approaches zero faster than any rational function. Formally:

Definition 3 *We say that a function $f : \mathbb{N} \to \mathbb{R}$ is negligible iff*

$$\forall k \in \mathbb{N}\ \exists \eta_0 \in \mathbb{N}\ \forall \eta \in \mathbb{N}.\ \eta > \eta_0 \implies |f(\eta)| < \eta^{-k}$$

For example, $2^{-\eta}$ is negligible, while η^{-2} is not.

We regard a protocol as *safe* if, for any adversary with polynomially bounded resources (with respect to the security parameter η), the probability of disclosing a secret (as a function of η) is *negligible*, provided that the cryptosystem is strong, i.e. it can be broken only with a *negligible* probability by such an adversary. This abstracts from the various notions of protocol security found in the computational approach, e.g. [12, 63, 23], and does not imply any of them. Our notion introduces only a few computational aspects, but it refines the standard notions of security used in the formal method approach.

The probability functions that we use have two arguments and so we constrain them to be negligible as follows.

Definition 4 *We call a function $\mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T}) : \mathbb{N} \times \mathbb{N} \to [0, 1]$ η–negligible iff*

1. $\forall \mathcal{T}, \mathcal{T}', \eta.\ \mathcal{T} \le \mathcal{T}' \implies \mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T}) \le \mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T}')$
2. $\forall c \in \mathbb{N}.\ \mathbb{P}_{\mathsf{guess}}(\eta, \eta^c)$ *is negligible*

Requirement 1 simply says that $\mathbb{P}_{\mathsf{guess}}$ is monotonic in its second argument. Requirement 2 imposes that the guessing probability is negligible when $\mathcal{T}$ is a polynomial in η. This condition expresses the robustness of the cryptosystem in presence of an adversary that performed a polynomial number of operations in the previous steps of the computation in hand.

We map the definition of *safe* protocol into our model. Again, we (polynomially) bound the resources of the adversary, letting $t = \eta^k$. Finally, we assume that the adversary initially knows only the terms 0 and 1. Accordingly to these assumptions, and recalling that P, the specification of the protocol, is canonical (see Def. 1), we write σ_0 for the initial state $(\{0,1\}, 0, P)$. We can now define *secrecy* in the following way:

Definition 5 *We say that a protocol P guarantees the $\mathsf{DY}_{\mathbb{P}}$–secrecy of a given term M iff, for any $\mathbb{P}_{\mathsf{guess}} : \mathbb{N} \times \mathbb{N} \to [0,1]$,*

$$\mathbb{P}_{\mathsf{guess}} \text{ is } \eta\text{–negligible} \implies \forall k \in \mathbb{N}.\ \mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0) \text{ is negligible}$$

The above definition can be read as: P guarantees secrecy if, provided we are using a strong cryptosystem, any adversary with polynomially bounded resources has only negligible probability of discovering M. Note that, when $\mathbb{P}_{\mathsf{guess}}$ is not null, bounding the resources of the adversary ($t = \eta^k$) is crucial: for any protocol P which outputs a term where M occurs, we have

$$\lim_{t\to\infty} \mathbb{S}^{t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0) = 1$$

Indeed, an unbounded adversary can repeatedly apply the DYGuess rule on any encryption until it eventually succeeds.

Towards a Comparison with Standard Secrecy.

We compare the above definition of $\mathsf{DY}_{\mathbb{P}}$–*secrecy* with the standard notion of secrecy used in formal models ($\mathsf{DY}_{\mathsf{std}}$–*secrecy*), which assumes a deterministic Dolev–Yao adversary.

Definition 6 *We say that a protocol P guarantees the* $\mathsf{DY}_{\mathsf{std}}$–secrecy *of a term M iff*

$$\forall \sigma.\ \ \sigma_0 \longrightarrow^* \sigma = (\mathcal{K}, \mathcal{T}, P') \implies M \not\in \mathcal{K}$$

where the arrow $\longrightarrow$ stands for either a $\xrightarrow{\mu}$ transition or a pair $\xrightarrow{\spadesuit}\xrightarrow{\spadesuit_s}$ derived from a deterministic *Dolev–Yao rule (i.e. not using a* DYGuess*).*

Note that this definition does *not* bound the number of transitions and therefore, unlike Definition 5, it is not resource–conscious.

It is convenient to rephrase the $\mathsf{DY}_{\mathsf{std}}$–*secrecy* property using $\mathbb{S}^{t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma)$ in order to simplify the comparison between Definitions 5 and 6. An obvious way of making the DYGuess rule harmless is assuming a null guessing probability: this makes our model deterministic.

Definition 7 *Let $\mathbb{P}_0$ be the constant null function. We say that a protocol P guarantees the* DY*–secrecy of a term M iff*

$$\forall t, \eta \in \mathbb{N}.\ \mathbb{S}^{t}_{M,\eta,\mathbb{P}_0}(\sigma_0) = 0$$

Proposition 1 *For any σ, the probability $\mathbb{S}^{t}_{M,\eta,\mathbb{P}_0}(\sigma)$ is either 0 or 1.*

Proof. Trivial induction on t. $\square$

Moreover, it is easy to show by induction that Definitions 6 and 7 are indeed equivalent, as the following proposition states:

Proposition 2 *P guarantees $\mathsf{DY}_{\mathsf{std}}$–secrecy (of M) iff P guarantees DY–secrecy (of M).*

Proof. Easy induction on t and the number of $\longrightarrow$ transitions (see Definition 6). $\square$

Proposition 2 enables us to compare $\mathsf{DY}_{\mathbb{P}}$–secrecy against $\mathsf{DY}_{\mathsf{std}}$–secrecy by comparing $\mathsf{DY}_{\mathbb{P}}$–secrecy against DY–secrecy, as shown in the next section.

2.4.2 Comparing DY and $\mathsf{DY}_{\mathbb{P}}$ Adversaries

The following lemma says that the probability an adversary has to break a protocol increases with the power of the adversary, as expected.

Lemma 1 (Monotonicity) *If $\forall \eta, \mathcal{T} \in \mathbb{N}.\ \mathbb{P}_{\mathsf{g1}}(\eta, \mathcal{T}) \le \mathbb{P}_{\mathsf{g2}}(\eta, \mathcal{T})$ and $t_1 \le t_2$, then*

$$\forall \eta \in \mathbb{N}\ \forall \sigma.\ \mathbb{S}^{t_1}_{M,\eta,\mathbb{P}_{\mathsf{g1}}}(\sigma) \le \mathbb{S}^{t_2}_{M,\eta,\mathbb{P}_{\mathsf{g2}}}(\sigma)$$

Proof. Easy induction on t. $\square$

We introduce two preliminary lemmata that state simple analytic properties. Their proofs are straightforward calculus arguments.

Lemma 2 $\exists \bar{x} > 0\ \forall x.\ \ 0 \le x < \bar{x} \implies 3^{-x} \le 1 - x$

Proof. The function

$$f(x) = 3^{-x} - 1 + x$$

is a $\mathcal{C}^{\infty}(\mathbb{R})$ function with derivative

$$f'(x) = -\ln 3 \cdot 3^{-x} + 1$$

Since $f(0) = 0$ and $f'(0) = -\ln 3 + 1 < 0$ there is some $\bar{x} > 0$ such that

$$\forall x.0 \le x < \bar{x}.\ f(x) \le 0$$

This completes the proof. $\square$

Lemma 3 *For any $c \in \mathbb{N}$,*

$$\lim_{x\to\infty} \frac{1 - 3^{-x^{-c-1}}}{x^{-c}} = 0$$

Proof. If $c = 0$, we have $\lim_{x\to\infty}(1 - 3^{-x^{-1}})/1 = 0$. Otherwise, for $c \geq 1$, the above is an indeterminate form $0/0$. By l'Hôpital's rule, we get

$$\lim_{x\to\infty} \frac{1 - 3^{-x^{-c-1}}}{x^{-c}} = \lim_{x\to\infty} \frac{-\ln 3 \cdot 3^{-x^{-c-1}} \cdot (-(-c-1)x^{-c-2})}{-c \cdot x^{-c-1}} =$$
$$= -\frac{\ln 3 \cdot (c+1)}{c} \cdot \lim_{x\to\infty} \frac{3^{-x^{-c-1}}}{x} = 0$$

□

Our main result states the equivalence of $\mathsf{DY}_{\mathbb{P}}$–secrecy and DY–secrecy.

Theorem 1 *P guarantees the $\mathsf{DY}_{\mathbb{P}}$–secrecy of M if and only if P guarantees the DY–secrecy of M.*

Proof. We rewrite the statement in the following way:

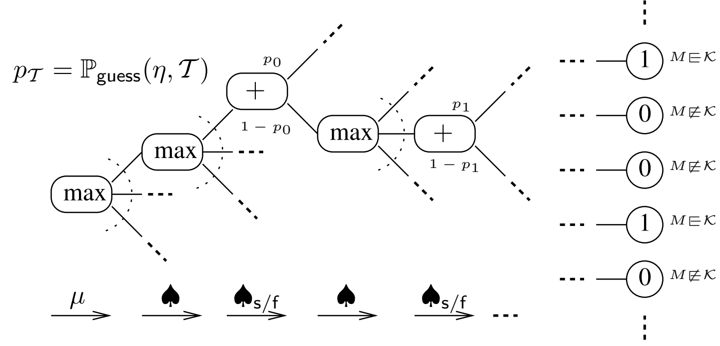
P does not guarantee $\mathsf{DY}_{\mathbb{P}}$–secrecy $\iff$ P does not guarantee DY–secrecy

($\Leftarrow$) We have that, for some $\varepsilon > 0$, $\exists t \in \mathbb{N}$. $\mathbb{S}^{t}_{M,\eta,\mathbb{P}_0}(\sigma_0) = \varepsilon$. Therefore we can apply the monotonicity lemma and state that, for every $\eta > t$, $\mathbb{S}^{\eta}_{M,\eta,\mathbb{P}_0}(\sigma_0) \geq \varepsilon$. This implies that $\mathbb{S}^{\eta}_{M,\eta,\mathbb{P}_0}(\sigma_0)$ is not negligible (as a function of η). Since $\mathbb{P}_0$ is η–negligible, we conclude that P does not guarantee the $\mathsf{DY}_{\mathbb{P}}$–secrecy of M.

($\Rightarrow$) Intuitively, we examine the tree whose branches represent each possible execution of P, considering that the operations of the adversary may or may not succeed. By hypothesis, the probability the adversary has to discover M is non–negligible. This probability can be expressed by a sum over all branches of the tree; however, we show that it is negligible the probability of discovering M summed over all branches that require at least one successful guess. Therefore, the probability of the branch where every guessing attempt fails must be non–negligible, and thus positive. This branch shows that a DY attack for P exists, so we conclude that P does not guarantee DY–secrecy.

We now proceed with the proof. By hypothesis, for some k and some η–negligible $\mathbb{P}_{\mathsf{guess}}$, $\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0)$ is not negligible.

We now examine the definition of $\mathbb{S}^{t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\mathcal{K}, \mathcal{T}, P)$. Given t, M, η, $\mathbb{P}_{\mathsf{guess}}$, $\mathcal{K}$, $\mathcal{T}$, and P, we can fully expand the definition to form a tree like the following one:



We name this tree $\bar{T}$. The tree $\bar{T}$ is formed by internal nodes, which can be either max nodes or *weighted sum* nodes, and leaf nodes, which can be either a constant 0 or a constant 1. The max nodes can have an arbitrary number of children (one for each $\xrightarrow{\mu}$ and $\xrightarrow{\spadesuit}$ transition). The *weighted sum* nodes have exactly two children, each with its weight (the probability of succeeding or failing to apply a Dolev–Yao rule) as a label on the edge.

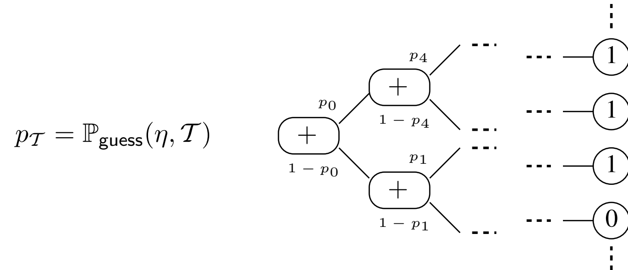
We denote the set of all root-to-leaf simple paths of $\bar{T}$ with $\bar{\Pi}$. For every $\bar{\pi} \in \bar{\Pi}$ we define a *weight*, by letting $\mathsf{weight}(\bar{\pi})$ be the product of all the edge labels in the path; we also define $\mathsf{length}(\bar{\pi})$ as the length of the path and $\mathsf{result}(\bar{\pi})$ as the value of the leaf at the end of the path.

A simple inductive reasoning shows that the following properties hold:

$$\forall \bar{\pi} \in \bar{\Pi}.\ \ \mathsf{length}(\bar{\pi}) \leq 2t \tag{2.5}$$

$$\forall \bar{\pi} \in \bar{\Pi}.\ \ \mathbb{S}^{t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\mathcal{K}, \mathcal{T}, P) \geq \mathsf{weight}(\bar{\pi}) \cdot \mathsf{result}(\bar{\pi}) \tag{2.6}$$

We now *simplify* the tree $\bar{T}$ by removing every max node from it and simply replacing it with one of its children which evaluate to the maximum. This new tree is made only of weighted sum nodes and leaf nodes. Simplify further the tree as follows: remove all the sums with (outgoing edges with) weights 0 and 1 derived from Dolev–Yao rules other than the $\mathsf{DYGuess}$ rule; replace them with the $\mathsf{success}$ subtree (the one reachable through the edge with weight 1). The resulting tree is thus formed only by weighted sums with weights $\mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T}_i)$ and $1 - \mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T}_i)$ and leaves. It has the following form.



We name this tree T and call it the simplified tree of $\bar{T}$. Note that, given t, M, η, $\mathbb{P}_{\mathsf{guess}}$, $\mathcal{K}$, $\mathcal{T}$, and P, the tree T is still equivalent to the old tree, in

that it still evaluates to $\mathbb{S}^{t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\mathcal{K},\mathcal{T},P)$. In a similar way to $\bar{\Pi}$, we write Π for the set of every root-to-leaf simple path of T. Moreover, we extend $\mathsf{weight}(-)$, $\mathsf{length}(-)$ and $\mathsf{result}(-)$ to every $\pi \in \Pi$.
Again, a simple inductive reasoning shows that the following properties hold:

$$\forall \pi \in \Pi.\ \ \mathsf{length}(\pi) \leq t \tag{2.7}$$

$$\sum_{\pi\in\Pi} \mathsf{weight}(\pi) = 1 \tag{2.8}$$

$$\mathbb{S}^{t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\mathcal{K},\mathcal{T},P) = \sum_{\substack{\pi \in \Pi \\ \mathsf{result}(\pi)=1}} \mathsf{weight}(\pi) \tag{2.9}$$

In order to prove the theorem, we now study how the tree $\bar{T}$ corresponding to $\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0)$ changes as η increases. For every η, we write $\bar{T}^\eta$ and $\bar{\Pi}^\eta$ (T^η and Π^η) respectively for the corresponding tree (simplified tree) and the set of its paths. Also, we write π^η_{fail} for the path of Π^η that represents the event in which the $\mathsf{DYGuess}$ rule always fails. We denote the labels of the edges of π^η_{fail} by $\mathbb{P}_{\mathsf{guess}}(\eta,\mathcal{T}^\eta_i)$, therefore $\mathsf{weight}(\pi^\eta_{\mathsf{fail}}) = \prod_{i=0}^{\mathsf{length}(\pi^\eta_{\mathsf{fail}})}(1-\mathbb{P}_{\mathsf{guess}}(\eta,\mathcal{T}^\eta_i))$.

We now claim that the hypothesis implies that, for some η, $\mathsf{result}(\pi^\eta_{\mathsf{fail}})$ is 1. The proof is by contradiction. Assume $\mathsf{result}(\pi^\eta_{\mathsf{fail}}) = 0$ for every η: we obtain

$$\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0) = \sum_{\substack{\pi \in \Pi^\eta \\ \mathsf{result}(\pi)=1}} \mathsf{weight}(\pi) =$$

$$= \sum_{\substack{\pi \in \Pi^\eta \setminus \{\pi^\eta_{\mathsf{fail}}\} \\ \mathsf{result}(\pi)=1}} \mathsf{weight}(\pi) \leq \sum_{\pi \in \Pi^\eta \setminus \{\pi^\eta_{\mathsf{fail}}\}} \mathsf{weight}(\pi)$$

which by (2.8), yields

$$\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0) \leq 1 - \mathsf{weight}(\pi^\eta_{\mathsf{fail}}) =$$

$$= 1 - \prod_{i=0}^{\mathsf{length}(\pi^\eta_{\mathsf{fail}})} \Big(1-\mathbb{P}_{\mathsf{guess}}(\eta,\mathcal{T}^\eta_i)\Big) \leq$$

$$\leq 1 - \Big(1-\mathbb{P}_{\mathsf{guess}}(\eta,\eta^k)\Big)^{\mathsf{length}(\pi^\eta_{\mathsf{fail}})} \tag{2.10}$$

The last inequality derives from $\mathbb{P}_{\mathsf{guess}}$ being η–negligible and $\mathcal{T}^\eta_i \leq \eta^k$. Applying (2.7) to T^η yields $\mathsf{length}(\pi^\eta_{\mathsf{fail}}) \leq \eta^k$. Therefore, from (2.10) we obtain the following asymptotic bound

$$\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0) \leq 1 - \Big(1-\mathbb{P}_{\mathsf{guess}}(\eta,\eta^k)\Big)^{\eta^k}$$

that, by Lemma (2), implies (asymptotically)

$$\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0) \leq 1 - 3^{-\mathbb{P}_{\mathsf{guess}}(\eta,\eta^k)\cdot\eta^k} \tag{2.11}$$

We now find a contradiction by showing that $\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0)$ is negligible. For any $c \in \mathbb{N}$, we show that $\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0) \leq \eta^{-c}$ if η is sufficiently large. Since by hypothesis $\mathbb{P}_{\mathsf{guess}}$ is η–negligible, we can assume η to be large enough such that $\mathbb{P}_{\mathsf{guess}}(\eta,\eta^k)\cdot\eta^k \leq \eta^{-c-1}$. From this and (2.11) we obtain

$$\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0) \leq 1 - 3^{-\eta^{-c-1}}$$

Therefore, by Lemma (3), $\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0)$ is negligible.

Now, having established that $\mathsf{result}(\pi^{\eta}_{\mathsf{fail}}) = 1$ for some η, we show that P does not guarantee DY–secrecy, i.e. that for some t, $\mathbb{S}^{t}_{M,\eta,\mathbb{P}_0}(\sigma_0) > 0$. We start by *lifting* the path $\pi^{\eta}_{\mathsf{fail}}$ from the simplified tree T^{η} to the non simplified tree $\bar{T}^{\eta}$, by choosing the path in $\bar{\Pi}^{\eta}$ leading to the same leaf which $\pi^{\eta}_{\mathsf{fail}}$ leads to. We name the path thus obtained $\bar{\pi}^{\eta}_{\mathsf{fail}}$.

We now study the tree $\bar{T}$ which corresponds to $\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_0}(\sigma_0)$, comparing it to the tree $\bar{T}^{\eta}$ which instead corresponds to $\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0)$. A simple inductive reasoning shows that the two trees are identical except for the labels of the edges. Therefore there is some path $\bar{\pi} \in \bar{\Pi}$ which corresponds to $\bar{\pi}^{\eta}_{\mathsf{fail}}$. As $\bar{\pi}^{\eta}_{\mathsf{fail}}$, the path $\bar{\pi}$ represents the event in which the DYGuess rule always fails and the other rules always succeed. In $\bar{T}$, the weight associated to the failure of a DYGuess rule is $1 - \mathbb{P}_0(\eta, \mathcal{T}^{\eta}_i) = 1 - 0 = 1$ and the weight associated to the success of other rules is 1 as well. Therefore, all the labels on the edges of $\bar{\pi}$ are equal to 1 and we have $\mathsf{weight}(\bar{\pi}) = 1$. Moreover, we also have $\mathsf{result}(\bar{\pi}) = 1$ since the path $\bar{\pi}$, as $\bar{\pi}^{\eta}_{\mathsf{fail}}$, leads to a leaf with a constant 1.

Applying (2.6) yields

$$\mathbb{S}^{\eta^k}_{M,\eta,\mathbb{P}_0}(\sigma_0) \geq \mathsf{weight}(\bar{\pi}) \cdot \mathsf{result}(\bar{\pi}) = 1$$

thus proving that P does not guarantee the DY–secrecy of M. □

Theorem 1 and Proposition 2 state that the definitions of $\mathsf{DY}_{\mathsf{std}}$–secrecy, DY–secrecy, and $\mathsf{DY}_{\mathbb{P}}$–secrecy are all equivalent. Intuitively, an adversary with unbounded resources, but no hope of guessing a key, is as powerful as an adversary that might guess a key (with a small probability) but can only act for a bounded amount of time. This equivalence, however, only holds asymptotically (with respect to the key length).

2.5 Authentication

We extend our model to deal with authentication protocols between an *initiator* and a *responder*. To this purpose, we use a *non-injective* authentication property [62, 35]. Informally, non injective authentication requires that, whenever the responder completes its protocol, it must have interacted with the initiator and must agree with the initiator on some set of values.

Mimicking the Woo–Lam approach [70], we introduce two annotations for when the initiator starts an authentication run ($\mathsf{begin}\ M$) and for when the responder completes it ($\mathsf{end}\ M$). These take the form of prefixes. The syntax of processes becomes the following.

$$\begin{array}{lll} P ::= \ldots & & \\ \quad\mid & \mathsf{begin}\ M.P & \text{authentication start} \\ \quad\mid & \mathsf{end}\ M.P & \text{authentication end} \end{array}$$

Intuitively, the initiator and the responder should agree on the value M.

We also augment the states σ of our LTS to include a set of authentication events $\mathcal{E}$. The generic state σ now is $(\mathcal{K}, \mathcal{T}, P, \mathcal{E})$. The set $\mathcal{E}$ records all the actions $\mathsf{begin}\ M$ and $\mathsf{end}\ M$ the process performs during its execution. Formally, we add the following rules to our operational semantics:

$$\mu\mathsf{Begin}\quad \frac{}{(\mathcal{K}, \mathcal{T}, \mathsf{begin}\ M.P, \mathcal{E}) \xrightarrow{\tau} (\mathcal{K}, \mathcal{T}, P, \mathcal{E} \cup \{\mathsf{begin}\ M\})}$$

$$\mu\mathsf{End}\quad \frac{}{(\mathcal{K}, \mathcal{T}, \mathsf{end}\ M.P, \mathcal{E}) \xrightarrow{\tau} (\mathcal{K}, \mathcal{T}, P, \mathcal{E} \cup \{\mathsf{end}\ M\})}$$

The rules $\mu\mathsf{Decl}$ and $\mu\mathsf{Par}$ change as follows, because they use they set $\mathcal{E}$. In all the other rules, the set $\mathcal{E}$ is added in the source and target states of transitions, with no change.

$$\mu\mathsf{Decl}\quad \frac{n \text{ does not occur in any term of } \mathcal{K} \text{ and of } \mathcal{E}}{(\mathcal{K}, \mathcal{T}, (\mathsf{new}\ n)P, \mathcal{E}) \xrightarrow{n} (\mathcal{K}, \mathcal{T}, P, \mathcal{E})}$$

$$\mu\mathsf{Par}\quad \frac{(\mathcal{K}, \mathcal{T}, P, \mathcal{E}) \xrightarrow{\mu} (\mathcal{K}', \mathcal{T}, P', \mathcal{E}') \qquad \mu = \tau \vee \mu \notin \mathsf{fn}(Q)}{(\mathcal{K}, \mathcal{T}, P|Q, \mathcal{E}) \xrightarrow{\mu} (\mathcal{K}', \mathcal{T}, P'|Q, \mathcal{E}')}$$

As done in the previous section, we write $\mathbb{A}^t_{\eta,\mathbb{P}_{\mathsf{guess}}}(\mathcal{K}, \mathcal{T}, P, \mathcal{E})$ for the probability the adversary has to disrupt authentication at a certain step of the computation of P, assuming that

- the adversary's initial knowledge is $\mathcal{K}$,
- the adversary has performed $\mathcal{T}$ operations in the previous steps,
- the adversary can guess keys with probability $\mathbb{P}_{\mathsf{guess}}(\eta)$,
- the number of transitions is limited by t, and

- the authentication actions performed in the previous steps were recorded in $\mathcal{E}$.

Under the same assumptions we made in Sect. 2.4, we define $\mathbb{A}^{t}_{\eta,\mathbb{P}_{\mathsf{guess}}}(\mathcal{K}, \mathcal{T}, P, \mathcal{E})$ as follows. The definition is essentially the same as Definition 2, except for the first item below. It specifies when the adversary succeeds: there is an $\mathsf{end}\ M$ action which is not preceded by a matching $\mathsf{begin}\ M$.

Definition 8 $\mathbb{A}^{t}_{\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma)$ *is defined by induction on t by the following equations.*

$$\begin{aligned}
\mathbb{A}^{t}(\sigma) &= 1 \qquad \text{if } \exists M.\ \mathsf{end}\ M \in \mathcal{E} \wedge \mathsf{begin}\ M \notin \mathcal{E} && (2.12)\\
\mathbb{A}^{0}(\sigma) &= 0 && (2.13)\\
\mathbb{A}^{t}(\sigma) &= \max\Big(\Big\{\mathbb{A}^{t-1}(\sigma')\Big|\ \sigma \xrightarrow{\mu} \sigma'\Big\} \cup \\
&\qquad \Big\{\mathbb{A}^{t}(\sigma^{p,N})\Big|\ \sigma \xrightarrow{\spadesuit} \sigma^{p,N}\Big\}\Big) && (2.14)\\
\mathbb{A}^{t}(\sigma^{p,N}) &= \sum \Big\{\Big| q * \mathbb{A}^{t-1}(\sigma')\ \Big|\ \sigma^{p,N} \xrightarrow[q]{\spadesuit_{\mathsf{s/f}}} \sigma' \Big|\Big\} && (2.15)
\end{aligned}$$

As in Definition 2, the topmost applicable equation is taken. Also, note that α–conversion does not affect the result of the equations above. Indeed, the new name replaces the old one in all the components of a state, notably in $\mathcal{E}$. Finally, in equation (2.14) the operator max is applied to a finite nonempty set. Thus the probability $\mathbb{A}^{t}_{\eta,\mathbb{P}_{\mathsf{guess}}}(\mathcal{K}, \mathcal{T}, P, \mathcal{E})$ is well-defined.

2.5.1 A Probabilistic Authentication Notion

As for secrecy, we can abstract from the parameters $t, \mathbb{P}_{\mathsf{guess}}, \mathcal{K}, \mathcal{T}$ and $\mathcal{E}$. We say that a protocol P guarantees authentication if any probabilistic polynomially bounded adversary with negligible guessing probability can disrupt authentication only with a negligible probability. We write σ_0 for the initial state $(\{0,1\}, 0, P, \emptyset)$.

Definition 9 *We say that a protocol P guarantees* $\mathsf{DY}_{\mathbb{P}}$*–authentication iff, for any function* $\mathbb{P}_{\mathsf{guess}} : \mathbb{N} \times \mathbb{N} \to [0,1]$,

$$\mathbb{P}_{\mathsf{guess}} \text{ is } \eta\text{–negligible} \implies \forall k \in \mathbb{N}.\ \mathbb{A}^{\eta^k}_{\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma_0) \text{ is negligible}$$

Below, we give the standard definition of authentication.

Definition 10 *We say that a protocol P guarantees* $\mathsf{DY}_{\mathsf{std}}$–authentication *iff*

$$\forall \sigma.\ (\sigma_0) \longrightarrow^{*} \sigma = (\mathcal{K}, \mathcal{T}, P', \mathcal{E}) \implies \nexists M.\ \mathsf{end}\ M \in \mathcal{E} \wedge \mathsf{begin}\ M \notin \mathcal{E}$$

where the arrow $\longrightarrow$ stands for either a $\xrightarrow{\mu}$ transition or a pair $\xrightarrow{\spadesuit}\xrightarrow{\spadesuit_{\mathsf{s}}}$ derived from a deterministic *Dolev–Yao rule (i.e. not using a* $\mathsf{DYGuess}$*).*

We can also rephrase the above using $\mathbb{A}^t_{\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma)$ as follows.

Definition 11 *Let $\mathbb{P}_0$ be the constant null function. We say that a protocol P guarantees* DY*–authentication iff*

$$\forall t, \eta \in \mathbb{N}.\ \mathbb{A}^t_{\eta,\mathbb{P}_0}(\sigma_0) = 0$$

Of course, the two definitions above are equivalent.

Proposition 3 *P guarantees* $\mathsf{DY}_{\mathsf{std}}$*–authentication if and only if P guarantees* DY*–authentication.*

Proof. Easy induction on t and the number of $\longrightarrow$ transitions (see Definition 10). □

We now compare $\mathsf{DY}_{\mathsf{std}}$–authentication and DY–authentication. As for secrecy, also for authentication the DY adversary is as powerful as the $\mathsf{DY}_{\mathbb{P}}$ one.

Theorem 2 *P guarantees* $\mathsf{DY}_{\mathbb{P}}$*–authentication if and only if P guarantees* DY*–authentication.*

The proof is similar to the proof of Theorem 1, and therefore we omit it.

We finally observe that Theorem 2 and Proposition 3 show the equivalence between $\mathsf{DY}_{\mathsf{std}}$–authentication, DY–authentication, and $\mathsf{DY}_{\mathbb{P}}$–authentication.

2.6 Non Constant–Cost Operations

In the definition of $\mathbb{S}^t_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\mathcal{K}, \mathcal{T}, P)$ the bound t put on the adversary is decreased by one unit for each application of an entailment rule. So, every operation of the adversary has a constant cost.

This is an oversimplification; a more adequate modeling can be obtained by relating the amount of resources consumed to the probability of success of the operation involved. For instance, we could have the following $\mathsf{DYGuess}$ rule.

$$\mathsf{DYGuess} \quad \{M\}_n \overset{\mathbb{P}_{\mathsf{guess}}(\eta,\mathcal{T},r)}{\underset{r}{\longmapsto}}_{\mathsf{DY}} n \qquad (r \geq 1)$$

The quantity r represents the amount of resources the adversary may decide to spend in guessing. Depending also on the parameter r, $\mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T}, r)$ computes a value that expresses the probability of success. The more resources are consumed, the more likely guessing is; thus we require $\mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T}, r)$ to be monotonic on r.

We slightly change the meaning of $\mathcal{T}$, by letting it count the number of resources units spent by the adversary in the previous steps of the computation, instead of the number of the operations performed. Note that, if only constant cost operations are considered, the two notions collapse.

Of course, we consider strong cryptosystems, so we still need to assume that $\mathbb{P}_{\mathsf{guess}}$ is η–negligible. The definition of η–negligible function is adapted as follows:

1. $\forall \mathcal{T}, \mathcal{T}', r, r', \eta.\ \mathcal{T} \leq \mathcal{T}' \wedge r \leq r' \implies \mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T}, r) \leq \mathbb{P}_{\mathsf{guess}}(\eta, \mathcal{T}', r')$

2. $\forall c \in \mathbb{N}.\ \mathbb{P}_{\mathsf{guess}}(\eta, \eta^c, \eta^c)$ is negligible

The second point requires that the guessing probability is negligible provided $\mathcal{T}$ and r are polynomials. Intuitively, if the adversary spent a polynomially bounded amount of resources in the past ($\mathcal{T}$ is a polynomial) and tries to guess spending only a polynomial amount of resources (r is a polynomial), the probability that guessing succeeds should be negligible.

We also need to adapt the semantics of the adversary actions. We assign $r = 1$ to operations other than a $\mathsf{DYGuess}$, and we use the following rules. Note that the parameter r is not a constant: it may be chosen at run–time by the adversary.

$$\spadesuit\mathsf{Decision}_1 \quad \frac{M \in \mathcal{K} \qquad M \overset{p}{\underset{r}{\longmapsto}}_{\mathsf{DY}} N}{(\mathcal{K}, \mathcal{T}, P) \xrightarrow{\spadesuit} (\mathcal{K}, \mathcal{T}, P)^{r,p,N}}$$

$$\spadesuit\mathsf{Decision}_2 \quad \frac{L, M \in \mathcal{K} \qquad L, M \overset{p}{\underset{r}{\longmapsto}}_{\mathsf{DY}} N}{(\mathcal{K}, \mathcal{T}, P) \xrightarrow{\spadesuit} (\mathcal{K}, \mathcal{T}, P)^{r,p,N}}$$

$$\spadesuit\mathsf{Success} \quad \frac{}{(\mathcal{K}, \mathcal{T}, P)^{r,p,N} \xrightarrow[p]{\spadesuit_{\mathsf{s}}} (\mathcal{K} \cup \{N\}, \mathcal{T} + r, P)}$$

$$\spadesuit\mathsf{Failure} \quad \frac{}{(\mathcal{K}, \mathcal{T}, P)^{r,p,N} \xrightarrow[1-p]{\spadesuit_{\mathsf{f}}} (\mathcal{K}, \mathcal{T} + r, P)}$$

Note that now $\mathcal{T}$ is incremented by r, instead of 1, to count the number of resource units spent by the adversary in the running computation.

Finally, we change the definition of $\mathbb{S}^{\,t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\mathcal{K}, P, \mathcal{E})$ so that, whenever an entailment rule requiring r resources is applied, the number t is decreased by r units. Formally, we have

$$\mathbb{S}^{\,t}_{M}(\sigma) = 1 \qquad \text{if } M \sqsubseteq \mathcal{K} \tag{2.16}$$

$$\mathbb{S}^{\,0}_{M}(\sigma) = 0 \tag{2.17}$$

$$\begin{aligned}\mathbb{S}^{\,t}_{M}(\sigma) &= \max\Big(\Big\{\mathbb{S}^{\,t-1}_{M}(\sigma')\Big|\ \sigma \xrightarrow{\mu} \sigma'\Big\} \cup \\ &\qquad \Big\{\mathbb{S}^{\,t}_{M}(\sigma^{r,p,N})\Big|\ \sigma \xrightarrow{\spadesuit} \sigma^{r,p,N} \wedge r \leq t\Big\}\Big) \qquad (2.18) \\ \mathbb{S}^{\,t}_{M}(\sigma^{r,p,N}) &= \sum \Big\{\Big| q * \mathbb{S}^{\,t-r}_{M}(\sigma') \ \Big|\ \sigma^{r,p,N} \xrightarrow[q]{\spadesuit_{\mathsf{s/f}}} \sigma' \Big|\Big\} \qquad (2.19)\end{aligned}$$

Note that equation (2.18) only considers the adversary actions that can be actually performed, i.e. those with $1 \leq r \leq t$. This implies that only a finitely branching fragment of the transition system is considered. Thus, the max in equation (2.18) is applied to a finite non empty set and therefore $\mathbb{S}^{\,t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma)$ is well–defined.

The definition of $\mathsf{DY}_{\mathbb{P}}$–secrecy needs no changes: it simply uses the new definition of η–negligible function and the above equations for $\mathbb{S}^{\,t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\sigma)$.

Of course, it is straightforward to change our authentication–related definitions in a similar way, thus using this more precise resource accounting also for $\mathsf{DY}_{\mathbb{P}}$–authentication.

Our results, notably Theorems 1 and 2, still hold when non constant cost operations are added to our model. The proofs change only in a marginal way. Similarly, following the line of this section, we could add more parameters to the function $\mathbb{P}_{\mathsf{guess}}$ to get an even more detailed, concrete view of the robustness of the cryptosystem. For instance, we could made $\mathbb{P}_{\mathsf{guess}}$ to increase as the adversary knowledge $\mathcal{K}$ becomes larger, i.e. when the adversary intercepts more messages. We foresee that our results still remain valid, provided the definition of η–negligible function is suitably changed.

2.7 Extensions

Our model can easily be further extended to include other adversary actions and cryptographic primitives. First, we present those extensions that do not significatively affect our results; in particular, it is easy to prove Theorems 1 and 2 also in the model enriched with all the four extensions below, provided that the probability to break the cryptosystem is kept negligible.

Private Channels

A typical way of specifying secure links is using private channels, as done in the Spi and the π–calculus. Adding private channels to our calculus is straightforward. Extending our adversary–aware semantics is easy, because terms sent through the private channels do not affect the knowledge of the adversary $\mathcal{K}$.

Breaking without Guessing

In our model we allowed the adversary to *guess*, i.e. to deduce the key from a given encryption. We can also give the adversary the ability to *break* encryptions and get the plaintext without guessing the key. We can do this by adding the rule

$$\mathsf{DYBreak} \quad \{M\}_n \stackrel{p}{\longmapsto}_{\mathsf{DY}} M$$

As for *guessing*, we assume of course that p is expressed as some negligible function $\mathbb{P}_{\mathsf{break}}(\eta)$. Note that this operation is less powerful than guessing: only the message M is disclosed with breaking, while guessing reveals both the key and M. Note however that assuming $\mathbb{P}_{\mathsf{break}}$ negligible is still insufficient to fully guarantee robustness in a computational sense.

Blind Guessing

We could also allow *blind guessing*, i.e. guessing without using any previous knowledge.

$$\mathsf{DYBlindGuess} \quad \stackrel{p}{\longmapsto}_{\mathsf{DY}} n$$

This rule differs from DYGuess in that it allows the adversary to guess, e.g., a key that has not yet been used by the protocol, or a nonce. Therefore, it is strictly more powerful than the DYGuess rule. Pragmatically, this random guessing operation requires the probability function to be $2^{-\eta}$.

Other Cryptographic Primitives

So far, we only considered symmetric encryption. Modeling asymmetric encryption and digital signatures requires additional rules that explicitly break the cryptosystem beyond the standard Dolev–Yao rules for asymmetric cryptography. Many different rules can be used: we just give an example.

We write n^+ and n^- for public and private keys, respectively. The DYInvert rule below allows the adversary to compute the private key from the corresponding public key. The DYFakeSign rule instead allows the adversary to sign a message using only the public key.

$$\mathsf{DYInvert} \quad n^+ \stackrel{p}{\longmapsto}_{\mathsf{DY}} n^- \qquad \mathsf{DYFakeSign} \quad M, n^+ \stackrel{q}{\longmapsto}_{\mathsf{DY}} [M]_{n^-}$$

As above, p and q should be negligible.

We now discuss some other extensions that require substantial and deeper modifications to our model, that we do not further investigate here.

Generalizing the Security Property

Currently, we considered only secrecy and authentication properties. However, we could extend our results to other security properties as well. In particular, it could be useful to develop some kind of probabilistic temporal logic in order to do that (see e.g. [45]). Besides specifying security properties, the logic might help us to better understand the difference between the DY and the $\mathsf{DY}_{\mathbb{P}}$ models. For example, we can consider any property φ expressed in the logic, and test whether these models are equivalent with respect to that property, i.e.

$$\forall P \text{ process. } \varphi(P) \text{ in } \mathsf{DY} \iff \varphi(P) \text{ in } \mathsf{DY}_{\mathbb{P}}$$

Actually, Sects. 2.4 and 2.5 establish the above proposition for secrecy and authentication respectively.

Random Choice

Sometimes protocols are specified using a toss–a–coin operation. In the computational approach this is easy to model, while the usual non–deterministic operator $+$ of process calculi is not adequate. A probabilistic choice operator $+_p$ (see e.g. Larsen and Skou [46]) accommodates well in our framework, originating transitions on the form $\xrightarrow[p]{\mu}$. The operator $+_p$ could help formalizing security properties that have been faced so far only within the computational approach, e.g. the correctness of zero-knowledge protocols [40].

Behavioural Equivalence

Some authors advocate the use of *behavioural equivalence* for security properties. Besides the work mentioned in the Introduction, there are proposals for studying non–interference. A classical deterministic approach is presented in [35], while a probabilistic version of non–interference is studied in [58]. All these proposals and those cited in the Introduction assume a standard Dolev–Yao adversary. It would be interesting to study if and how these notions change when a $\mathsf{DY}_{\mathbb{P}}$ adversary is assumed, instead.

Partial Information

In the real world, the adversary may not be able to guess keys but still be able to perform statistical attacks and gather some partial information from intercepted messages. Unlike the computational complexity model, ours is not apt to study this kind of attacks. This is because we use a set to represent the knowledge of the adversary, thus implicitly assuming that the adversary either has complete knowledge of a term or no knowledge at all. In [27],

Clark, Hunt, and Malacaria deal with information leakage using Shannon's entropy. Further investigation is needed to see whether this approach can also be applied to process calculi and formal methods.

Average–case vs. Worst–case Analysis

For studying secrecy, in the definition of $\mathbb{S}^{t}_{M,\eta,\mathbb{P}_{\mathsf{guess}}}(\mathcal{K}, \mathcal{T}, P)$, we only consider the transition which is the best one for the adversary (i.e., the worst case for P). The same holds for authentication in the definition of $\mathbb{A}^{t}_{\eta,\mathbb{P}_{\mathsf{guess}}}(\mathcal{K}, \mathcal{T}, P, \mathcal{E})$. Thus, we perform a *worst–case* analysis. An alternative would be an *average–case* analysis: we could consider all the transitions and weight each of them according to a given distribution. The hard point is to define a distribution that faithfully reflects both the scheduling of the concurrent actions by the protocol and the choices the adversary makes among its operations.

2.8 Conclusions

In this chapter we presented a simple process calculus that can be used to specify cryptographic protocols. We introduced two distinct notions of secrecy, one borrowed from formal methods (DY–secrecy) and one adapted from the computational complexity theory ($\mathsf{DY}_{\mathbb{P}}$–secrecy). Under suitable assumptions, we proved the equivalence of these two secrecy definitions. A similar result holds for authentication.

Consequently, the perfect encryption assumption of the formal method approach can be weakened, as we propose. Thus, one can use standard techniques for protocol analysis (e.g. type systems [1], control flow analysis [16, 14], model checkers [50, 53], etc.) to formally verify whether a given protocol is $\mathsf{DY}_{\mathbb{P}}$–safe. Indeed, in Chapters 3 and 4 we shall define a technique for protocol verification that can be used to this purpose.

As the reader may have noticed, our proofs only depend on the transition system and are independent of the calculus and of the definition of its semantics. Therefore our result can be easily extended to other calculi, with non–standard operations, and to other properties, involving different measures of transitions than probability. Indeed, we showed that protocol analysis does not require a full inspection of transition systems: only those paths do suffice, that can be taken by a Dolev–Yao adversary.

Chapter 3

Tree Automata

Abstract

A technique is presented for computing a finite over-approximation of a language of terms, modulo rewriting. The language is represented by an arbitrary term automata. The approximation is a term automata as well, and it already embodies rewriting, i.e. its language includes all the original terms *and* all their possible rewritings. In Chapter 4, we shall use this technique to statically verify cryptographic protocols, expressed in process calculi involving non-free algebras of terms.

3.1 Introduction

In the last years, various process calculi have been used to specify cryptographic protocols [52, 4, 14], making it possible to build automatic tools for reasoning on them. Among these, we mention dynamic analyzers [53, 50] and static analyzers [15, 16, 17, 18, 14, 24, 1, 59].

Each of the above calculi fixes once and for all a specific set of cryptographic primitives, and relies on having a *single* representation for each value. Further, destructing composed terms is only possible through pattern matching, which is a feature of these calculi, and *not* of their term algebras which are free.

For instance, the simple process calculus of Chapter 2 considers pairs and encryptions, only. There, terms are built using constructors, and destruction is performed only through processes split M as (x, y) in P and decrypt M as $\{x\}_N$ in P. So, there we indeed use pattern matching for destruction, and terms form a free algebra.

Process analysis can often exploit the freeness of the term algebra. For instance, the *control flow analysis* (CFA) of calculi for cryptographic protocols [57] succeeds in computing a finite representation for (an approximation of) the set of terms that can be dynamically bound to variables occurring in the protocol specification. Briefly, this is done for the Spi–calculus [4] and related calculi by generating constraints over set of terms, and then solve them using the Succinct Solver [65]. We shall give more details about this process in Chapter 4. Here, we simply note that the constraint solving procedure is simplified when the terms are free. Indeed, under the freeness assumption the solver can assume that syntactically different terms have distinct meaning. Basically, each term is always in normal form.

Unfortunately, as we argued in Sect. 1.1, not all cryptographic primitives admit a single representation, e.g. XOR. In this case, if we want to follow the CFA approach for protocol analysis, we need a technique for solving constraints over sets of terms that does not rely on term freeness. For instance, we could choose a set of the most common primitives, including non-free ones, and try to adapt the solving process to this case. Rather, we

shall follow a more general approach, and not use a fixed set of primitives. Instead, we allow the primitives to be *specified* along the protocol, through a term *rewriting system* $\mathcal{R}$. Further, we put *no* restrictions on $\mathcal{R}$. In particular, we do not require $\mathcal{R}$ to be confluent or terminating.

In this chapter, we present a static analysis technique for approximating term sets modulo rewriting. This approximation technique is the foundation of the static analysis of protocols we shall define in Chapter 4. Here, we focus on how to finitely represent term sets, and how to compute an over-approximation of *all* their $\mathcal{R}$-equivalent forms.

The representation we use for sets of terms exploits non-deterministic finite *tree automata* (NTFA) [28]. In fact, an NTFA $\mathcal{A}$ has an associated language of terms, which are the terms reachable trough the derivation relation $\rightarrow_{\mathcal{A}}$ defined by $\mathcal{A}$. Note that the language can be an infinite set of terms, while $\mathcal{A}$ finitely represents it. Also, NTFA are a convenient representation, since they allow a number of common operations on languages, e.g. union, to be easily performed. Because of this, when a term set $\mathcal{T}$ is not the language of a NTFA, one might still be able to define an automaton such that its language is a *superset* of $\mathcal{T}$, as an over-approximation of it.

Our goal is over-approximating the language of $\mathcal{A}$ up to rewritings in $\mathcal{R}$. In other words, we approximate the relation defined by $\rightarrow^*_{\mathcal{A}}\rightarrow^*_{\mathcal{R}}$, where $\rightarrow_{\mathcal{R}}$ denotes the term rewriting relation defined by $\mathcal{R}$. For this, it is convenient to approximate instead the larger relation $(\rightarrow^*_{\mathcal{A}}\rightarrow^*_{\mathcal{R}})^*$ which allows for interleaving derivations with rewritings. To do that, we look for a new tree automaton $\mathcal{A}'$ such that

$$\rightarrow^*_{\mathcal{A}'} = (\rightarrow^*_{\mathcal{A}'}\rightarrow^*_{\mathcal{R}})^* \supseteq (\rightarrow^*_{\mathcal{A}}\rightarrow^*_{\mathcal{R}})^* \supseteq \rightarrow^*_{\mathcal{A}}\rightarrow^*_{\mathcal{R}}$$

We call such an $\mathcal{A}'$ *fully-exposing* because its language is *already closed* under rewritings in $\mathcal{R}$, so each term in the language is exposed in $\mathcal{A}'$ under all its possible rewritings. We shall define a (polynomial-time) *completion* algorithm to compute such an $\mathcal{A}'$ given $\mathcal{A}$ and $\mathcal{R}$. Further, we also implemented a tool for computing such an approximation.

The idea of approximating reachability in rewriting systems appeared in a work by Jacquemard [44]. The completion algorithm was discussed in several works by Genet et al. [38, 37, 33, 66]. These works anticipated ours in [72], where we essentially re-developed the ideas behind the completion algorithm and NTFA, without being aware such techniques were already known. Unfortunately, a part of our research efforts resulted in duplicating previous work. Our use of a different terminology surely contributed in preventing us to find these works.

Here, while we do adopt the standard terminology, we follow the presentation of [72], extending its results in order to simplify the static analysis of protocols of Chapter 4. Also, we discuss some implementation details, such as the actual heuristics we adopted, that improved both the performance

and the precision of the results of our tool. We also provide examples, and report on the successes and failures. We anticipate that in Chapter 4 we shall establish forward secrecy for a Diffie-Hellman–based protocol that involves exponentials, which are difficult to approximate since they are subject to many rewritings.

Another tool that exploits the completion algorithm is Timbuk [67]. Notably, when some forms of rewriting systems are used, this tool is able to compute *exact* representations for languages up to rewriting. Our tool lacks this feature. Another difference is that Timbuk, at least in the current version 2.2, sometimes relies on user interaction, while our tool always uses heuristics in order to compute the approximation in a completely automatic way. Further, our tool ensures termination for any rewriting system, while this is not the case for Timbuk. We however believe that techniques similar to ours could be used within Timbuk, so to merge the features of the two tools. Indeed, always terminating variants of Timbuk started to appear [19].

Summary Background and notation are in Section 3.2. Section 3.3 establishes the semantic properties we exploit throughout this chapter. An algorithm for left-linear rewritings is defined and discussed in Section 3.4, while in Section 3.5 we relax the left-linearity assumption, adapting our algorithm to any rewriting system. Finally, Section 3.9 reports on our experiments with our tool.

3.2 Preliminaries

We shall use the following denumerable sets: the set $\mathcal{Q}$ of *states* ($@q$, $@a$, $@b$, ...), the set $\mathcal{X}$ of *variables* (X, Y,...), the set $\mathcal{F}$ of *function symbols* (f, g, $+$, 1, ...).

Definition 12 *The set of* terms $\mathcal{T}$ *is inductively defined by*

$$\begin{array}{lll} T & ::= & X & \textit{variable} \\ & & @q & \textit{state} \\ & & \mathsf{f}(\cdots, T, \cdots) & \textit{application} \end{array}$$

By definition, we have $\mathcal{Q} \cup \mathcal{X} \subseteq \mathcal{T}$. Moreover, we assume that each function symbol f has a fixed associated arity. When the arity is zero (as for the function symbol 1), we simply write f instead of $\mathsf{f}()$.

The *size* of a term is the number of the applications occurring in it. The *depth* of a term is the maximum number of *nested* applications. For instance, g, $\mathsf{f}(X)$, and $\mathsf{f}(@q)$ have depth and size 1, while $\mathsf{f}(\mathsf{g}, \mathsf{h})$ has depth 2 and size 3, and X and $@q$ have depth and size zero. The variables occurring in a term T are denoted by $\mathsf{vars}(T)$.

Also, we adopt the following conventions:

- a *ground* term ($\in \mathcal{T}_{\mathsf{gr}}$) is a term with no occurring variables;
- a *simple* term ($\in \mathcal{T}_{\mathsf{sm}}$) is a term with no occurring states;
- a *pure* term ($\in \mathcal{T}_{\mathsf{pr}}$) is a term which is both ground and simple;
- a *plain* term ($\in \mathcal{T}_{\mathsf{pl}}$) is a ground term of depth at most one.

Similarly, we shall use the sets $\mathcal{C}_{\mathsf{gr}}, \mathcal{C}_{\mathsf{sm}}, \mathcal{C}_{\mathsf{pr}}$ to denote the sets of ground, simple, and pure *contexts* $C[\bullet]$, respectively. That is, $C[\bullet] \in \mathcal{C}_{\mathsf{type}}$ iff $C[T] \in \mathcal{T}_{\mathsf{type}}$, where T is an arbitrary pure term, and type ranges over $\mathsf{gr}, \mathsf{sm}, \mathsf{pr}$. $\mathcal{C}$ is the set of all contexts. Sometimes, we also use contexts with many placeholders $C[\bullet, \ldots, \bullet]$.

A *substitution* is a function $\sigma : \mathcal{X} \to \mathcal{T}_{\mathsf{gr}}$. We adopt the usual notation σx for the application $\sigma(x)$. When $\sigma x = T$, we say that σ replaces x with T. Substitutions are homomorphically extended to terms. So, we write σT as the application of the extension of σ to term T. We shall refer to substitutions resulting from pattern matching as *bindings*.

Below, we give the definitions for rewriting systems and automata.

Definition 13 *A* term rewriting system $\mathcal{R}$ *is a finite set of rewriting rules* $\mathcal{R}$ *having the form* $L \Rightarrow R$*, where* $L, R \in T_{\mathsf{sm}}$ *and* $\mathsf{vars}(R) \subseteq \mathsf{vars}(L)$*.*

Example The following rules model the usual rules for pairs.

$$\mathcal{R} = \{\mathsf{fst}(\mathsf{cons}(X, Y)) \Rightarrow X, \quad \mathsf{snd}(\mathsf{cons}(X, Y)) \Rightarrow Y\}$$

Definition 14 *Given a term rewriting system* $\mathcal{R}$*, we define the relation* $\to_{\mathcal{R}} \subseteq \mathcal{T}_{\mathsf{gr}} \times \mathcal{T}_{\mathsf{gr}}$ *to be the minimum relation closed under ground contexts such that for each* $(L \Rightarrow R) \in \mathcal{R}$*, and for each substitution* $\sigma : \mathcal{X} \to \mathcal{T}_{\mathsf{gr}}$*, we have* $\sigma L \to_{\mathcal{R}} \sigma R$*.*

Example Using the above $\mathcal{R}$, we have, for any $T_i \in \mathcal{T}_{\mathsf{gr}}$

$$\mathsf{fst}(\mathsf{fst}(\mathsf{cons}(\mathsf{cons}(T_1, T_2), T_3))) \to_{\mathcal{R}} \mathsf{fst}(\mathsf{cons}(T_1, T_2)) \to_{\mathcal{R}} T_1$$

Definition 15 *A* non deterministic finite tree automaton *(NTFA)* $\mathcal{A}$ *is a pair* $(\mathcal{Q}_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}})$*, where* $\mathcal{Q}_{\mathcal{A}} = \{@a, @b, \ldots\}$ *is a finite set of* states *and* $\mathcal{T}_{\mathcal{A}}$ *is the finite set of* transitions*. Transitions have the form* $@q \to T$*, where* $T \in \mathcal{T}_{\mathsf{pl}}$*. To simplify notation, we identify* $\mathcal{T}_{\mathcal{A}}$ *with* $\mathcal{A}$*, and simply write* $(@q \to T) \in \mathcal{A}$ *instead of* $(@q \to T) \in \mathcal{T}_{\mathcal{A}}$*. We sometimes write several transitions in the compact form* $@q \to T_1, \ldots, T_n$*, meaning* $@q \to T_1, \ldots, @q \to T_n$*.*

All automata we shall use here are NTFA, so we will refer to them as "automata", simply. Each state of the automaton has an associated

language, which intuitively is the set of terms reachable through transitions. For example, consider the following automaton:

$$@a \quad \rightarrow \quad 0, \mathsf{s}(@a) \qquad @b \quad \rightarrow \quad \mathsf{cons}(@a, @b)$$

The transitions for $@a$ define the language of naturals, with the usual Peano encoding. The language of $@b$ is instead the language of the infinite lists (streams) of such naturals, encoded through cons. We now formalize this fact.

Definition 16 *Given $\mathcal{A}$ we define the relation $\rightarrow_{\mathcal{A}} \subseteq \mathcal{T}_{\mathsf{gr}} \times \mathcal{T}_{\mathsf{gr}}$ as the minimum relation closed under ground contexts such that, for each $@q \rightarrow T \in \mathcal{A}$, we have $@q \rightarrow_{\mathcal{A}} T$.*

Definition 17 *Let $\rightarrow_{\mathcal{R},\mathcal{A}}$ be $\rightarrow_{\mathcal{R}} \cup \rightarrow_{\mathcal{A}}$. We write $[\![T]\!]_{\mathcal{A}}$ (resp. $[\![T]\!]_{\mathcal{R}}$ or $[\![T]\!]_{\mathcal{R},\mathcal{A}}$) for the set of* pure *terms reachable through $\rightarrow_{\mathcal{A}}$ (resp. $\rightarrow_{\mathcal{R}}$ or $\rightarrow_{\mathcal{R},\mathcal{A}}$) from the term $T \in \mathcal{T}_{\mathsf{gr}}$:*

$$\begin{aligned}
[\![T]\!]_{\mathcal{A}} &= \{T' \mid T \rightarrow^*_{\mathcal{A}} T' \in \mathcal{T}_{\mathsf{pr}}\} \\
[\![T]\!]_{\mathcal{R}} &= \{T' \mid T \rightarrow^*_{\mathcal{R}} T' \in \mathcal{T}_{\mathsf{pr}}\} \\
[\![T]\!]_{\mathcal{R},\mathcal{A}} &= \{T' \mid T \rightarrow^*_{\mathcal{R},\mathcal{A}} T' \in \mathcal{T}_{\mathsf{pr}}\}
\end{aligned}$$

Further, we let

$$[\![T]\!]_{\mathcal{A}}/\mathcal{R} = \bigcup \{[\![T']\!]_{\mathcal{R}} \mid T' \in [\![T]\!]_{\mathcal{A}}\}$$

Of course, $[\![@q]\!]_{\mathcal{A}}$ is the language related to the state $@q$, while $[\![@q]\!]_{\mathcal{A}}/\mathcal{R}$ is the same language up to rewriting. Instead, $[\![@q]\!]_{\mathcal{R},\mathcal{A}}$ is the set of terms reachable through automaton transitions and rewriting, allowing arbitrary interleaving between them. Obviously, the inclusion $[\![T]\!]_{\mathcal{A}} \subseteq [\![T]\!]_{\mathcal{R},\mathcal{A}}$ holds. We also have the inclusion $[\![T]\!]_{\mathcal{A}}/\mathcal{R} \subseteq [\![T]\!]_{\mathcal{R},\mathcal{A}}$. This last inclusion, in general, is strict: in fact it might be interesting to note that it is possible for relations $\rightarrow^*_{\mathcal{A}}$ and $\rightarrow^*_{\mathcal{R}}$ not to commute, even if the target term is restricted to be in $\mathcal{T}_{\mathsf{pr}}$. Formally:

$$\begin{aligned}
&\exists \mathcal{R}, \mathcal{A} \; . \; \rightarrow^*_{\mathcal{R}} \rightarrow^*_{\mathcal{A}} \cap (\mathcal{T}_{\mathsf{gr}} \times \mathcal{T}_{\mathsf{pr}}) \not\subseteq \rightarrow^*_{\mathcal{A}} \rightarrow^*_{\mathcal{R}} \\
&\exists \mathcal{R}, \mathcal{A} \; . \; \rightarrow^*_{\mathcal{A}} \rightarrow^*_{\mathcal{R}} \cap (\mathcal{T}_{\mathsf{gr}} \times \mathcal{T}_{\mathsf{pr}}) \not\subseteq \rightarrow^*_{\mathcal{R}} \rightarrow^*_{\mathcal{A}}
\end{aligned}$$

A counterexample to the first statement is

$$\begin{aligned}
&\mathcal{R} = \{\mathsf{f}(X) \Rightarrow \mathsf{g}(X, X)\} \qquad \mathcal{A} = \{@q \rightarrow \mathsf{a}, \mathsf{b}\} \\
&T_b = \mathsf{f}(@q) \rightarrow_{\mathcal{R}} \mathsf{g}(@q, @q) \rightarrow_{\mathcal{A}} \mathsf{g}(\mathsf{a}, @q) \rightarrow_{\mathcal{A}} \mathsf{g}(\mathsf{a}, \mathsf{b}) = T_e
\end{aligned}$$

Indeed, we do not have $T_b \rightarrow^*_{\mathcal{A}} \rightarrow^*_{\mathcal{R}} T_e$. For the second statement, take

$$\begin{aligned}
&\mathcal{R} = \{\mathsf{f}(X) \Rightarrow \mathsf{a}\} \qquad \mathcal{A} = \{@q \rightarrow \mathsf{f}(@m) \;\; ; \;\; @m \rightarrow \mathsf{b}\} \\
&T_b = @q \rightarrow_{\mathcal{A}} \mathsf{f}(@m) \rightarrow_{\mathcal{R}} \mathsf{a} = T_e
\end{aligned}$$

and note that we do not have $T_b \to_{\mathcal{R}}^{*} \to_{\mathcal{A}}^{*} T_e$. Therefore, the language up to rewriting $[\![@a]\!]_{\mathcal{A}}/\mathcal{R}$ may be smaller than the set $[\![@a]\!]_{\mathcal{R},\mathcal{A}}$ we focus on. We will return to this point in Sect. 3.6.

Now, we characterize those automata the languages of which include also the terms up to rewriting. We name these automata "fully-exposing", since they make explicit every possible rewriting. The goal of our approximation technique will be to compute such an automaton.

Definition 18 *An automaton $\mathcal{A}$ is said to be* fully-exposing *(w.r.t $\mathcal{R}$) iff for every $@q \in \mathcal{Q}_{\mathcal{A}}$, we have $[\![@q]\!]_{\mathcal{A}} = [\![@q]\!]_{\mathcal{R},\mathcal{A}}$.*

The reader might have expected the above equation to read $[\![@q]\!]_{\mathcal{A}} = [\![@q]\!]_{\mathcal{A}}/\mathcal{R}$, which would precisely state that the languages are closed under rewriting. The definition of "fully-exposing" we give uses instead the set $[\![@q]\!]_{\mathcal{R},\mathcal{A}}$, which is in general larger than $[\![@q]\!]_{\mathcal{A}}/\mathcal{R}$. So, the above requirement $[\![@q]\!]_{\mathcal{A}} = [\![@q]\!]_{\mathcal{R},\mathcal{A}}$ is actually stronger than the one $[\![@q]\!]_{\mathcal{A}} = [\![@q]\!]_{\mathcal{A}}/\mathcal{R}$. Our choice for this definition of "fully-exposing" is mainly driven by the fact that the first approximations we shall construct will satisfy this stronger requirement, and therefore this simplifies our presentation. However, in Sect. 3.6 we will return to this point, and consider alternative definitions, closer to the one $[\![@q]\!]_{\mathcal{A}} = [\![@q]\!]_{\mathcal{A}}/\mathcal{R}$. There, we shall also define approximations agreeing with these new definitions of "fully-exposing".

Having a fully-exposing automaton $\mathcal{A}$ makes it easy to test whether a term belongs to $[\![@q]\!]_{\mathcal{R},\mathcal{A}}$, since we can look into $[\![@q]\!]_{\mathcal{A}}$ instead. This can be done through a bounded exhaustive search, as we shall see in Sect. 3.2.1.

Normalization In Definition 15, for each transition $@q \to T$ we require $T \in \mathcal{T}_{\mathsf{pl}}$. One might instead allow for a broader class of automata using the weaker requirement $T \in \mathcal{T}_{\mathsf{gr}}$. It turns out that the expressive power of these two kinds of automata is the same. In fact every automaton having transitions where $T \in \mathcal{T}_{\mathsf{gr}}$ can be *normalized* into an equivalent automaton that only uses plain terms, as the following example shows.

Example The automaton

$$@a \;\to\; \mathsf{f}(@a), 1 \qquad @q \;\to\; \mathsf{cons}(\mathsf{cons}(@a, @a), @a)$$

has the following normal form, where $@q_1$ is a fresh state:

$$@a \;\to\; \mathsf{f}(@a), 1 \qquad @q \;\to\; \mathsf{cons}(@q_1, @a) \qquad @q_1 \;\to\; \mathsf{cons}(@a, @a)$$

In the above normal form, only plain terms are used. We shall make use of this translation later on.

Notation The notation introduced here slightly deviates from more standard ones, e.g. the one in [28]. For instance, we write transitions backwards with respect to the more standard notation $@q \leftarrow T$, since we think this makes the overall notation of $\rightarrow, \rightarrow_{\mathcal{A}}$, and $\rightarrow_{\mathcal{R}}$ more consistent.

Also, it is also common to include in the definition of automaton a set of *final states* $\mathcal{Q}_{\mathsf{F}}$, and consider the automaton as a way to define the language

$$\{T \mid \exists @q \in \mathcal{Q}_{\mathsf{F}}.\ @q \rightarrow^*_{\mathcal{A}} T \in \mathcal{T}_{\mathsf{pr}}\}$$

Instead, we simply use automata for defining a *class* of languages, one for each state. So, we consider the languages $[\![@q]\!]_{\mathcal{A}}$ for any $@q \in \mathcal{Q}_{\mathcal{A}}$.

3.2.1 Matching

Beneficial to studying $\rightarrow_{\mathcal{R},\mathcal{A}}$ is understanding which terms T, with $@q \rightarrow^*_{\mathcal{A}} T$, can be rewritten by a rule $L \Rightarrow R$, i.e. which such terms T *match* L. To this aim, we define the result of a *match* of the state $@q$ with the pattern L as the set of the bindings $\sigma : \mathcal{X} \rightarrow \mathcal{T}_{\mathsf{gr}}$ for which $@q \rightarrow^*_{\mathcal{A}} \sigma L$. Note that the number of such bindings may be infinite, in general.

A very interesting subset of the bindings resulting from a match is the one formed by *state* bindings, i.e. those $\sigma : \mathcal{X} \rightarrow \mathcal{Q}$. Indeed, there are only a *finite* number of state bindings, since the variables occurring in L are finite, and so are the states of $\mathcal{A}$. The set of the state bindings that match against L can be computed in a natural way under the following assumption.

Left Linearity We assume each rule $L \Rightarrow R \in \mathcal{R}$ be *left–linear*, i.e. each variable occurs at most once in L.

Intuitively, the above assumption means that each variable in L can be matched independently. To compute state bindings, just expand $@q$ by applying all the transitions in a non deterministic fashion, and discarding the mismatching branches. Below, $@q$ is matched against the pattern $\mathsf{f}(\mathsf{g}(X), \mathsf{h}(Y))$.

$$@q \rightarrow_{\mathcal{A}} \mathsf{f}(@a, @b) \rightarrow_{\mathcal{A}} \mathsf{f}(\mathsf{g}(@c), @b) \rightarrow_{\mathcal{A}} \mathsf{f}(\mathsf{g}(@c), \mathsf{h}(@d)) = \sigma \mathsf{f}(\mathsf{g}(X), \mathsf{h}(Y))$$

Note that we need to search roughly as deep as the size of the pattern L. That is, not counting the transitions of the form $@a \rightarrow @b$ (thus counting only those that generate a new term head), we can limit the exploration of the $\rightarrow_{\mathcal{A}}$ tree to depth equal to the size of L. For the most common rewriting rule sets this depth is 2 or 3. Therefore, even though we adopt an exponential-time exhaustive search for this, the impact on the performance of our tool will still be acceptable. The number of state bindings is at most $|\mathcal{Q}_{\mathcal{A}}|^k$ where k is the number of variables in L. Note that this is polynomial in the number of states.

Example Given the automaton $\mathcal{A}$

$$@a \to 1, 2, 3 \qquad @b \to +(@a, @a)$$
$$@q \to \mathsf{cons}(@a, @q), \mathsf{fst}(@m) \qquad @m \to \mathsf{cons}(@q, @b), @q$$

the match of $@q$ with the pattern $L = \mathsf{fst}(\mathsf{cons}(X, Y))$ produces the state bindings $\sigma_1 = \{X \mapsto @q, Y \mapsto @b\}, \sigma_2 = \{X \mapsto @a, Y \mapsto @q\}$.

The importance of state bindings lies in that they are a good approximation for the set of *all* the bindings $\sigma : \mathcal{X} \to \mathcal{T}_{\mathsf{gr}}$, as we shall show in the next section.

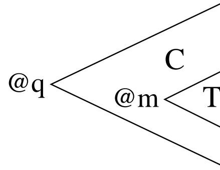
Moreover, the same exhaustive search used for matching can also be used for testing whether some given pure term T belongs to $[\![@q]\!]_{\mathcal{A}}$. In fact, T can be seen as a pattern with no variables. Again, the cost of the test is exponential on the size of T, but polynomial on the number of states.

Finally, we note that left linearity makes matching rather convenient, but it prevents us from using many well-known rewriting rules, such as $\mathsf{xor}(X, X) \Rightarrow 0$. We will return to this hypothesis, and relax it, in Sect. 3.5.

3.3 Semantics

A nice direct consequence of having $\mathcal{A}$ in normal form is that each *sub*-term occurring in $[\![@q]\!]_{\mathcal{A}}$ also belongs to the language $[\![@m]\!]_{\mathcal{A}}$ for some $@m \in \mathcal{Q}_{\mathcal{A}}$, as established by the lemma below.

Lemma 4 (State-in-the-middle) *If $@q \to^*_{\mathcal{A}} C[T]$ for some $@q \in \mathcal{Q}_{\mathcal{A}}$, $C[\bullet] \in \mathcal{C}_{\mathsf{gr}}$ and $T \in \mathcal{T}_{\mathsf{gr}}$, then for some state $@m \in \mathcal{Q}_{\mathcal{A}}$ we have $@q \to^*_{\mathcal{A}} C[@m]$ and $@m \to^*_{\mathcal{A}} T$.*



Proof. By induction on the number of $\to_{\mathcal{A}}$ steps.

For the base case, we have zero steps and $C[T] = @q$, so $T = @q$ and the lemma holds for $@m = @q$.

For the inductive case, assume that the last step was due to $@s \to_{\mathcal{A}} T_s$ within a context C', and therefore $@q \to^*_{\mathcal{A}} C'[@s] \to_{\mathcal{A}} C'[T_s] = C[T]$. If $C' = C$, we have $T_s = T$, and $@m = @s$ satisfies the lemma. Otherwise, we have $C'[\bullet] = C''[\bullet, T]$ and $@q \to^*_{\mathcal{A}} C''[@s, T] \to_{\mathcal{A}} C''[T_s, T]$. By inductive hypothesis applied to the context $C''[@s, \bullet]$, we have $@q \to^*_{\mathcal{A}} C''[@s, @m] \to^*_{\mathcal{A}} C''[@s, T] \to_{\mathcal{A}} C''[T_2, T]$, completing the proof. □

Intuitively, the "state-in-the-middle" lemma implies that $\mathcal{A}$ provides us with a convenient abstraction (namely, the state $@m$) for many terms that

are affected by rewriting. Indeed, whenever a rewriting $\sigma L \to_{\mathcal{R}} \sigma R$ occurs and $\sigma L \in [\![@q]\!]_{\mathcal{A}}$, we know that there is some state $@m_i$ in $\mathcal{A}$ that generates σX_i, for each variable X_i of L. This suggests us a way for proving some properties involving the whole, potentially infinite, set of bindings $\sigma : \mathcal{X} \to \mathcal{T}_{\mathsf{gr}}$. Proving a property Φ for all σ might reduce to checking Φ on the set $\mathcal{B}$ of state bindings $\sigma : \mathcal{X} \to \mathcal{Q}$. This turns out to be convenient when only variables in $\mathcal{R}$ and states in $\mathcal{A}$ are used, because $\mathcal{B}$ is finite.

Quite interestingly, this is the case when proving that an automaton is fully-exposing. The next lemma gives a convenient criterion for this, thus paving the way towards an algorithmic treatment of automata and rewritings.

Lemma 5 (State Bindings Criterion) *If $\mathcal{A}$ satisfies*

$$\forall (L \Rightarrow R) \in \mathcal{R}.\ \forall \sigma : \mathcal{X} \to \mathcal{Q}\ .\ \forall @q \in \mathcal{Q}_{\mathcal{A}}.\ @q \to^*_{\mathcal{A}} \sigma L \implies @q \to^*_{\mathcal{A}} \sigma R \tag{3.1}$$

then $\mathcal{A}$ is fully-exposing.

Proof. First, property (3.1) suffices to ensure the following apparently stronger property, which involves *any* binding σ:

$$\forall (L \Rightarrow R) \in \mathcal{R}.\ \forall \sigma : \mathcal{X} \to \mathcal{T}_{\mathsf{gr}}\ .\ \forall @q \in \mathcal{Q}_{\mathcal{A}}.\ @q \to^*_{\mathcal{A}} \sigma L \implies @q \to^*_{\mathcal{A}} \sigma R \tag{3.2}$$

This is because, whenever $\sigma X \notin \mathcal{Q}$, we can apply the "state-in-the-middle" lemma, and find a state which is responsible for σX, i.e. a state $@m \in \mathcal{Q}_{\mathcal{A}}$ such that $@q \to^*_{\mathcal{A}} C[@m] \to^*_{\mathcal{A}} C[\sigma X] = \sigma L$. Exploiting the left–linearity of $\mathcal{R}$, we can then update the binding to $\sigma' = \sigma[X \mapsto @m]$ and still have $@q \to^*_{\mathcal{A}} \sigma' L$. We repeat this process for any variable $X_1, \ldots, X_n$ occurring in L. In this way we obtain $\sigma'' \in \mathcal{X} \to \mathcal{Q}$ such that $@q \to^*_{\mathcal{A}} C[@m_1, \ldots, @m_n] = \sigma'' L$ and, for each i, $@m_i \to^*_{\mathcal{A}} \sigma X_i$. By hypothesis (3.1), this implies

$$@q \to^*_{\mathcal{A}} \sigma'' R = C'[\ldots, @m_{i_k}, \ldots] \to^*_{\mathcal{A}} C'[\ldots, \sigma X_{i_k}, \ldots] = \sigma R.$$

and therefore $@q \to^*_{\mathcal{A}} \sigma R$, completing the proof for (3.2).

We now proceed and show $[\![@q]\!]_{\mathcal{R},\mathcal{A}} = [\![@q]\!]_{\mathcal{A}}$, for each $@q \in \mathcal{Q}_{\mathcal{A}}$. Since $[\![T]\!]_{\mathcal{A}} \subseteq [\![T]\!]_{\mathcal{R},\mathcal{A}}$ for any $T \in \mathcal{T}_{\mathsf{gr}}$, we only need to show $[\![@q]\!]_{\mathcal{R},\mathcal{A}} \subseteq [\![@q]\!]_{\mathcal{A}}$.

This proof is by contradiction: assume that some T exists such that $@q \to^*_{\mathcal{A}} \to_{\mathcal{R}} T$ but for which $@q \not\to^*_{\mathcal{A}} T$. This means that, for some rule $(L \Rightarrow R) \in \mathcal{R}$, we have

$$@q \to^*_{\mathcal{A}} C[\sigma L] \to_{\mathcal{R}} C[\sigma R] = T$$

By the "state-in-the-middle" lemma, for some state $@m \in \mathcal{Q}_\mathcal{A}$, $@q \to^*_\mathcal{A} C[@m]$ and $@m \to^*_\mathcal{A} \sigma L$, which by (3.2) implies $@m \to^*_\mathcal{A} \sigma R$. This shows that

$$@q \to^*_\mathcal{A} C[@m] \to^*_\mathcal{A} C[\sigma R] = T$$

which contradicts $@q \not\to^*_\mathcal{A} T$. □

3.3.1 Automaton Transformations

We now define a way to modify $\mathcal{A}$ preserving its semantics, in the sense made precise by the following preorder.

Definition 19 *Given $\mathcal{R}$ and $\mathcal{Q}' \subseteq \mathcal{Q}$, we define a preorder $\sqsubseteq^{\mathcal{Q}'}$ over the set of automata as follows:*

$$\mathcal{A} \sqsubseteq^{\mathcal{Q}'} \mathcal{A}' \iff \forall @q \in \mathcal{Q}' \;.\; [\![@q]\!]_\mathcal{A} \subseteq [\![@q]\!]_{\mathcal{A}'}$$

We also write $\mathcal{A} \equiv^{\mathcal{Q}'} \mathcal{A}'$ when $\mathcal{A} \sqsubseteq^{\mathcal{Q}'} \mathcal{A}' \wedge \mathcal{A}' \sqsubseteq^{\mathcal{Q}'} \mathcal{A}$.

Intuitively, the states in $\mathcal{Q}'$ represent the languages we are interested in. The other states are auxiliary, and, while they contribute to the overall definition of the automaton, they have no particular meaning. The above preorder ensures that $\mathcal{A}'$ over-approximates $\mathcal{A}$ for the "important" languages of $\mathcal{Q}'$, disregarding the languages of other states.

To modify $\mathcal{A}$ into an automaton $\mathcal{A}'$ such that $\mathcal{A} \equiv^{\mathcal{Q}'} \mathcal{A}'$, each state in $\mathcal{Q}_\mathcal{A} \cap \mathcal{Q}'$ needs to be in $\mathcal{A}'$, as well. Also, we may rename or remove states in $\mathcal{Q}_\mathcal{A} \setminus \mathcal{Q}'$, as long as, in $\mathcal{A}'$, the languages of the states in $\mathcal{Q}_\mathcal{A} \cap \mathcal{Q}'$ are larger than (or equal to) the corresponding ones in $\mathcal{A}$. Below, we define three transformations that preserve our preorder.

Augment $$\frac{}{\mathcal{A} \sqsubseteq^{\mathcal{Q}'} \mathcal{A} \cup (@q \to T)}$$

Transitivity $$\frac{(@q_1 \to @q_2) \in \mathcal{A} \quad (@q_2 \to T) \in \mathcal{A}}{\mathcal{A} \equiv^{\mathcal{Q}'} \mathcal{A} \cup (@q_1 \to T)}$$

Join $$\frac{\forall T \;.\; (@q_1 \to T) \in \mathcal{A} \iff (@q_2 \to T) \in \mathcal{A}}{\mathcal{A} \equiv^{\mathcal{Q}'} \mathcal{A}\{@q_2/@q_1\}} \quad @q_1 \notin \mathcal{Q}'$$

Rule Augment simply adds a transition to $\mathcal{A}$, clearly preserving our preorder. Rule Transitivity short-cuts two transitions, yielding a completely equivalent automaton. Note that both rules Augment and Transitivity increase the size of $\mathcal{A}$.

Rule Join is the most peculiar. It replaces one state $@q_1$ with another state $@q_2$, under the assumption they have the same transitions (and therefore define the same language). This rule may decrease the size of $\mathcal{A}$, both

because all the transitions having the form $@q_1 \to T$ are removed, and because pairs of transitions such as $@m \to \mathsf{f}(@q_1), \mathsf{f}(@q_2)$ collapse into the single transition $@m \to \mathsf{f}(@q_2)$. The languages generated by the states are unaffected, except for $[\![@q_1]\!]_{\mathcal{R},\mathcal{A}}$ that is not of interest, as $@q_1 \notin \mathcal{Q}'$.

The following lemma states that if we approximate (w.r.t. $\sqsubseteq^{\mathcal{Q}'}$) an automaton $\mathcal{A}$ with a fully-exposing automaton $\mathcal{A}'$, then $\mathcal{A}'$ actually encompasses the languages of $\mathcal{A}$ up to rewriting.

Lemma 6 *If $\mathcal{A} \sqsubseteq^{\mathcal{Q}'} \mathcal{A}'$ and $\mathcal{A}'$ is fully-exposing, then*

$$\forall @q \in \mathcal{Q}'. \;\; [\![@q]\!]_{\mathcal{A}}/\mathcal{R} \subseteq [\![@q]\!]_{\mathcal{A}'}$$

Proof. Assume $T \in [\![@q]\!]_{\mathcal{A}}/\mathcal{R}$. We have $@q \to^*_{\mathcal{A}} T_p \to^*_{\mathcal{R}} T$ for a pure T_p. Since $@q \in \mathcal{Q}'$, we also have $@q \to^*_{\mathcal{A}'} T_p \to^*_{\mathcal{R}} T$. Therefore, since $\mathcal{A}'$ is fully-exposing, $T \in [\![@q]\!]_{\mathcal{R},\mathcal{A}'} = [\![@q]\!]_{\mathcal{A}'}$. $\square$

3.4 Algorithm

In Fig. 3.1 we give an algorithm that approximates a pair $\mathcal{A}, \mathcal{R}$ with a fully-exposing automaton $\mathcal{A}'$. In other words, it computes a $\mathcal{A}'$ such that $\mathcal{A} \sqsubseteq^{\mathcal{Q}_{\mathcal{A}}} \mathcal{A}'$.

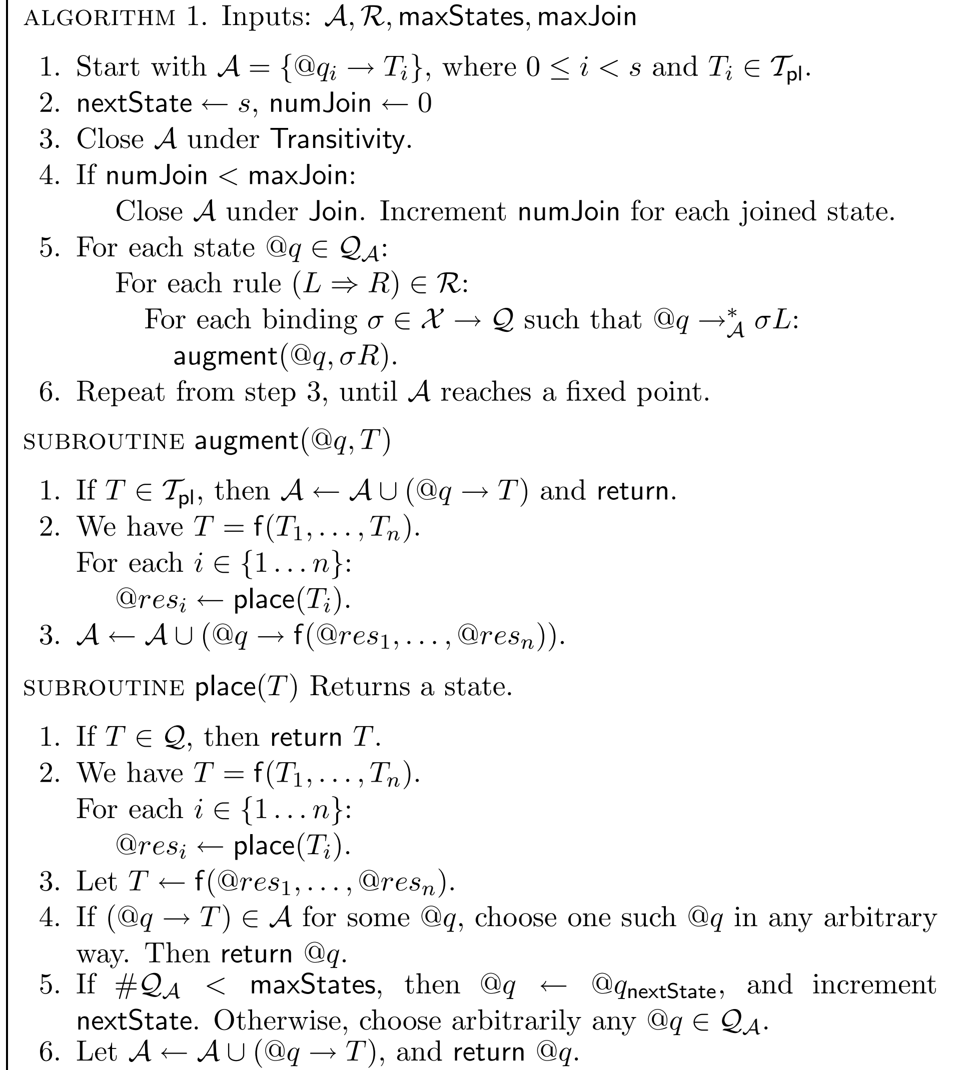
At the beginning, we choose an arbitrary bound maxStates on the number of distinct states that can be used in constructing $\mathcal{A}'$. During the execution of our algorithm this bound can be reached, and still a transition for a ground term T has to be added to $\mathcal{A}$. If the term T is not plain, a normalization is in order, that requires a fresh, unavailable state. Rule Augment comes to our rescue: we can safely reuse any of the states already employed. Of course, this causes a loss of precision in the language of the selected state, yet the result is sound w.r.t. $\sqsubseteq^{\mathcal{Q}_{\mathcal{A}}}$. This is done in step 5 of the subroutine place of the algorithm.

We stress that the result is sound for *any* maxStates bound. In fact, having just one state is enough to produce a sound approximation: the language consisting of *all* pure terms is generated by all the transitions $@o : \mathsf{f}(\ldots, @o, \ldots)$ which only involve one state. Of course, one wants a more precise result than this, and therefore runs the algorithm with more resources.

We also put a bound maxJoin on the number of Join operations we allow during the algorithm run. The bounds maxJoin and maxStates help in ensuring termination of the algorithm; its correctness is established below.

Theorem 3 *Algorithm 1 is correct, i.e. given $\mathcal{R}, \mathcal{A}$, it always computes a fully-exposing $\mathcal{A}'$ such that $\mathcal{A} \sqsubseteq^{\mathcal{Q}_{\mathcal{A}}} \mathcal{A}'$. Also, $\mathcal{A}'$ is closed by* Transitivity.

Proof. We start by showing the termination of the algorithm. First, we show that the size of $\mathcal{A}$ is bounded because:



ALGORITHM 1. Inputs: $\mathcal{A}, \mathcal{R}, \mathsf{maxStates}, \mathsf{maxJoin}$

1. Start with $\mathcal{A} = \{@q_i \to T_i\}$, where $0 \leq i < s$ and $T_i \in \mathcal{T}_{\mathsf{pl}}$.
2. $\mathsf{nextState} \leftarrow s$, $\mathsf{numJoin} \leftarrow 0$
3. Close $\mathcal{A}$ under Transitivity.
4. If $\mathsf{numJoin} < \mathsf{maxJoin}$:
   Close $\mathcal{A}$ under Join. Increment numJoin for each joined state.
5. For each state $@q \in \mathcal{Q}_\mathcal{A}$:
   For each rule $(L \Rightarrow R) \in \mathcal{R}$:
   For each binding $\sigma \in \mathcal{X} \to \mathcal{Q}$ such that $@q \to^*_\mathcal{A} \sigma L$:
   $\mathsf{augment}(@q, \sigma R)$.
6. Repeat from step 3, until $\mathcal{A}$ reaches a fixed point.

SUBROUTINE $\mathsf{augment}(@q, T)$

1. If $T \in \mathcal{T}_{\mathsf{pl}}$, then $\mathcal{A} \leftarrow \mathcal{A} \cup (@q \to T)$ and return.
2. We have $T = \mathsf{f}(T_1, \ldots, T_n)$.
   For each $i \in \{1 \ldots n\}$:
   $@res_i \leftarrow \mathsf{place}(T_i)$.
3. $\mathcal{A} \leftarrow \mathcal{A} \cup (@q \to \mathsf{f}(@res_1, \ldots, @res_n))$.

SUBROUTINE $\mathsf{place}(T)$ Returns a state.

1. If $T \in \mathcal{Q}$, then return T.
2. We have $T = \mathsf{f}(T_1, \ldots, T_n)$.
   For each $i \in \{1 \ldots n\}$:
   $@res_i \leftarrow \mathsf{place}(T_i)$.
3. Let $T \leftarrow \mathsf{f}(@res_1, \ldots, @res_n)$.
4. If $(@q \to T) \in \mathcal{A}$ for some $@q$, choose one such $@q$ in any arbitrary way. Then return $@q$.
5. If $\#\mathcal{Q}_\mathcal{A} < \mathsf{maxStates}$, then $@q \leftarrow @q_{\mathsf{nextState}}$, and increment nextState. Otherwise, choose arbitrarily any $@q \in \mathcal{Q}_\mathcal{A}$.
6. Let $\mathcal{A} \leftarrow \mathcal{A} \cup (@q \to T)$, and return $@q$.

Figure 3.1: The completion algorithm

- the transitions in $\mathcal{A}$ only carry *plain* terms, which have bounded depth;
- no new function symbols are introduced;
- the function symbols have *fixed* arity;
- the number of states in $\mathcal{A}$ is bounded by maxStates.

Termination is proved by contradiction, assuming our algorithm loops indefinitely in steps $3 \ldots 6$. Since we never exit the loop, and the number of Join operations is bounded by maxJoin, eventually only rules Augment and Transitivity have to be applied. These rules make the size of $\mathcal{A}$ to indefinitely grow, so contradicting the first fact proved above.

Having established termination, by Lemma 5 the resulting automaton $\mathcal{A}'$ is fully-exposing. Moreover, since the algorithm only applies the automaton transformations of Sect. 3.3.1, we have $\mathcal{A} \sqsubseteq^{\mathcal{Q}_\mathcal{A}} \mathcal{A}'$. □

3.5 Relaxing the Left Linearity Assumption

Unfortunately, many useful rewriting rules are not left–linear, e.g.

$$\mathcal{R} = \{\ \mathsf{xor}(X, X) \ \Rightarrow\ 0 \qquad \mathsf{dec}(\mathsf{enc}(M, K), K) \ \Rightarrow\ M\ \}$$

In order to apply our technique to the above rules, we could start by approximating the rule set $\mathcal{R}$ with another left–linear rule set $\mathcal{R}_l$ for which $\rightarrow_\mathcal{R} \subseteq \rightarrow_{\mathcal{R}_l}$. Such a $\mathcal{R}_l$ could be

$$\mathcal{R}_l = \{\ \mathsf{xor}(X_1, X_2) \ \Rightarrow\ 0 \quad \mathsf{dec}(\mathsf{enc}(M, K_1), K_2) \ \Rightarrow\ M\ \}$$

While this would lead to a correct approximation of $\rightarrow^*_{\mathcal{R},\mathcal{A}}$, the result would be very imprecise. In the example above $\mathcal{R}_l$ allows us to decrypt any message, even if garbage is used instead of the proper key. Such an approximation of $\mathcal{R}$ will unlikely lead to the proof of any interesting security property.

Instead, we extend rewriting rules to have a set of equalities (which we write as $\widetilde{X} = \widetilde{Y}$) between variables as a side condition. We thus rewrite $\mathcal{R}$ as

$$\mathcal{R}' = \left\{ \begin{array}{lll} \mathsf{xor}(X_1, X_2) & {}_{X_1 = X_2} \Rightarrow & 0 \\ \mathsf{dec}(\mathsf{enc}(M, K_1), K_2) & {}_{K_1 = K_2} \Rightarrow & M \end{array} \right\}$$

Now the $\rightarrow_{\mathcal{R}'}$ rewritings are only allowed when the side condition is met; for instance, in $\mathcal{R}'$, X_1 and X_2 must actually match the same ground terms, as well as K_1 and K_2. This precisely reflects the intended meaning of the rules in $\mathcal{R}$.

In the general case, if we have n occurrences ($n > 1$) of the same variable X in a left hand side of a rule, we replace each of them with X_i and add $n-1$ equations as a side condition, that is $X_1 = X_2, X_2 = X_3, \ldots X_{n-1} = X_n$. Having renamed X in the left hand side, we must also rename it in the right hand side. When rewriting happens, all the variables X_i are bound to the same term, so we can replace X in the right hand side with any of them. By convention, we choose X_1 for this.

We update the definition of $\rightarrow_\mathcal{R}$, so to take into account the side condition.

Definition 20 *Given an automaton $\mathcal{A}$ and a term rewriting system $\mathcal{R}$, we define the relation $\rightarrow_\mathcal{R} \subseteq \mathcal{T}_{\mathsf{gr}} \times \mathcal{T}_{\mathsf{gr}}$ to be the minimum relation closed under ground contexts such that for each $(L_{\widetilde{X}=\widetilde{Y}} \Rightarrow R) \in \mathcal{R}$, and for each substitution $\sigma : \mathcal{X} \rightarrow \mathcal{T}_{\mathsf{gr}}$, when we have $\sigma\widetilde{X} = \sigma\widetilde{Y}$ we also have $\sigma L \rightarrow_\mathcal{R} \sigma R$.*

Accordingly, we also update the definitions of $\to_{\mathcal{R},\mathcal{A}}$, $[\![-]\!]_{\mathcal{R},\mathcal{A}}$, and the notion of "fully-exposing" to use the rewriting above.

Now we can reuse our technique with minor changes, only: indeed, a revised version of Lemma 5 still holds. Here, we use the relation $\simeq$, defined below, to check whether two states may lead to the same ground term.

Lemma 7 (State Bindings Criterion) *Given $\mathcal{A}, \mathcal{R}$, let*

$$\simeq \ = \{\langle @q_1, @q_2\rangle \mid \exists T \in \mathcal{T}_{\mathsf{gr}}.\ @q_1 \to^*_{\mathcal{A}} T \wedge @q_2 \to^*_{\mathcal{A}} T\} \tag{3.3}$$

If $\mathcal{A}$ satisfies

$$\left.\begin{array}{r} \forall (L_{\widetilde{X}=\widetilde{Y}} \Rightarrow R) \in \mathcal{R} \\ \forall \sigma : \mathcal{X} \to \mathcal{Q} \\ \forall @q \in \mathcal{Q}_{\mathcal{A}} \end{array}\right\} @q \to^*_{\mathcal{A}} \sigma L \wedge \forall i.\ \sigma\widetilde{X}_i \simeq \sigma\widetilde{Y}_i \implies @q \to^*_{\mathcal{A}} \sigma R \tag{3.4}$$

then $\mathcal{A}$ is fully-exposing.

Proof. As for Lemma 5, we establish property (3.4) for *any* binding $\sigma : \mathcal{X} \to \mathcal{T}_{\mathsf{gr}}$ and replacing $\simeq$ with identity.

$$\left.\begin{array}{r} \forall (L_{\widetilde{X}=\widetilde{Y}} \Rightarrow R) \in \mathcal{R} \\ \forall \sigma : \mathcal{X} \to \mathcal{T}_{\mathsf{gr}} \\ \forall @q \in \mathcal{Q}_{\mathcal{A}} \end{array}\right\} @q \to^*_{\mathcal{A}} \sigma L \wedge \forall i.\ \sigma\widetilde{X}_i = \sigma\widetilde{Y}_i \implies @q \to^*_{\mathcal{A}} \sigma R \tag{3.5}$$

For this, assume $@q \to^*_{\mathcal{A}} \sigma L$ and $\forall i.\ \sigma\widetilde{X}_i = \sigma\widetilde{Y}_i$. By the "state-in-the-middle" lemma, we can state $@q \to^*_{\mathcal{A}} C[@m_1, \ldots, @m_n] = \sigma' L \to^*_{\mathcal{A}} \sigma L$ for some $@m_i$ and their related state binding σ'. Since $\sigma\widetilde{X}_i = \sigma\widetilde{Y}_i$, we have that the $@m_i$ corresponding to those variables have a common descendant, i.e. $@m_{x_i} \to^*_{\mathcal{A}} \sigma\widetilde{X}_i = T_i$ $@m_{y_i} \to^*_{\mathcal{A}} \sigma\widetilde{Y}_i = T_i$. This implies that $\forall i. @m_{x_i} \simeq @m_{y_i}$. Applying (3.4) we get $@q \to^*_{\mathcal{A}} \sigma' R \to^*_{\mathcal{A}} \sigma R$. This completes the proof for (3.5).

Finally we need to show that (3.5) implies that $\mathcal{A}$ is fully-exposing. This is done exactly as in the last part of the proof of Lemma 5, using (3.5) in lieu of (3.2). □

Note that properties (3.3) and (3.4) depend on both $\mathcal{A}$ and $\simeq$. Thus, we must adapt our algorithm to compute both an automata $\mathcal{A}$ and a relation $\simeq$ that simultaneously satisfy these two properties: this is similar to a double fixed point computation.

To give an intuition of the new algorithm, consider the above $\mathcal{R}'$, and in particular $L = \mathsf{xor}(X_1, X_2)$. During the execution of step 5 of the (old) algorithm, assume to have a $\simeq$ satisfying (3.3) for the current $\mathcal{A}$. We look for some state binding σ such that $@q \to^*_{\mathcal{A}} \sigma L$. Now, before calling the subroutine $\mathsf{augment}(@q, \sigma R = 0)$ that would add the rewritten term to $\mathcal{A}$, we check $\sigma X_1 \simeq \sigma X_2$, i.e. the side condition. If the check succeeds, then we

call augment($@q, 0$); otherwise, we do nothing. When $\mathcal{A}$ changes, we must ensure that the relation $\simeq$ still satisfies (3.3). This forces us to update the relation $\simeq$ at each main loop iteration.

Note that our side condition check is a safe approximation. Intuitively, we should check two terms T_1, T_2 for equality. However, we do not know the actual terms to compare, but merely their approximations: we know that the states σX_1 and σX_2 are such that $\sigma X_1 \to^*_{\mathcal{R},\mathcal{A}} T_1$ and $\sigma X_2 \to^*_{\mathcal{R},\mathcal{A}} T_2$. Conservatively, we check whether these approximations overlap, i.e. if the states might represent the same ground term: for this, we use $\simeq$.

The new algorithm is in Fig. 3.2. Note that when $\mathcal{A}$ reaches its fixed point in the main loop, so has $\simeq$ since it only depends on $\mathcal{A}$ as per step 5.

```
ALGORITHM 2. Inputs: A, R, maxStates, maxJoin
 1. Start with A = {@q_i → T_i}, where 0 ≤ i < s and T_i ∈ T_pl.
 2. nextState ← s, numJoin ← 0
 3. Close A under Transitivity.
 4. If numJoin < maxJoin:
        Close A under Join. Increment numJoin for each joined state.
 5. ≃ ← approxIntersection(A).
 6. For each state @q ∈ Q_A:
        For each rule (L_{X̃=Ỹ} ⇒ R) ∈ R:
           For each binding σ ∈ X → Q such that @q →*_A σL:
              If ∀i. σX̃_i ≃ σỸ_i:
                 augment(@q, σR).
 7. Repeat from step 3, until A reaches a fixed point.
SUBROUTINE approxIntersection(A) Returns ≃.
 1. ≃ ← {⟨@q, @q⟩ | @q ∈ Q_A}
 2. If (@a → f(@a_1, ..., @a_n)), (@b → f(@b_1, ..., @b_n)) ∈ A,
    and ∀i. @a_i ≃ @b_i:
        ≃ ← ≃ ∪ {⟨@a, @b⟩}
 3. If (@a → @b) ∈ A and @b ≃ @c:
        ≃ ← ≃ ∪ {⟨@a, @c⟩}
 4. Repeat from step 2 until ≃ reaches a fixed point.
 5. return ≃.
```

Figure 3.2: Revised completion algorithm

Lemma 8 *In algorithm 2, after each call to* approxIntersection*, $\simeq$ satisfies (3.3).*

Proof. The termination of approxIntersection is ensured by $\simeq \subseteq \mathcal{Q}_\mathcal{A} \times \mathcal{Q}_\mathcal{A}$, and the fact that the size of $\simeq$ increases in the subroutine loop.

Assuming that $@a \to^*_\mathcal{A} T$ in n steps, and that $@b \to^*_\mathcal{A} T$ in m steps, we proceed by induction on the pair (n, m), under pointwise ordering. The base

case, i.e. $n = m = 0$, is handled by initializing $\simeq$ to the identity relation in step 1 of the subroutine.

For the inductive case, assume that the ground term $T = \mathsf{f}(T_1, \dots, T_n)$, is reachable from $@a$ and $@b$. Without loss of generality, we can assume that $@a$ reaches T in more than zero steps (otherwise, swap $@a$ with $@b$). So, we have $@a \rightarrow_{\mathcal{A}} T_a \rightarrow^*_{\mathcal{A}} T$. If $T_a = @ta$, by inductive hypothesis $@ta \simeq @b$, and after step 3, we also have $@a \simeq @b$. Otherwise, we have $T_a = \mathsf{f}(@a_1, \dots, @a_n)$ and therefore $@b$ also reaches T in more than zero steps, i.e. $@b \rightarrow_{\mathcal{A}} T_b \rightarrow^*_{\mathcal{A}} T$. If $T_b = @tb$, we proceed as in the $@ta$ case. Otherwise, we have $T_b = \mathsf{f}(@b_1, \dots, @b_n)$ (note that the function symbol must be the same of T_a, since T is a common descendant). By inductive hypothesis, we have $@a_i \simeq @b_i$ for all i. When this is found in step 2, we update $\simeq$ so that $@a \simeq @b$. □

The correctness of Algorithm 2 is established below.

Theorem 4 *Given $\mathcal{A}, \mathcal{R}$, Algorithm 2 always outputs a fully-exposing $\mathcal{A}'$ such that $\mathcal{A} \sqsubseteq^{\mathcal{Q}_\mathcal{A}} \mathcal{A}'$. Also, $\mathcal{A}'$ is closed under* Transitivity.

Proof. Termination is established as for Theorem 3. Therefore, by Lemma 8 and Lemma 7, $\mathcal{A}'$ is fully-exposing. Finally, we have that $\mathcal{A} \sqsubseteq^{\mathcal{Q}_\mathcal{A}} \mathcal{A}'$ since the algorithm only applies the $\sqsubseteq^{\mathcal{Q}_\mathcal{A}}$–preserving transformations of Sect. 3.3.1. □

A Remark When translating non left-linear rules to left-linear ones, we rename variables. For the right hand sides, we simply rename a variable X with any one of the linear variables X_i generated, since at match time all of them are matched to the same ground term.

We note, however, that our approximation algorithm *is* affected by this choice. This is because for the approximation we only consider *state* bindings, and use $\simeq$ rather than equality. For example, consider the rule $\mathsf{f}(X, X) \Rightarrow \mathsf{g}(X)$. We can translate this rule to one of the following

$$\mathsf{f}(X_1, X_2)_{X_1 = X_2} \Rightarrow \mathsf{g}(X_1)$$
$$\mathsf{f}(X_1, X_2)_{X_1 = X_2} \Rightarrow \mathsf{g}(X_2)$$

Now, assume that we are approximating $\mathsf{f}(@a, @b)$ in step 6 of the algorithm, and that $@a \simeq @b$. Under the first rule above, we will call augment with $\mathsf{g}(@a)$; under the second rule, we use $\mathsf{g}(@b)$ instead. So, if $[\![\mathsf{g}(@a)]\!]_\mathcal{A} \neq [\![\mathsf{g}(@b)]\!]_\mathcal{A}$, our approximation computes different (but *sound*) results in these cases.

In Sect. 3.8 we shall consider handling the rewriting above by computing the *intersection* of the languages of $@a$ and $@b$, and then using this for the augment call.

3.6 Improving Precision

In Sect. 3.2, we hinted at the differences between considering languages up to rewriting (i.e. $[\![@q]\!]_{\mathcal{A}}/\mathcal{R}$) and reachability through $\rightarrow^*_{\mathcal{R},\mathcal{A}}$ (i.e. $[\![@q]\!]_{\mathcal{R},\mathcal{A}}$). We now return to this point, and examine how our approximation technique uses these sets.

So far, in order to compute a sound approximation of the language $[\![@a]\!]_{\mathcal{A}}/\mathcal{R}$, we used algorithms for approximating instead the set $[\![@a]\!]_{\mathcal{R},\mathcal{A}}$. The latter is a correct over-approximation for the former, as Lemma 6 states, so the computed approximation is indeed sound. However, by approximating the set $[\![@a]\!]_{\mathcal{R},\mathcal{A}}$, which is in general larger than the language up to rewriting, we could compute approximations larger than necessary.

In fact, by definition, $T \in [\![@q]\!]_{\mathcal{A}}/\mathcal{R}$ means that $@q \rightarrow^*_{\mathcal{A}} T_p \rightarrow^*_{\mathcal{R}} T$ (with T_p pure), so here $\rightarrow_{\mathcal{A}}$ and $\rightarrow_{\mathcal{R}}$ are applied sequentially, one after the other. In contrast, $T \in [\![@q]\!]_{\mathcal{R},\mathcal{A}}$ means that $@q(\rightarrow_{\mathcal{A}} \cup \rightarrow_{\mathcal{R}})^*T$, allowing for arbitrary interleaving of $\rightarrow_{\mathcal{A}}$ and $\rightarrow_{\mathcal{R}}$.

To consider a concrete example, take $\mathcal{R} = \{\mathsf{f}(X) \Rightarrow 1\}$ and $\mathcal{A} = \{@q \rightarrow \mathsf{f}(@q)\}$. Here $@q$ generates an empty language, thus the language up to rewriting $[\![@q]\!]_{\mathcal{A}}/\mathcal{R}$ is empty as well. However, $1 \in [\![@q]\!]_{\mathcal{R},\mathcal{A}}$ since we have $@q \rightarrow_{\mathcal{A}} \mathsf{f}(@q) \rightarrow_{\mathcal{R}} 1$. By computing a correct w.r.t. $\sqsubseteq^{\mathcal{Q}_{\mathcal{A}}}$, fully-exposing approximation $\mathcal{A}'$ we actually require $@q \rightarrow^*_{\mathcal{A}'} 1$, losing precision.

If we want to achieve a more precise result, we have to replacement $[\![@q]\!]_{\mathcal{R},\mathcal{A}}$ with a closer approximation of $[\![@q]\!]_{\mathcal{A}}/\mathcal{R}$. A way to achieve this is to restrict rewriting so that it is applicable to *inhabited* terms T only, i.e. to terms such that $[\![T]\!]_{\mathcal{A}} \neq \emptyset$. In this way, term $\mathsf{f}(@q)$ in the example above can not be rewritten, and therefore we are not forced to include 1 in the approximation. So, from now on, we shall always use the following restricted rewriting.

Definition 21 *Given an automaton $\mathcal{A}$ and a term rewriting system $\mathcal{R}$, let $\rightarrow_{\mathcal{R}'}$ be the rewriting relation as defined in Definition 20. We define $\rightarrow_{\mathcal{R}}$ as $\rightarrow_{\mathcal{R}'} \cap (\mathcal{H} \times \mathcal{T}_{\mathsf{gr}})$, where $\mathcal{H} = \{T \in \mathcal{T}_{\mathsf{gr}} | [\![T]\!]_{\mathcal{A}} \neq \emptyset\}$.*

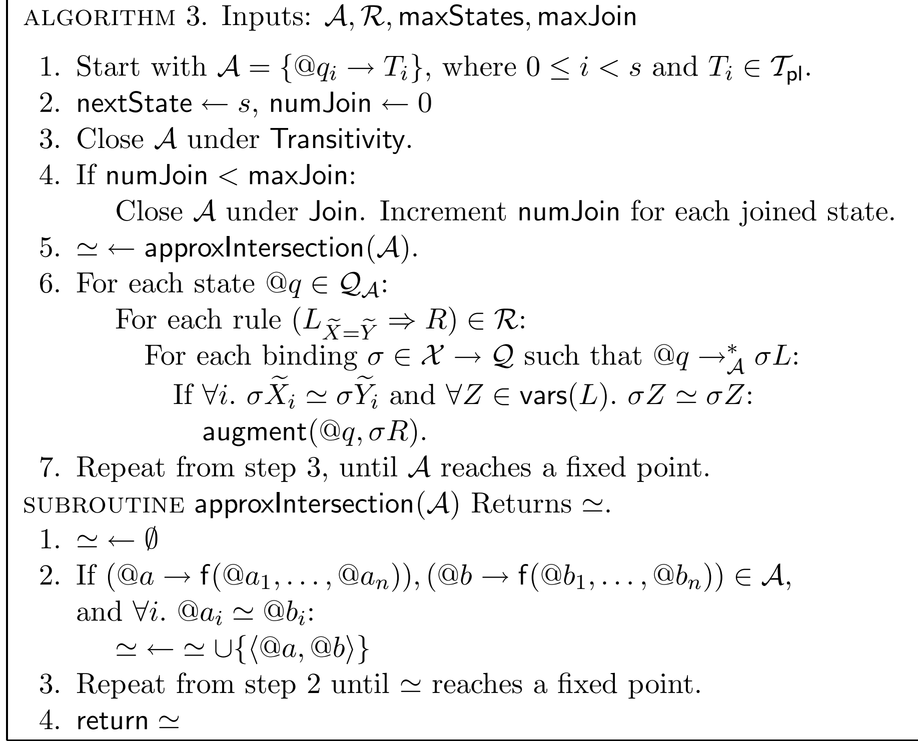
We also update the definitions of $\rightarrow_{\mathcal{R},\mathcal{A}}$, $[\![-]\!]_{\mathcal{R},\mathcal{A}}$, and the notion of "fully-exposing" to use the restricted rewriting, using the automaton at hand as the automaton $\mathcal{A}$ in the definition above.

Lemma 6 is restated as follows, ensuring that fully-exposing approximations indeed include terms up to rewriting.

Lemma 9 *If $\mathcal{A} \sqsubseteq^{\mathcal{Q}'} \mathcal{A}'$ and $\mathcal{A}'$ is fully-exposing, then*

$$\forall @q \in \mathcal{Q}'.\ \ [\![@q]\!]_{\mathcal{A}}/\mathcal{R} \subseteq [\![@q]\!]_{\mathcal{A}'}$$

Proof. Assume $T \in [\![@q]\!]_{\mathcal{A}}/\mathcal{R}$. We have $@q \rightarrow^*_{\mathcal{A}} T_p \rightarrow^*_{\mathcal{R}} T$ for a pure T_p. Since $@q \in \mathcal{Q}'$, we also have $@q \rightarrow'_{\mathcal{A}'} T_p$. Of course, a pure term is always



ALGORITHM 3. Inputs: $\mathcal{A}, \mathcal{R}, \mathsf{maxStates}, \mathsf{maxJoin}$

1. Start with $\mathcal{A} = \{@q_i \to T_i\}$, where $0 \leq i < s$ and $T_i \in \mathcal{T}_{\mathsf{pl}}$.
2. $\mathsf{nextState} \leftarrow s$, $\mathsf{numJoin} \leftarrow 0$
3. Close $\mathcal{A}$ under $\mathsf{Transitivity}$.
4. If $\mathsf{numJoin} < \mathsf{maxJoin}$:
   Close $\mathcal{A}$ under Join. Increment $\mathsf{numJoin}$ for each joined state.
5. $\simeq \leftarrow \mathsf{approxIntersection}(\mathcal{A})$.
6. For each state $@q \in \mathcal{Q}_{\mathcal{A}}$:
   For each rule $(L_{\widetilde{X}=\widetilde{Y}} \Rightarrow R) \in \mathcal{R}$:
   For each binding $\sigma \in \mathcal{X} \to \mathcal{Q}$ such that $@q \to^*_{\mathcal{A}} \sigma L$:
   If $\forall i.\ \sigma\widetilde{X}_i \simeq \sigma\widetilde{Y}_i$ and $\forall Z \in \mathsf{vars}(L).\ \sigma Z \simeq \sigma Z$:
   $\mathsf{augment}(@q, \sigma R)$.
7. Repeat from step 3, until $\mathcal{A}$ reaches a fixed point.

SUBROUTINE $\mathsf{approxIntersection}(\mathcal{A})$ Returns $\simeq$.

1. $\simeq \leftarrow \emptyset$
2. If $(@a \to \mathsf{f}(@a_1, \ldots, @a_n)), (@b \to \mathsf{f}(@b_1, \ldots, @b_n)) \in \mathcal{A}$, and $\forall i.\ @a_i \simeq @b_i$:
   $\simeq \leftarrow \simeq \cup \{\langle @a, @b\rangle\}$
3. Repeat from step 2 until $\simeq$ reaches a fixed point.
4. $\mathsf{return} \simeq$

Figure 3.3: Improved algorithm

inhabited, that is $[\![T_p]\!]_{\mathcal{A}'} \neq \emptyset$. So, we have $@q \to'_{\mathcal{A}'} T_p \to^*_{\mathcal{R}} T$. Therefore, since $\mathcal{A}'$ is fully-exposing, $T \in [\![@q]\!]_{\mathcal{R},\mathcal{A}'} = [\![@q]\!]_{\mathcal{A}'}$. $\square$

We now study how we can exploit these new definitions in our algorithm. Before describing the actual changes we consider another example, the rewriting of $T_1 = \mathsf{g}(@a, @b, @c)$ by the rule $\mathsf{g}(X, Y, Z)_{Y=Z} \Rightarrow 1$. Under the new definition, rewriting only happens if $T_1 \to^*_{\mathcal{A}} T_2 = \mathsf{g}(@a, T', T') \to_{\mathcal{R}} 1$ for some *ground* T', *and* T_2 is inhabited. However, if T_2 is inhabited, so are $@a$ and T'. This means that the languages of $@a, @b$ and $@c$ satisfy

(i) $[\![@a]\!]_{\mathcal{A}} \neq \emptyset$.

(ii) $[\![@b]\!]_{\mathcal{A}} \cap [\![@c]\!]_{\mathcal{A}} \neq \emptyset$ and

We can generalize the above example to arbitrary rewriting rules $L_{\widetilde{X}=\widetilde{Y}} \Rightarrow R$. Assume we are considering whether this rule applies to some term σL, where σ is a state binding. Rewriting is only possible if

(i) for each $Z \in \mathsf{vars}(L)$ we have $[\![\sigma Z]\!]_{\mathcal{F}} \neq \emptyset$, and

(ii) for each side condition $X = Y$ we have $[\![\sigma X]\!]_{\mathcal{A}} \cap [\![\sigma Y]\!]_{\mathcal{A}} \neq \emptyset$

In Fig. 3.3, we show our changes to the algorithm for checking (i) and (ii). First, we modify the subroutine $\mathsf{approxIntersection}$ so that the $\simeq$ relation

tests whether the languages of two states intersect, as established by the following lemma.

Lemma 10 *In Algorithm 3, after each call to* $\mathsf{approxIntersection}$, $\simeq$ *satisfies*

$$\simeq \ = \{\langle @q_1, @q_2\rangle \mid \exists T \in \mathcal{T}_{\mathsf{pr}}.\ @q_1 \to^*_{\mathcal{A}} T \wedge @q_2 \to^*_{\mathcal{A}} T\} \tag{3.6}$$

Proof. Analogous to Lemma 8. Here we also exploit the closure under $\mathsf{Transitivity}$ to avoid the case $@a \to_{\mathcal{A}} @b$. □

Exploiting Lemma 10, we check for (i) and (ii) in step 6 of Algorithm 3. There, we express $[\![\sigma Z]\!]_{\mathcal{F}} \neq \emptyset$ in the equivalent form $\sigma Z \simeq \sigma Z$, so to be able to reuse $\simeq$.

A revised form of Lemma 7 also holds:

Lemma 11 (State Bindings Criterion) *Given* $\mathcal{A}, \mathcal{R}, \simeq$, *such that* $\simeq$ *satisfies (3.6), and* $\mathcal{A}$ *satisfies*

$$\begin{array}{l} \forall (L_{\widetilde{X}=\widetilde{Y}} \Rightarrow R) \in \mathcal{R} \quad \forall \sigma : \mathcal{X} \to \mathcal{Q} \quad \forall @q \in \mathcal{Q}_{\mathcal{A}} \quad . \\ \left.\begin{array}{l} @q \to^*_{\mathcal{A}} \sigma L \\ \forall i.\ \sigma \widetilde{X}_i \simeq \sigma \widetilde{Y}_i \\ \forall Z \in \mathsf{vars}(L).\ \sigma Z \simeq \sigma Z \end{array}\right\} \implies @q \to^*_{\mathcal{A}} \sigma R \end{array} \tag{3.7}$$

then $\mathcal{A}$ *is fully-exposing.*

Proof. Mostly analogous to Lemma 7. We can show that (3.7) implies the following property, involving *any inhabited* (w.r.t. $\mathcal{A}$) term bindings, and equality instead of $\simeq$. We let $\mathcal{H} = \{T \in \mathcal{T}_{\mathsf{gr}} | [\![T]\!]_{\mathcal{A}} \neq \emptyset\}$.

$$\left.\begin{array}{r} \forall (L_{\widetilde{X}=\widetilde{Y}} \Rightarrow R) \in \mathcal{R} \\ \forall \sigma : \mathcal{X} \to \mathcal{H} \\ \forall @q \in \mathcal{Q}_{\mathcal{A}} \end{array}\right\} @q \to^*_{\mathcal{A}} \sigma L \wedge \forall i.\ \sigma \widetilde{X}_i = \sigma \widetilde{Y}_i \implies @q \to^*_{\mathcal{A}} \sigma R \tag{3.8}$$

Assume the hypothesis of the above. As done for Lemma 7, we apply the "state-in-the-middle" lemma and derive $@q \to^*_{\mathcal{A}} C[@m_1, \ldots, @m_n] = \sigma' L \to^*_{\mathcal{A}} \sigma L$, for some $@m_i$ and their related state binding σ'. Here, we have $@m_i \in \mathcal{H}$, so $@m_i \simeq @m_i$. Also, since σL satisfies the equations in the side condition, we have $@m_{x_i} \simeq @m_{y_i}$ for the corresponding states. Therefore, the hypotheses of (3.7) hold, and we can infer $@q \to^*_{\mathcal{A}} \sigma' R \to^*_{\mathcal{A}} \sigma' L \to^*_{\mathcal{A}} \sigma L$. This proves (3.8).

The last part of the proof is identical to that of Lemmata 5 and 7. □

Theorem 5 *Given* $\mathcal{A}, \mathcal{R}$, *Algorithm 3 always outputs a fully-exposing automaton* $\mathcal{A}'$ *such that* $\mathcal{A} \sqsubseteq^{\mathcal{Q}_{\mathcal{A}}} \mathcal{A}'$. *Also,* $\mathcal{A}'$ *is closed under* $\mathsf{Transitivity}$.

Proof. Analogous to Theorem 4. □

3.7 Intersecting Languages

Our algorithm can be adapted to compute an over-approximation for the *intersection* of the languages of a given pair of states, e.g. $[\![@a]\!]_{\mathcal{A}} \cap [\![@b]\!]_{\mathcal{A}}$. This is done by using a third state, e.g. $@c$, to represent the result of the over-approximation.

In more detail, we provide a set $\mathcal{I}$ of *intersection constraints* as a further input to the algorithm. We write such constraints as $@a \cap @b \subseteq @c$, meaning that $[\![@a]\!]_{\mathcal{A}} \cap [\![@b]\!]_{\mathcal{A}} \subseteq [\![@c]\!]_{\mathcal{A}}$, where $@a, @b, @c$ are states of $\mathcal{A}$. The new algorithm is in Fig. 3.4. At each main loop iteration, we call the subroutine handleConstraint for each constraint in $\mathcal{I}$. The subroutine searches for (a superset of) the terms in the intersection of $[\![@a]\!]_{\mathcal{A}}$ and $[\![@b]\!]_{\mathcal{A}}$ exactly as the subroutine approxIntersection: we look for transitions $@a \rightarrow \mathsf{f}(@a_1, \ldots, @a_n)$ and $@b \rightarrow \mathsf{f}(@b_1, \ldots, @b_n)$ such $@a_i \simeq @b_i$ for all i. We then add the transition $@c \rightarrow \mathsf{f}(@a_1, \ldots, @a_n)$ to $\mathcal{A}$ to ensure $[\![@c]\!]_{\mathcal{A}}$ satisfies the constraint.

ALGORITHM 4. Inputs: $\mathcal{A}, \mathcal{I}, \mathcal{R}$, maxStates, maxJoin

1. Start with $\mathcal{A} = \{@q_i \rightarrow T_i\}$, where $0 \leq i < s$ and $T_i \in \mathcal{T}_{\mathsf{pl}}$.
2. nextState $\leftarrow s$, numJoin $\leftarrow 0$
3. Close $\mathcal{A}$ under Transitivity.
4. If numJoin $<$ maxJoin:
   Close $\mathcal{A}$ under Join. Increment numJoin for each joined state.
5. $\simeq \leftarrow$ approxIntersection($\mathcal{A}$).
6. For each $c \in \mathcal{I}$ call handleConstraint(c).
7. For each state $@q \in \mathcal{Q}_{\mathcal{A}}$:
   For each rule $(L_{\widetilde{X}=\widetilde{Y}} \Rightarrow R) \in \mathcal{R}$:
      For each binding $\sigma \in \mathcal{X} \rightarrow \mathcal{Q}$ such that $@q \rightarrow^*_{\mathcal{A}} \sigma L$:
         If $\forall i.\ \sigma\widetilde{X}_i \simeq \sigma\widetilde{Y}_i$ and $\forall Z \in \mathsf{vars}(L).\ \sigma Z \simeq \sigma Z$:
            augment($@q, \sigma R$).
8. Repeat from step 3, until $\mathcal{A}$ reaches a fixed point.

SUBROUTINE handleConstraint($@a \cap @b \subseteq @c$)

1. For each $(@a \rightarrow \mathsf{f}(@a_1, \ldots, @a_n))$, $(@b \rightarrow \mathsf{f}(@b_1, \ldots, @b_n)) \in \mathcal{A}$:
   If $\forall i. @a_i \simeq @b_i$:
      $\mathcal{A} \leftarrow \mathcal{A} \cup (@c \rightarrow \mathsf{f}(@a_1, \ldots, @a_n))$

Figure 3.4: Handling intersection constraints

Theorem 6 *Given $\mathcal{A}, \mathcal{I}, \mathcal{R}$, Algorithm 4 always outputs an automaton $\mathcal{A}'$ such that*

- $\mathcal{A} \sqsubseteq^{\mathcal{Q}_{\mathcal{A}}} \mathcal{A}'$
- *$\mathcal{A}'$ is fully-exposing*

- $\mathcal{A}'$ *is closed under* Transitivity
- $\mathcal{A}'$ *satisfies the constraints in* $\mathcal{I}$.

Proof. Termination of the algorithm follows as for Theorem 5, since the subroutine handleConstraint only adds transitions to $\mathcal{A}'$. Correctness is also shown by the same arguments, except for the constraints satisfaction, which we prove below.

Assume that $\mathsf{f}(T_1, \ldots, T_n)$ is a (pure) term in $[\![@a]\!]_{\mathcal{A}'} \cap [\![@b]\!]_{\mathcal{A}'}$. Since $\mathcal{A}'$ is closed under Transitivity, there are transitions $@a \to \mathsf{f}(@a_1, \ldots, @a_n)$ and $@b \to \mathsf{f}(@b_1, \ldots, @b_n)$, where $@a_i \to^*_{\mathcal{A}'} T_i$ and $@b_i \to^*_{\mathcal{A}'} T_i$. By Lemma 10, at the last main loop iteration, we have $@a_i \simeq @b_i$, and therefore $\mathsf{f}(T_1, \ldots, T_n) \in [\![@c]\!]_{\mathcal{A}'}$. □

Time Complexity From the complexity point of view, our algorithms are quite naïve, and would benefit from the use of more adequate data structures and more efficient subroutines. Yet, the following theorem establishes a polynomial (on the number of the states of $\mathcal{A}$) time complexity bound for Algorithm 4.

However, we believe there is still room for improvement. Indeed, in the theorem we consider the most straightforward implementation of the algorithm, only, often with sub-optimal complexity. For instance, we do not consider the Floyd-Warshall algorithm [34, 69] for computing the transitive closure of a graph, which would help in closing $\mathcal{A}$ under Transitivity, obtaining a lower complexity bound. Also, we do not claim the bound shown below is the best one that can be achieved for Algorithm 4: for the proof we often use the most straightforward bounds, rather than seeking better ones.

Theorem 7 *Assume that*

- $n = \mathsf{maxStates} + \mathsf{maxJoin}$,
- s *is the number of function symbols that occur in* $\mathcal{R}, \mathcal{A}$,
- maxArity *is the maximum arity of such function symbols,*
- $\mathsf{maxLSize} = \max\{\mathsf{size}(L) | (L_{X=Y} \Rightarrow R) \in \mathcal{R}\}$,
- $\mathsf{maxRSize} = \max\{\mathsf{size}(R) | (L_{X=Y} \Rightarrow R) \in \mathcal{R}\}$,
- $|\mathcal{I}|$ *is the number of intersection constraints in* $\mathcal{I}$,
- $|\mathcal{R}|$ *is the number of rewriting rules in* $\mathcal{R}$.

Algorithm 4 runs in worst–case time complexity

$$\mathcal{O}(n^k \cdot (\log s + \mathsf{maxArity} \log n) \cdot s^l \cdot c \cdot (|\mathcal{I}| + |\mathcal{R}|))$$

where

- $k = \max(5 + 4 \cdot \mathsf{maxArity}, 2 + \mathsf{maxArity} \cdot (1 + \mathsf{maxLSize}))$,
- $l = \max(1 + \mathsf{maxLSize}, 4)$,
- $c = \mathsf{maxArity} + \mathsf{maxLSize} + \mathsf{maxRSize}$.

Proof. During the whole run, we use at most n states for $\mathcal{A}$: $@q_0, \ldots, @q_{n-1}$. Using only these states, we can build at most $t = s \cdot n^{\mathsf{maxArity}}$ plain terms. So, in the whole run we consider at most $m = nt = s \cdot n^{\mathsf{maxArity}+1}$ transitions. Note that this also takes into account the transitions for states that at some point are merged with some other state through Join.

At each iteration, if we have not yet reached the fixed point, we modify $\mathcal{A}$. This means, we either perform a Join or add a new transition through augment. The number of Join is bound by maxJoin. The number of transitions added is bound by m. Since $\mathsf{maxJoin} \in \mathcal{O}(m)$, we conclude that we have $\mathcal{O}(m)$ main loop iterations.

We now study the complexity of each iteration. While examining each step, we enclose the factors that contribute to the complexity inside parentheses. We use balanced trees to represent the transitions of $\mathcal{A}$ so we can perform queries and insertions in logarithmic time $\mathcal{O}(\log m) = \mathcal{O}(\mathsf{maxArity} \log n + \log s)$. We also use balanced trees for $\simeq$, performing queries and insertions in time $\mathcal{O}(\log(n^2)) = \mathcal{O}(\log n)$. Further, with these representations a scan of all the transitions of $\mathcal{A}$ takes linear time.

Step 3 closes $\mathcal{A}$ under Transitivity. For each transition $@q \to T$ ($\mathcal{O}(m)$), we check whether $T = @r$ and in that case we add all the transitions of $@r$ to $@q$ ($\mathcal{O}(t \log m)$). We repeat this process until $\mathcal{A}$ is closed, with $\mathcal{O}(m)$ iterations. So, step 3 takes $\mathcal{O}(m^2 t \log m) = \mathcal{O}(s^3 n^{3\mathsf{maxArity}+2} \log m)$.

Step 4 closes $\mathcal{A}$ under Join. We consider all the pairs $@q, @r$ ($\mathcal{O}(n^2)$) having the same transitions ($\mathcal{O}(t)$ check). When we found such a pair, we substitute a state for the joined one at every occurrence in $\mathcal{A}$ ($\mathcal{O}(m)$). When substituting, we delete the old transition and insert the new one ($\mathcal{O}(\log m)$). To reach the closure, we repeat this process at most maxJoin ($\mathcal{O}(n)$) times. Therefore, step 4 takes $\mathcal{O}(n^2(t + m \log m)n) = \mathcal{O}(n^3 m \log m)$ $= \mathcal{O}(s n^{4+\mathsf{maxArity}} \log m)$.

For step 5, we run approxIntersection. In step 2 of the subroutine we scan for transition pairs ($\mathcal{O}(m^2)$, and check if the states involved by them intersect ($\mathcal{O}(\mathsf{maxArity} \log n)$). If this is the case, we insert the state pair into the $\simeq$ relation ($\mathcal{O}(\log n)$). To reach the fixed point, we run this process at most n^2 times. We conclude that step 5 takes time $\mathcal{O}(m^2(\mathsf{maxArity} \log n + \log n) n^2) = \mathcal{O}(s^2 n^{2\mathsf{maxArity}+4} \log n \cdot \mathsf{maxArity})$.

In step 6, for each intersection constraint ($|\mathcal{I}|$), we call handleConstraint. We consider transition pairs from two selected states ($\mathcal{O}(t^2)$), check for $\simeq$ ($\mathcal{O}(\mathsf{maxArity} \log n)$), and optionally insert a transition ($\mathcal{O}(\log m)$). So, the whole step 6 takes $\mathcal{O}(t^2(\mathsf{maxArity} \log n + \log m) \cdot |\mathcal{I}|) \leq \mathcal{O}(s^2 n^{2\mathsf{maxArity}} \log m \cdot \mathsf{maxArity} \cdot |\mathcal{I}|)$.

In step 7, we consider each state $@q$ ($\mathcal{O}(n)$) and each rule (in $|\mathcal{R}|$). We match $@q$ with the left hand side of the rule, possibly expanding states into terms $\mathsf{maxLSize}$ times ($\mathcal{O}(t^{\mathsf{maxLSize}})$). In each case, we check equations for the rule ($\mathcal{O}(\mathsf{maxLSize} \log n)$). If everything succeeds we call $\mathsf{augment}(\sigma R)$. Depending of the size of R ($\mathsf{maxRSize}$), we insert a number of transitions in $\mathcal{A}$ (each $\mathcal{O}(\log m)$). Hence, the overall complexity of step 7 is

$$\begin{aligned}&\mathcal{O}(n \cdot |\mathcal{R}| \cdot t^{\mathsf{maxLSize}}(\mathsf{maxLSize} \log n + \mathsf{maxRSize} \log m)) \leq \\ \leq\ &\mathcal{O}(s^{\mathsf{maxLSize}} n^{\mathsf{maxArity} \cdot \mathsf{maxLSize}+1} \log m \cdot |\mathcal{R}| \cdot (\mathsf{maxLSize} + \mathsf{maxRSize}))\end{aligned}$$

Summing up the complexity of steps $3 \ldots 7$, we obtain the complexity of one main loop iteration:

$$\mathcal{O}(s^{\max(\mathsf{maxLSize},3)} \cdot n^{\max(\mathsf{maxArity} \cdot \mathsf{maxLSize}+1, 3\mathsf{maxArity}+4)} \cdot \log m \cdot c \cdot (|\mathcal{I}| + |\mathcal{R}|))$$

where $c = \mathsf{maxArity} + \mathsf{maxLSize} + \mathsf{maxRSize}$.

Multiplying the above by the number of iterations $\mathcal{O}(m)$, we get

$$\mathcal{O}(s^l \cdot n^k \cdot \log m \cdot c \cdot (|\mathcal{I}| + |\mathcal{R}|)$$

where $l = \max(\mathsf{maxLSize} + 1, 4)$ and $k = \max(\mathsf{maxArity} \cdot (1 + \mathsf{maxLSize}) + 2, 4\mathsf{maxArity} + 5)$.

Since $\mathcal{O}(\log m) = \mathcal{O}(\log(sn^{\mathsf{maxArity}+1})) = \mathcal{O}(\log s + \mathsf{maxArity} \log n)$, we get the bound

$$\mathcal{O}(s^l \cdot n^k \cdot (\log s + \mathsf{maxArity} \log n) \cdot c \cdot (|\mathcal{I}| + |\mathcal{R}|))$$

□

3.8 Further Improvements

Of course, our handling of intersection constraints is approximate: it only looks at terms reachable through $\rightarrow_{\mathcal{A}}$ in a single step. When we discover that $@a \rightarrow_{\mathcal{A}} T_a$ and $@b \rightarrow_{\mathcal{A}} T_b$ witness an intersection, we simply add a transition $@c \rightarrow T_a$ rather than approximating the intersection in a more precise way, possibly through several transitions $@c \rightarrow T_{ci}$, where the T_{ci} may involve fresh states. This would require a careful exploration of the descendants of $@a$ and $@b$, trying to find a good trade-off between the precision of the result and the cost of that search. Particular care should also be used if, in order to express the result of the intersection, fresh states are needed. Generating new states eventually leads to reusing previous states, much as it happens for the subroutine place, which causes a loss of precision. So, it is possible for "smarter" approximations to be worse both in complexity and in precision.

This is somehow related to our handling of rewriting rules such as $\mathsf{f}(X, X) \Rightarrow \mathsf{g}(X)$, by using instead $\mathsf{f}(X_1, X_2)_{X_1=X_2} \Rightarrow \mathsf{g}(X_1)$. Also here, when running our algorithm, we rewrite $T_l = \mathsf{f}(@a, @b)$ with $T_r = \mathsf{g}(@a)$ as an approximation of the intersection $@a \cap @b$. This approximation is actually less precise than the one used for intersection constraints: here we do not search at all for the descendants of the states. As before, this is so to avoid generating fresh states, as it would be needed when T_r is not a plain term and we would need to normalize transitions. However, for the special case of rewriting rules having right hand sides of depth zero, e.g. $\mathsf{f}(X, X) \Rightarrow X$, it is possible to use the same approximation used for intersection constraints, since it produces only plain terms.

3.9 Implementation

We implemented the proposed algorithm, with the changes discussed in sections 3.6 and 3.7. Our implementation is rather faithful to the algorithm as we have presented it. As such, the most important aspects of the implementation have already been covered in this chapter. However, we discuss here some implementation details we still find noteworthy.

3.9.1 Representing Transitions and $\simeq$

In our implementation, we store the transitions of $\mathcal{A}$ as a map from states to plain term sets. That is, if we have $@a \rightarrow \mathsf{f}(@b)$ and $@a \rightarrow \mathsf{g}(@c, @a)$, we simply map the state $@a$ to the set $\{\mathsf{f}(@b), \mathsf{g}(@c, @a)\}$.

Similarly, we treat $\simeq$ as a set of pairs of states. Note that in the last version of our algorithm we never need to perform complex queries on $\simeq$, but we only check whether $\simeq$ holds between given pairs of states. Our implementation actually stores the $\simeq$ set along the transitions of the automaton $\mathcal{A}$, employing a dummy state `@!Inters` and the following transitions:

$$@!Inters \rightarrow \mathsf{cons}(@a, @a), \mathsf{cons}(@a, @b), \ldots$$

This representation simplifies joining states, since substituting $\{@b/@a\}$ everywhere in the automaton updates $\simeq$ accordingly. Moreover, this representation allows for the specification of some advanced rules, which we discuss below.

3.9.2 More Expressive Rules

In the implementation, we translate rewriting rules such as

$$\mathsf{f}(\mathsf{g}(X), Y) \Rightarrow \mathsf{h}(\mathsf{l}(X))$$

in a representation that makes pattern matching on the left hand side explicit, using a conjunction of clauses $S : T$, as follows:

$$S : \mathsf{f}(S', Y) \wedge S' : \mathsf{g}(X) \Rightarrow S : \mathsf{h}(\mathsf{l}(X))$$

These rules only match a single constructor at a time. In the example above, we first search for f, and then for g. The newly introduced variables, S and S', are fresh and, as the other variables, are only matched against states. Also, in the left hand side, only terms of depth one are used. In the right hand side, however, any (simple) term is allowed. In each clause $S : T$ the same variable may occur only once, unless it also occurs in a previous clause.

Note that this kind of rule is strictly more expressive than rewriting rules: for instance

$$S : \mathsf{1} \wedge S : \mathsf{2} \Rightarrow S : \mathsf{3}$$
$$@a : \mathsf{1} \Rightarrow @b : \mathsf{1}$$
$$S : \mathsf{f}(S') \Rightarrow S' : \mathsf{1}$$
$$S : \mathsf{1} \wedge @a : \mathsf{g}(S, S) \Rightarrow S : \mathsf{1}$$

cannot be simulated through usual rewriting rules. The effect of the first rule above is to add the term 3 to a language, provided that language includes terms 1 and 2. The second rule instead performs a check on the language $[\![@a]\!]_{\mathcal{A}}$ to update *another* language $[\![@b]\!]_{\mathcal{A}}$. The third rule is a generalization of the second, in which the state to be updated is only known at match time. The fourth rule checks $[\![@a]\!]_{\mathcal{A}}$ for terms of the form $\mathsf{g}(@b, @b)$ where $\mathsf{1} \in [\![@b]\!]_{\mathcal{A}}$; note that transitions such as $\mathsf{g}(@b, @c)$ do not pass the test, even if the languages of $@b$ and $@c$ are the same.

While this kind of rule is more expressive, these rules still operate monotonically on $\mathcal{A}$, only *adding* transitions. Therefore, termination is still guaranteed, as it is the closure under the rules on termination.

Rewriting rules with equations on variables such as

$$\mathsf{f}(X, Y)_{X=Y} \Rightarrow \mathsf{h}(X)$$

are translated exploiting the special `@!Inters` state:

$$S : \mathsf{f}(X, Y) \wedge @!Inters : \mathsf{cons}(X, Y) \Rightarrow S : \mathsf{h}(X)$$

Having $\simeq$ directly available in rules is sometimes convenient, making it possible to specify "witness" tests such as

$$@!Inters : \mathsf{cons}(@analysisResult, @badValues) \Rightarrow @witness : \mathsf{analysisFailed}$$

so that we can easily check if the tool was able to prove $[\![@analysisResult]\!]_{\mathcal{A}}$ and $[\![@badValues]\!]_{\mathcal{A}}$ disjoint by simply observing $@witness$. Indeed, our implementation provides a special witness state $@zzResult$: if a transition $@zzResult \rightarrow \mathsf{!FAIL!}$ is ever found, the tool aborts immediately reporting the failure. Our tests always include some rule such as $\ldots \Rightarrow @zzResult : \mathsf{!FAIL!}$ to check whether the approximation meets our expectations.

3.9.3 Heuristics

In Algorithm 4, after a successful match of a rule, we call the subroutine augment to add the transition $@q \to \sigma R$ to the automata $\mathcal{A}$. However, if σR is not a plain term, the transition must be normalized: this is done by the subroutine place (see Fig. 3.1). In the presented algorithm, place sometimes has to choose a state $@q$ among some given set. While the soundness of the algorithm is not affected by these choices, the precision of the result might be. For instance, consider the following $\mathcal{A}$

$$@a \to 1, 2 \qquad @b \to 1, 3 \qquad @c \to 0$$

and let $\mathcal{R} = \{0 \Rightarrow \mathsf{f}(1)\}$. Running the algorithm, we have to normalize $@c \to \mathsf{f}(1)$. If we already reached the maxStates bound, we have to reuse a state $@q \in \{@a, @b, @c\}$ and add the transitions $@c \to \mathsf{f}(@q)$ and $@q \to 1$. If we choose $@q = @a$, we will have $\mathsf{f}(1), \mathsf{f}(2) \in [\![@c]\!]_{\mathcal{A}}$. If we instead choose $@q = @b$, we will have $\mathsf{f}(1), \mathsf{f}(3) \in [\![@c]\!]_{\mathcal{A}}$. Finally, we can choose $@q = @c$, and have $1, \mathsf{f}(0), \mathsf{f}(1), \mathsf{f}(\mathsf{f}(1)), \ldots \in [\![@c]\!]_{\mathcal{A}}$. No alternative yields a more precise result than the others. Which is the best one actually depends on what these terms represent for the user who is running the algorithm: if, for instance, all the user wants is to prove $\mathsf{f}(2) \notin [\![@c]\!]_{\mathcal{A}}$, then any choice but $@q = @a$ will yield an approximation precise enough.

In general, however, we have to resort to some heuristics for choosing which state to reuse. In our implementation we used the ones below. Some of these are already included in our definition of place. Assume below we are normalizing to $@r \to \mathsf{f}(\ldots, @q, \ldots)$ and $@q \to T$.

- If we already have some transition $@s \to T$ in the current $\mathcal{A}$, we reuse one of those $@s$. This somehow simple heuristic turned out to be quite significant, since it avoids a subtle resource wasting problem caused by following phenomenon.

  At each main loop iteration we call augment for each match of $\mathcal{R}$ against $\mathcal{A}$, including those matches that were already discovered in previous iterations. Therefore, we perform the same normalizations many times. If we kept generating new states for this, many states would be generated for the same normalization, and therefore for the same language. Then, we would compute the same rewritings for all these languages, and keep distinct the states $@a, @b, @c, \ldots$ for different instances of the same language.

  $$@c \to 0 \qquad @b \to 0, 1 \qquad @a \to 0, 1, 2$$

  Above, a new state $@q$ for $@q \to 0$ is being generated at each iteration, and then its language is rewritten under rules $\mathcal{R} = \{0 \Rightarrow 1, 1 \Rightarrow 2, 2 \Rightarrow 3\}$. These languages will eventually converge to the same representation and then merged with a Join operation into the same state.

However, the languages may require several steps to converge and trigger Join, and we are likely to hit the maxStates bound before states are freed. This causes the reuse of states, possibly even those that are going to converge: if this happens, the languages may get perturbed so that Join does not happen.

Using the heuristic, we generate a state only the first time we normalize a transition (if at all). The next times we attempt the same normalization, we will find at least the state we generate before (modulo Join) and reuse that state.

An alternative to this heuristic would be recording the results of the normalizations in some data structure, and checking whether we can reuse them later. In our tests, this often increased the memory footprint and the overall complexity without significantly improving the results.

- If there are many states $@s$ such that $@s \to T$ exists, prefer those that were not automatically generated, i.e. those that were in the input $\mathcal{A}$. The rationale behind this heuristic is that user-defined states might carry more semantically significant languages, and therefore their use might produce better approximations. This is somehow in contrast to the fact that generated states, at least before they are reused, start with only one term and so have usually smaller languages with respect to user-defined ones. Our implementation has a switch for enabling or disabling this heuristic.

- If we have not yet reached the maxStates bound, and no transition $@s \to T$ is in $\mathcal{A}$, we generate a fresh state $@q$.

- If everything else fails, fallback to reusing the last generated state.

Further, we use these heuristics, that are not related to the normalization of transitions:

- We fix an order on the states. In our implementation, we simply use the lexicographic ordering over the *names* of the states. Also, generated states are greater than user-defined ones, and are ordered so that newer states are greater. When scanning a sets of states, we scan them in increasing order. When choosing, we prefer the smallest.

  Note that our algorithm may produce different (sound) approximations depending on the actual order of matches and normalizations. The choice of a fixed ordering for these guarantees *reproducible* results, essentially making our implementation resilient to some changes to the low-level routines, e.g. the routine that serializes a set into a list of elements.

As a side effect, using a fixed ordering also allows the user to specify "high priority" states ($@aa$) and "low-priority" ones ($@zz$). In this way, while providing the input $\mathcal{A}$, the user specifies a hint about which normalizations should be considered first.

- We apply Join aggressively. Since Join can avoid duplicating work, the sooner it is applied, the better. Moreover, when using rewriting rules such as associativity or commutativity, many equivalent languages are generated, so early joining is crucial.

- When possible, we also apply Join to states in $\mathcal{Q}_\mathcal{A}$.

  In our previous sections, we showed how to compute approximations of $\mathcal{A}$ respecting the ordering given by $\sqsubseteq^{\mathcal{Q}_\mathcal{A}}$. We do this by applying only automaton transformations preserving that preorder. In particular, we never join states in $\mathcal{Q}_\mathcal{A}$, as the side condition of rule Join requires. This ensures that we compute a sound approximation for the languages of *all* the states of $\mathcal{A}$.

  However, in practice, most of the states of $\mathcal{A}$ are auxiliary: we are interested only in the languages of a few states, disregarding the rest. So, we do not need to compute approximations for all the states. We rather can require the computed approximation to preserve the preorder $\sqsubseteq^{\mathcal{Q}'}$, where $\mathcal{Q}'$ is a subset of $\mathcal{Q}_\mathcal{A}$ selecting only those states we care about. In this way, we allow joining for all the states not in $\mathcal{Q}'$, usually including most of the states of $\mathcal{A}$.

  In our implementation, we chose $\mathcal{Q}'$ as the set of the states that occur in the (extended) rules and in intersection constraints. This follows the common practice of encoding a "witness" test in the rules. In this case, we ensure that the languages involved in the test are soundly approximated, while the others can be arbitrarily joined.

- We perform normalizations that do not generate new states before the others. This is to reuse existing states as much as possible.

3.9.4 Other Optimizations

We briefly comment on some minor optimizations we found in the development of our tool.

- We do not recompute $\simeq$ from scratch at every main loop iteration. Rather, we exploit the fact that if $@a \simeq @b$ for the $\mathcal{A}$ in the previous loop iteration, we still have $@a \simeq @b$ for the current $\mathcal{A}$, provided no Join merged $@a$ or $@b$ with some other state. So, we consider for addition to $\simeq$ only the pairs $(@a, @b)$ such that $@a \not\simeq @b$ in the previous iteration.

- When translating non left-linear rules into left-linear ones, we add as many equations as possible in side conditions. For instance, we translate $\mathsf{f}(X, X, X) \Rightarrow 0$ to

  $$\mathsf{f}(X_1, X_2, X_3)_{X_1=X_2, X_2=X_3, X_1=X_3} \Rightarrow 0$$

  Above, the third equation is redundant, since it is implied by the other two. That is, removing the third equation yields the same $\to_{\mathcal{R}}$ relation. However, our approximation algorithm may produce a more precise result if we keep all the equations above. This is because, when we have to determine whether the rewriting rule applies to a term as $\mathsf{f}(@a, @b, @c)$ we check for $@a \simeq @b$, $@b \simeq @c$, and $@a \simeq @c$. If any of these checks fails, we deduce that rewriting does not apply, and therefore we do not add 0 through augment to the language at hand. Note that it is possible to have $@a \simeq @b$, $@b \simeq @c$, but $@a \not\simeq @c$. Removing the third equation, and consequently not performing the third check, would cause adding 0, with a loss of precision. Therefore, the last check can indeed help in achieving a better result.

  However, in practice, most non left-linear rewriting rules do not use more than *two* occurrences for each variable in their left hand sides. So, in many cases this optimization has no effect. Yet, its implementation is simple, so we included it in our tool.

3.9.5 Test Cases

We applied our tool to several rewriting systems, including those for pairs (cons, fst, snd), encryptions (enc, dec), and also those for xor and exponentials (exp, $*$, inv, 1) that do not admit simple normal forms. These experiments often produced approximations precise enough to show the security properties we were looking for. For instance, we successfully checked a one-time pad example, and other examples for xor. Also, a flawed version of the WEP protocol [21] did *not* pass the test, as expected. We also successfully checked the Common Crypto API (CCA) [25], that involves hashes, xor, and a number of black-box operations to be executed by trusted hardware. The tool showed its limitations when applied to an exotic definition of even and odd naturals: in that case the tool over-approximated the sets yielding the whole $\mathbb{N}$. In this section, we report on the results of these experiments.

Finally, we shall use our completion tool in Chapter 4 to define a protocol verification technique. We shall be able to statically check protocols specified in a process calculus. Therefore, the examples of Chapter 4 also contribute to confirm the validity of the approximation techniques presented in this chapter. Notably, there we shall analyze a protocol based on the Diffie-Hellman key exchange [31], dealing with exp, $*$, inv, 1.

One-time Pad

Here, a single message msg, xored with a secret key key, is sent through the network. The key is used only for this purpose and only once. A recipient knowing the secret key can recover the original message by applying xor once more so that

$$\mathsf{xor}(\mathsf{xor}(\mathsf{msg}, \mathsf{key}), \mathsf{key}) = \mathsf{msg}$$

We represent the network as the language of the state $@net$. We assume the presence of an adversary, able to manipulate messages through xor operations. Further, the adversary is also allowed to use 0. We model this adversary by making $@net$ closed under the operations available to the adversary.

We expect the adversary not to be able to derive the plaintext message msg. In other words, we expect $\mathsf{msg} \notin [\![@net]\!]_{\mathcal{A}}/\mathcal{R}$. Our tool succeeds in proving this.

While the security of this simple protocol has already been established, even for low-level models, it is comforting to know that our tool is able to achieve the corresponding result in a formal, high-level model. Note that, despite the simplicity of the protocol, the rewriting rules include associativity and commutativity rules.

Below, we provide the actual input for our tool.

```
# xor rules

^(X,Y) => ^(Y,X) .
^(^(X,Y),Z) => ^(X,^(Y,Z)) .
^(X,X) => 0 .
^(0,X) => X .

# Expected Result
| @net : msg => @zzResult : !FAIL! .

%%

@msg : msg .
@key : key .
@net : ^(@msg,@key) .

# adversary rules

@net : 0 , ^(@net,@net) .

@zzResult : soFarSoGood .
```

Another xor Example

We experiment with xor again. We define two languages using mutual recursion, with the following transitions.

$$@res1 \to \mathsf{xor}(\mathsf{a}, @res2)$$
$$@res2 \to \mathsf{xor}(\mathsf{d}, @res1), \mathsf{e}$$

Under these definitions, we expect $[\![@res1]\!]_{\mathcal{A}}/\mathcal{R} = \{\mathsf{xor}(\mathsf{a},\mathsf{e}), \mathsf{xor}(\mathsf{d},\mathsf{e})\}$ and $[\![@res2]\!]_{\mathcal{A}}/\mathcal{R} = \{\mathsf{e}, \mathsf{xor}(\mathsf{d}, \mathsf{xor}(\mathsf{a},\mathsf{e}))\}$, both to be closed under rewriting. Below, we check that a number of other terms are not included in the language $@res1$, so that the computed approximation is reasonably tight. Our tool succeeds in this test.

```
# Rules for xor.

^(0,Y) => Y .
^(X,Y) => ^(Y,X) .
^(^(X,Y),Z) => ^(X,^(Y,Z)) .
^(X,X) => 0 .

# @zz summarizes @res1

| @res1 : ^(X,Y) , X : a , Y : b => @zz : ab .
| @res1 : ^(X,Y) , X : a , Y : c => @zz : ac .
| @res1 : ^(X,Y) , X : a , Y : d => @zz : ad .
| @res1 : ^(X,Y) , X : a , Y : e => @zz : ae .

| @res1 : ^(X,Y) , X : b , Y : c => @zz : bc .
| @res1 : ^(X,Y) , X : b , Y : d => @zz : bd .
| @res1 : ^(X,Y) , X : b , Y : e => @zz : be .

| @res1 : ^(X,Y) , X : c , Y : d => @zz : cd .
| @res1 : ^(X,Y) , X : c , Y : e => @zz : ce .

| @res1 : ^(X,Y) , X : d , Y : e => @zz : de .

# If @zz contains unwanted terms, the analysis fails.

| @zz : ab => @zzResult : !FAIL! .
| @zz : ac => @zzResult : !FAIL! .
| @zz : ad => @zzResult : !FAIL! .

| @zz : bc => @zzResult : !FAIL! .
| @zz : bd => @zzResult : !FAIL! .
```

```
| @zz : be => @zzResult : !FAIL! .

| @zz : cd => @zzResult : !FAIL! .
| @zz : ce => @zzResult : !FAIL! .

%%

# Constants

@0 : 0 .
@a : a .
@b : b .
@c : c .
@d : d .
@e : e .

@res1 : ^(@a,@res2) .        # @res1 = ae, de .
@res2 : ^(@d,@res1) , @e .   # @res2 = e, dae .

@zzResult : soFarSoGood.
```

WEP

We now consider a flawed version of the WEP protocol [21]. Here, the protocol sends over the network the messages

$$\mathsf{n1}, \mathsf{n2}, \mathsf{v}$$
$$\mathsf{cons}(\mathsf{v}, \mathsf{xor}(\mathsf{cons}(\mathsf{n1}, \mathsf{c}(\mathsf{n1})), \mathsf{rc4}(\mathsf{v}, \mathsf{k})))$$

For the actual intended meaning, we refer the reader to [21]. For our purposes, it is sufficient to know that the protocol, after having sent the above messages, relies on the adversary *not* to be able to derive

$$\mathsf{xor}(\mathsf{cons}(\mathsf{n2}, \mathsf{c}(\mathsf{n2})), \mathsf{rc4}(\mathsf{v}, \mathsf{k}))$$

However, an adversary able to manipulate messages using pair construction and destruction ($\mathsf{cons}, \mathsf{fst}, \mathsf{snd}$), operation c, and xor is actually able to construct the above, disrupting the protocol.

We run the tool, asking it to prove the secrecy of the above message. The tool fails, as it should, because of the flaw.

```
# Rules for xor.

^(0,Y) => Y .
^(X,Y) => ^(Y,X) .
```

```
^(X,^(Y,Z)) => ^(^(X,Y),Z) .
^(X,X) => 0 .

# Pairs: fst/snd/cons

fst(cons(X,Y)) => X .
snd(cons(X,Y)) => Y .

# Expected result:
# The value ^(cons(n2,c(c2)) , rc4(v,k)) should be kept secret,
# but it is not, since the protocol is flawed. We fail the test.

| @net : ^(P,RC4) , P : cons(N2, CN2) ,
        N2 : n2 , CN2 : c(N2b) , N2b : n2 ,
        RC4 : rc4(V,K) , V : v , K : k
        => @zzResult : !FAIL! .

%%

# Constants.
@0 : 0 .

@n1 : n1 .
@cn1 : c(@n1) .
@p : cons(@n1,@cn1) .
@v : v .
@k : k .
@rc4 : rc4(@v,@k) .
@c : ^(@p,@rc4) .

@net : n1 , n2 .
@net : v .
@net : cons(@v,@c) .

# Adversary rules
@net : cons(@net,@net) , fst(@net) , snd(@net) ,
       ^(@net,@net), c(@net) .

@zzResult : soFarSoGood .
```

Common Crypto API

The Common Crypto API (CCA) [25] defines a set of primitives used in every Automated Teller Machine (ATM) over the world. In the ATMs there

is a trusted hardware cryptographic component that stores a number of secret keys used for the ATM operations. This component can only be accessed through the CCA primitives, and so it forms a black-box device only allowing some interactions with the secrets it holds. Of course, this trusted component should never disclose its secrets, even when the operations of the API are run in an unusual, malicious order.

In [30], Courant and Monin provide a formal proof of the correctness of the CCA. For the sake of brevity, we do not describe here the ideas behind the complex CCA primitives, and its exact secrecy property that was shown in [30]. We refer the reader to [30] for a more adequate description. For our purposes, we simply note that the property shown in [30] states that, using the six primitives of CCA, plus public operations (e.g. hashes, xor), one can not derive any of the secrets stored in the trusted component.

The proof in [30] was written and verified with the help of the Coq proof assistant [29] to ensure its soundness. The files containing the proof contain 2100 lines of hand-made definitions, lemmata, and theorems.

We ran our approximation tool on the CCA specification. We define the CCA primitives $f1, \dots, f6$ using suitable rewriting rules, as we do for hashes, encryptions and xor. Then, we define the language of $@k$ so to include public information and to be closed under the primitives. Finally, we check that the language of $@k$ does not contain any secret, by testing its intersection with $@secret$. Below, we give the complete specification.

Our tool succeeded in proving the same secrecy property of [30] with no manual intervention.

```
# Encryption

dec(enc(M,K),K) => M .

# Rules for XOR

+(X,Y) => +(Y,X) .
+(+(X,Y),Z) => +(X,+(Y,Z)) .
+(0,X) => X .
+(X,X) => 0 .

# Hash (no rules required)

###
# The six CCA operations

# K_key_part_import_notcompleting
f1(X,Y,enc(Z,h(h(X,kp),km))) => enc(+(Z,Y),h(h(X,kp),km)) .
```

```
# K_key_part_import_notcompleting
f2(X,Y,enc(Z,h(h(X,kp),km))) => enc(+(Z,Y),h(X,km)) .

# K_key_import
f3(T,enc(K,h(imp,km)),enc(X,h(T,K))) => enc(X,h(T,km)) .

# K_key_export
f4(T,enc(K,h(exp,km)),enc(X,h(T,km))) => enc(X,h(T,K)) .

# K_key_encrypt_using_data_key
f5(X,enc(K,h(data,km))) => enc(X,K) .

# K_key_decrypt_using_data_key
f6(enc(X,K),enc(K,h(data,km))) => X .

| @!Inters : cons(@k, @forbidden) => @zzResult : !FAIL! .

%%

@public : data,pin,imp,k3,acc,kp.

@secret : km,kek,p,kek2,exp1 .

@forbidden : kek,km,p,enc(acc,p).

@k : 0 ,
     @public,
     enc(p,h(pin,kek)) ,
     enc(+(kek,k3),h(h(imp,kp),km)) ,
     enc(kek2, h(h(imp,kp),km)) ,
     enc(kek2, h(h(exp,kp),km)) ,
     enc(exp1, h(exp,km)) .

@k : +(@k,@k) , h(@k,@k) ,
     enc(@k,@k) , dec(@k,@k) ,
     f1(@k,@k,@k) , f2(@k,@k,@k) ,
     f3(@k,@k,@k) , f4(@k,@k,@k) ,
     f5(@k,@k) , f6(@k,@k) .
```

Even and Odd

Here, we construct the sets of even and odd naturals, with the usual Peano encoding. We use sum (+) and successor (s) operations. We check that in the approximation some even numbers do not appear between the odd ones,

and vice versa. Our tool succeeds in providing a good enough approximation.

```
# Peano's axioms for naturals

+(0,X)     => X .
+(s(X),Y) => s(+(X,Y)) .

# @resA summarizes @a , @resB summarizes @b

|                                  @a : 0 => @resA : 0 .
|                      @a : s(X) , X : 0 => @resA : 1 .
|           @a : s(Y) , Y : s(X) , X : 0 => @resA : 2 .
| @a : s(Z) , Z : s(Y) , Y : s(X) , X : 0 => @resA : 3 .

|                                  @b : 0 => @resB : 0 .
|                      @b : s(X) , X : 0 => @resB : 1 .
|           @b : s(Y) , Y : s(X) , X : 0 => @resB : 2 .
| @b : s(Z) , Z : s(Y) , Y : s(X) , X : 0 => @resB : 3 .

# Expected result: no odds in @a, no evens in @b

| @resA : 1 => @zzResult : !FAIL! .
| @resA : 3 => @zzResult : !FAIL! .
| @resB : 0 => @zzResult : !FAIL! .
| @resB : 2 => @zzResult : !FAIL! .

%%

# @a has only even numbers

@a : 0 , @a , s(@b) , +(@a,@a) .

# @b has only odd numbers

@b : s(@a) .

@zzResult : soFarSoGood .
```

Even and Odd, Revisited

We again define even and odd naturals. However, here we also use a "$-2(x)$" operation, meaning the subtraction of 2 from x. The purpose of using such a operation is to try to confuse the tool into mixing even with odds.

Our tool was indeed confused. The provided approximation used the

whole set of naturals, for both even and odd numbers. So, in this case, our tool failed the test. We stress that the result *is* sound, but not as precise as we hoped it to be.

```
# Peano's axioms for naturals 0,s(),+(,)
# We also use -2() as "subtract 2"

+(0,Y)       => Y .
+(s(X),Y)    => s(+(X,Y)) .
-2(s(s(X))) => X .

# @res summarizes @a

|                            @a : 0 => @res : 0 .
|              @a : s(X0) , X0 : 0 => @res : 1 .
| @a : s(X1) , X1 : s(X0) , X0 : 0 => @res : 2 .
| @a : s(X2) ,
  X2 : s(X1) , X1 : s(X0) , X0 : 0 => @res : 3 .
| @a : s(X3) , X3 : s(X2) ,
  X2 : s(X1) , X1 : s(X0) , X0 : 0 => @res : 4 .
| @a : s(X4) ,
  X4 : s(X3) , X3 : s(X2) ,
  X2 : s(X1) , X1 : s(X0) , X0 : 0 => @res : 5 .
| @a : s(X5) , X5 : s(X4) , X4 : s(X3) , X3 : s(X2) ,
  X2 : s(X1) , X1 : s(X0) , X0 : 0 => @res : 6 .

# Expected result: @res should contain only even numbers

| @res : 1 => @zzResult : !FAIL! .
| @res : 3 => @zzResult : !FAIL! .
| @res : 5 => @zzResult : !FAIL! .

%%

@zzResult : soFarSoGood .

# constants
@n0 : 0 .
@n1 : s(@n0) .
@n2 : s(@n1) .
@n3 : s(@n2) .
@n4 : s(@n3) .

# @a has only even numbers
```

```
@a : @n4 , +(@a,@a) , -2(@a) .

# Even and odd numbers

@even : @n0 , s(@odd) .
@odd  : s(@even) .
```

Pathological Cases

Our algorithm uses the maxJoin bound to ensure that only a finite number of Join operations are performed. This is to ensure that eventually only monotonic operations are applied to $\mathcal{A}$, making it converge. However, one might ask whether maxJoin is actually needed, i.e. if there is such an $\mathcal{A}$ that can trigger an unbounded number of Join under some $\mathcal{R}$. We call such an $\mathcal{A}$ *wild* (w.r.t. $\mathcal{R}$). If no wild $\mathcal{A}$ exists, we could avoid requiring the user to specify maxJoin, and avoid using such an arbitrary restriction on Join applications.

A wild $\mathcal{A}$ automaton has to achieve two contrasting goals: making states joinable by equating their languages, and keep generating new states during normalization. The latter turns quite difficult if we use some heuristic to reuse states, since $\mathcal{A}$ has to *force* the generation of an unlimited number of states.

Quite interestingly, the existence of a wild $\mathcal{A}$ depends on the actual heuristics we are using. For instance, consider a dumb heuristic that uses a single fixed state for every normalization. In this case, we ignore maxStates, no state is ever generated, so there is no wild $\mathcal{A}$. On the other hand, consider the opposite heuristic: we never reuse states unless we exceeded maxStates. Now, take $\mathcal{A} = \{@a \to \mathsf{k}\}$ and $\mathcal{R} = \{\mathsf{k} \Rightarrow \mathsf{f}(\mathsf{k})\}$. At each main loop iteration i we perform the normalization, creating a new state $@q_i$ with a $@q_i \to \mathsf{k}$ transition. So, the state $@q_i$ is joined with $@q_0$ in the next loop, provided $\mathsf{maxStates} \geq 3$. Therefore, such an $\mathcal{A}$ is wild.

If more adequate heuristics are used, tackling the existence of a wild $\mathcal{A}$ is quite challenging. For the heuristics we used in our implementation presented in Sect. 3.9.3 we were nevertheless able to construct a wild $\mathcal{A}$. Below, we discuss how we constructed such an $\mathcal{A}$. Further, we only use left-linear rewriting rules for $\mathcal{R}$: no extended rules are used.

We consider the normalization of the following rule. We read c as "child" and p as "parent".

$$\mathsf{c}(X) \Rightarrow \mathsf{p}(\mathsf{c}(\mathsf{c}(X)))$$

Our heuristic will try to reuse the state carrying $\mathsf{c}(X)$, thus implementing the rule above as if it were

$$S : \mathsf{c}(X) \Rightarrow S : \mathsf{p}(\mathsf{c}(S))$$

So, we can rewrite

$$@b_0 \to \mathsf{p}(@b_1), \mathsf{root}$$
$$@b_1 \to \mathsf{c}(@b_0)$$

as follows

$$@b_0 \to \mathsf{p}(@b_1), \mathsf{root}$$
$$@b_1 \to \mathsf{c}(@b_0), \mathsf{p}(@b_2)$$
$$@b_2 \to \mathsf{c}(@b_1)$$

This will generate an unbounded number of $@b_i$. These states form a doubly-linked list, rooted at $@b_0$, with c and p providing the child and parent link, respectively, as shown below. The root constant term is used to make the languages non-empty, so that rewritings actually happen.

$$@b_0 \to \mathsf{p}(@b_1), \mathsf{root}$$
$$@b_1 \to \mathsf{c}(@b_0), \mathsf{p}(@b_2)$$
$$@b_2 \to \mathsf{c}(@b_1), \mathsf{p}(@b_3)$$
$$\ldots$$
$$@b_n \to \mathsf{c}(@b_{n-1}), \mathsf{p}(@b_{n+1})$$
$$@b_{n+1} \to \mathsf{c}(@b_n)$$

Now, we construct a superset $@everything$ of all the languages of the $@b_i$. Below, we use $\mathsf{d} = \mathsf{p}^{-1}$.

$$\mathsf{d}(\mathsf{p}(X)) \Rightarrow X$$
$$@everything \to @b_0, \mathsf{d}(@everything), \mathsf{extra}$$

This will trigger adding $@b_1$ to $@everything$, followed by $@b_2$ and so on. We now force all the $@b_n$ to join with $@everything$. However, we should be careful here, since if *all* the $@b_n$ are joined *at the same time*, the state generation stops. In other words, we need to join a $@b_n$ only if it already spawned its child. This is accomplished through the following rewriting rules.

$$\mathsf{id}(X) \Rightarrow X$$
$$\mathsf{p}(X) \Rightarrow \mathsf{id}(\mathsf{extra})$$

Of course, id is the identity function. The second rule adds the term id(extra) only to states that have a child. This will trigger a normalization, and heuristics will reuse for extra the only state that holds it: $@everything$. So,

we get

$$\ldots$$
$$@b_n \to \mathsf{c}(@b_{n-1}), \mathsf{p}(@b_{n+1}), \mathsf{id}(@everything)$$
$$\ldots$$
$$@everything \to \mathsf{extra}, \ldots, @b_n, \ldots$$

After one more rewriting, we have both the transitions $@b_n \to @everything$ and $@everything \to @b_n$. We then close under Transitivity, and then Join the states. Therefore, this automaton keeps on triggering new state generations as well as new joins, and so it is *wild*.

Above, we rely on the fact that the term extra occurs only in the transitions of $@everything$. Actually, when a $@b_n$ is about to be joined with $@everything$, it shares the same transitions, so the term extra occurs also in the transitions of $@b_n$. Because of this it is possible for another state $@b_m$ to be joined with $@b_n$ instead of $@everything$, if the normalization chooses $@b_n$ as the state approximating extra. However, note that, even if this happens, at this time there is nothing that can prevent $@b_n$ to be joined with $@everything$. So, the overall effect will be the same as joining both $@b_n$ and $@b_m$ with $@everything$. A similar argument can also be made for longer chains of states: it is possible that $@b_i$ is joined with $@b_j$ which is in turn joined with $@b_k$, and so on. However, all of them will eventually be merged with $@everything$.

Here is the wild $\mathcal{A}$ we defined above.

```
c(X)    => p(c(c(X))) .
p(X)    => id(extra) .
d(p(X)) => X .
id(X)   => X .

%%

@b0 : p(@b1) , root .
@b1 : c(@b0) .
@everything : @b0 , d(@everything) , extra.
```

In practice, however, the maxJoin bound seems to be mostly irrelevant. Usually, when a newly generated state is joined with a previous state, the language of previous state is augmented, leading to convergence. In fact, all our other tests converge even if we choose $\mathsf{maxJoin} = +\infty$.

3.9.6 Further Information

Our tool was implemented in Haskell [42], using the Glasgow Haskell Compiler (GHC) [39]. At the time of this writing, the development tree contains

2998 lines of source code. This figure includes the parser, the pretty printing routines, some sanity checks, and, for certain subroutines, an alternative version. The actual core of the tool is about 1500 lines of code. Our tool was released [61] under the terms of the GNU General Public License.

Just for a rough comparison, the Succinct Solver [65] version 2 is 8887 lines of Standard ML source code. The Timbuk tool for tree automata [67], version 2.2, is 16319 lines of OCaml code.

Chapter 4

Static Analysis for Security Protocols

Abstract

We present a static analysis technique for the verification of cryptographic protocols, specified in a process calculus. Rather than assuming a specific, fixed set of cryptographic primitives, we only require them to be specified through a term rewriting system, with no restrictions, as discussed in Chapter 3. Examples are provided to support our analysis. First, we tackle forward secrecy for a Diffie-Hellman-based protocol involving exponentiation, multiplication and inversion. Then, a simplified version of Kerberos is analyzed, showing that its use of timestamps succeeds in preventing replay attacks.

4.1 Introduction

Process calculi [52] have been extensively used for cryptographic protocol specification and verification, exploiting formal methods. In the introduction of this work, Sect. 1.1, we discussed how several of these calculi (e.g. Spi [4]) use a specific set of cryptographic primitives, which is often entwined with the definition of the process syntax and semantics, e.g. by introducing pattern matching on encrypted messages. We also addressed the main advantages this approach has. For instance, the presentation of the calculus is simplified; also, the verification tools only need to consider the given set of primitives. Instead, the main disadvantage lies in that protocols using different primitives cannot be specified in the calculus as it is: one has to suitably extend it and to adapt existing tools to cope with the extensions. Of course, the new tools also need new, adapted soundness proofs.

In Chapter 3 we saw how we can compute static over-approximations of set of terms modulo rewriting. Now, we can leverage these techniques in order to apply them to the verification of cryptographic protocols. Here, we shall not use a fixed set of cryptographic primitives. Rather, we shall assume that such primitives are specified through an an *arbitrary* rewriting system $\mathcal{R}$. Indeed, we do not put any restrictions on $\mathcal{R}$: it needs neither to be confluent nor terminating, since our algorithms of Chapter 3 do not require it to be so. This will allow us to add new cryptographic primitives just by adding the relevant rewriting rules to $\mathcal{R}$.

For the specification of protocols, we shall use a process calculus that allows processes to exchange arbitrary terms. To this purpose, the applied pi calculus [3] provides a solid ground. In the applied pi, terms are handled up to some equivalence relation, entirely defined by the user. In our case, this equivalence is simply the one given by the rewriting rules. Exploiting this equivalence, the definition of the calculus provide the semantics of processes exchanging terms. In this way, the semantics of the processes, which is fixed, is clearly separated from the semantics of the terms, which is user-defined.

For instance, adding new operations over terms does not affect the syntax of the processes. In Sect. 1.1 we discussed how this instead happens for those calculi that rely on pattern matching to manipulate terms.

In this chapter, we shall present a (slight) variant of the applied pi. Then, we shall address the problem of statically proving properties about processes. For this, we adapt the control flow analysis (CFA) approach [57, 16] to our calculus. Following this approach, we extract a number of constraints from a protocol specification, expressed as a process. Then, we solve the constraints using the algorithms on automata of Chapter 3, computing a sound approximation up to rewriting. The result is a tree automata $\mathcal{F}$ describing a language which is an over-approximation of the set of terms exchanged by the protocol, in *all* their possible equivalent forms according to $\mathcal{R}$. Finally, the automaton $\mathcal{F}$ can be inspected to check a number of security properties of the protocol. Essentially, this technique is a reachability analysis.

We then study an enhancement of the CFA, which we name CFA_{io}, to better exploit the features of the automata tools. In doing this, we revisit some choices made for the CFA, and derive a way to generate an alternative set of constraints. These constraints allow for a precise handling of the parallel composition of processes, replication and matching, which are features heavily exploited in the protocols specifications we consider. We compare the CFA to the CFA_{io}, considering the result of the analysis of some given processes by both analyses.

Exploiting the CFA_{io} technique, we analyzed protocols using both standard cryptographic primitives, such as encryptions and signatures, as well as more "problematic" primitives such as exponentials. Exponentials are hard to deal with because their equational theory has many equations, and therefore equivalent terms may assume very different shapes. As a consequence, it is difficult to find an accurate over-approximation for them. The literature often reports on studies carried out assuming only a few equations. For instance, from the web page of the AVISPA project [7] one sees examples with three equations for exp and inversion $(\bullet)^{-1}$ over finite-fields, analyzed through the TA4SP tool in [19]. In the example of Sect. 4.3.1, we consider exp, $\times$, 1, and $(\bullet)^{-1}$, axiomatizing their interactions with twelve equations. Yet, our implementation [61] of the presented analysis was able to prove the *forward secrecy* property for a protocol based on the Diffie-Hellman key exchange [31].

We also study the Wide-Mouthed Frog protocol [22], establishing a simple secrecy property. This protocol performs a key exchange using symmetric encryption, only. Because of this, the verification of this protocol is rather simple. Yet, it is comforting that our technique can handle this as well.

Our technique also offers a limited treatment of time. Here we report on the success of our tool for the verification of a simplified version of the Kerberos protocol [64, 56], involving timestamps. In our specification, we allow the disclosure of an old session key, mimicking a secret leak. The

tool was able to prove the secrecy of messages exchanged in newer sessions, confirming the protocol is resilient to replay attacks with the compromised old key.

Finally, our technique allows for some composition of results. Albeit with some limitations, it is possible to analyze the components of a system independently, and then merge the results later to derive a sound analysis for the whole system.

A related approach to ours is by Blanchet, Abadi and Fournet in [13]. However, they only consider certain equational theories, e.g. without associativity, and define a semi-algorithm to obtain rewriting rules with "partial normal forms." They then use ProVerif to check processes equivalent, thus establishing security properties.

We also mention the work in [2], also by Abadi and Blanchet. Here typing and logic programming is used to derive security properties. The results are mostly about theories of constructors and destructors. The authors hint at handling more general equational theories such as XOR or exponentials, leaving this for future work.

Also, some decidability results for (a significant fragment of) the exponential theory are discussed by Millen and Shmatikov in [51]. Another application of tree automata to security by Genet et al. can be found in [37]. There, protocols are specified through rewriting, rather than process calculi. A similar approach is in [36], by Genet and Klay. Another interesting work is by Goubault [41], dealing with exponentials through rewriting. There, however, only exponentials with a fixed base are considered. Monniaux in [55] also uses tree automata for verifying protocols, when crypto primitives can be expressed through left-linear rewritings.

Finally, there is an earlier analysis for the applied pi calculus in [71], developed by us. However, it only applies to free terms, subject to no rewriting.

Summary We present our calculus in Sect. 4.2, defining its dynamic semantics in Sect. 4.3. The same section has the Diffie-Hellman example. Sect. 4.4 describes the static analysis and its application to Diffie-Hellman, Wide-Mouthed Frog and Kerberos. In Sect. 4.6 we discuss compositionality.

4.2 Syntax

Our process calculus is a simplified version of the applied pi calculus [3], in that processes exchange values using a global public network channel. Reusing the definitions given in Chapter 3, values are simply represented as terms, up to the equivalence specified by a rewriting system $\mathcal{R}$. We write $\mathcal{T}_{\mathsf{pr}}$ for the set of pure terms.

We also use $\mathcal{V} = \{x, y, \ldots\}$ as the set of variables used by our processes. The syntax of our calculus is rather standard. Below, $M \in \mathcal{T}_{\mathsf{pr}}$.

$$\begin{array}{lcl} \pi & ::= & \mathsf{in}\ x \ \mid \mathsf{out}\ M \ \mid [x = y] \mid \mathsf{let}\ x = M \ \mid \mathsf{new}\ x \ \mid \mathsf{repl} \mid \mathsf{chk} \\ P & ::= & \mathsf{nil} \mid \pi\ .\ P \mid (P|P) \end{array}$$

We now briefly describe our calculus: its semantics will be given in Sect. 4.3. Intuitively, nil is a process that performs no actions; $\pi.P$ executes the prefix π and then behaves as P; $P_1|P_2$ runs concurrently the processes P_1 and P_2. Prefixes perform the following actions: $\mathsf{in}\ x$ reads a term from the network and binds x to it; $\mathsf{out}\ M$ sends a term to the network; $[x = y]$ compares the terms bound to x and y and stops the process if they differ; $\mathsf{let}\ x = M$ simply locally binds x to the value of M; $\mathsf{new}\ x$ generates a fresh value and binds x to it; repl spawns an unlimited number of copies of the running process, which will run independently; chk is a special action that we use to model certain kinds of attacks, which we shall address in Sect. 4.3.

Note that match $[x = y]$ is only allowed between variables. This is actually not a restriction, since matching between arbitrary terms, e.g. $[M = N].P$ can be expressed by $\mathsf{let}\ x = M\ .\mathsf{let}\ y = N\ .[x = y].P$.

Unlike the applied pi calculus, our calculus uses a single global public channel for sending and receiving data. Our previous calculus of Sect. 2.2 also used a single channel. Indeed, the motivations underlying this choice are the same. First, this keeps our static and dynamic semantics simple. Further, it models the fact that the adversary is able to reroute messages between different public channels, so these channels actually behave like a single one.

As usual, the *bound* variables in a process are those under a let,new, or in prefix; the others are *free*. A process with no free variables is *closed*.

Given a process, we use addresses $\theta \in \{\mathsf{n}, \mathsf{l}, \mathsf{r}\}^*$ to point to its subprocesses. Intuitively, n chooses the continuation P for a process $\pi.P$, while l and r choose the left and right branch of a parallel $P_1|P_2$, respectively. An address θ is a concatenation of these selectors, singling out the subprocess $P@\theta$ as defined below. We write ϵ for the empty string, that is the address pointing at the top-level process.

$$\begin{array}{lclclcl} P@\epsilon & = & P & \qquad & (P_1|P_2)@\theta\mathsf{l} & = & P_1@\theta \\ \pi.P@\theta\mathsf{n} & = & P@\theta & & (P_1|P_2)@\theta\mathsf{r} & = & P_2@\theta \end{array}$$

The *size* of a process P is the number of prefixes and parallel operators that occur in P.

4.3 Dynamic Semantics

Given a closed process P, we define its semantics through a multiset rewriting system [49, 26]. A state is a multiset σ of parallel threads. Each thread

is formed by an environment $\rho \in \mathcal{V} \to \mathcal{T}_{\mathsf{pr}}$ and a continuation address θ singling out a subprocess of P. We write such a thread as $\langle \rho, \theta \rangle$. Intuitively, $\langle \rho, \theta \rangle$ runs the process $P@\theta$ under the bindings in ρ. The initial state is $\langle \emptyset, \epsilon \rangle$.

We extend ρ homomorphically to terms: $\rho(M)$ replaces variables in M with the value they are bound to in ρ. We denote updates to the environment with $\rho[x \mapsto M]$: the updated environment binds x to M, and binds any other variable y to $\rho(y)$. Also, as a handy convention, if $P@\theta = P_1|P_2$, we write $\langle \rho, \theta \rangle$ for the multiset $\{\langle \rho, \mathsf{l}\theta \rangle, \langle \rho, \mathsf{r}\theta \rangle\}$, or its further expansion, so that threads in the state never have continuation addresses θ' such that $P@\theta'$ has the form $P_1|P_2$.

Our semantics is given by the rules in Fig. 4.1. Local rules only care about one or two elements of the current state: these elements are rewritten independently of the rest of the state, which does not change. All the rules fire a prefix, advancing the current continuation address θ to $\mathsf{n}\theta$, except for rule Rew.

Local Rules

$$\textsf{Comm}\ \frac{P@\theta_1 = \mathsf{in}\ x\ .P' \quad P@\theta_2 = \mathsf{out}\ M\ .P'' \quad \rho_1' = \rho_1[x \mapsto \rho_2(M)]}{\langle \rho_1, \theta_1 \rangle, \langle \rho_2, \theta_2 \rangle \xrightarrow{\mathsf{comm}\ \theta_1, \theta_2, \rho_2(M)} \langle \rho_1', \mathsf{n}\theta_1 \rangle, \langle \rho_2, \mathsf{n}\theta_2 \rangle}$$

$$\textsf{Out}\ \frac{P@\theta = \mathsf{out}\ M\ .P'}{\langle \rho, \theta \rangle \xrightarrow{\mathsf{out}\ \theta, \rho(M)} \langle \rho, \mathsf{n}\theta \rangle}$$

$$\textsf{Match}\ \frac{P@\theta = [x = y].P' \quad \rho(x) = \rho(y)}{\langle \rho, \theta \rangle \xrightarrow{\tau} \langle \rho, \mathsf{n}\theta \rangle} \qquad \textsf{Let}\ \frac{P@\theta = \mathsf{let}\ x = M\ .P'}{\langle \rho, \theta \rangle \xrightarrow{\tau} \langle \rho[x \mapsto \rho(M)], \mathsf{n}\theta \rangle}$$

$$\textsf{Repl}\ \frac{P@\theta = \mathsf{repl}.P'}{\langle \rho, \theta \rangle \xrightarrow{\tau} \langle \rho, \theta \rangle, \langle \rho, \mathsf{n}\theta \rangle} \qquad \textsf{Rew}\ \frac{\rho(x) \to_{\mathcal{R}} M}{\langle \rho, \theta \rangle \xrightarrow{\tau} \langle \rho[x \mapsto M], \theta \rangle}$$

Global Rules

$$\textsf{New}\ \frac{P@\theta = \mathsf{new}\ x\ .P' \quad \hat{x} = \mathsf{genFresh}()}{\sigma, \langle \rho, \theta \rangle \xrightarrow{\tau} \sigma, \langle \rho[x \mapsto \hat{x}], \mathsf{n}\theta \rangle} \qquad \textsf{Chk}\ \frac{P@\theta = \mathsf{chk}.P'}{\sigma, \langle \rho, \theta \rangle \xrightarrow{\mathsf{chk}} \langle \rho, \mathsf{n}\theta \rangle}$$

Figure 4.1: Multiset Rewriting Rules

Rule Comm performs communication between threads. Rule Out outputs a term to the external environment. Since out M may be handled by either Comm or Out, there is no guarantee that outputs have a corresponding input; instead, they may simply cause a *barb*, i.e. an action observed only by the external environment. Note that there is no rule for input, and therefore processes can never receive a value from the environment – for studying security issues our processes will explicitly contain an adversary.

Rule Match, allows a process to continue only if x and y are bound to the same term. Rule Let simply updates ρ with the new binding. Rule Repl allows for spawning a new copy of P'. In Rew, the thread rewrites the term bound by x, thus performing an internal computation step; note that these internal steps may lead a matching to succeed.

Global rules instead look at the whole state. Rule New is not completely standard, and it generates a *fresh* value $\hat{x}$ for the variable x. Here we postulate that 1) a constant (nullary function) symbol $\bar{x}$ exists for each variable bound by new in P, and 2) two distinguished function symbols val, next exist, subject to no rewriting in $\mathcal{R}$. Note that we only need a finite number of such $\bar{x}$, since there are only finitely many variables in P and we have *no* α-conversion. When rule New is applied, the $\hat{x}$ is generated by a genFresh() primitive, which we assume to choose among $\mathsf{val}(\bar{x}), \mathsf{val}(\mathsf{next}(\bar{x}))$, $\mathsf{val}(\mathsf{next}(\mathsf{next}(\bar{x}))), \ldots$. To make it possible to track new-generated values to their new prefix in P, we require that all new-bound variables are distinct, and therefore so are their related constants $\bar{x}$. Note that this representation prevents an adversary Adv to deduce any instance of $\hat{x}$ from other instances he knows, even if Adv can use val and next.

Rule Chk is peculiar: when a chk prefix is fired *all* the other threads are aborted, and the thread continues its execution alone. For simplicity, we admit only one firing of chk. We use this special prefix to model some kinds of attacks. For instance, suppose we want to study the case in which the adversary learns some secret term S, maybe by corrupting some participant to the protocol. A straightforward way to model this attack would be simply adding out S to the protocol, disclosing S. While this would work, in many cases giving this kind of power to the adversary might allow for trivial attacks. Instead, to keep the game fair, we could restrict the interaction between the adversary and the participants after the disclosure of S. For example, we could imagine that it would take a long time for the adversary to obtain S, and meanwhile the participants have terminated the protocol run, either normally or because of a time-out.

A possible usage of chk is the following. The adversary, after having learnt S, is only allowed to run alone, and possibly use this new knowledge to decrypt messages it learnt in the past. In our calculus, we model this scenario as

$$(Proto|Adv)|\mathsf{in}\ know\ .\mathsf{chk}.(\mathsf{out}\ know\ .\mathsf{out}\ S\ .\mathsf{nil}|Adv)$$

Note that we include the adversary process twice. First, the adversary can interact with the protocol. Later, when chk is fired, the adversary can learn S and go on with its computation, without being able to communicate with the protocol participants. Since we want to allow the adversary to keep its knowledge across the chk firing, we simply save this knowledge in the variable *know* before the chk, and make it again available to the adversary

later on. Note that, while *know* is only a single term, it can be a cons-list of all the terms known by the adversary. Therefore *know* actually can bring all the old adversary knowledge into the new world, provided we have a primitive for pairing. In the next section, we show such a use of chk.

Another interesting use of chk we found is for modeling *timestamps*, as we shall show in Sect. 4.5.4. Finally, in Sect. 4.7 we discuss an alternative semantics for chk.

A Brief Comparison to the Calculus of Chapter 2

The calculus we use in this chapter is different from the one used in Chapter 2. The main differences can be summarized as follows.

- In the calculus of Chapter 2, the adversary is fixed and it is defined in the semantics. Here, the adversary is a process, a part of the system P at hand.
- In Chapter 2, the knowledge of the adversary was made explicit in the process state $(\mathcal{K}, \mathcal{T}, P)$. Instead, here this knowledge is not directly represented.
- In Chapter 2, processes used pattern matching for destructing pairs and encryptions. Here, we use rewriting rules.
- In Chapter 2 we considered guessing keys via the DYGuess operation.

Despite these differences, it is possible to define in the new calculus the same Dolev-Yao adversary of Chapter 2. recall that, by Theorem 1, we can neglect the DYGuess operation. First, we adopt the usual rewriting rules for pairs and encryptions.

$$\mathcal{R} = \left\{ \begin{array}{ll} \mathsf{dec}(\mathsf{enc}(X,K),K) & \Rightarrow X \\ \mathsf{fst}(\mathsf{cons}(X,Y)) & \Rightarrow X \\ \mathsf{snd}(\mathsf{cons}(X,Y)) & \Rightarrow Y \end{array} \right\}$$

Then we define the adversary as follows.

$$\begin{array}{rl} DY = & \mathsf{repl.in}\ x\ \mathsf{.in}\ y\ .(\\ & \mathsf{repl.out}\ 0\ \mathsf{.nil} \mid \mathsf{repl.out}\ 1\ \mathsf{.nil} \mid \\ & \mathsf{repl.out}\ x\ \mathsf{.nil} \mid \mathsf{repl.out}\ y\ \mathsf{.nil} \mid \\ & \mathsf{repl.out}\ \mathsf{cons}(x,y)\ \mathsf{.nil} \mid \mathsf{repl.out}\ \mathsf{fst}(x)\ \mathsf{.nil} \mid \mathsf{repl.out}\ \mathsf{snd}(x)\ \mathsf{.nil} \mid \\ & \mathsf{repl.out}\ \mathsf{enc}(x,y)\ \mathsf{.nil} \mid \mathsf{repl.out}\ \mathsf{dec}(x,y)\ \mathsf{.nil}) \end{array}$$

This process simply runs every operation non-deterministically in all the possible ways. We can now define the knowledge $\mathcal{K}$ of this DY adversary. Take $P = Proto|DY$, and any state σ reachable from P, i.e. such that $\langle \emptyset, \epsilon \rangle \rightarrow^* \sigma$. The knowledge of the adversary at σ is

$$\mathcal{K} = \{\rho(M) \mid \langle \rho, \theta \rangle \in \sigma \land P@\theta = \mathsf{repl.out}\ M\ \mathsf{.nil}\}$$

Above, we inspect σ for all active outputs under a replication, and we define the knowledge as the set of terms appearing in these outputs. Clearly, these are the terms constructed by the *DY* process above, or by *Proto*. In both cases they are available to the adversary, so they form its knowledge $\mathcal{K}$.

This formulation of $\mathcal{K}$ is strictly related to the one given in Chapter 2. Indeed, it is possible to see that the two adversaries can simulate each other, so that if one is able learn a term, so does the other. This holds for any term that the calculus of Chapter 2 is able to represent. The new calculus can also have unevaluated terms, e.g. $\mathsf{fst}(\mathsf{cons}(0,1))$, that have no correspondent in the other calculus (this also includes non-well-formed terms such as $\mathsf{fst}(0)$). Neglecting these terms, the two calculi define the same adversary knowledge $\mathcal{K}$, and therefore share the security properties that only depend on $\mathcal{K}$, such as secrecy.

4.3.1 Diffie-Hellman Example

We consider the following key-exchange protocol, based on Diffie-Hellman's [31].

1. $A \to \mathsf{all}$	:	g
2. $A \to B$	:	$\{\mathsf{g}^{\mathsf{a}}\}_{\mathsf{k1}}$
3. $B \to A$	:	$\{\mathsf{g}^{\mathsf{b}}\}_{\mathsf{k2}}$
4. $A \to B$	:	$\{\mathsf{m}\}_{\mathsf{g}^{\mathsf{ab}}}$
5. ... (time passes)		
6. $A \to \mathsf{all}$	:	$\mathsf{k1}, \mathsf{k2}$

Initially, the principals A and B share two long term secret keys $\mathsf{k1}, \mathsf{k2}$, and agree on a public finite field $\mathsf{GF}[p]$ (where p is a large prime), and public generator g of $\mathsf{GF}[p]^*$. In the second step, principal A generates a nonce a and sends B the result of $\mathsf{g}^{\mathsf{a}} (\bmod\ p)$, encrypted with the key $\mathsf{k1}$. In the third step, B does the same, with its own nonce b and key $\mathsf{k2}$. Since both principals know the long term keys, they can compute $(\mathsf{g}^{\mathsf{b}})^{\mathsf{a}} = \mathsf{g}^{\mathsf{ab}} = (\mathsf{g}^{\mathsf{a}})^{\mathsf{b}} \pmod p$ and use this value as a session key to exchange the message m in the fourth step.

We study the robustness of this protocol against the active Dolev–Yao [32] adversary. We discussed this kind of adversary in Chapter 2. Here, we briefly remind that such an adversary has full control over the public network, can reroute, discard or forge messages; further, he can apply any algebraic operation to terms learnt before. In particular, we are interested in the *forward secrecy* of the message m. That is, we want m to be kept secret even though later on the long term keys $\mathsf{k1}, \mathsf{k2}$ are disclosed (last step).

We define the algebra by adapting the rewriting rules for encryption,

multiplication, exponentiation, and inversion from [51]:

$$\mathcal{R} = \left\{ \begin{array}{ll} \mathsf{dec}(\mathsf{enc}(X,K),K) & \Rightarrow X \\ \mathsf{fst}(\mathsf{cons}(X,Y)) & \Rightarrow X \\ \mathsf{snd}(\mathsf{cons}(X,Y)) & \Rightarrow Y \\ *(1,X) & \Rightarrow X \\ *(X,Y) & \Rightarrow *(Y,X) \\ *(X,*(Y,Z)) & \Rightarrow *(*(X,Y),Z) \\ \mathsf{exp}(X,1) & \Rightarrow X \\ \mathsf{exp}(1,X) & \Rightarrow 1 \\ \mathsf{exp}(\mathsf{exp}(X,Y),Z) & \Rightarrow \mathsf{exp}(X,*(Y,Z)) \\ \mathsf{exp}(*(Y,Z),X) & \Rightarrow *(\mathsf{exp}(Y,X),\mathsf{exp}(Z,X)) \\ \mathsf{inv}(1) & \Rightarrow 1 \\ \mathsf{inv}(\mathsf{inv}(X)) & \Rightarrow X \\ *(X,\mathsf{inv}(X)) & \Rightarrow 1 \\ \mathsf{inv}(*(X,Y)) & \Rightarrow *(\mathsf{inv}(X),\mathsf{inv}(Y)) \\ \mathsf{exp}(\mathsf{inv}(X),Y) & \Rightarrow \mathsf{inv}(\mathsf{exp}(X,Y)) \end{array} \right\}$$

Note that the algebra $\mathcal{A}$ defined by the above rewriting rules is not the same algebra of $\mathsf{GF}[p]^*$. In fact, $\mathcal{A}$ satisfies more equations than the ones that hold in $\mathsf{GF}[p]^*$. For instance, operations in $\mathcal{A}$ do not specify which modulus is being used; e.g., inversion modulo n is simply written as $\mathsf{inv}(X)$ rather than $\mathsf{inv}(X,n)$. Therefore, we have *(a)* $*(X,\mathsf{inv}(X)) = 1$ and *(b)* $\mathsf{exp}(Y,*(X,\mathsf{inv}(X))) = Y$. However, *(a)* holds in $\mathsf{GF}[p]^*$ only if $\mathsf{inv}(X)$ is performed modulo p, while *(b)* holds only if $\mathsf{inv}(X)$ is performed modulo $\varphi(p) = p-1$ (where φ is the Euler function). In spite of $\mathcal{A}$ being not equal to the $\mathsf{GF}[p]^*$ algebra, we believe it is a rather faithful approximation, satisfying most of the the equations that hold in $\mathsf{GF}[p]^*$.

We use the following process:

$$\begin{array}{rl} P = & \mathsf{new}\ g\ .(DY|\mathsf{new}\ k1\ .\mathsf{new}\ k2\ .(Proto|Chk)) \\ Chk = & \mathsf{in}\ know\ .\mathsf{chk}.(\mathsf{out}\ know\ .\mathsf{out}\ k1\ .\mathsf{out}\ k2\ .\mathsf{nil}|DY)) \\ Proto = & A|B \\ A = & \mathsf{repl.new}\ a\ .\mathsf{out}\ \mathsf{enc}(\mathsf{exp}(g,a),k1)\ .\mathsf{in}\ x\ . \\ & \mathsf{let}\ k = \mathsf{exp}(\mathsf{dec}(x,k2),a)\ .\mathsf{out}\ \mathsf{enc}(\mathsf{m},k)\ .\mathsf{nil} \\ B = & \mathsf{repl.new}\ b\ .\mathsf{in}\ x\ .\mathsf{out}\ \mathsf{enc}(\mathsf{exp}(g,b),k2)\ . \\ & \mathsf{let}\ k = \mathsf{exp}(\mathsf{dec}(x,k1),b)\ .\mathsf{in}\ n\ .\mathsf{out}\ \mathsf{hash}(\mathsf{dec}(n,k))\ .\mathsf{nil} \\ DY = & \mathsf{repl}.((\mathsf{new}\ nonce\ .\mathsf{out}\ nonce\ .\mathsf{nil}|\mathsf{out}\ g\ .\mathsf{out}\ 1\ .\mathsf{nil})| \\ & \mathsf{in}\ x\ .\mathsf{in}\ y\ .\mathsf{out}\ x\ .\mathsf{out}\ y\ .\mathsf{out}\ \mathsf{enc}(x,y)\ .\mathsf{out}\ \mathsf{dec}(x,y)\ . \\ & \mathsf{out}\ \mathsf{exp}(x,y)\ .\mathsf{out}\ *(x,y)\ .\mathsf{out}\ \mathsf{inv}(x)\ . \\ & \mathsf{out}\ \mathsf{cons}(x,y)\ .\mathsf{out}\ \mathsf{fst}(x)\ .\mathsf{out}\ \mathsf{snd}(x)\ . \\ & \mathsf{out}\ \mathsf{hash}(x)\ .\mathsf{out}\ \mathsf{val}(x)\ .\mathsf{out}\ \mathsf{next}(x)\ .\mathsf{nil}) \end{array}$$

The specification P combines the protocol participants with the DY adversary. The principal B outputs the hash of the exchanged message, just as a witness. We also add a Chk process for the explicit disclosure of the secret keys: of course, this only happens after the chk prefix is fired.

We expect P to ensure the secrecy of the message m. Further, we expect this secrecy property to still hold even after the chk fires and thus the long term keys $k1, k2$ are disclosed. This is the *forward secrecy* property we want to prove through static analysis.

4.4 Static Semantics

We now present two techniques for the static analysis of processes. The first one is the *control flow analysis* (CFA) [57], which we adapted to our calculus. The second one, CFA_{io}, is a refinement of the first one, and exploits our completion algorithm to often provide a better approximation for parallel composition and matching.

4.4.1 Control Flow Analysis

In this section, we adapt the *control flow analysis* (CFA) [57] to our language. The adaptation we use is however mild, in that it is still very faithful to the original CFA. Because of this, we do not introduce another process calculus just to present the CFA of [57] over it, but rather we directly present its (slight) adaptation to the calculus we have used throughout this chapter. Doing this, we shall *not* require the reader to be familiar with the original CFA. Rather, we shall assume no previous knowledge on the subject, and present the analysis from scratch. At the end of this section we will comment on some minor differences due to our adaptation. From now on, we will refer to the analysis defined below in this section simply by “CFA”.

The goal of the CFA is to over-approximate the values that processes exchange at run-time. In other words, we want to compute (a finite representation for) a superset of all the terms that flow through the network. In order to compute this approximation, the CFA first generates from a given process P a set of *constraints* over sets of values. Then, these constraints are solved. We represent these sets of values as the languages associated with the states of a finite tree automaton. In this way, the approximation of the network flow will simply be the language of a single state @net. Some of the constraints extracted from P are expressed as transitions forming an automaton $\mathcal{A}$. For instance, generating the (to be normalized) transition $\texttt{@z} \rightarrow \mathsf{f}(\mathsf{g}(\texttt{@x}), \texttt{@y})$ put a constraint on the languages of the automata, in that we have $\{\mathsf{f}(\mathsf{g}(x), y) | x \in [\![\texttt{@x}]\!]_{\mathcal{A}} \wedge y \in [\![\texttt{@y}]\!]_{\mathcal{A}}\} \subseteq [\![\texttt{@z}]\!]_{\mathcal{A}}$. The other constraints have the form $\texttt{@x} \simeq \texttt{@y} \Rightarrow \texttt{@a} \rightarrow \texttt{@b}$, and exploit the extended rules of Sect. 3.9.2. Of course, we also require our sets of values be closed under

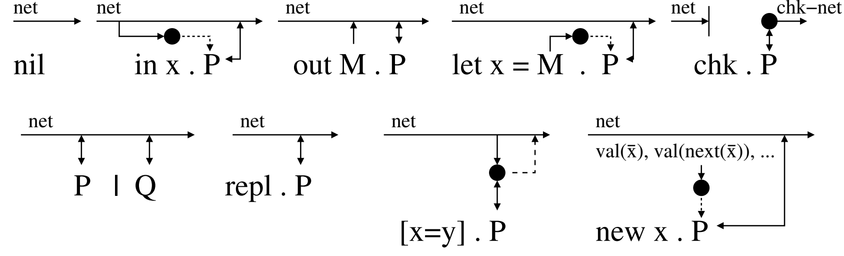


Figure 4.2: Schematic CFA Data Flow

rewritings in a rule set $\mathcal{R}$, defining the behaviour of the protocol primitives. For the CFA we use no intersection constraint, so $\mathcal{I} = \emptyset$.

After we generate these constraints, obtaining $\mathcal{A}$ and $\mathcal{R}$, we run the completion tool and compute an automaton $\mathcal{F}$ such that its languages include those of $\mathcal{A}$ and are closed under rewritings in $\mathcal{R}$. Once done that, we can check a number of properties about P by simply inspecting $\mathcal{F}$.

We first give some intuition behind the construction of $\mathcal{A}$ and $\mathcal{R}$. Roughly, we follow the data-flow between processes depicted in Fig. 4.2. In the figure, the double arrows towards/from processes represent communication, while bullets represent the data-flow points which we focus on in our analysis. For each bullet, we compute an approximation for the set of values that flow through it.

More in detail, we generate a dedicated state of $\mathcal{A}$ for each bullet, and add transitions between states following the arrows in the figure. Formally, the states of $\mathcal{A}$ are:

- `@net`, `@chk-net`;
- `@net-`θ, for each θ such that $P@\theta = [x = y].P'$;
- `@x` and `@x-val`, for each `new` x occurring in P;
- `@x`, for each `let` $x = M$ and `in` x occurring in P.

We write $\zeta : \mathcal{T}_{\mathsf{gr}} \to \mathcal{Q}$ for the homomorphic extension of $\zeta(x) =$ `@x`. Given this ζ, the generation of constraints is done through the function *gen* as defined in Fig. 4.3. The expression $gen(\theta, P', net)$ generates the constraints for $P' = P@\theta$, a subprocess of P[1]. The *net* parameter defines the state for the approximation of the values that can be received and sent by P'. Initially, *gen* is called as $gen(\epsilon, P,$ `@net`$)$.

No transitions are generated for `nil`. For `in` $x\ .P'$, we generate a transition from `@x` to *net* to include inputs in its language. Then, we proceed

[1]The parameter P' of *gen* is actually redundant since it is determined by θ, but its presence allows for a simple definition.

$$gen(\theta, \mathsf{nil}, net) = \emptyset$$
$$gen(\theta, \mathsf{in}\ x\ .P, net) = (\texttt{@x} \to net), gen(\mathsf{n}\theta, P, net)$$
$$gen(\theta, \mathsf{out}\ M\ .P, net) = (net \to \zeta(M)), gen(\mathsf{n}\theta, P, net)$$
$$gen(\theta, (P|Q), net) = gen(\mathsf{l}\theta, P, net), gen(\mathsf{r}\theta, Q, net)$$
$$gen(\theta, \mathsf{repl}.P, net) = gen(\mathsf{n}\theta, P, net)$$
$$gen(\theta, \mathsf{let}\ x = M\ .P, net) = (\texttt{@x} \to \zeta(M)), gen(\mathsf{n}\theta, P, net)$$
$$gen(\theta, \mathsf{new}\ x\ .P, net) = (\texttt{@x} \to \mathsf{val}(\texttt{@x-val})), (\texttt{@x-val} \to \bar{x}), (\texttt{@x-val} \to \mathsf{next}(\texttt{@x-val})), gen(\mathsf{n}\theta, P, net)$$
$$gen(\theta, [x = y].P, net) = (\texttt{@x} \simeq \texttt{@y} \Rightarrow net \to \texttt{@net-}\theta), (\texttt{@net-}\theta \to net), gen(\mathsf{n}\theta, P, \texttt{@net-}\theta)$$
$$gen(\theta, \mathsf{chk}.P, net) = gen(\mathsf{n}\theta, P, \texttt{@chk-net})$$

Figure 4.3: Extraction of CFA constraints from a process.

recursively with the continuation P'. Outputs as $\mathsf{out}\ M\ .P'$ generate a transition from the net state to $\zeta(M)$; we then proceed recursively for P'. For example, the generated transitions for $P = \mathsf{in}\ x\ .\mathsf{out}\ \mathsf{f}(x)\ .\mathsf{out}\ \mathsf{g}(x)\ .\mathsf{nil}$ are $\texttt{@x} \to \texttt{@net}, \texttt{@net} \to \mathsf{f}(\texttt{@x}), \texttt{@net} \to \mathsf{g}(\texttt{@x})$.

Parallel processes such as $P|Q$ are handled independently: both have access to net. Note that processes such as $P|P$ generate twice the constraints for P. For this reason, the result of our CFA for $P|P$ is equal to the one for P alone: our CFA does not take into account multiple copies of the same process. This concept is reinforced by the replication case: the constraints for $\mathsf{repl}.P$ are, by definition, those for P.

For let bindings, we simply create a new state for the approximation of the bound value. A $\mathsf{new}\ x\ .P'$ causes the generation of transitions for the language $\mathsf{val}(\bar{x}), \mathsf{val}(\mathsf{next}(\bar{x})), \mathsf{val}(\mathsf{next}(\mathsf{next}(\bar{x}))), \ldots$ using the two states `@x` and `@x-val`.

A match $[x = y].P'$ generates the constraints for P' using an alternative state for net, i.e. $\texttt{@net-}\theta$. Then, we put a simple constraint: if there is some common term to `@x` and `@y`, we join $\texttt{@net-}\theta$ with the previous net through a transition. This transition is shown as a dashed line in Fig. 4.3. If instead the states `@x` and `@y` have disjoint languages, that means that the match will be never satisfied at run-time, so we do not join the two nets. Note that this is quite a drastic approach: $[x = y].P'$ is handled as nil or as P' depending on whether we can statically decide that the match will be never passed through, i.e. it is dead code. In other words, when we meet non-dead code, we simply ignore the match. A better alternative would be to approximate the intersection between the languages of `@x` and `@y`: we shall return to this point in Sect. 4.4.3.

When a chk is fired, the continuation runs in an isolated world, therefore in the analysis we simply reset *net* to a new independent state @chk-net and proceed recursively.

Example Consider the following process.

$$P = \mathsf{out}\ 0\ .\mathsf{nil} \mid \ \mathsf{in}\ x\ .\mathsf{out}\ \mathsf{f}(x)\ .\mathsf{out}\ 1\ .\mathsf{nil}$$

Following the definition of $gen()$, we generate the following constraints, forming an automata:

$$\texttt{@net} \to 0$$
$$\texttt{@x} \to \texttt{@net}$$
$$\texttt{@net} \to \mathsf{f}(\texttt{@x})$$
$$\texttt{@net} \to 1$$

Therefore, the language of @net includes $0, 1$ and is closed under f. Note that this is a proper *over*-approximation of the behaviour of P. In fact, according to the dynamic semantics, P can only output $0, \mathsf{f}(0)$ and 1, while our static semantics also includes $\mathsf{f}(1), \mathsf{f}(\mathsf{f}(0)), \ldots$.

We will provide further examples in Sect. 4.4.5.

Time Complexity We now study the performance of the CFA analysis of a process P, with respect to the size of P.

Running $gen()$ for P takes linear time: the definition of $gen()$ closely follows the syntax of P so that it never consider the same subprocess twice. This also implies that the generated automaton $\mathcal{A}$ has a linear number of transitions. Further, the rules added to $\mathcal{R}$ through $gen()$ are also bounded linearly.

So, the overall worst-case complexity bound is that of Theorem 7 in Chapter 3.

$$\mathcal{O}(n^k \cdot (\log s + \mathsf{maxArity} \log n) \cdot s^l \cdot c \cdot (|\mathcal{I}| + |\mathcal{R}|))$$

Here, variable n represents 1) the number of states of $\mathcal{A}$, plus 2) any additional states that can be generated by normalization, plus 3) the number of joined states. We saw that term 1) grows linearly with (the size of) P. Terms 2) and 3) are chosen by the user, depending on how much resources the user wants to provide for the analysis.

The size of $\mathcal{R}$, written above as $|\mathcal{R}|$, accounts both for the rewriting rules given in the specification of the protocol and the generated rules, the latter being linear in P. Note, however, that the bound above was established considering only regular rewriting rules, and our analysis generates some extended rules, discussed in Sect. 3.9.2. In spite of this, the above bound

still holds. In fact, our rules use only left hand sides of the form $\texttt{@x} \simeq \texttt{@y}$, and that is checked through a simple query to $\simeq$, the cost of which ($\mathcal{O}(\log n)$) is less than the cost of matching the left hand side of a regular rewriting rule (greater than n).

The size of $\mathcal{I}$ is zero, as we use no intersection constraints.

The other variables occurring in the bound above depend on the user-specified $\mathcal{R}$, i.e. on the actual primitives that the protocol employs. For a detailed discussion we refer to Theorem 7. Here, we simply stress that CFA runs in polynomial time w.r.t. the size of the process. Further, the degree of this polynomial depends only on the algebra of terms modulo rewriting.

A Note on Our Adaptation We briefly summarize here the differences between our adaptation of the CFA and the original CFA in [57].

First, we consider terms up to rewriting, while the original CFA assumed a fixed set of operations, and relied on a free algebra. For destruction of terms, pattern matching is used in the calculus of the original CFA. Both analyses use a representation based on tree languages for their static approximations. The actual constraint solver used for the original CFA is the Succinct Solver [65], allowing constraints to be expressed in a fragment of first order logic. Instead, we use the approximation algorithms of Chapter 3, and write most constraints as transitions of an automaton.

The process calculus used for the original CFA allows the use of multiple (private) channels, as the π-calculus. Our calculus, as explained in Sect. 4.2 only uses one public channel. Of course, we could adapt our CFA for multiple channels, e.g. by encoding the multiple-channels calculus into our single-channel one. However, recent works [18, 14], using an enhanced version the original CFA, seem to confirm that even using only one single channel one can still study many real-world protocols.

The original CFA has some further dead code detection mechanism, which is not to be found in our CFA. The main reason behind this is that it does not seem to be very useful in practice, since protocol specifications rarely contain dead code. We shall return to this point in Sect. 4.4.5.

4.4.2 Subject Reduction for CFA

We now establish the soundness for our CFA. The following theorem ensures that the result of the static analysis actually encompasses the dynamic behaviour. More precisely, all the outputs fired before chk have been statically foreseen in the language $[\![\texttt{@net}]\!]_{\mathcal{F}}$. Similarly, all the outputs after the chk have been predicted by $[\![\texttt{@chk-net}]\!]_{\mathcal{F}}$.

Theorem 8 (Subject Reduction) *Given P, let $\mathcal{F}$ be the automata resulting from the control flow analysis. Assume $\langle\emptyset, \epsilon\rangle \rightarrow^* \xrightarrow{\alpha} \sigma, \langle\rho, \theta\rangle$.*

1. $\forall x \in \mathsf{dom}(\rho).\ \rho(x) \in [\texttt{@}x]_{\mathcal{F}}$

2. *if* $\alpha = (\mathsf{out}\ \theta\ , M)$ *and* chk *was not fired before* α*, then* $M \in [\texttt{@net}]_{\mathcal{F}}$

3. *if* $\alpha = (\mathsf{out}\ \theta\ , M)$ *and* chk *was fired before* α*, then* $M \in [\texttt{@chk-net}]_{\mathcal{F}}$

Proof. By induction on the number of computation steps, and by case analysis on the last step. First, we consider property (1): for this, we only need to check the rules that update the environment ρ.

Assume the Comm rule is applied to threads $\langle \theta_1, \rho_1 \rangle$ and $\langle \theta_2, \rho_2 \rangle$, yielding to comm $\theta_1, \theta_2, \rho_2(M)$. We look for the transitions driven by in x and out M , as generated by $gen()$. These transitions have the form $\texttt{@x} \rightarrow net_1$ and $net_2 \rightarrow \zeta(M)$. By inductive hypothesis and structural induction on M, we have $\rho_2(M) \in [\![\zeta(M)]\!]_{\mathcal{F}} \subseteq [\![net_2]\!]_{\mathcal{F}}$.

Now, consider all the matches $[w = z]$ we went through to reach $P@\theta_1$ and $P@\theta_2$, i.e. the matches between addresses ϵ and θ_i, for $i \in \{1, 2\}$. When these matches had been crossed, we had $\rho'(w) = \rho'(z)$, where ρ' is the environment at the time of the match. By inductive hypothesis, we have $\rho'(w) \in [\![\texttt{@w}]\!]_{\mathcal{F}}$ and $\rho'(z) \in [\![\texttt{@z}]\!]_{\mathcal{F}}$, and therefore $[\![\texttt{@w}]\!]_{\mathcal{F}} \cap [\![\texttt{@z}]\!]_{\mathcal{F}} \neq \emptyset$. Since we generated the constraints $\texttt{@w} \simeq \texttt{@z} \Rightarrow net \rightarrow net_i$ and $net_i \rightarrow net$, we can state $[\![net_i]\!]_{\mathcal{F}} = [\![net]\!]_{\mathcal{F}}$. If chk has not been fired, we can apply this argument to every match, obtaining $[\![net_i]\!]_{\mathcal{F}} = [\![\texttt{@net}]\!]_{\mathcal{F}}$. If chk was fired, the induction above stops at the chk prefix, i.e. it holds for processes between addresses θ_{chk} and θ_i. In that case, we have instead $[\![net_i]\!]_{\mathcal{F}} = [\![\texttt{@chk-net}]\!]_{\mathcal{F}}$. However, regardless of the firing of chk, we get $[\![net_1]\!]_{\mathcal{F}} = [\![net_2]\!]_{\mathcal{F}}$. This shows that $\rho_2(M) \in [\![net_2]\!]_{\mathcal{F}} = [\![net_1]\!]_{\mathcal{F}} \subseteq [\![\texttt{@x}]\!]_{\mathcal{F}}$, and so property (1) is preserved by Comm.

We now tackle property (1) for the other rules. The Let case is straightforward: we generated the transition $\texttt{@x} \rightarrow \zeta(M)$, so, by structural induction on M, we have $\rho(M) \in [\![\texttt{@x}]\!]_{\mathcal{F}}$. Rule New also poses no problem, because the fresh term returned by genFresh() is chosen among the terms in the language of @x. Finally, environment updates by rule Rew are harmless, being the languages of $\mathcal{F}$ closed under rewritings.

For properties (2,3), only rule Out may cause out $\theta, \rho(M)$. For the output process, we generated the constraint $net \rightarrow \zeta(M)$. As we did for property (1), we consider all the matches $[w = z]$ the process went through. As before, we can infer that the premises of constraints $\texttt{@w} \simeq \texttt{@z} \Rightarrow net' \rightarrow net''$ are satisfied. Therefore, we can track net back to @net (or to @chk-net, if chk was fired), and have $[\![net]\!]_{\mathcal{F}} = [\![\texttt{@net}]\!]_{\mathcal{F}}$ (resp. $[\![net]\!]_{\mathcal{F}} = [\![\texttt{@chk-net}]\!]_{\mathcal{F}}$). By structural induction on M, we also have that $\rho(M) \in [\![\zeta(M)]\!]_{\mathcal{F}}$. Therefore, we can conclude $\rho(M) \in [\![\zeta(M)]\!]_{\mathcal{F}} \subseteq [\![net]\!]_{\mathcal{F}} = [\![\texttt{@net}]\!]_{\mathcal{F}}$ (resp $\rho(M) \in [\![\texttt{@chk-net}]\!]_{\mathcal{F}}$). □

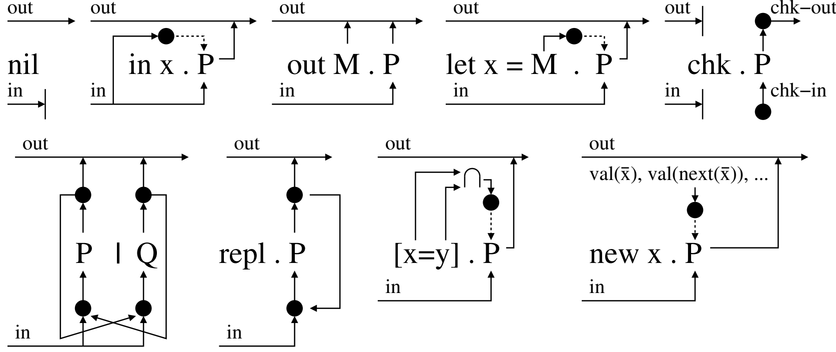


Figure 4.4: Schematic $\mathrm{CFA}_{\mathsf{io}}$ Data Flow

4.4.3 A Refined CFA Analysis: $\mathrm{CFA}_{\mathsf{io}}$

We now refine the CFA analysis to better exploit the features of our completion algorithm. As for the CFA, we generate constraints from a given process P, and solve them to over-approximate the values that processes exchange at run-time. We name this analysis $\mathrm{CFA}_{\mathsf{io}}$, since we shall separate the analysis of inputs from that of outputs. We shall also provide a more careful handling of parallel composition and matching.

During the constraints generation, we produce the transitions of an automata $\mathcal{A}$, together with a set of intersection constraints $\mathcal{I}$, with typical element $\texttt{@a} \cap \texttt{@b} \subseteq \texttt{@c}$. Of course, we shall also require our sets of values be closed under rewritings in $\mathcal{R}$. Note that, unlike for CFA, we shall not generate any additional rewriting rule: $\mathcal{R}$ will completely defined by the protocol specification. We shall then run the completion algorithm on $\mathcal{A}, \mathcal{I}, \mathcal{R}$, computing an automaton $\mathcal{F}$ such that its languages include those of $\mathcal{A}$, satisfy $\mathcal{I}$, and are closed under $\mathcal{R}$. Once done that, we can check a number of properties about P by inspecting the result $\mathcal{F}$.

Fig. 4.4. show the data flow we use. The main difference from the CFA of Sect. 4.4.1 is that we do not use *net*, but two distinct states, *in* and *out*, for the inputs and outputs of the processes, respectively. As before, bullets in the figure represent the languages we approximate in our $\mathrm{CFA}_{\mathsf{io}}$. Formally, the states of $\mathcal{A}$ are:

- `@in`, `@out`, `@chk-in`, `@chk-out`;
- `@in-`$b\theta$ and `@out-`$b\theta$, for each θ such that $P@\theta = P_1|P_2$, and $b \in \{\mathsf{l}, \mathsf{r}\}$;
- `@in-n`θ and `@out-n`θ, for each θ such that $P@\theta = \mathsf{repl}.P'$;
- `@inters-`θ, for each θ such that $P@\theta = [x = y].P'$;
- `@x` and `@x-val`, for each $\mathsf{new}\ x$ occurring in P;

- $\texttt{@x}$, for each $\mathsf{let}\ x = M$ and $\mathsf{in}\ x$ occurring in P.

We generate the transitions of the automaton $\mathcal{A}$ and the intersection constraints $\mathcal{I}$ using the function $gen(\theta, P', \zeta, in, out)$, recursively defined in Fig. 4.5. As for the CFA, we pass θ and P' to select a subprocess of P. Here however, we do not always use the state $\texttt{@x}$ for the language of the variable x, so we pass the *static environment* $\zeta \in \mathcal{V} \to \mathcal{Q}$ to specify which state corresponds to each variable. As before, we use $\zeta(M)$ for its homomorphic extension to ground terms. The *in* and *out* parameters define the states for the approximation of the values that can be received and sent by P', respectively. Initially, *gen* is called as $gen(\epsilon, P, \emptyset, \texttt{@in}, \texttt{@out})$ to generate $\mathcal{A}, \mathcal{I}$ for the whole process P.

$$
\begin{array}{l}
gen(\theta, \mathsf{nil}, \zeta, in, out) = \emptyset \\
gen(\theta, \mathsf{in}\ x\ .P, \zeta, in, out) = (\texttt{@x} \to in), gen(\mathsf{n}\theta, P, \zeta[x \mapsto \texttt{@x}], in, out) \\
gen(\theta, \mathsf{out}\ M\ .P, \zeta, in, out) = (out \to \zeta(M)), gen(\mathsf{n}\theta, P, \zeta, in, out) \\
gen(\theta, (P|Q), \zeta, in, out) = (out \to \texttt{@out-}\mathsf{l}\theta), (out \to \texttt{@out-}\mathsf{r}\theta), \\
\quad (\texttt{@in-}\mathsf{l}\theta \to in), (\texttt{@in-}\mathsf{l}\theta \to \texttt{@out-}\mathsf{r}\theta), (\texttt{@in-}\mathsf{r}\theta \to in), (\texttt{@in-}\mathsf{r}\theta \to \texttt{@out-}\mathsf{l}\theta), \\
\quad gen(\mathsf{l}\theta, P, \zeta, \texttt{@in-}\mathsf{l}\theta, \texttt{@out-}\mathsf{l}\theta), gen(\mathsf{r}\theta, Q, \zeta, \texttt{@in-}\mathsf{r}\theta, \texttt{@out-}\mathsf{r}\theta) \\
gen(\theta, \mathsf{repl}.P, \zeta, in, out) = (out \to \texttt{@out-}\mathsf{n}\theta), (\texttt{@in-}\mathsf{n}\theta \to in), \\
\quad (\texttt{@in-}\mathsf{n}\theta \to \texttt{@out-}\mathsf{n}\theta), gen(\mathsf{n}\theta, P, \zeta, \texttt{@in-}\mathsf{n}\theta, \texttt{@out-}\mathsf{n}\theta) \\
gen(\theta, \mathsf{let}\ x = M\ .P, \zeta, in, out) = (\texttt{@x} \to \zeta(M)), gen(\mathsf{n}\theta, P, \zeta[x \mapsto \texttt{@x}], in, out) \\
gen(\theta, \mathsf{new}\ x\ .P, \zeta, in, out) = (\texttt{@x} \to \mathsf{val}(\texttt{@x-val})), \\
\quad (\texttt{@x-val} \to \bar{x}), (\texttt{@x-val} \to \mathsf{next}(\texttt{@x-val})), gen(\mathsf{n}\theta, P, \zeta[x \mapsto \texttt{@x}], in, out) \\
gen(\theta, [x = y].P, \zeta, in, out) = (\zeta(x) \cap \zeta(y) \subseteq \texttt{@inters-}\theta), \\
\quad gen(\mathsf{n}\theta, P, \zeta[x, y \mapsto \texttt{@inters-}\theta], in, out) \\
gen(\theta, \mathsf{chk}.P, \zeta, in, out) = gen(\mathsf{n}\theta, P, \zeta, \texttt{@chk-in}, \texttt{@chk-out})
\end{array}
$$

Figure 4.5: Extraction of $\mathrm{CFA}_{\mathsf{io}}$ constraints from a process.

No transitions are generated for nil. For the input $\mathsf{in}\ x\ .P'$, we generate a new state $\texttt{@x}$, and a transition from it to *in* to include inputs in its language. Then, we update ζ (the dotted line in Fig. 4.4) by binding x to $\texttt{@x}$, and proceed recursively with the continuation P'. Outputs as $\mathsf{out}\ M\ .P'$ generate a transition from the *out* state to $\zeta(M)$, the term obtained by replacing all the variables in M with their corresponding states; we then proceed recursively for P'. For example, the generated transitions for $P = \mathsf{in}\ x\ .\mathsf{out}\ \mathsf{f}(x)\ .\mathsf{out}\ \mathsf{g}(x)\ .\mathsf{nil}$ are $\texttt{@x} \to \texttt{@in}, \texttt{@out} \to \mathsf{f}(\texttt{@x}), \texttt{@out} \to \mathsf{g}(\texttt{@x})$. Note that each output *contributes* to the language of $\texttt{@out}$ by adding transitions to those already generated. This is depicted in Fig. 4.4 by the *out* arrow going straight from left to the right and collecting possible outputs

from below. As seen in the figure, this happens for all processes, except for chk.

Parallel processes such as $P|Q$ are handled by creating four dedicated states for input and output of the left and right branch, then adding transitions to cross-connect inputs and outputs as in Fig. 4.4. Replication repl.P is done in a similar fashion, with a loopback transition.

For let bindings, we simply create a new state for the approximation of the bound value, and update ζ accordingly. A new x .P' causes the generation of transitions for the language $\mathsf{val}(\bar{x}), \mathsf{val}(\mathsf{next}(\bar{x})), \mathsf{val}(\mathsf{next}(\mathsf{next}(\bar{x}))), \ldots$ using the two states `@x` and `@x-val`; then, we update ζ to bind x to this language.

A match $[x = y].P'$ creates a new state `@inters-`θ for the (approximation of the) intersection of values hold by x and y, together with the associated intersection constraint; in the analysis of P' we use this new state for both $\zeta(x)$ and $\zeta(y)$.

When a chk is fired, the continuation runs in an isolated world, therefore in the analysis we simply reset in, out to new independent states and proceed recursively. Note that ζ is not changed, and that bound variables bring their values into the new world (e.g. in x .chk.out x .nil).

Note that our CFA_{io} generates no transitions for states `@in` and `@chk-in`: their language is therefore empty. In fact, top-level processes receive no value from their environment; this reflects the absence of an input rule in our dynamic semantics.

Example We consider again the example given in Sect. 4.4.1.

$P =$out 0 .nil | in x .out f(x) .out 1 .nil

Applying $gen()$, we generate the following constraints:

`@out` $\rightarrow$ `@out-l`, `@out-r`

`@in-l` $\rightarrow$ `@in`, `@out-r`

`@in-r` $\rightarrow$ `@in`, `@out-l`

`@out-l` $\rightarrow 0$

`@x` $\rightarrow$ `@in-r`

`@out-r` $\rightarrow$ f(`@x`)

`@out-r` $\rightarrow 1$

Here, variable x is approximated with the inputs of the right branch of the parallel `@in-r`. In turn, these inputs include the outputs of the left branch `@out-l`. Finally, the outputs include 0, that therefore is propagated to the language of `@x`.

The language of `@out` defined by the above automaton includes the terms $0, 1$ and f(0) only. These are exactly the terms that can be output dynamically. So, in this case the static semantics coincides with the dynamic one.

Time Complexity We now study how the overall complexity of the $\mathrm{CFA}_{\mathsf{io}}$ depends on the size of the process P.

As for the CFA, here the function $gen()$ is defined recursively by simple induction on the syntax of P. Since we never consider twice the same subprocess P, the constraint generation takes linear time. Also, the number of generated transitions forming $\mathcal{A}$ is linear. The number of generated intersection constraints, forming $\mathcal{I}$, is linear as well.

So, once again we get the worst-case complexity bound from Theorem 7 in Chapter 3.

$$\mathcal{O}(n^k \cdot (\log s + \mathsf{maxArity} \log n) \cdot s^l \cdot c \cdot (|\mathcal{I}| + |\mathcal{R}|))$$

Here variable n is the number of generated states, which is linear on (the size of P), plus any further states the user want to use for the approximation. Variable $|\mathcal{R}|$ is just the number of rewriting rules used in the specification of the protocol, since our $\mathrm{CFA}_{\mathsf{io}}$ does not add any further rule. Variable $|\mathcal{I}|$ is the number of generated constraints, so is linear on P.

The other variables depend only on the particular algebra of terms modulo rewriting used by the protocol, and not on the protocol itself. For the precise meaning of those, we refer to Theorem 7. Here, we merely note that the overall complexity is polynomial.

4.4.4 Subject Reduction for $\mathrm{CFA}_{\mathsf{io}}$

Here, we establish the soundness for the $\mathrm{CFA}_{\mathsf{io}}$.

First, we define a compatibility relation $\sim$ over addresses. Roughly speaking, $\theta_1 \sim \theta_2$ means that at run-time a thread running $P@\theta_1$ could communicate with a thread running $P@\theta_2$. The actual $\sim$ over-approximates run-time communication, and simply checks if the two addresses point to processes either at different branches of the same parallel, or under the same replication. We also take into account the presence of the chk prefix, since its continuation cannot interact with previously spawned threads.

Definition 22 *Given a process P, the address compatibility relation $\sim$ is the minimum symmetric relation over the addresses of P such that*

- $\mathsf{l}\theta \sim \mathsf{r}\theta$
- $P@\theta = \mathsf{repl}.P' \implies \theta \sim \theta$
- $\theta_1 \sim \theta_2 \wedge P@\theta_1 \neq \mathsf{chk}.P' \implies \mathsf{n}\theta_1 \sim \theta_2 \wedge \mathsf{l}\theta_1 \sim \theta_2 \wedge \mathsf{r}\theta_1 \sim \theta_2$

provided that the above are valid addresses for P.

Fig. 4.6 shows the syntax tree for three processes; the bifurcations correspond to the occurrence of parallel operators. In each case, we represent by bullets the addresses compatible with a selected address θ.

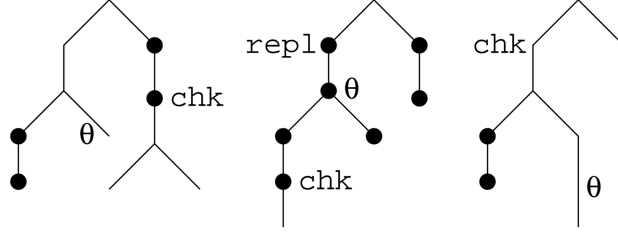


Figure 4.6: Address compatibility

The following lemmata ensure that the relation $\sim$ actually encompasses all run-time communications.

Lemma 12 *If* $\langle\emptyset,\epsilon\rangle\to^*\sigma,\langle\rho_1,\theta_1\rangle,\langle\rho_2,\theta_2\rangle$, *then* $\theta_1\sim\theta_2$.

Proof. We proceed by induction on the number of computation steps, and then by case analysis on the last step.

If the last step was a chk, the lemma trivially holds because we have only one thread in the final state. The Rew rule only affects ρ.

All other rules change θ to $\mathsf{n}\theta$: by the definition of $\simeq$, an address $\mathsf{n}\theta$ is compatible with all addresses θ is compatible with, so inductive hypothesis suffices.

Finally, we need to consider the split of a parallel $P@\theta$ into $P@\mathsf{l}\theta$ and $P@\mathsf{r}\theta$. The new addresses are compatible with each other, and they are compatible with all the addresses θ is compatible with. So, we conclude by inductive hypothesis. $\Box$

It is convenient to study the values of *in* and *out* used during the constraints generation, when $gen()$ considered a subprocess $P@\theta$. The following functions capture the actual states used there.

Definition 23 *We inductively define the function* $\mathrm{out}(\theta)$ *as follows.*

$$\begin{array}{lll}
\mathrm{out}(\epsilon) & = \texttt{@out} & \\
\mathrm{out}(\mathsf{l}\theta) & = \texttt{@out}\text{-}\mathsf{l}\theta & \\
\mathrm{out}(\mathsf{r}\theta) & = \texttt{@out}\text{-}\mathsf{r}\theta & \\
\mathrm{out}(\mathsf{n}\theta) & = \texttt{@out}\text{-}\mathsf{n}\theta & \textit{if } P@\theta = \mathsf{repl}.P' \\
\mathrm{out}(\mathsf{n}\theta) & = \texttt{@chk-out} & \textit{if } P@\theta = \mathsf{chk}.P' \\
\mathrm{out}(\mathsf{n}\theta) & = \mathrm{out}(\theta) & \textit{otherwise}
\end{array}$$

Similarly, the function $\mathrm{in}(\theta)$ *is defined as follows.*

$$\begin{array}{lll}
\mathrm{in}(\epsilon) & = \texttt{@in} & \\
\mathrm{in}(\mathsf{l}\theta) & = \texttt{@in}\text{-}\mathsf{l}\theta & \\
\mathrm{in}(\mathsf{r}\theta) & = \texttt{@in}\text{-}\mathsf{r}\theta & \\
\mathrm{in}(\mathsf{n}\theta) & = \texttt{@in}\text{-}\mathsf{n}\theta & \textit{if } P@\theta = \mathsf{repl}.P' \\
\mathrm{in}(\mathsf{n}\theta) & = \texttt{@chk-in} & \textit{if } P@\theta = \mathsf{chk}.P' \\
\mathrm{in}(\mathsf{n}\theta) & = \mathrm{in}(\theta) & \textit{otherwise}
\end{array}$$

Indeed, these are the values used by $gen()$, as the following lemma states.

Lemma 13 *When unfolding* $gen(\epsilon, P, \emptyset, \texttt{@}\boldsymbol{in}, \texttt{@}\boldsymbol{out})$*, the values of in and out used for* $P@\theta$ *are* $\mathrm{in}(\theta)$ *and* $\mathrm{out}(\theta)$*, respectively.*

Proof. By structural induction on θ. Following the definition of $gen()$, the values of *in* and *out* change exactly following the functions in Definition 23. □

Now we can relate the address compatibility relation with functions in() and out(). In fact, the input and output states of compatible addresses satisfy the following inclusion property.

Lemma 14 *Given* P*, let* $\mathcal{F}$ *be the automata resulting from the* CFA_{io}*. If* $\theta_1 \sim \theta_2$*, then we have* $[\mathrm{out}(\theta_1)]_{\mathcal{F}} \subseteq [\mathrm{in}(\theta_2)]_{\mathcal{F}}$*.*

Proof. By induction on the rules defining $\sim$, as for Definition 22. We also analyze the cases of Definition 23. In each step, we prove the above property both for $\theta_1 \sim \theta_2$ and its symmetric version $\theta_2 \sim \theta_1$.

If θ_1 and θ_2 point to different branches of the same parallel, i.e. $\theta_1 = \mathsf{l}\theta$ and $\theta_2 = \mathsf{r}\theta$, where $P@\theta$ is a parallel, then $gen()$ generated the transition $\texttt{@in-}\theta_1 \to \texttt{@out-}\theta_2$. This proves the wanted property. The symmetric case is analogous.

If $\theta_1 = \theta_2 = \theta$ and $P@\theta$ is a replication, we generated through $gen()$ the transition $\texttt{@in-}\theta \to \texttt{@out-}\theta$. So, the property holds. Here the symmetric case is identical.

For the inductive case, assume the lemma holds for $\theta_1 \sim \theta_2$, and that $P@\theta_1$ is not a chk, as in the third case of Definition 22. By inductive hypothesis, $[\mathrm{out}(\theta_1)]_{\mathcal{F}} \subseteq [\mathrm{in}(\theta_2)]_{\mathcal{F}}$ (exchange "in" and "out" for the symmetric case). We now prove the lemma for $\theta'\theta_1 \sim \theta_2$, with $\theta' \in \{\mathsf{n}, \mathsf{l}, \mathsf{r}\}$. Let $out = \mathrm{out}(\theta'\theta_1)$ and $in = \mathrm{in}(\theta'\theta_1)$

For $\theta' \neq \mathsf{n}$, the subprocess $P@\theta$ is a parallel. By Lemma 13, we generated the constraint $\mathrm{out}(\theta_1) \to out$ when we ran $gen()$ over $P@\theta$. This, and inductive hypothesis prove the lemma for this case. For the symmetric case, we note that $gen()$ also generated the transition $in \to \mathrm{in}(\theta_1)$, and proceed similarly.

When $\theta' = \mathsf{n}$ and $P@\theta_1$ is a replication, we proceed as for the parallel case. In fact, also here we generated the transition $\mathrm{out}(\theta_1) \to out$ when we ran $gen()$ over $P@\theta$. The symmetric case is analogous, considering $in \to \mathrm{in}(\theta_1)$.

In all the other cases, $\theta' = \mathsf{n}$ and $P@\theta \neq \mathsf{chk}.P'$. We have $out = \mathrm{out}(\theta_1)$ and $in = \mathrm{in}(\theta_1)$, since we fall in the last case of Definition 23. So, here inductive hypothesis suffices. □

The following theorem ensures that our $\mathrm{CFA}_{\mathsf{io}}$ is sound, relating the dynamic semantics to the static one.

Theorem 9 (Subject Reduction) *Given P, let $\mathcal{F}$ be the automata resulting from the CFA$_{\mathsf{io}}$. Assume $\langle\emptyset, \epsilon\rangle \rightarrow^* \xrightarrow{\alpha} \sigma, \langle\rho, \theta\rangle$.*

1. $\forall x \in \mathsf{dom}(\rho).\ \rho(x) \in [\texttt{@x}]_{\mathcal{F}}$
2. *if* $\alpha = (\mathsf{out}\ \theta\ , M)$ *and* chk *was not fired before* α*, then* $M \in [\texttt{@out}]_{\mathcal{F}}$
3. *if* $\alpha = (\mathsf{out}\ \theta\ , M)$ *and* chk *was fired before* α*, then* $M \in [\texttt{@chk-out}]_{\mathcal{F}}$

Proof. By induction on the number of computation steps, and by case analysis on the last step. First, we consider property (1): for this, we only need to check the rules that update the environment ρ.

When the Comm rule is applied, yielding to comm $\theta_1, \theta_2, \rho_2(M)$, by Lemma 12 we have $\theta_1 \sim \theta_2$. We look for the transitions for in x and out M generated by $gen()$. By Lemma 13, these transitions have the form $\texttt{@x} \rightarrow in$ and $out \rightarrow \zeta(M)$, where $in = \mathrm{in}(\theta_1)$ and $out = \mathrm{out}(\theta_2)$. By Lemma 14, $\rho'_1(x) = \rho_2(M) \in [out]_{\mathcal{F}} \subseteq [in]_{\mathcal{F}} \subseteq [\texttt{@x}]_{\mathcal{F}}$, provided that $\zeta(M)$ is a correct approximation of $\rho_2(M)$, i.e. $\rho_2(M) \in [\![\zeta(M)]\!]_{\mathcal{F}}$.

For this last proof obligation, we only need to show that $\rho_2(x) \in [\![\zeta(x)]\!]_{\mathcal{F}}$ for all variables in x occurring in M, and apply structural induction on M. Take any such x. If there is no match involving x, we have $\zeta(x) = \texttt{@x}$, so inductive hypothesis suffices. Otherwise, there are matches. Roughly, the process P has this form:

$$P = \cdots\ .[x = y].\ \cdots\ .\mathsf{out}\ M_x\ .\ \cdots$$

Here, we have $\zeta(x) = \texttt{@inters-}\theta_m$, where θ_m is the address of the match.

First, assume the above match is the earliest match involving x, i.e. the one nearest to address ϵ. In this case, $gen()$ generated the constraints $\texttt{@x} \cap \texttt{@some-y} \subseteq \texttt{@inters-}\theta_m$, for some state @some-y. This implies that $[\![\texttt{@x}]\!]_{\mathcal{F}} \subseteq [\![\texttt{@inters-}\theta_m]\!]_{\mathcal{F}}$. By inductive hypothesis, we have $\rho_2(x) \in [\![\texttt{@x}]\!]_{\mathcal{F}}$, completing the proof for this case.

If there are more matches, we proceed by induction on the number n of the matches. Assume that the $(n-1)$-th match, counting from the nearest to ϵ towards θ_2, is at address θ_l. By inductive hypothesis (on n) we have $\rho_2(x) \in [\![\texttt{@inters-}\theta_l]\!]_{\mathcal{F}}$. Further, running $gen()$ on $P@\theta_m$ generated the constraint $\texttt{@inters-}\theta_l \cap \texttt{@some-z} \subseteq \texttt{@inters-}\theta_m$, for some state @some-z. This means that $[\![\texttt{@inters-}\theta_l]\!]_{\mathcal{F}} \subseteq [\![\texttt{@inters-}\theta_m]\!]_{\mathcal{F}}$. So, $\rho_2(x) \in [\![\texttt{@inters-}\theta_m]\!]_{\mathcal{F}}$. This completes the proof that property (1) is preserved by Comm.

We now tackle property (1) for the other rules. The Let case is straightforward: we generated the transition $\texttt{@x} \rightarrow \zeta(M)$, so we have $\rho(M) \in [\texttt{@x}]_{\mathcal{F}}$. Rule New also poses no problem, because the fresh term returned by genFresh() is chosen among the terms in the language of @x. Finally, environment updates by rule Rew are harmless, the languages of $\mathcal{F}$ being closed under rewritings.

For properties (2,3), only rule Out may cause out $\theta, \rho(M)$. Here we apply the same argument used for property (1) to derive that $\rho(M) \in [\![\zeta(M)]\!]_{\mathcal{F}}$. Moreover, $gen()$ generated the constraint $out' \to \zeta(M)$, where $out' = \text{out}(\theta)$. Therefore, we have $\rho(M) \in [\![out]\!]_{\mathcal{F}}$.

From here, we just examine how the variable out changes in the definition of $gen()$ when we recurse on subprocesses. Every time it is changed from out_1 to out_2, we also generate the transition $out_1 \to out_2$, except when crossing a chk. This ensures that $[\![out_2]\!]_{\mathcal{F}} \subseteq [\![out_1]\!]_{\mathcal{F}}$. By induction on these changes, we can see that $[\![out']\!]_{\mathcal{F}} \subseteq [\![out_{top}]\!]_{\mathcal{F}}$, where the state out_{top} is `@chk-out` or `@out`, depending on whether the output $P@\theta$ is under a chk or not, respectively. Of course, the output is under a chk if and only if it was fired in the process run. This completes the proof for properties (2) and (3). □

4.4.5 Examples and Comparison

We now present the result of the analysis for several processes. For each of these, we compare the result of the CFA of Sect. 4.4.1 with the one obtained through the CFA_{io} of Sect. 4.4.3.

Parallel Example In Sect. 4.4.1 and 4.4.3, we considered the following process:

$$P = \text{out } 0 \text{ .nil} \mid \text{ in } x \text{ .out f}(x) \text{ .out } 1 \text{ .nil}$$

We noted that P, according to our dynamic semantics, can only output $0, \mathsf{f}(0)$ and 1. We already showed that our CFA_{io} is able to compute exactly that set of terms as its static approximation. We also showed that CFA instead produced a less precise result, e.g. including also $\mathsf{f}(1)$. Therefore, in this case, the CFA_{io} produced a better approximation.

There is an easy way to visualize why the CFA of Sect. 4.4.1 is not very precise for this example. Recall that our CFA does not distinguish between P and repl.P, i.e. processes are analyzed by the CFA just as they were under a replication. Therefore, the process considered above is handled as

$$P = \text{out } 0 \text{ .nil} \mid \text{ repl.in } x \text{ .out f}(x) \text{ .out } 1 \text{ .nil}$$

which indeed can output $\mathsf{f}(1)$. So, since the CFA is sound, it has to include $\mathsf{f}(1)$ in its computed approximation.

Matching Example 1 Consider the following process:

$$P = \text{out } 0 \text{ .out } 1 \text{ .nil} \mid \text{ in } x \text{ .let } z = 0 \text{ .}[x = z]\text{.out f}(x) \text{ .nil}$$

At run-time, the last $\mathsf{out}\ \mathsf{f}(x)$ can output $\mathsf{f}(0)$, only. Running the CFA will generate the following constraints:

$$\texttt{@net} \rightarrow 0, 1$$
$$\texttt{@x} \rightarrow \texttt{@net}$$
$$\texttt{@z} \rightarrow 0$$
$$\texttt{@x} \simeq \texttt{@z} \Rightarrow \texttt{@net} \rightarrow \texttt{@net1}$$
$$\texttt{@net1} \rightarrow \texttt{@net}$$
$$\texttt{@net1} \rightarrow \mathsf{f}(\texttt{@x})$$

Solving these constraints, we discover that $\texttt{@x} \simeq \texttt{@z}$ and therefore we merge $\texttt{@net1}$ with $\texttt{@net}$. So, we obtain $\texttt{@net} \rightarrow^*_{\mathcal{A}} \mathsf{f}(\texttt{@x})$. Since $0, 1 \in [\![\texttt{@x}]\!]_{\mathcal{F}}$, we get $\mathsf{f}(0), \mathsf{f}(1) \in [\![\texttt{@net}]\!]_{\mathcal{F}}$. In this case the CFA was not able to deduce that $\mathsf{f}(1)$ can not be output.

The CFA_{io} provides a better approximation: we start from the constraints

$$\texttt{@out} \rightarrow \texttt{@out-l}, \texttt{@out-r}$$
$$\texttt{@in-l} \rightarrow \texttt{@in}, \texttt{@out-r}$$
$$\texttt{@in-r} \rightarrow \texttt{@in}, \texttt{@out-l}$$
$$\texttt{@out-l} \rightarrow 0, 1$$
$$\texttt{@x} \rightarrow \texttt{@in-r}$$
$$\texttt{@z} \rightarrow 0$$
$$\texttt{@x} \cap \texttt{@z} \subseteq \texttt{@inters-nnr}$$
$$\texttt{@out-r} \rightarrow \mathsf{f}(\texttt{@inters-nnr})$$

Here, the intersection constraint computes $\texttt{@x} \cap \texttt{@z}$, resulting in $\texttt{@inters-nnr}$, and uses it in the last constraint for the output. So, we only have $0 \in [\![\texttt{@inters-nnr}]\!]_{\mathcal{F}}$, and therefore we only get $\mathsf{f}(0) \in [\![\texttt{@out}]\!]_{\mathcal{F}}$, as we expected.

Matching Example 2 Consider the following process:

$$P_1 = \mathsf{out}\ \mathsf{cons}(0,0)\ .\mathsf{out}\ \mathsf{cons}(1,1)\ .\mathsf{nil}\ |$$
$$\quad \mathsf{in}\ x\ .\mathsf{let}\ f = \mathsf{fst}(x)\ .\mathsf{let}\ z = 0\ .[f = z].\mathsf{out}\ \mathsf{snd}(x)\ .\mathsf{nil}$$

At run-time, the last $\mathsf{out}\ \mathsf{snd}(x)$ can output 0, only. Running the CFA will generate the following constraints:

$$\begin{aligned}
&\texttt{@net} \to \mathsf{cons}(0,0), \mathsf{cons}(1,1)\\
&\texttt{@x} \to \texttt{@net}\\
&\texttt{@f} \to \mathsf{fst}(\texttt{@x})\\
&\texttt{@z} \to 0\\
&\texttt{@f} \simeq \texttt{@z} \Rightarrow \texttt{@net} \to \texttt{@net1}\\
&\texttt{@net1} \to \texttt{@net}\\
&\texttt{@net1} \to \mathsf{snd}(\texttt{@x})
\end{aligned}$$

Solving these constraints, we discover that $\texttt{@f} \simeq \texttt{@z}$ and therefore we merge $\texttt{@net1}$ with $\texttt{@net}$. So, we obtain $0, 1 \in [\![\texttt{@net}]\!]_{\mathcal{F}}$.

We now try the CFA_{io}. The generated constraints are

$$\begin{aligned}
&\texttt{@out} \to \texttt{@out-l}, \texttt{@out-r}\\
&\texttt{@in-l} \to \texttt{@in}, \texttt{@out-r}\\
&\texttt{@in-r} \to \texttt{@in}, \texttt{@out-l}\\
&\texttt{@out-l} \to \mathsf{cons}(0,0), \mathsf{cons}(1,1)\\
&\texttt{@x} \to \texttt{@in-r}\\
&\texttt{@f} \to \mathsf{fst}(\texttt{@x})\\
&\texttt{@z} \to 0\\
&\texttt{@f} \cap \texttt{@z} \subseteq \texttt{@inters-nnnr}\\
&\texttt{@out-r} \to \mathsf{snd}(\texttt{@x})
\end{aligned}$$

Note the presence of the intersection constraint. The CFA_{io} computes the intersection $\texttt{@f} \cap \texttt{@z}$ and uses the result **@inters-nnnr** later on for f and z after the match. Unfortunately there actually are *no* such uses, so the intersection computation is wasted. Moreover, while the analysis refines the approximation of f and z, it does not refine the one of x. In fact, a single state is used for the values of x before and after the match. So, the result is that the last out may output either 0 or 1, i.e. $0, 1 \in [\![@out]\!]_{\mathcal{F}}$.

We can however rewrite our process into a semantically equivalent form that yields to a more precise approximation.

$$\begin{aligned}
P_2 = &\mathsf{out}\ \mathsf{cons}(0,0)\ .\mathsf{out}\ \mathsf{cons}(1,1)\ .\mathsf{nil}\ |\\
&\quad \mathsf{in}\ x\ .\mathsf{let}\ y = \mathsf{cons}(0, \mathsf{snd}(x))\ .[x = y].\mathsf{out}\ \mathsf{snd}(x)\ .\mathsf{nil}
\end{aligned}$$

Here, we match x itself rather than its first component. We do this by computing another term, y, which is equal to x except that its first component is set to 0. Of course, matching $x = y$ is equivalent to matching $\mathsf{fst}(x) = 0$.

The CFA constraints are similar to the previous ones:

$$\texttt{@net} \to \mathsf{cons}(0,0), \mathsf{cons}(1,1)$$
$$\texttt{@x} \to \texttt{@net}$$
$$\texttt{@y} \to \mathsf{cons}(0, \mathsf{snd}(\texttt{@x}))$$
$$\texttt{@x} \simeq \texttt{@y} \Rightarrow \texttt{@net} \to \texttt{@net1}$$
$$\texttt{@net1} \to \texttt{@net}$$
$$\texttt{@net1} \to \mathsf{snd}(\texttt{@x})$$

Once again, we get $\texttt{@x} \simeq \texttt{@y}$, and therefore `@net1` is merged with `@net`. The CFA result does not improve, since we still have $0, 1 \in [\![\texttt{@out}]\!]_{\mathcal{F}}$.

For the $\mathrm{CFA}_{\mathsf{io}}$, we get

$$\texttt{@out} \to \texttt{@out-l}, \texttt{@out-r}$$
$$\texttt{@in-l} \to \texttt{@in}, \texttt{@out-r}$$
$$\texttt{@in-r} \to \texttt{@in}, \texttt{@out-l}$$
$$\texttt{@out-l} \to \mathsf{cons}(0,0), \mathsf{cons}(1,1)$$
$$\texttt{@x} \to \texttt{@in-r}$$
$$\texttt{@y} \to \mathsf{cons}(0, \mathsf{snd}(\texttt{@x}))$$
$$\texttt{@x} \cap \texttt{@y} \subseteq \texttt{@inters-nnr}$$
$$\texttt{@out-r} \to \mathsf{snd}(\texttt{@inters-nnr})$$

Here, the result of the intersection, `@inters-nnr`, is indeed used in the last constraint. This is because we use the same variable x both in the match and in the output. We now have a better approximation, as $1 \notin [\![\texttt{@out}]\!]_{\mathcal{F}}$.

When possible, we shall always adopt the style of matching used in P_2. Roughly, this is possible when we can deconstruct x into its components, change some of them into fixed values, and then build with them a term y to be matched with x. This is easy to do when the term rewriting rules deal with constructors and destructors. For instance, we used pairs in the example above. Also encryptions work similarly.

Sometimes, even more careful matching might help. For instance, take $\mathsf{let}\ f = \mathsf{fst}(x)\ .\mathsf{let}\ s = \mathsf{snd}(x)\ .[f = s].P$. We can translate it into one of the following

$$\mathsf{let}\ y = \mathsf{cons}(\mathsf{fst}(x), \mathsf{fst}(x))\ .\ [x = y].\ P$$
$$\mathsf{let}\ y = \mathsf{cons}(\mathsf{snd}(x), \mathsf{snd}(x))\ .\ [x = y].\ P$$
$$\mathsf{let}\ y = \mathsf{cons}(\mathsf{snd}(x), \mathsf{fst}(x))\ .\ [x = y].\ P$$

The last one will likely lead to a better result, since we will use non-trivial intersections to approximate both the first and the second component of x.

Further, consider $\mathsf{let}\ f = \mathsf{f}(x)\ .\mathsf{let}\ g = \mathsf{g}(y)\ .[f = g].P$. If there are no rewriting rules involving f or g, we can not translate the match as we did

for previous examples. However, we can add some rules to make it possible, such as

$$\mathsf{fi}(\mathsf{f}(X)) \Rightarrow X$$
$$\mathsf{gi}(\mathsf{g}(X)) \Rightarrow X$$

The above provide "destructors" for f and g, allowing us to rewrite the match as $\mathsf{let}\ w = \mathsf{gi}(\mathsf{f}(x))\ .\mathsf{let}\ z = \mathsf{fi}(\mathsf{g}(y))\ .[y = w].[x = z].P$. This form directly matches variables x and y, so we can expect precise results. Note that, if the destructors fi and gi are not used elsewhere in the process, the semantics of the process is unchanged.

A Case for the CFA We consider

$$P = \mathsf{let}\ x = 0\ .\mathsf{let}\ y = 1\ .[x = y].\mathsf{out}\ 2\ .\mathsf{nil}$$

Clearly P contains dead code. In fact, after two steps, P reaches the match $0 = 1$, which is not satisfied, and it stops. So, the output of 2 never happens. The CFA is able to detect this dead code. Here are the generated constraints:

$$\texttt{@x} \to 0$$
$$\texttt{@y} \to 1$$
$$\texttt{@net-nn} \to \texttt{@net}$$
$$\texttt{@x} \simeq \texttt{@y} \Rightarrow \texttt{@net} \to \texttt{@net-nn}$$
$$\texttt{@net-nn} \to 2$$

The condition $\texttt{@x} \simeq \texttt{@y}$ is not satisfied, and so we never merge `@net-nn` with `@net`. Therefore, the CFA is able to prove $2 \notin [\![\texttt{@net}]\!]_{\mathcal{F}}$.

The CFA_{io} is not as precise as the CFA for this case. We generate

$$\texttt{@x} \to 0$$
$$\texttt{@y} \to 1$$
$$\texttt{@x} \cap \texttt{@y} \subseteq \texttt{@inters-nn}$$
$$\texttt{@out} \to 2$$

Note that we have $2 \in [\![\texttt{@out}]\!]_{\mathcal{F}}$.

We could include the same dead code detection found in the CFA in the CFA_{io}. However, we found this particular feature to be of little importance in our applications. This is mostly because protocol specifications never include dead code. Also, our handling of match is often equivalent to dead code detection, in practice. For instance, consider

$$[x = y].\mathsf{out}\ \mathsf{f}(x)\ .\mathsf{nil}$$

If $\texttt{@x} \not\simeq \texttt{@y}$, the above generates the transition $out \to \mathtt{f}(\texttt{@inters-}\theta)$. The last term, however, is empty, so it never gets rewritten and does not contribute to the language of *out*. In this case, our CFA_{io} encompasses dead code detection.

4.5 Implementation and Test Cases

We implemented the analyses of Sect. 4.4.1 and 4.4.3. Our tool takes a rewriting system $\mathcal{R}$ (including the extended rules of Sect. 3.9.2), a set of intersection constraints $\mathcal{I}$ (usually empty) and a process P. The tool then runs the function *gen*() and generates the constraints to form an automaton $\mathcal{A}$. This also generates some intersection constraints, which are added to $\mathcal{I}$. Finally, $\mathcal{A}, \mathcal{I}$, and $\mathcal{R}$ are passed to the automata approximation tool of Chapter 3.

Our implementation is rather simple. Ignoring the parsing and pretty printing routines, the actual tool core is about 170 lines of Haskell code. This conciseness mostly derives from the simple definition of *gen*(). The tool is available [61] under the terms of the GNU General Public License.

Below, we report on our experiments with this tool.

4.5.1 Diffie-Hellman Example (continued)

We ran the CFA_{io} of Sect. 4.4.3 on the protocol specified in Sect. 4.3.1. We provide the actual specification below. computing the result $\mathcal{F}$. Our tool generated an $\mathcal{F}$ having 47 states and 865 transitions. Our development machine[2] took about one minute for computing the approximation. Our analysis was able to establish *forward secrecy*, as $\mathsf{m} \notin [\![\texttt{@chk-out}]\!]_{\mathcal{F}}$.

```
### Inverse

!(1) => 1 .
!(!(X)) => X .
!(*(X,Y)) => *(!(X),!(Y)) .

### Times

*(1,X) => X .
*(X,*(Y,Z)) => *(*(X,Y),Z) .
*(X,Y) => *(Y,X) .
*(X,!(X)) => 1 .

### Exp

^(1,X) => 1 .
^(X,1) => X .
^(^(X,Y),Z) => ^(X,*(Y,Z)) .
^(*(Y,Z),X) => *(^(Y,X),^(Z,X)) .
^(!(X),Y) => !(^(X,Y)) .
```

[2] Ref.: PowerPC G4 1.25GHz, 768MB RAM, $\mathsf{maxStates} = \#\mathcal{Q}_{\mathcal{A}} + 6$, $\mathsf{maxJoin} = +\infty$

```
### Enc/Dec

dec(enc(M,K),K) => M .

## Pairs

fst(cons(X,Y)) => X .
snd(cons(X,Y)) => Y .

## Expected Result

| => @zzResult : soFarSoGood .

# We check for val(m), both before and after the chk.

| @out     : val(X) , X : kM  => @zzResult : !FAIL! .
| @chk-out : val(X) , X : kM  => @zzResult : !FAIL! .

%%

new G .
( # Dolev-Yao adversary.
  ! . new Nonce . out Nonce . ()
| out G . out 1 . ()
| ! . in X . in Y . out X . out Y .
        out enc(X,Y) . out dec(X,Y) .
        out ^(X,Y) . out *(X,Y) . out !(X) .
        out cons(X,Y) . out fst(X) . out snd(X) . out f(X) .
        out val(X) . out next(X) . ()
# Process.
| new K1 . new K2 .
( # Participant A .
  ! . new A . out enc(^(G,A),K1) . in X .
      let K = ^(dec(X,K2),A) .
      new M . out enc(M,K) . ()
| # Participant B .
  ! . new B . in X . out enc(^(G,B),K2) .
      let K = ^(dec(X,K1),B) .
      in N . out f(dec(N,K)) . ()
| in Know . chk .
        ( # Keys are revealed here.
          out Know . out K1 . out K2 . ()
        | # Dolev-Yao adversary (again)
```

```
          ! . new Nonce2 . out Nonce2 . ()
        | out G . out 1 . ()
        | ! . in X . in Y . out X . out Y .
            out enc(X,Y) . out dec(X,Y) .
            out ^(X,Y) . out *(X,Y) . out !(X) .
            out cons(X,Y) . out fst(X) . out snd(X) .
            out f(X) . out val(X) . out next(X) . ()
        )
)
)
```

4.5.2 Wide-Mouthed Frog

We studied a simplified version of the Wide-Mouthed Frog protocol [22].

1. $A \rightarrow S$: $\{\mathsf{k}\}_{\mathsf{kAS}}$
2. $S \rightarrow B$: $\{\mathsf{k}\}_{\mathsf{kBS}}$
3. $A \rightarrow B$: $\{\mathsf{msg}\}_{\mathsf{k}}$
4. $B \rightarrow \mathsf{all}$: $\mathsf{f}(\mathsf{msg})$

Participants A and B interact with a server S. Both participants share a private symmetric key with the server: kAS and kBS. Then, A generates a session key k and sends it to the server in step one, encrypted with their shared key. The server decrypts the key, and re-encrypts it using the key it shares with B. Then, the encrypted key is sent to B in step 2. In step 3 A send a message msg to B encrypting it with the session key. Finally, B uses the message. In step 4, B publishes the hash of the message.

The goal of this protocol is to exchange the message msg preserving its secrecy. To this purpose, we approximate the protocol behaviour in presence of a Dolev-Yao adversary.

We specify the protocol in our model. The rewriting system $\mathcal{R}$ contains the standard rewriting rule for symmetric encryption.

$$\mathsf{dec}(\mathsf{enc}(M, K), K) \Rightarrow M$$

We also add the following "surjective encryption" rule to $\mathcal{R}$.

$$\mathsf{enc}(\mathsf{dec}(M, K), K) \Rightarrow M$$

Note that the protocol does not actually rely on this rule, and we could avoid including it in $\mathcal{R}$. However, this rule might get exploited by the adversary. So, its presence *helps* the adversary to disrupt the protocol. Moreover, using both rules makes the algebra not definable through constructors (e.g. enc) and destructors (e.g. dec). This fact makes the algebra hard to use for tools that need such requirements. So, we can test our technique on a slightly more complex scenario than the ones used for these tools.

We specify the protocol as follows, including the Dolev-Yao adversary, which is able to perform encryption and decryption on known terms.

$$
\begin{aligned}
P =& DY|\mathsf{repl.new}\ kAS\ \mathsf{.new}\ kBS\ .(A|(B|S)) \\
A =& \mathsf{new}\ k\ \mathsf{.out\ enc}(k, kAS)\ \mathsf{.out\ enc}(\mathsf{msg}, k)\ \mathsf{.nil} \\
B =& \mathsf{in}\ key\ \mathsf{.in}\ msgE\ \mathsf{.let}\ k2 = \mathsf{dec}(key, kBS)\ . \\
& \mathsf{let}\ m2 = \mathsf{dec}(msgE, k2)\ \mathsf{.out\ f}(m2)\ \mathsf{.nil} \\
S =& \mathsf{in}\ x\ \mathsf{.out\ enc}(\mathsf{dec}(x, kAS), kBS)\ \mathsf{.nil} \\
DY =& \mathsf{repl.(in}\ w\ \mathsf{.in}\ z\ \mathsf{.out}\ w\ \mathsf{.out}\ z\ \mathsf{.out\ enc}(x, y)\ \mathsf{.out\ dec}(x, y)\ \mathsf{.nil}| \\
& \mathsf{new}\ nonce\ \mathsf{.out}\ nonce\ \mathsf{.nil)}
\end{aligned}
$$

We run the analysis of this specification. On our development machine, the tool computed the over-approximation $\mathcal{F}$ in under one minute, having 23 states and 170 productions. The tool was able to prove the secrecy of the message, as $\mathsf{msg} \notin [\![\texttt{@out}]\!]_{\mathcal{F}}$.

Below, we provide the actual specification of the protocol.

```
# Wide Mouthed Frog

# Encryption rules .

enc(dec(M,K),K) => M .
dec(enc(M,K),K) => M .

# Expected result.

| @out : f(X) , X : msg => @zzResult : expected .
| @out : msg            => @zzResult : !FAIL! .

%%

( # Dolev-Yao adversary.
 ! . ( in W . in Z . out W . out Z .
       out enc(W, Z) . out dec(W, Z) .
       out val(W) . out next(W) . ()
     | new Nonce . out Nonce . ()
     )
| # Process.
! . new AS . new BS .
  ( in X . out enc(dec(X,AS),BS) . ()
  | new Key . out enc(Key,AS) . out enc(msg,Key) . ()
  |  in Key1 . in N .
    let Key2 = dec(Key1,BS) .
```

```
    let Msg = dec(N,Key2) .
    out f(Msg) . ()
  )
)
```

4.5.3 Session Keys via Asymmetric Key Pairs

The following protocol establishes a session key, much like the Diffie-Hellman protocol of Sect. 4.3.1.

1. $A \to B$: $\{\mathsf{pub(t)}\}_\mathsf{k}$
2. $B \to A$: $\{\mathsf{s}\}_{\mathsf{pub(t)}}$
3. $B \to A$: $\{\mathsf{m}\}_\mathsf{s}$
4. ... (time passes)
5. $A \to \mathsf{all}$: k

Participants A and B share a long term symmetric key k. In step 1, participant A generates a fresh public key pair $\mathsf{pub(t)}, \mathsf{pri(t)}$ and sends the public part to B, encrypted with k. Participant B therefore obtains $\mathsf{pub(t)}$. In step 2, B generates the session symmetric key s, encrypt it with $\mathsf{pub(t)}$, and sends it to A. In the next step, A and B can exchange messages using the established session key.

We study the forward secrecy of the message m. To this purpose, we explicitly disclose the long term key k in the last step. This should not affect completed sessions: to learn s, the adversary would need to decrypt $\{\mathsf{s}\}_{\mathsf{pub(t)}}$ using only $\mathsf{pub(t)}$, which is unfeasible. So the protocol should indeed ensure the forward secrecy of m.

We model this protocol using our technique. We use only the standard rewriting rules for symmetric and asymmetric encryption.

$$\mathcal{R} = \left\{ \begin{array}{ll} \mathsf{dec}(\mathsf{enc}(X,K),K) & \Rightarrow X \\ \mathsf{adec}(\mathsf{aenc}(X,\mathsf{pub}(K)),\mathsf{pri}(K)) & \Rightarrow X \\ \mathsf{fst}(\mathsf{cons}(X,Y)) & \Rightarrow X \\ \mathsf{snd}(\mathsf{cons}(X,Y)) & \Rightarrow Y \end{array} \right\}$$

We specify the protocol as in Fig. 4.7. We ran our tool on that specification: the actual input of our tool is shown below. Our tool computed the result of the analysis in a few seconds, generating an automaton $\mathcal{F}$ with 35 states and 582 transitions. Our analysis showed the wanted forward secrecy property: $\mathsf{m} \notin [\![\texttt{@chk-out}]\!]_\mathcal{F}$.

```
# Asymmetric encryption
```

$$\begin{aligned}
P =& (DY|\mathsf{new}\ k\ .(Proto|Chk)) \\
Chk =& \mathsf{in}\ know\ .\mathsf{chk}.(\mathsf{out}\ know\ .\mathsf{out}\ k\ .\mathsf{nil}|DY)) \\
Proto =& A|B \\
A =& \mathsf{repl.new}\ t\ .\mathsf{out}\ \mathsf{enc}(\mathsf{pub}(t), k)\ .\mathsf{in}\ kSessEnc\ . \\
& \mathsf{let}\ s = \mathsf{adec}(kSessEnc, \mathsf{pri}(t))\ .\mathsf{in}\ msgEnc\ . \\
& \mathsf{let}\ msg = \mathsf{dec}(msgEnc, s)\ .\mathsf{out}\ \mathsf{hash}(msg)\ .\mathsf{nil} \\
B =& \mathsf{repl.in}\ pubKenc\ .\mathsf{let}\ pubK = \mathsf{dec}(pubKenc, k)\ . \\
& \mathsf{new}\ s\ .\mathsf{out}\ \mathsf{aenc}(s, pubK)\ .\mathsf{out}\ \mathsf{enc}(m, s)\ .\mathsf{nil} \\
DY =& \mathsf{repl}.(\mathsf{new}\ nonce\ .\mathsf{out}\ nonce\ .\mathsf{nil}| \\
& \mathsf{in}\ x\ .\mathsf{in}\ y\ .\mathsf{out}\ x\ .\mathsf{out}\ y\ .\mathsf{out}\ \mathsf{enc}(x, y)\ .\mathsf{out}\ \mathsf{dec}(x, y)\ . \\
& \mathsf{out}\ \mathsf{aenc}(x, y)\ .\mathsf{out}\ \mathsf{adec}(x, y)\ .\mathsf{out}\ \mathsf{pri}(x)\ .\mathsf{out}\ \mathsf{pub}(x)\ . \\
& \mathsf{out}\ \mathsf{cons}(x, y)\ .\mathsf{out}\ \mathsf{fst}(x)\ .\mathsf{out}\ \mathsf{snd}(x)\ . \\
& \mathsf{out}\ \mathsf{hash}(x)\ .\mathsf{out}\ \mathsf{val}(x)\ .\mathsf{out}\ \mathsf{next}(x)\ .\mathsf{nil})
\end{aligned}$$

Figure 4.7: Forward secrecy via public keys

```
adec(aenc(M,pub(K)),pri(K)) => M .

# Symmetryc ecryption

dec(enc(M,K),K) => M .

# Pairs

fst(cons(X,Y)) => X .
snd(cons(X,Y)) => Y .

# Expected Result

| @out     : message  => @zzResult : !FAIL! .
| @chk-out : message  => @zzResult : !FAIL! .

%%

new LongTerm .
  ( # Participant A
    ! . new K .
    out enc(pub(K), LongTerm) .
    in KsessEnc . let Ksess = adec(KsessEnc,pri(K)) .
```

```
    in Message . let Message = dec(Message, Ksess) .
    out hash(Message) . ()
  | # Participant B
    ! .
    in PubKEnc . let PubK = dec(PubKEnc, LongTerm) .
    new Ksess .
    out aenc(Ksess, PubK) .
    out enc(message, Ksess) . ()
  | # Adversary
    ( ! . new Nonce . out Nonce . ()
    | ! . in X . in Y . out X . out Y .
          out aenc(X,Y) . out adec(X,Y) .
          out enc(X,Y) . out dec(X,Y) .
          out cons(X,Y) . out fst(X) . out snd(X) .
          out val(X) . out next(X) . ()
    )
  | # Corruption
    in Know . chk .
    ( # Reveal secret
      out LongTerm . out Know . ()
    | # Adversary
      ( ! . new Nonce . out Nonce . ()
      | ! . in X . in Y . out X . out Y .
            out aenc(X,Y) . out adec(X,Y) .
            out pri(X) . out pub(X) .
            out enc(X,Y) . out dec(X,Y) .
            out cons(X,Y) . out fst(X) . out snd(X) .
            out val(X) . out next(X) . ()
      )
    )
  )
```

4.5.4 Kerberos

We now study a protocol involving timestamps. We chose a simplified version of Kerberos [64, 56].

In this protocol, a key exchange is performed by an authentication server AS, a client C and a server S. Initially, the authentication server shares long term keys with the client (kc) and with the server (ks). Upon request from the client, AS generates a fresh key kcs and sends it to the client encrypted with kc. Further, AS also provides a certificate for the freshness of kcs, made of the kcs key itself and the current time, both encrypted by ks. The server S can decrypt the certificate and ensure that kcs is indeed fresh by

checking the timestamp. After that, C and S use kcs to exchange a session key $ksess$, and then proceed exchanging messages encrypted with $ksess$.

We study the rôle of timestamps in the protocol. To that purpose, we introduce a vulnerability in the server S. In our implementation, we let the server to disclose kcs, potentially mining the security of the protocol. However, to keep the game fair, disclosure may only happen after a long time since the timestamp for kcs has been generated. We model this through the occurrence of a chk. Hopefully, if timestamps are properly checked, disclosing a old kcs will not disrupt new sessions of the protocol.

In our specification, we abstractly represent the actual timestamps values as two constants before and after. Initially, the protocol uses only before: any other timestamp value is considered not valid, being in the far past or far future. After chk, the before timestamp has expired, and the protocol has moved to newer timestamps, represented by after. Similarly, we use msg1 and msg2 for the messages exchanged by C and S before and after the chk, respectively.

We expect this faulty protocol implementation not to disclose msg1 until a chk occurs. After chk, we do expect msg1 to be disclosed, but we hope any new msg2 messages to be kept secret.

We specify the above as follows: (we omit parentheses in $P_1|\cdots|P_n$ for readability)

$$
\begin{aligned}
P =& DY|\mathsf{new}\ kc\ .\mathsf{new}\ ks\ .(AS|C|S) \\
AS =& \mathsf{repl.new}\ kcs\ .\mathsf{in}\ nonce\ . \\
& \mathsf{out}\ \mathsf{enc}(\mathsf{cons}(nonce, kcs), kc)\ . \\
& \mathsf{out}\ \mathsf{enc}(\mathsf{cons}(kcs, \mathsf{before}), ks)\ .\mathsf{nil} \\
C =& \mathsf{repl.new}\ nonce\ .\mathsf{out}\ nonce\ .\mathsf{in}\ ticket\ .\mathsf{in}\ cert\ . \\
& \mathsf{let}\ ticketCorrect = \mathsf{enc}(\mathsf{cons}(nonce, \mathsf{snd}(\mathsf{dec}(ticket, kc))), kc)\ . \\
& [ticket = ticketCorrect].\mathsf{let}\ kcs = \mathsf{snd}(\mathsf{dec}(ticket, kc))\ . \\
& \mathsf{new}\ ksess\ .\mathsf{out}\ \mathsf{enc}(ksess, kcs)\ .\mathsf{out}\ cert\ .\mathsf{out}\ \mathsf{enc}(\mathsf{msg1}, ksess)\ . \\
& \mathsf{nil} \\
S =& \mathsf{repl.in}\ tsess\ .\mathsf{in}\ cert\ .\mathsf{let}\ sess = \mathsf{dec}(cert, ks)\ . \\
& \mathsf{let}\ sessCorrect = \mathsf{cons}(\mathsf{fst}(sess), \mathsf{before})\ .[sess = sessCorrect]. \\
& \mathsf{let}\ ksess = \mathsf{dec}(tsess, \mathsf{fst}(sess))\ .\mathsf{in}\ m\ .\mathsf{out}\ \mathsf{hash}(\mathsf{dec}(m, ksess))\ . \\
& Chk \\
Chk =& \mathsf{in}\ know\ .\mathsf{chk}.(\mathsf{out}\ know\ .\mathsf{out}\ sess\ .\mathsf{nil}|AS'|C'|S'|DY) \\
DY =& \mathsf{repl.out}\ \mathsf{before}\ .\mathsf{out}\ \mathsf{after}\ .\mathsf{new}\ nonceDY\ .\mathsf{out}\ nonceDY\ .\mathsf{nil}| \\
& \mathsf{repl.in}\ x\ .\mathsf{in}\ y\ .\mathsf{out}\ x\ .\mathsf{out}\ y\ .\mathsf{out}\ \mathsf{cons}(x, y)\ .\mathsf{out}\ \mathsf{fst}(x)\ .\mathsf{out}\ \mathsf{snd}(x)\ . \\
& \mathsf{out}\ \mathsf{dec}(x, y)\ .\mathsf{out}\ \mathsf{enc}(x, y)\ .\mathsf{out}\ \mathsf{hash}(x)\ .\mathsf{out}\ \mathsf{val}(x)\ .\mathsf{out}\ \mathsf{next}(x)\ . \\
& \mathsf{nil}
\end{aligned}
$$

where AS', C', S' are the same as AS, C, S except that before is replaced with after, msg1 is replaced with msg2, and Chk is replaced with nil. As in the Diffie-Hellman example, our specification, once exchanged a message msg1 or msg2, output its hash.

Using our tool, we generated $\mathcal{F}$ (77 states, 1424 transitions) and verified that msg1 $\notin$ $[$`@out`$]_{\mathcal{F}}$ and msg2 $\notin$ [`@chk-out`], thus establishing the wanted properties. On a side note, we also have msg1 $\in$ [`@chk-out`], as it should be, since msg1 is actually disclosed and our analysis is sound. Our development machine computed the approximation in under one minute.

Below, we give the actual specification passed to our tool.

```
# Encryptions

dec(enc(M,K),K) => M .

# Pairs

fst(cons(X,Y)) => X .
snd(cons(X,Y)) => Y .

# Expected Result

# msg1 should not be disclosed before chk.
| @out     : msg1 => @zzResult : !FAIL! .

# msg1 is *expected* to be disclosed after chk.
| @chk-out : msg1 => @zzResult : expected .

# However, msg2 should be kept secret, even after chk.
| @chk-out : msg2 => @zzResult : !FAIL! .

%%

# Paricipants:
#  AS  Authentication Server
#  C   Client
#  S   Server
#
# Long Term Keys:
#  Kc  shared between AS and C
#  Ks  shared between AS and S
#
# Short Term Keys:
#  Kcs   generated by AS, for C and S; timestamped by AS
```

```
#  Ksess generated by C, for C and S (protected by Kcs)

new Kc . new Ks .
   ( # Authentication server
     ! . new Kcs . in Nonce .
       # Ticket
       out enc(cons(Nonce,Kcs), Kc) .
       # Auth
       out enc(cons(Kcs,before), Ks) . ()
   | # Client
     ! . new Nonce . out Nonce . in Ticket . in Auth .
     let TicketCorrect =
         enc(cons(Nonce,snd(dec(Ticket,Kc))),Kc) .
     [ Ticket = TicketCorrect ] .
     let Kcs = snd(dec(Ticket, Kc)) .
     new Ksess .
     out enc(Ksess, Kcs) . out Auth . out enc(msg1, Ksess) . ()
   | # Server
     ! . in T . in Auth . let Sess = dec(Auth,Ks) .
     let SessCorrect = cons(fst(Sess), before) .
     [ Sess = SessCorrect ] .
     let Ksess = dec(T,fst(Sess)) .
     in M . out f(dec(M,Ksess)) .
     #####
     # Corruption
     in X . chk .
      ( # Corruption data
        out X . out Sess . ()  # Reveals Kcs, Ksess, msg1
      | # Authentication Sever
        ! . new Kcs2 . in Nonce2 .
            out enc(cons(Nonce2,Kcs2), Kc) .
            out enc(cons(Kcs2,after), Ks) . ()
      | # Client
        ! . new Nonce2 . out Nonce2 . in Ticket2 . in Auth2 .
        let TicketCorrect2 =
            enc(cons(Nonce2,snd(dec(Ticket2,Kc))),Kc) .
        [ Ticket2 = TicketCorrect2 ] .
        let Kcs2 = snd(dec(Ticket2, Kc)) .
        new Ksess2 . out enc(Ksess2, Kcs2) . out Auth2 .
        out enc(msg2, Ksess2) . ()
      | # Server
        ! . in T2 . in Auth2 . let Sess2 = dec(Auth2,Ks) .
        let SessCorrect2 = cons(fst(Sess2), after) .
        [ Sess2 = SessCorrect2 ] .
```

```
      let Ksess2 = dec(T2,fst(Sess2)) .
      in M2 . out f(dec(M2,Ksess2)) . ()
    | # Dolev-Yao Adversary
      ! . out before . out after .
          new NonceDY2 . out NonceDY2 . ()
    | ! . in X . in Y . out X . out Y .
          out cons(X,Y) . out fst(X) . out snd(X) .
          out dec(X,Y) . out enc(X,Y) . out f(X) .
          out val(X) . out next(X) . ()
    )
| # Dolev-Yao Adversary
  ! . out before . out after . new NonceDY . out NonceDY . ()
| ! . in X . in Y . out X . out Y .
      out cons(X,Y) . out fst(X) . out snd(X) .
      out dec(X,Y) . out enc(X,Y) . out f(X) .
      out val(X) . out next(X) . ()
)
```

4.6 A Bit of Compositionality

Real-world systems often run many different protocols in a concurrent fashion. However, one usually studies the security properties of each protocol independently. This may not be enough to ensure the integrity of a system, since two otherwise safe protocols may have unwanted interactions, especially if the protocols share secrets. One would rather be able to derive properties about $P_1|P_2$ from the studies of P_1 and P_2.

Our CFA_{io} offers some opportunities for composing security results. Assume P_1 and P_2 were analyzed beforehand, yielding the automata $\mathcal{F}_1$ and $\mathcal{F}_2$. We can build an $\mathcal{F}$ for $P_1|P_2$ by merging the transitions of $\mathcal{F}_1$ and $\mathcal{F}_2$ and adding

$$\begin{aligned}
&\texttt{@in}_1 \rightarrow \texttt{@in}\\
&\texttt{@in}_1 \rightarrow \texttt{@out}_2\\
&\texttt{@in}_2 \rightarrow \texttt{@in}\\
&\texttt{@in}_2 \rightarrow \texttt{@out}_1\\
&\texttt{@out} \rightarrow \texttt{@out}_1\\
&\texttt{@out} \rightarrow \texttt{@out}_2
\end{aligned}$$

just as it happens for the analysis of the parallel operator. Such an $\mathcal{F}$ is sound, provided that $[\texttt{@out}_1]_{\mathcal{F}_1} \subseteq [\texttt{@in}_2]_{\mathcal{F}_2}$ and $[\texttt{@out}_2]_{\mathcal{F}_2} \subseteq [\texttt{@in}_1]_{\mathcal{F}_1}$. This last proof obligation might be checked by static analysis. If the obligations do not hold (or cannot be proved), the completion algorithm can be restarted from the above $\mathcal{F}$ to compute a sound approximation. This could be less expensive than rebuilding the approximation from scratch, since parts of the work have been already done when computing $\mathcal{F}_1$ and $\mathcal{F}_2$.

4.7 Future Work

The handling of chk in our static semantics is quite simple: for $\mathrm{CFA}_{\mathsf{io}}$, we simply proceed using the states `@chk-in` and `@chk-out` for *in* and *out*. Yet, this is enough to ensure soundness. The fact that handling chk is so simple is mainly due to the corresponding simplicity of rule Chk in the dynamic semantics. In fact, when a chk prefix is fired, all running threads are aborted, and the system restarts from a fixed, statically-known configuration.

While this allows for the study of some limited causal dependencies, one might wish to use a less drastic semantic rule for chk. Ideally, chk would just cause the current point in time to be marked, and have no visible effect on running concurrent threads. In this scenario, one would use the rule

$$\mathsf{Chk}' \ \frac{P@\theta = \mathsf{chk}.P'}{\sigma, \langle \rho, \theta \rangle \xrightarrow{\mathsf{chk}} \sigma, \langle \rho, \mathsf{n}\theta \rangle}$$

However, our $\mathrm{CFA}_{\mathsf{io}}$ of Sect. 4.4 is not sound w.r.t. this rule: we need a more complex static semantics. We attempted to define such an analysis. Our current best attempt, which we do not present here, generates a number of states which is quadratic in the size of the process. In comparison, the $\mathrm{CFA}_{\mathsf{io}}$ generates a linear number of states. Further, during our tests, the analysis often reused some inappropriate state, causing a loss of precision, and leading to the failure of the analysis goal. This problem was amplified by the large number of states.

In order to proceed in this direction, we need more precise heuristics. Currently, no information about the processes is given to the completion tool other than $\mathcal{A}, \mathcal{I}, \mathcal{R}$. For instance, the tool can not try to separate the "before chk" states from the "after chk" ones, so that we reuse a state of a kind only if we are dealing with states of the same kind. More information about the high-level meaning of the states might provide better hints to the state reuse heuristics. Also, we could consider a more careful allocation of new states, so that we do not waste all the resources to precisely approximate non significant languages. We could also try to separate our constraints into "mostly independent" parts and solve them individually, as described in Sect. 4.6, and merge them later: this would allow to specify how much resources we assign to each part, as well as effectively hinder state reuse from different parts.

4.8 Related Techniques and Tools

Our technique exploits concepts from the CFA [57, 16] and from the tree automaton approximation techniques [28] presented in Chapter 3. In Sect. 3.1 and 3.9.6, we briefly compared our approximation tool to other related tools, namely to the Succinct Solver [65] and Timbuk [67]. Here, we discuss

the differences of our approach w.r.t. related ones, focusing more on the aspects relevant to security protocol analysis.

Succinct Solver The Succinct Solver [65] does not use term rewriting. For representing sets of terms, it uses a abstract domain based on tree automata, but the term algebra is assumed to be free. Constraints on these sets of terms are not expressed through automata transitions, rather with Alternation-free Least FixPoint (ALFP) clauses, a fragment of first-order logic. This allows one to express constraints in a familiar way, using the usual logic quantifiers and connectives, as long as the resulting clauses are within the ALFP fragment. Of course, one could use these constraints to require the sets of terms to be closed under some rewriting. This is possible for some rewriting rule sets, e.g. for algebras of constructors and destructors. However, some rules, e.g. associativity, can not be expressed without making the tool diverge, hindering the verification of protocols involving exponentials. The cause of this limitation is to be found in the fact that the Succinct Solver performs no normalization of transitions, so terms are not always kept *plain* but can have arbitrary depth, possibly leading to an infinite domain and hence to non termination. Related to this, the tool never generates fresh automaton states, which are necessary for normalization.

Timbuk The Timbuk tool [67] performs many interesting operations on tree automata, including the completion for closing a tree language under rewriting. A large amount of work has been done to ensure that, for certain classes of rewritings, the result of the completion not only is sound – including all possible rewritten terms – but also complete – including no other terms. However, in the general case completeness can not be achieved due undecidability issues. Here, Timbuk requires the user to specify its own normalizations, either through a specification file, or through manual user interaction, when the file does not handle all the cases. This might be a problem for constructing tools for unattended, fully automatic verification. Indeed, TA4SP [19] provides a modified version of Timbuk that does not require user interaction.

TA4SP Tree Automata for Security Protocols (TA4SP) [19] is a tool for protocol verification based on Timbuk. TA4SP is part of the tool suite AVISPA [7]. A main difference between our technique and that of TA4SP is the choice of the protocol specification language. TA4SP, as the other tools in AVISPA, use the High Level Protocol Specification Language (HLPSL), a very rich language: protocols are specified by describing the states of the participants, and the transitions between states. Instead, we use a fairly simple process algebra, based on the applied pi calculus. We feel our specifications simpler, and more intuitive, than similar ones in HLPSL. We invite

the reader to compare the specifications we included in this chapter with the ones in [7].

Moreover, our technique extracts from the protocol a set of constraints forming an automaton $\mathcal{A}$, while the rewriting system $\mathcal{R}$ is independent of the protocol, but only contains the rules for the cryptographic primitives. Instead, in TA4SP, protocol logic affects both $\mathcal{A}$ and $\mathcal{R}$.

Another difference between our tool and TA4SP is that, at the time of this writing, TA4SP does not handle general rewriting rules. Currently, the user can not specify its own rules: the term algebra is fixed, and only allows for some cryptographic primitives, e.g. encryption. In particular, the exp operator is not supported. This prevents us to make a detailed comparison on the handling of the exponentials by TA4SP/Timbuk and our tool. For future research, we could try to use the amended Timbuk as our back-end. Finally, some support for handling more general rewriting rules in TA4SP is starting to appear: in [20] an example involving xor is presented.

Dynamic analyzers While our technique, as most static analyses, aim to prove protocol secure, dynamic analyzers usually try to find attacks, i.e. to prove protocols *insecure*. The two goals being complementary, it is not possible to make a direct comparison of these kinds of results[3]. However, we mention here the On-the-Fly Model-Checker (OFMC) [54]. OFMC takes algebraic properties into account, and is therefore able to handle exponentials in finding attack traces.

4.9 Conclusions

We presented a simple model for the specification of cryptographic protocols, based on process calculi and term rewriting. We stress that we allow *any* rewriting system for defining the cryptographic primitives. This is a nice consequence of adopting our algorithms on tree automata of Chapter 3. Further, the model deals with some basic temporal aspects, and therefore it is suitable to express certain security properties involving time, such as forward secrecy.

We addressed the problem of statically verifying protocols. To this purpose we studied two techniques, CFA and CFA_{io}, for such static analysis. In both cases, we compute a static approximation of the dynamic behaviour so that it is closed under rewritings. We also focused on foreseeing the protocol behaviour before and after a selected point in time, represented by the firing of chk. Also, we explored some opportunities for composing results of our CFA_{io}.

[3] Of course, producing a contradiction through two tools, e.g. "proving" a protocol as secure and insecure at the same time, would point out a flaw in one of the tools.

We implemented these analyses [61], and used our tool to check some significant protocols. The tool confirmed that we can handle complex rewriting rules, such as those of exponentials, and protocols involving timestamps.

Chapter 5

Conclusions

In this work, we studied techniques for reasoning about the security of distributed systems. Mainly, we focused on methods for actually proving systems secure. For this, we considered models for both the *specification* and the *verification* of cryptographic protocols. We argued that *formal methods* offer tools for simplifying both these tasks, so the designer of a protocol can

- write a reasonably short and clear specification, and
- prove security properties in a convenient way.

We also discussed some common *assumptions* of formal models, and relaxed them. First, in Chapter 2 we weakened the *perfect encryption* assumption. We allowed the formal Dolev-Yao adversary to break encryptions with low probability, as it happens in *computational* models. We then showed that the classes of secure protocols w.r.t. the standard adversary and the enhanced one are actually the same class.

In the next chapters we relaxed the *free algebra* assumption. We first defined in Chapter 3 some techniques for approximating sets of terms up to rewriting. Then, in Chapter 4, we applied these techniques and derived a static analysis for protocols, including those involving non-free primitives. Finally, we implemented the analysis and built a tool for verifying protocols. Our experiments with the tool confirmed the effectiveness of our analysis.

It is worth noting that the results of Chapters 2, 3, and 4 can be applied *together* to the same protocol. Assume that we run our static analysis tool, compute the result of the analysis for a protocol using encryptions, and derive the secrecy of a message. This secrecy result still holds when we consider non-perfect encryptions, as we did in Chapter 2. Indeed, we can relax at the same time both the free algebra assumption and the perfect encryption one.

Of course, the gap between formal models, including ours, and computational models is still open, but is slowly narrowing. In the next years, we

hope to see other contributions in this line, establishing more connections between the two worlds, helping to bridging this gap.

Acknowledgments

I wish to thank all the people that helped me in developing the ideas behind this thesis. This includes virtually every person I talked with at the Dipartimento di Informatica, as well as at the conferences I attended. The support of friends and colleagues was invaluable. Also, the anonymous reviewers of my papers often provided nice suggestions, good advices, and useful pointers.

Pierpaolo Degano constantly gave me many insightful comments. It is very hard for me to estimate the enormous amount of time he dedicated to me and our works. His contribute surely had a significant impact on the overall quality of our results. From general suggestions to detailed corrections, his support was outstanding.

Finally, I warmly thank the referees of this thesis, Chris Hankin and Luca Viganò. Their useful remarks, sharp suggestions, and constructive criticism greatly helped me in improving the overall presentation.

Roberto Zunino

Bibliography

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 5(46):18–36, 1999.

[2] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of ACM*, 52(1):102–146, 2005.

[3] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL'01), pages 104-115.*, 2001.

[4] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Journal of Information and Computation*, 148(1):1–70, 1999.

[5] M. Abadi and J. Jürjens. Formal eavesdropping and its computational interpretation. In *Proceedings of TACS 2001,* Lecture Notes in Computer Science, volume 2215, page 82, 2001.

[6] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *Proceedings of IFIP TCS2000,* Lecture Notes in Computer Science*, Vol. 1872*, pages 3–22, 2000.

[7] AVISPA project home page. `http://www.avispa-project.org`.

[8] M. Backes and C. Jacobi. Cryptographically sound and machine-assisted verification of security protocols. In *Proceedings of 20th International Symposium on Theoretical Aspects of Computer Science, STACS '03*, pages 675–686, 2003.

[9] M. Backes, B. Pfitzmann, and M. Waidner. Symmetric authentication within a simulatable cryptographic library. In *8th European Symposium on Research in Computer Security (ESORICS 2003),* Lecture Notes in Computer Science*, Vol. 2808*, pages 271–290, 2003.

[10] M. Baudet. Random polynomial-time attacks and Dolev-Yao models. In Siva Anantharaman, editor, *Proceedings of the Workshop on Security*

of Systems: Formalism and Tools (SASYFT'04), Orléans, France, June 2004.

[11] M. Baudet. Random polynomial-time attacks and Dolev-Yao models. *Journal of Automata, Languages and Combinatorics*, 2005. To appear.

[12] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Crypto 93,* Lecture Notes in Computer Science*, Vol. 773*, page 232, 1994.

[13] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, 2005.

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

[15] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for secrecy and non-interference in networks of processes. *Lecture Notes in Computer Science*, 2127, 2001.

[16] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for the π-calculus with application to security. *Journal of Information and Computation*, 168(1):68–92, 2001.

[17] C. Bodei, P. Degano, H. Riis Nielson, and F. Nielson. Flow logic for Dolev-Yao secrecy in cryptographic processes. *Future Generation Computer Systems*, 18(6), 2002.

[18] C. Bodei, P. Degano, and C. Priami. Checking security policies through an enhanced control flow analysis. *Journal of Computer Security*, 13(1):49–85, 2005.

[19] Y. Boichut. Tree automata for security protocols (TA4SP) tool. `http://lifc.univ-fcomte.fr/~boichut/TA4SP/TA4SP.html`.

[20] Y. Boichut, P. Héam, and O. Kouchnarenko. Handling algebraic properties in automatic analysis of security protocols. Research Report 5857, INRIA, 2006.

[21] N. Borisov, I. Goldberg, and D. Wagner. Intercepting mobile communications: The insecurity of 802.11. `http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html`, 2001.

[22] M. Burrows, M. Abadi, and R.M. Needham. A logic of authentication. In *Proc. of the Royal Society of London A*, volume 426, pages 233–271, 1989.

[23] R. Canetti and H. Krawczyk. Universally composable notions of key exchange and secure channels. Cryptology ePrint Archive, Report 2002/059, 2002.

[24] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proc. of POPL'99*, pages 79–92. ACM Press, 1999.

[25] IBM CCA basic services reference and guide, release 2.54.

[26] I. Cervesato, N. A. Durgin, J. C. Mitchell, P. D. Lincoln, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *13-th IEEE Computer Security Foundations Workshop*, pages 35–51, 2000.

[27] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In Alessandra Di Pierro and Herbert Wiklicky, editors, *Electronic Notes in Theoretical Computer Science*, volume 59(3), 2002.

[28] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release October, 1rst 2002.

[29] The Coq proof assistant. `http://coq.inria.fr`.

[30] J. Courant and J. Monin. Defending the bank with a proof assistant. In *Sixth International IFIP WG 1.7 Workshop on Issues in the Theory of Security (WITS)*, pages 87–98, 2006.

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

[32] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, 1983.

[33] G. Feuillade, T. Genet, and V. V. T. Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 33:341–383, 2004.

[34] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[35] R. Focardi, R. Gorrieri, and F. Martinelli. Non interference for the analysis of cryptographic protocols. In *Proceedings of ICALP 2000,* Lecture Notes in Computer Science, Vol. 1853, pages 354–372, 2000.

[36] T. Genet and F. Klay. Rewriting for cryptographic protocol verification. In *Proceeding of CADE*, pages 271–290, 2000.

[37] T. Genet, Y. T. Tang-Talpin, and V. V. T. Tong. Verification of copy-protection cryptographic protocol using approximations of term rewriting systems. In *Proc. of Workshop on Issues in the Theory of Security*, 2003.

[38] T. Genet and V. V. T. Tong. Reachability analysis of term rewriting systems with timbuk. In *Proc. of Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2250 of *Lecture Notes in Computer Science*, 2001.

[39] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[40] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the 17th Annual ACM Symposium on the Theory of Computing*, pages 291–304, 1985.

[41] J. Goubault-Larrecq, M. Roger, and K. N. Verma. Abstraction and resolution modulo AC: How to verify Diffie-Hellman-like protocols automatically. *Journal of Logic and Algebraic Programming*, 64(2):219–251, 2005.

[42] The Haskell home page. `http://www.haskell.org`.

[43] J. Herzog. A computational interpretation of dolev-yao adversaries. In *Proceedings of WITS 2003*, pages 146–155, 2003.

[44] F. Jacquemard. Decidable approximations of term rewriting systems. In *Proc. of Rewriting Techniques and Applications*, pages 362–376, 1996.

[45] C. W. Johnson. A probabilistic logic for the development of safety-critical, interactive systems. *International Journal of Man-Machine Studies*, 39(2):333–351, 1993.

[46] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Journal of Information and Computation*, 94(1):1–28, 1991.

[47] P. Laud. Secrecy types for a simulatable cryptographic library. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 26–35, New York, NY, USA, 2005. ACM Press.

[48] G. Lowe. Analysing protocols subject to guessing attacks. In *Proceedings of WITS 2002*, 2002.

[49] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[50] J. K. Millen. The interrogator: A tool for cryptographic protocol security. In *Proceedings of the 1984 Symposium on Security and Privacy (SSP '84)*, pages 134–141, 1990.

[51] J. K. Millen and V. Shmatikov. Symbolic protocol analysis with products and Diffie-Hellman exponentiation. In *Computer Security Foundations Workshop*, 2003.

[52] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[53] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murϕ. In *Proceedings of the 1997 Conference on Security and Privacy*, pages 141–153, 1997.

[54] S. Mödersheim, P. H. Drielsma, and P. Schaller. On-the-fly model-checker. `http://www.inf.ethz.ch/personal/moederss/`.

[55] D. Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 47(2–3):177–202, 2003.

[56] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32:33–38, 1994.

[57] F. Nielson, H. Riis Nielson, and H. Seidl. Cryptographic analysis in cubic time. *Electronic Notes in Theoretical Computer Science*, 62, 2002.

[58] A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non–interference. In *Proceedings of CSFW'02 – 15th IEEE Computer Security Foundations Workshop*, pages 3–17, 2002.

[59] F. Pottier. A simple view of type-secure information flow in the π-calculus. In *CSFW-15*, pages 320–330, 2002.

[60] A. Ramanathan, J. Mitchell, A. Scedrov, and V. Teague. Probabilistic bisimulation and equivalence for security analysis of network protocols. In *FOSSACS 2004,* Lecture Notes in Computer Science*, Vol. 2987*, pages 468–483, 2004.

[61] The Rewrite completion and protocol analysis tool. `http://www.di.unipi.it/~zunino/software`.

[62] S. Schneider. Security Properties and CSP. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 174–187, 1996.

[63] V. Shoup. On formal models for secure key exchange. *1999 revision of IBM Research Report RZ 3120*, 1999.

[64] J. G. Steiner, B. C. Neuman, and J. I. Shiller. Kerberos: An authentication service for open network systems. In *Proc. of the Winter 1988 Usenix Conference*, pages 191–201, 1988.

[65] Succinct solver. `http://www2.imm.dtu.dk/cs_SuccinctSolver/index.htm`.

[66] T. Takai. A verification technique using term rewriting systems and abstract interpretation. In *Proc. of Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 119 – 133, 2004.

[67] Timbuk tree automata tool. `http://www.irisa.fr/lande/genet/timbuk`.

[68] A. Troina, A. Aldini, and R. Gorrieri. A probabilistic formulation of imperfect cryptography. In N. Busi, R. Gorrieri, and F. Martinelli, editors, *Proceedings of Int. Workshop on Issues in Security and Petri Nets (WISP'03)*, pages 41–55, 2003.

[69] S. Warshall. A theorem on boolean matrices. *Journal of ACM*, 9(1):11–12, 1962.

[70] 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*, page 178, 1993.

[71] R. Zunino. Control flow analysis for the applied π–calculus. In *Proceedings of the MEFISTO Project 2003*, volume 99 of *ENTCS*, pages 87–110, 2004.

[72] R. Zunino and P. Degano. Finite approximations of terms up to rewriting. `http://www.di.unipi.it/~zunino/papers/completion.html`.

[73] R. Zunino and P. Degano. A note on the perfect encryption assumption in a process calculus. In *Proceeding of FoSSaCS 2004*, volume 2987 of *Lecture Notes in Computer Science*, pages 514–528, Jan 2004.

[74] R. Zunino and P. Degano. Weakening the perfect encryption assumption in Dolev-Yao adversaries. *Theoretical Computer Science*, 340(1):154–178, 2005.

[75] R. Zunino and P. Degano. Handling exp, $\times$ (and timestamps) in protocol analysis. In *Proceedings of FoSSaCS 2006*, volume 3921 of *Lecture Notes in Computer Science*, pages 413–427, 2006.